Introduction to Computational Data Science Using ScalaTion

John A. Miller
Department of Computer Science
University of Georgia

November 10, 2021
# Probability

- **3.1 Probability Measure** .......................... 43
  - 3.1.1 Joint Probability .......................... 43
  - 3.1.2 Conditional Probability ................. 44
- **3.2 Random Variable** ............................ 45
  - 3.2.1 Discrete Random Variable ............... 45
  - 3.2.2 Continuous Random Variable ......... 45
- **3.3 Probability Distribution** ................. 46
  - 3.3.1 Cumulative Distribution Function ...... 46
  - 3.3.2 Probability Mass Function ............ 46
  - 3.3.3 Probability Density Function .......... 47
- **3.4 Empirical Distribution** ................... 50
- **3.5 Expectation** ................................ 51
  - 3.5.1 Continuous Case .......................... 51
  - 3.5.2 Discrete Case ............................ 51
  - 3.5.3 Variance .................................. 51
  - 3.5.4 Covariance .................................. 52
- **3.6 Algebra of Random Variables** ............ 54
  - 3.6.1 Expectation is a Linear Operator ...... 54
  - 3.6.2 Variance is not a Linear Operator ...... 54
  - 3.6.3 Convolution of Probability Distributions .................................. 54
  - 3.6.4 Central Limit Theorem .................... 55
- **3.7 Median, Mode and Quantiles** ............ 57
  - 3.7.1 Median ..................................... 57
  - 3.7.2 Quantile .................................... 57
  - 3.7.3 Mode ....................................... 57
- **3.8 Joint, Marginal and Conditional Distributions** .......... 58
  - 3.8.1 Discrete Case: Joint and Marginal Mass ............. 58
  - 3.8.2 Continuous Case: Joint and Marginal Density ................. 59
  - 3.8.3 Discrete Case: Conditional Mass .............. 60
  - 3.8.4 Continuous Case: Conditional Density .......... 60
  - 3.8.5 Independence .................................. 61
  - 3.8.6 Conditional Expectation ................... 62
  - 3.8.7 Conditional Independence ............... 63
- **3.9 Odds** ........................................ 64
- **3.10 Example Problems** ......................... 65
- **3.11 Estimating Parameters from Samples** .... 67
  - 3.11.1 Sample Mean .................................. 67
  - 3.11.2 Confidence Interval ..................... 68
  - 3.11.3 Estimation for Discrete Outcomes/Responses .......... 70
- **3.12 Entropy** .................................... 72
  - 3.12.1 Joint Entropy ............................. 73
  - 3.12.2 Conditional Entropy ..................... 73
### Data Management

#### 4.1 Introduction

#### 4.2 Columnar Database Model

#### 4.3 Columnar Relational Algebra API

#### 4.4 SQL-Like API for Columnar Database

#### 4.5 Exercises

#### 4.6 Graph Databases

#### 4.7 Directed Graphs

#### 4.8 Directed Multi-Graphs

#### 4.9 Property Graphs

#### 4.10 Special Types of Graph Databases

#### 4.11 Exercises

### Data Preprocessing

#### 5.1 Basic Operations

#### 5.2 Detect Outliers

#### 5.3 Imputation Techniques

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.12.3 Relative Entropy</td>
<td>74</td>
</tr>
<tr>
<td>3.12.4 Cross Entropy</td>
<td>75</td>
</tr>
<tr>
<td>3.12.5 Mutual Information</td>
<td>75</td>
</tr>
<tr>
<td>3.13 Exercises</td>
<td>78</td>
</tr>
<tr>
<td>3.14 Further Reading</td>
<td>81</td>
</tr>
<tr>
<td>3.15 Notational Conventions</td>
<td>82</td>
</tr>
<tr>
<td>3.16 Model</td>
<td>83</td>
</tr>
<tr>
<td>4 Data Management</td>
<td>87</td>
</tr>
<tr>
<td>4.1 Introduction</td>
<td>87</td>
</tr>
<tr>
<td>4.1.1 Analytics Databases</td>
<td>88</td>
</tr>
<tr>
<td>4.2 Columnar Database Model</td>
<td>88</td>
</tr>
<tr>
<td>4.3 Columnar Relational Algebra API</td>
<td>90</td>
</tr>
<tr>
<td>4.3.1 Relation Creation</td>
<td>90</td>
</tr>
<tr>
<td>4.3.2 Relation Population</td>
<td>91</td>
</tr>
<tr>
<td>4.3.3 Columnar Relational Algebra Operators</td>
<td>91</td>
</tr>
<tr>
<td>4.3.4 Example Queries</td>
<td>94</td>
</tr>
<tr>
<td>4.4 SQL-Like API for Columnar Database</td>
<td>96</td>
</tr>
<tr>
<td>4.4.1 Relation Creation</td>
<td>96</td>
</tr>
<tr>
<td>4.4.2 Sample Queries</td>
<td>96</td>
</tr>
<tr>
<td>4.5 Exercises</td>
<td>99</td>
</tr>
<tr>
<td>4.6 Graph Databases</td>
<td>100</td>
</tr>
<tr>
<td>4.7 Directed Graphs</td>
<td>101</td>
</tr>
<tr>
<td>4.8 Directed Multi-Graphs</td>
<td>103</td>
</tr>
<tr>
<td>4.9 Property Graphs</td>
<td>105</td>
</tr>
<tr>
<td>4.9.1 Structure of a Property Graph</td>
<td>105</td>
</tr>
<tr>
<td>4.9.2 Native Storage</td>
<td>108</td>
</tr>
<tr>
<td>4.9.3 High-Level Query Language for Graph Databases</td>
<td>110</td>
</tr>
<tr>
<td>4.9.4 Graph Algebra</td>
<td>110</td>
</tr>
<tr>
<td>4.9.5 Query Processing in Graph Databases</td>
<td>112</td>
</tr>
<tr>
<td>4.10 Special Types of Graph Databases</td>
<td>113</td>
</tr>
<tr>
<td>4.10.1 Embedding Relationships in Vertex Types</td>
<td>113</td>
</tr>
<tr>
<td>4.10.2 Resource Description Framework (RDF) Graphs</td>
<td>114</td>
</tr>
<tr>
<td>4.10.3 Knowledge Graphs</td>
<td>114</td>
</tr>
<tr>
<td>4.11 Exercises</td>
<td>116</td>
</tr>
<tr>
<td>5 Data Preprocessing</td>
<td>117</td>
</tr>
<tr>
<td>5.1 Basic Operations</td>
<td>117</td>
</tr>
<tr>
<td>5.1.1 Remove Identifiers</td>
<td>117</td>
</tr>
<tr>
<td>5.1.2 Convert String Columns to Numeric Columns</td>
<td>117</td>
</tr>
<tr>
<td>5.1.3 Identify Missing Values</td>
<td>117</td>
</tr>
<tr>
<td>5.2 Detect Outliers</td>
<td>117</td>
</tr>
<tr>
<td>5.3 Imputation Techniques</td>
<td>118</td>
</tr>
<tr>
<td>5.3.1 Imputation Trait</td>
<td>118</td>
</tr>
</tbody>
</table>
# Modeling

## Prediction

6.1 Predictor

6.1.1 Fit

6.1.2 PredictorMat

6.2 Null Model

6.2.1 Model Equation

6.2.2 Training

6.2.3 Optimization - Derivative

6.2.4 Example Calculation

6.2.5 Exercises

6.3 Simpler Regression

6.3.1 Model Equation

6.3.2 Training

6.3.3 Optimization - Derivative

6.3.4 Example Calculation

6.3.5 Exercises

6.4 Simple Regression

6.4.1 Model Equation

6.4.2 Training

6.4.3 Optimization - Gradient

6.4.4 Example Calculation

6.4.5 Exploratory Data Analysis

6.4.6 Exercises

6.5 Regression

6.5.1 Model Equation

6.5.2 Training

6.5.3 Optimization - Gradient

6.5.4 Matrix Inversion Technique

6.5.5 LU Factorization Technique

6.5.6 Cholesky Factorization Technique

6.5.7 QR Factorization Technique

6.5.8 Use of Factorization in Regression

6.5.9 Model Assessment

6.5.10 Model Validation

6.5.11 Collinearity

6.5.12 Feature Selection
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.9</td>
<td>Random Forest</td>
<td>278</td>
</tr>
<tr>
<td>8.10</td>
<td>Support Vector Machine</td>
<td>279</td>
</tr>
<tr>
<td>9</td>
<td>Generalized Linear and Additive Models</td>
<td>281</td>
</tr>
<tr>
<td>9.1</td>
<td>Generalized Linear Model</td>
<td>281</td>
</tr>
<tr>
<td>9.2</td>
<td>Maximum Likelihood Estimation</td>
<td>284</td>
</tr>
<tr>
<td>9.2.1</td>
<td>Akaike Information Criterion</td>
<td>285</td>
</tr>
<tr>
<td>9.2.2</td>
<td>MLE for Generalized Linear Models</td>
<td>285</td>
</tr>
<tr>
<td>9.3</td>
<td>Poisson Regression</td>
<td>287</td>
</tr>
<tr>
<td>9.4</td>
<td>Generalized Additive Models</td>
<td>289</td>
</tr>
<tr>
<td>9.5</td>
<td>Regression Trees</td>
<td>290</td>
</tr>
<tr>
<td>9.5.1</td>
<td>Example Problem</td>
<td>290</td>
</tr>
<tr>
<td>9.5.2</td>
<td>Regions</td>
<td>291</td>
</tr>
<tr>
<td>9.5.3</td>
<td>Determining Thresholds</td>
<td>291</td>
</tr>
<tr>
<td>9.6</td>
<td>Exercises</td>
<td>292</td>
</tr>
<tr>
<td>9.7</td>
<td>Further Reading</td>
<td>292</td>
</tr>
<tr>
<td>10</td>
<td>Nonlinear Models and Neural Networks</td>
<td>293</td>
</tr>
<tr>
<td>10.1</td>
<td>Nonlinear Regression</td>
<td>294</td>
</tr>
<tr>
<td>10.1.1</td>
<td>Model Equation</td>
<td>294</td>
</tr>
<tr>
<td>10.1.2</td>
<td>Training</td>
<td>294</td>
</tr>
<tr>
<td>10.1.3</td>
<td>Optimization</td>
<td>296</td>
</tr>
<tr>
<td>10.1.4</td>
<td>Exercises</td>
<td>298</td>
</tr>
<tr>
<td>10.2</td>
<td>Simple Exponential Regression</td>
<td>299</td>
</tr>
<tr>
<td>10.2.1</td>
<td>Model Equation</td>
<td>299</td>
</tr>
<tr>
<td>10.2.2</td>
<td>Training</td>
<td>299</td>
</tr>
<tr>
<td>10.2.3</td>
<td>Optimization</td>
<td>299</td>
</tr>
<tr>
<td>10.2.4</td>
<td>Linearization</td>
<td>300</td>
</tr>
<tr>
<td>10.2.5</td>
<td>Exercises</td>
<td>301</td>
</tr>
<tr>
<td>10.3</td>
<td>Exponential Regression</td>
<td>303</td>
</tr>
<tr>
<td>10.3.1</td>
<td>Exercises</td>
<td>304</td>
</tr>
<tr>
<td>10.4</td>
<td>Perceptron</td>
<td>305</td>
</tr>
<tr>
<td>10.4.1</td>
<td>Model Equation</td>
<td>305</td>
</tr>
<tr>
<td>10.4.2</td>
<td>Training</td>
<td>306</td>
</tr>
<tr>
<td>10.4.3</td>
<td>Optimization</td>
<td>307</td>
</tr>
<tr>
<td>10.4.4</td>
<td>Example Calculation for $\epsilon$ and $\delta$</td>
<td>309</td>
</tr>
<tr>
<td>10.4.5</td>
<td>Initializing Weights/Parameters</td>
<td>312</td>
</tr>
<tr>
<td>10.4.6</td>
<td>Activation Functions</td>
<td>312</td>
</tr>
<tr>
<td>10.4.7</td>
<td>Basic Gradient Descent Algorithm</td>
<td>314</td>
</tr>
<tr>
<td>10.4.8</td>
<td>Exercises</td>
<td>317</td>
</tr>
<tr>
<td>10.5</td>
<td>Multi-Output Prediction</td>
<td>320</td>
</tr>
<tr>
<td>10.5.1</td>
<td>Model Equation</td>
<td>320</td>
</tr>
<tr>
<td>10.5.2</td>
<td>Training</td>
<td>320</td>
</tr>
<tr>
<td>10.6</td>
<td>Two-Layer Neural Networks</td>
<td>323</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>10.6.1 Model Equation</td>
<td>323</td>
<td></td>
</tr>
<tr>
<td>10.6.2 Training</td>
<td>324</td>
<td></td>
</tr>
<tr>
<td>10.6.3 Optimization</td>
<td>324</td>
<td></td>
</tr>
<tr>
<td>10.6.4 Matrix Version</td>
<td>325</td>
<td></td>
</tr>
<tr>
<td>10.6.5 Exercises</td>
<td>327</td>
<td></td>
</tr>
<tr>
<td>10.7 Three-Layer Neural Networks</td>
<td>328</td>
<td></td>
</tr>
<tr>
<td>10.7.1 Model Equation</td>
<td>328</td>
<td></td>
</tr>
<tr>
<td>10.7.2 Training</td>
<td>329</td>
<td></td>
</tr>
<tr>
<td>10.7.3 Optimization</td>
<td>330</td>
<td></td>
</tr>
<tr>
<td>10.7.4 Matrix Version</td>
<td>332</td>
<td></td>
</tr>
<tr>
<td>10.7.5 Gradient Descent Algorithm</td>
<td>333</td>
<td></td>
</tr>
<tr>
<td>10.7.6 Example Error Calculation Problem</td>
<td>334</td>
<td></td>
</tr>
<tr>
<td>10.7.7 Exercises</td>
<td>336</td>
<td></td>
</tr>
<tr>
<td>10.8 Multi-Hidden Layer Neural Networks</td>
<td>338</td>
<td></td>
</tr>
<tr>
<td>10.8.1 Model Equation</td>
<td>338</td>
<td></td>
</tr>
<tr>
<td>10.8.2 Training</td>
<td>339</td>
<td></td>
</tr>
<tr>
<td>10.8.3 Optimization</td>
<td>339</td>
<td></td>
</tr>
<tr>
<td>10.8.4 Number of Nodes in Hidden Layers</td>
<td>340</td>
<td></td>
</tr>
<tr>
<td>10.8.5 Avoidance of Overfitting</td>
<td>341</td>
<td></td>
</tr>
<tr>
<td>10.8.6 Deep Learning</td>
<td>341</td>
<td></td>
</tr>
<tr>
<td>10.8.7 Exercises</td>
<td>342</td>
<td></td>
</tr>
<tr>
<td>10.9 Convolutional Networks</td>
<td>345</td>
<td></td>
</tr>
<tr>
<td>10.10.1D Conv-Net</td>
<td>346</td>
<td></td>
</tr>
<tr>
<td>10.10.1 Model Equation</td>
<td>347</td>
<td></td>
</tr>
<tr>
<td>10.10.2 Training</td>
<td>348</td>
<td></td>
</tr>
<tr>
<td>10.10.3 Optimization</td>
<td>348</td>
<td></td>
</tr>
<tr>
<td>10.10.4 Matrix Version</td>
<td>350</td>
<td></td>
</tr>
<tr>
<td>10.10.5 Gradient Descent Algorithm</td>
<td>350</td>
<td></td>
</tr>
<tr>
<td>10.10.6 Example Error Calculation Problem</td>
<td>351</td>
<td></td>
</tr>
<tr>
<td>10.10.7 Two Convolutional Filters</td>
<td>352</td>
<td></td>
</tr>
<tr>
<td>10.10.8 Exercises</td>
<td>354</td>
<td></td>
</tr>
<tr>
<td>10.11D Conv-Net</td>
<td>355</td>
<td></td>
</tr>
<tr>
<td>10.11.1 Filtering Operation</td>
<td>355</td>
<td></td>
</tr>
<tr>
<td>10.11.2 Pooling Operation</td>
<td>357</td>
<td></td>
</tr>
<tr>
<td>10.11.3 Flattening Operation</td>
<td>357</td>
<td></td>
</tr>
<tr>
<td>10.11.4 Model Equation</td>
<td>358</td>
<td></td>
</tr>
<tr>
<td>10.11.5 Training</td>
<td>358</td>
<td></td>
</tr>
<tr>
<td>10.11.6 Optimization</td>
<td>358</td>
<td></td>
</tr>
<tr>
<td>10.11.7 Exercises</td>
<td>358</td>
<td></td>
</tr>
<tr>
<td>10.12 Graph Neural Networks</td>
<td>360</td>
<td></td>
</tr>
<tr>
<td>10.12.1 AGGREGATE and COMBINE Operations</td>
<td>360</td>
<td></td>
</tr>
<tr>
<td>10.13 Transfer Learning</td>
<td>363</td>
<td></td>
</tr>
<tr>
<td>10.13.1 Definition of Transfer Learning</td>
<td>363</td>
<td></td>
</tr>
</tbody>
</table>
10.13.2 Type of Transfer Learning ........................................... 364
10.13.3 Exercises .............................................................. 366
10.14 Extreme Learning Machines ............................................. 367
10.14.1 Model Equation ......................................................... 367
10.14.2 Training ................................................................. 367
10.14.3 Optimization ........................................................... 367
10.14.4 Exercises ............................................................... 368

11 Time-Series/Temporal Models .............................................. 369
11.1 ForecasterVec ............................................................... 370
11.1.1 Auto-Correlation Function ........................................... 371
11.1.2 Quality of Fit (QoF) for Time-Series Data .......................... 372
11.2 Random Walk ............................................................. 373
11.2.1 White Noise ............................................................. 373
11.2.2 Detecting Random Walks ............................................ 373
11.2.3 Forecasting Lake Levels .............................................. 373
11.2.4 Random Walk with Drift ............................................ 375
11.2.5 Exercises ............................................................... 375
11.3 Simple Exponential Smoothing ......................................... 376
11.3.1 Model Equation ......................................................... 376
11.3.2 Training ................................................................. 376
11.3.3 Effect of the Smoothing Parameter ................................. 377
11.3.4 Simple Exponential Smoothing with Growth ...................... 377
11.3.5 Exercises ............................................................... 378
11.4 Auto-Regressive (AR) Models ........................................... 379
11.4.1 AR(1) Model ............................................................ 380
11.4.2 AR(p) Model ........................................................... 381
11.4.3 Training ................................................................. 383
11.4.4 Forecasting ............................................................. 384
11.4.5 Exercises ............................................................... 385
11.5 Moving-Average (MA) Models .......................................... 388
11.5.1 MA(q) Model .......................................................... 388
11.5.2 Training ................................................................. 390
11.5.3 Exercises ............................................................... 391
11.6 ARMA ................................................................. 392
11.6.1 Selection Based on ACF and PACF ................................. 392
11.6.2 Training ................................................................. 393
11.6.3 Exercises ............................................................... 394
11.7 Rolling-Validation ......................................................... 396
11.7.1 1-Fold Rolling Validation ........................................... 396
11.7.2 k-Fold Rolling-Validation .......................................... 399
11.8 ARIMA ................................................................. 400
11.8.1 Differencing ........................................................... 400
11.8.2 Forecasting ............................................................ 401
13.3.3 Training ........................................... 430
13.3.4 Exercises ........................................ 432
13.4 K-Means Clustering - Hartigan-Wong .................. 434
13.4.1 Adjusted Distance ................................ 435
13.4.2 Exercises ......................................... 435
13.5 K-Means++ Clustering .................................. 436
13.5.1 Picking Initial Centroids ............................ 436
13.5.2 Exercises ......................................... 437
13.6 Clustering Predictor .................................... 438
13.6.1 Training ........................................... 438
13.6.2 Exercises ......................................... 439
13.7 Hierarchical Clustering ................................ 440
13.7.1 Exercises ......................................... 441
13.8 Markov Clustering ..................................... 442
13.8.1 Exercises ......................................... 442

III Simulation .................................................. 445

14 Simulation Foundations .................................. 447
14.1 Basic Concepts ....................................... 449
14.2 Types of Models ....................................... 450
14.2.1 Example: Modeling an M/M/1 Queue .............. 450
14.3 Random Number Generation ......................... 451
14.3.1 Example RNG: Random0 ......................... 451
14.3.2 Testing Random Number Generators ............. 452
14.3.3 Example RNG: Random3 ......................... 453
14.3.4 Exercises ......................................... 453
14.4 Random Variate Generation .......................... 455
14.4.1 Inverse Transform Method ........................ 455
14.4.2 Convolution Method .............................. 457
14.4.3 Exercises ......................................... 458
14.5 Poisson Process ...................................... 460
14.5.1 Generating a Poisson Process .................... 461
14.5.2 Generating a Non-Homogeneous Poisson Process 463
14.5.3 Exercises ......................................... 463
14.6 Monte Carlo Simulation ................................ 465
14.6.1 Simulation of Card Games ....................... 465
14.6.2 Integral of a Complex Function .................. 466
14.6.3 Grain Dropping Experiment ...................... 468
14.6.4 Simulation of the Monty Hall Problem ........... 469
14.6.5 Exercises ......................................... 469
14.7 Hand Simulation ..................................... 472
14.7.1 Little’s Law ...................................... 473
16.3 Event Scheduling

16.3.1 Event Class

16.3.2 Example: Bank Model

16.3.3 Example: Call Center Model

16.3.4 Entity Class

16.3.5 WaitQueue Class

16.3.6 WaitQueue_LCFS Class

16.3.7 Model Class

16.3.8 Example: Machine Shop Model

16.4 Event Graphs

16.4.1 Example: Bank Model

16.4.2 EventNode Class

16.4.3 CausalLink Class

16.5 Exercises

17 Process-Oriented Models

17.1 Base Traits and Classes for Process-Oriented Models

17.1.1 Identifiable Trait

17.1.2 Locatable Trait

17.1.3 Modelable Trait

17.1.4 Temporal Trait

17.2 Concurrent Processing of Actors

17.2.1 Java's Thread Class

17.2.2 ScalaTion's Coroutine Class

17.3 Process Interaction

17.3.1 Model Template

17.3.2 Component Trait

17.3.3 Example: BankModel

17.3.4 Executing the Bank Model

17.3.5 Network Diagram

17.3.6 Comparison to Event Scheduling

17.3.7 SimActor Class

17.3.8 Source Class

17.3.9 Sink Class

17.3.10 Transport Class

17.3.11 Resource Class

17.3.12 WaitQueue Class

17.3.13 WaitQueue_LCFS Class

17.3.14 Junction Class

17.3.15 Gate Class

17.3.16 Route Class

17.3.17 Model Class

17.3.18 Vehicle Traffic Model

17.3.19 Model_MBM Class
B  Functional Data Analysis 603
B.1  Basis Functions .......................................................... 603
B.2  Functional Smoothing .................................................... 603
B.3  Functional Principal Component Analysis ......................... 603
B.4  Functional Regression ................................................... 603
C  Parallel and Distributed Computing 605
C.1  MIMD - Multithreading .................................................. 605
C.2  SIMD - Vector Instructions ............................................. 605
C.3  Message Passing .......................................................... 605
C.4  Distributed Shared Memory ............................................. 605
C.5  Microservices ............................................................. 605
C.6  Distributed Functional Programming ................................. 605
Chapter 1

Introduction to Data Science

1.1 Data Science

The field of Data Science can be defined in many ways. To its left is Machine Learning that emphasizes algorithms for learning, while to its right is Statistics that focuses on procedures for estimating parameters of models and determining statistical properties of those parameters. Both fields develop models to describe/predict reality based on one or more datasets. Statistics has a greater interest in making inferences or testing hypotheses based upon datasets. It also has a greater interest in fitting probability distributions (e.g., are the residuals normally or exponentially distributed).

The common thread is modeling. A model should be able to make predictions (where is the hurricane likely to make landfall, when will the next recession occur, etc.). In addition, it may be desirable for a model to enhance the understanding of the system under study and to address what-if type questions (perspective analytics), e.g., how will traffic flow improve/degrade if a light-controlled intersection is replaced with a round-about.

A model may be viewed as replacement for a real system, phenomena to process. A model will map inputs into outputs with the goal being that for a given input, the model will produce output that approximates the output that the real system would produce. In addition to inputs and outputs, some models include state information. For example, the output of a heat pump will depend if it is in the heating or cooling state (internally this determines the direction of flow of the refrigerant). Further, some types of models are intended to mimic the behavior of the actual system and facilitate believable animation. Examples of such models are simulation models. They support prescriptive analytics which enables changes to a system to tested on the model, before the often costly changes to the actual system are under taken.

Broad categories of modeling are dependent of the type output (also called response) of the model. When the response is treated as a continuous variable, a predictive model (e.g., regression) is used. If the goal is to forecast into the future (or there is dependency among the response values), a forecasting model (e.g., ARIMA) is used. When the response is treated as a categorical variable, a classification model (e.g., support vector machine) is used. When the response values are largely missing, a clustering model may be used. Finally, when values are missing from a data matrix, an imputation model (k-nearest neighbors) or recommendation model (e.g., low-rank approximation using singular value decomposition) may be used. Dimensionality reduction (e.g., principal component analysis) can be useful across categories.

Computational Data Science puts more emphasis on computational issues, such as optimization algo-
Mathematical derivations are provided for the loss functions that are used to train the models. Short Scala code snippets are provided to illustrate how the algorithms work. The Scala object-oriented, functional language allows the creation of coincide code that looks very much like the mathematical expressions. Modeling based on ordinary differential equations and simulation models are also provided.

The prerequisite material for data science includes Vector Calculus, Applied Linear Algebra and Calculus-based Probability and Statistics. Datasets can be stored as vectors and matrices, learning/parameter estimation often involves taking gradients, and probability and statistics are needed to handle uncertainty.
1.2 ScalaTion

ScalaTion supports multi-paradigm modeling that can be used for simulation, optimization and analytics.

In ScalaTion, the analytics package provides tools for performing data analytics. Datasets are becoming so large that statistical analysis or machine learning software should utilize parallel and/or distributed processing. Databases are also scaling up to handle greater amounts of data, while at the same time increasing their analytics capabilities beyond the traditional On-Line Analytic Processing (OLAP). ScalaTion provides many analytics techniques found in tools like MATLAB, R and Weka. The analytics component contains six types of tools: predictors, classifiers, forecasters, clusterers, recommenders and reducers. A trait is defined for each type.

To use ScalaTion, go to the Website [http://www.cs.uga.edu/~jam/scalation.html](http://www.cs.uga.edu/~jam/scalation.html) and click on the most recent version of ScalaTion and follow the first three steps: download, unzip, build.

Current projects are targeting Big Data Analytics in four ways: (i) use of sparse matrices, (ii) parallel implementations using Scala’s support for parallelism (e.g., .par methods, parallel collections and actors), (iii) distributed implementations using Akka, and (iv) high performance data stores including columnar databases (e.g., Vertica), document databases (e.g., MongoDB), graph databases (e.g., Neo4j) and distributed file systems (e.g., HDFS).
1.3  A Data Science Project

The orientation of this textbook is that of developing modeling techniques and the understanding of how to apply them. A secondary goal is to explain the mathematics behind the models in sufficient detail to understand the algorithms implementing the modeling techniques. Concise code based on the mathematics is included and explained in the textbook. Readers may drill down to see the actual ScalaTiOn code.

The textbook is intended to facilitate trying out the modeling techniques as they are learned and to support a group-based term project that includes the following ten elements. The term project is to culminate in a presentation that explains what was done concerning these ten elements.

1. **Problem Statement.** Imagine that your group is hired as consultants to solve some problem for a company or government agency. The answers and recommendations that your group produces should not depend solely on prior knowledge, but rather on sophisticated analytics performed on multiple large-scale datasets. In particular, the study should be focused and the purpose of the study should clearly stated. What not to do: The following datasets are relevant to the company, so we ran them through an analytics package (e.g., R) and obtained the following results.

2. **Collection and Description of Datasets.** To reduce the chances of results being relevant only to a single dataset, multiple datasets should be used for the study (at least two). Explanation must be given to how each dataset relates to the other datasets as well as to the problem statement. When a dataset in the form of a matrix, metadata should be collected for each column/variable. In some cases the response column(s)/variable(s) will be obvious, in others it will depend on the purpose of the study. Initially, the result of columns/variables may be considered as features that may be useful in predicting responses. Ideally, the datasets should loaded into a well-designed database. ScalaTiOn provides two high-performance database systems: a column-oriented relational database system and a graph database system.

3. **Data Preprocessing Techniques Applied.** During the preprocessing phase (before the modeling techniques are applied), the data should be clean-up. This includes elimination of features with zero variance or too many missing values, as well as the elimination of key columns (e.g., on the training data, the employee-id could perfectly predict the salary of an employee, but is unlikely to be of any value in making predictions on the test data). For the remaining columns, strings should be converted to integers and imputation techniques should be used to replace missing values.

4. **Visual Examination.** At this point, Exploratory Data Analysis (EDA) should be applied. Commonly, one column of a dataset in the combined data matrix will be chosen as the response column, call it the response vector $y$, and the rest of the columns that remain after preprocessing form $m$-by-$n$ data matrix $X$. In general models are of the form

$$y = f(x) + \epsilon$$

where $f$ is function mapping feature vector $x$ into a predicted value for response $y$. The last term may be viewed as random error $\epsilon$. In an ideal model, the last term will be error (e.g., white noise). Since most models are approximations, technically the last term should be referred to as a residual (that which is not explained by the model). During exploratory data analysis, the value of $y$, should be plotted against each feature/column $x_j$ of data matrix $X$. The relationships between the columns should
be examined by computing a correlation matrix. Two columns that are very highly correlated are supplying redundant information, and typically, one should be removed. For a regression type problem, where \( y \) is treated as continuous random variable, a simple linear regression model should be created for each feature \( x_j \),

\[
y = b_0 + b_1 x_j + \epsilon
\]  

(1.2)

where the parameters \( b = [b_0, b_1] \) are to be estimated. The line generated by the model should be plotted along with the \( \{(x_{ij}, y_i)\} \) data points. Visually, look for patterns such white noise, linear relationship, quadratic relationship, etc. Plotting the residuals \( \{(x_{ij}, \epsilon_i)\} \) will also be useful.

5. Modeling Techniques Chosen. For every type of modeling problem, there is the notion of a NullModel: For prediction it is guess the mean, i.e., given a feature vector \( z \), predict the value \( E[y] \), regardless of the value of \( z \). The coefficient of determination \( R^2 \) for such models will be zero. If a more sophisticated model cannot beat the NullModel, it is not helpful in predicting or explaining the phenomena. Projects should include four classes of models: (i) NullModel, (ii) simple, easy to explain models (e.g., Multiple Linear Regression), (iii) complex, performant models (e.g., Quadratic Regression, Extreme Learning Machines) (iv) complex, time-consuming models (e.g., Neural Networks). If classes (ii-iv) do not improve upon class (i) models, new datasets should be collected. If this does not help, a new problem should be sought. On the flip side, if class (ii) models are nearly perfect \( (R^2 \text{ close to } 1) \), the problem being addressed may be too simple for a term project. At least one modeling technique should be chosen from each class.

6. Explanation of Why Techniques Were Chosen. As a consultant to a company, a likely question will be, “why did you chose those particular modeling techniques”? There are an enormous number of possible modeling techniques. Your group should explain how the candidate techniques were narrowed down and ultimately how the techniques were chosen. A review of how well the selected modeling techniques worked, as well as suggested changes for future work, should be given at the end of the presentation.

7. Feature Selection. Although feature selection can occur during multiple phases in a modeling study, an overview should be given at this point in the presentation. Explain which features were eliminated and why they were eliminated prior to building the models. During model building, what features were eliminated, e.g., using forward selection, backward selection, Lasso Regression, dimensionality reduction, etc. Also address the relative importance of the remaining features.

8. Reporting of Results. First the experimental setup should be described in sufficient detail to facilitate reproducibility of your results. One way to show overall results is to plot predicted responses \( \hat{y} \) and actual responses \( y \) versus the instance index \( i = 0 \) until \( m \). Reports are to include the Quality of Fit (QoF) for the various models and datasets in the form of tables, figures and explanation of the results. Besides the overall model, for many modeling techniques the importance/significance of model parameters/variables may be assessed as well. Tables and figures must include descriptive captions and color/shape schemes should be consistent across figures.

9. Interpretation of Results. With the results clearly presented, they need to be given insightful interpretations. What are the ramifications of the results? Are the modeling techniques useful in making predictions, classifications or forecasts?
10. **Recommendations of Study.** The organization that hired your group would like some take home messages that may result in improvements to the organization (e.g., what to produce, what processes to adapt, how to market, etc.). A brief discussion of how the study could be improved (possibly leading to further consulting work) should be given.
1.4 Additional Textbooks

More detailed development of this material can be found in textbooks on statistical learning, such as “An Introduction to Statistical Learning” (ISL) [46] and “The Elements of Statistical Learning” (ESL) [38]. See Table 1.1 for a mapping between the chapters in three textbooks.

Table 1.1: Source Material Chapter Mappings

<table>
<thead>
<tr>
<th>Topic</th>
<th>SCALATION</th>
<th>ISL</th>
<th>ESL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear Algebra</td>
<td>Ch. 2</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Probability</td>
<td>Ch. 3</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Data Management</td>
<td>Ch. 4</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Data Preprocessing</td>
<td>Ch. 5</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Prediction</td>
<td>Ch. 6</td>
<td>Ch. 3, 5, 6</td>
<td>Ch. 3</td>
</tr>
<tr>
<td>Classification</td>
<td>Ch. 7</td>
<td>Ch. 2, 5, 8</td>
<td>Ch. 4, 12, 13, 15</td>
</tr>
<tr>
<td>Classification - Continuous</td>
<td>Ch. 8</td>
<td>Ch. 4, 8, 9</td>
<td>Ch. 4, 12, 13, 15</td>
</tr>
<tr>
<td>Generalized Models</td>
<td>Ch. 9</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Nonlinear Models and Neural Networks</td>
<td>Ch. 10</td>
<td>Ch. 7</td>
<td>Ch. 11</td>
</tr>
<tr>
<td>Time-Series/Temporal Models</td>
<td>Ch. 11</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Dimensionality Reduction</td>
<td>Ch. 12</td>
<td>Ch. 6, 10</td>
<td>Ch. 14</td>
</tr>
<tr>
<td>Clustering</td>
<td>Ch. 13</td>
<td>Ch. 10</td>
<td>Ch. 14</td>
</tr>
<tr>
<td>Simulation Foundations</td>
<td>Ch. 14</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>State Space Models</td>
<td>Ch. 15</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Event-Oriented Models</td>
<td>Ch. 16</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Process-Oriented Models</td>
<td>Ch. 17</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Simulation Output Analysis</td>
<td>Ch. 18</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Optimization in Data Science</td>
<td>Appendix A</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Functional Data Analysis</td>
<td>Appendix B</td>
<td>Ch. 7</td>
<td>Ch. 5</td>
</tr>
<tr>
<td>Parallel/Distributed Computing</td>
<td>Appendix C</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

The next two chapters serve as quick reviews of the two principal mathematical foundations for data science: linear algebra and probability.
Part I

Foundations
Chapter 2

Linear Algebra

Data science and analytics make extensive use of linear algebra. For example, let $y_i$ be the income of the $i^{th}$ individual and $x_{ij}$ be the value of the $j^{th}$ predictor/feature (age, education, etc.) for the $i^{th}$ individual. The responses (outcomes of interest) are collected into a vector $y$, the values for predictors/features are collected in a matrix $X$ and the parameters/coefficients $b$ are fit to the data.

2.1 Linear System of Equations

The study of linear algebra starts with solving systems of equations, e.g.,

\[
\begin{align*}
  y_0 &= x_{00}b_0 + x_{01}b_1 \\
  y_1 &= x_{10}b_0 + x_{11}b_1
\end{align*}
\]

This linear system has two equations with two variables having unknown values, $b_0$ and $b_1$. Such linear systems can be used to solve problems like the following: Suppose a movie theatre charges 10 dollars per child and 20 dollars per adult. The evening attendance is 100, while the revenue is 1600 dollars. How many children ($b_0$) and adults ($b_1$) were in attendance?

\[
\begin{align*}
  100 &= 1b_0 + 1b_1 \\
  1600 &= 10b_0 + 20b_1
\end{align*}
\]

The solution is $b_0 = 40$ children and $b_1 = 60$ adults.

In general, linear systems may be written using matrix notation.

\[
y = Xb \tag{2.1}
\]

where $y$ is an $m$-dimensional vector, $X$ is an $m$-by-$n$ dimensional matrix and $b$ is an $n$-dimensional vector.
2.2 Matrix Inversion

If the matrix is of full rank with \( m = n \), then the unknown vector \( \mathbf{b} \) may be uniquely determined by multiplying both sides of the equation by the inverse of \( X \), \( X^{-1} \)

\[
\mathbf{b} = X^{-1} \mathbf{y}
\]  

(2.2)

Multiplying matrix \( X \) and its inverse \( X^{-1} \), \( X^{-1}X \) results in an \( n \)-by-\( n \) identity matrix \( I_n = [1_{i=j}] \), where the indicator function \( 1_{i=j} \) equals 1 when \( i = j \) and 0 otherwise.

A faster and more numerically stable way to solve for \( \mathbf{b} \) is to perform Lower-Upper (LU) Factorization. This is done by factoring matrix \( X \) into lower \( L \) and upper \( U \) triangular matrices.

\[
X = LU
\]  

(2.3)

Then \( LU \mathbf{b} = \mathbf{y} \), so multiplying both sides by \( L^{-1} \) gives \( U \mathbf{b} = L^{-1} \mathbf{y} \). Taking an augmented matrix

\[
\begin{bmatrix}
1 & 3 & 1 \\
2 & 1 & 7
\end{bmatrix}
\]

and performing row operations to make it upper right triangular has the effect of multiplying by \( L^{-1} \). In this case, the first row multiplied by -2 is added to second row to give.

\[
\begin{bmatrix}
1 & 3 & 1 \\
0 & -5 & 5
\end{bmatrix}
\]

From this, backward substitution can be used to determine \( b_1 = -1 \) and then that \( b_0 = 4 \), i.e.,

\[
\mathbf{b} = \begin{bmatrix}
4 \\
-1
\end{bmatrix}
\]

In cases where \( m > n \), the system may be overdetermined, and no solution will exist. Values for \( \mathbf{b} \) are then often determined to make \( \mathbf{y} \) and \( X \mathbf{b} \) agree as closely as possible, e.g., minimize absolute or squared differences.

Vector notation is used in this book, with vectors shown in boldface and matrices in uppercase. Note, matrices in SCALATion are in lowercase, since by convention, uppercase indicates a type, not a variable. SCALATion supports vectors and matrices in its mathstat package. A commonly used operation is the dot (inner) product, \( \mathbf{x} \cdot \mathbf{y} \), or in SCALATion, \( \mathbf{x} \text{ dot } \mathbf{y} \).
2.3 Vector

A vector may be viewed a point in multi-dimensional space, e.g., in three space, we may have

\[
\begin{align*}
\mathbf{x} &= [x_0, x_1, x_2] = [0.57735, 0.55735, 0.57735] \\
\mathbf{y} &= [y_0, y_1, y_2] = [1.0, 1.0, 0.0]
\end{align*}
\]

where \( \mathbf{x} \) is a point on the unit sphere and \( \mathbf{y} \) is a point in the plane determined by the first two coordinates.

2.3.1 Vector Addition and Subtraction

Vectors may be added (\( \mathbf{x} + \mathbf{y} \)), subtracted (\( \mathbf{x} - \mathbf{y} \)).

2.3.2 Elementwise Multiplication and Division

Vectors may be multiplied element-by-element (Hadamard product) (\( \mathbf{x} \ast \mathbf{y} \)), and divided element-by-element (\( \mathbf{x}/\mathbf{y} \)). These operations are also supported when one of the arguments is a scalar.

2.3.3 Vector Dot Product

A particularly important operation, the dot product (or inner product) of two vectors is simply the sum of the products of their elements.

\[
\mathbf{x} \cdot \mathbf{y} = \sum_{i=0}^{n-1} x_i y_i = 1.1547 \tag{2.4}
\]

2.3.4 Norm

The norm of a vector is its length. Assuming Euclidean distance, the norm is

\[
\|\mathbf{x}\| = \sqrt{\sum_{i=0}^{n-1} x_i^2} = 1 \tag{2.5}
\]

The norm of \( \mathbf{y} \) is \( \sqrt{2} \). If \( \theta \) is the angle between the \( \mathbf{x} \) and \( \mathbf{y} \) vectors, then the dot product is the product of their norms and the cosine of the angle.

\[
\mathbf{x} \cdot \mathbf{y} = \|\mathbf{x}\|\|\mathbf{y}\| \cos(\theta) \tag{2.6}
\]

Thus, the cosine of \( \theta \) is,

\[
\cos(\theta) = \frac{\mathbf{x} \cdot \mathbf{y}}{\|\mathbf{x}\|\|\mathbf{y}\|} = \frac{1.1547}{1 \cdot \sqrt{2}} = 0.8165
\]

so the angle \( \theta = .616 \) radians. Vectors \( \mathbf{x} \) and \( \mathbf{y} \) are orthogonal if the angle \( \theta = \pi/2 \) radians (90 degrees).

In general there are \( \ell_p \) norms. The two that are used here are the \( \ell_2 \) norm \( \|\mathbf{x}\| = \|\mathbf{x}\|_2 \) (Euclidean distance) and the \( \ell_1 \) norm \( \|\mathbf{x}\|_1 \) (Manhattan distance).

\[
\|\mathbf{x}\|_1 = \sum_{i=0}^{n-1} |x_i| \tag{2.7}
\]
Vector notation facilitates concise mathematical expressions. Many common statistical measures for populations or samples can be given in vector notation. For an $m$ dimensional vector ($m$-vector) the following may be defined.

$$\mu(x) = \mu_x = \frac{1 \cdot x}{m}$$

$$\sigma^2(x) = \sigma^2_x = \frac{(x - \mu_x) \cdot (x - \mu_x)}{m} = \frac{x \cdot x}{m} - \mu_x^2$$

$$\sigma(x, y) = \sigma_{x,y} = \frac{(x - \mu_x) \cdot (y - \mu_y)}{m} = \frac{x \cdot y}{m} - \mu_x \mu_y$$

$$\rho(x, y) = \rho_{x,y} = \frac{\sigma_{x,y}}{\sigma_x \sigma_y}$$

which are the population mean, variance, covariance and correlation, respectively.

The size of the population is $m$, which corresponds to the number of elements in the vector. A vector of all ones is denoted by $\mathbf{1}$. For an $m$-vector $||\mathbf{1}||^2 = \mathbf{1} \cdot \mathbf{1} = m$. Note, the sample mean uses the same formula, while the sample variance and covariance divide by $m - 1$, rather than $m$ (sample indicates that only some fraction of population is used in the calculation).

Vectors may be used for describing the motion of an object through space over time. Let $u(t)$ be the location of an object (e.g., golf ball) in three dimensional space $\mathbb{R}^3$ at time $t$,

$$u(t) = [x(t), y(t), z(t)]$$

To describe the motion, let $v(t)$ be the velocity at time $t$, and $a$ be the constant acceleration, then according to Newton’s Second Law of Motion,

$$u(t) = u(0) + v(0) t + \frac{1}{2} a t^2$$

The time varying function $u(t)$ over time will show the trajectory of the golf ball.

### 2.3.5 Vector Operations in ScalaTion

Vector operations are implemented by multiple classes, such as the VectorD class.

```scala
@constructor private VectorD (val dim: Int,
                         @param v the 1D array used to store vector elements

class VectorD (val dim: Int,
               private [mathstat] var v: Array[Double] = null)
  extends IndexedSeq[Double]
  with PartiallyOrdered [VectorD]
  with Cloneable [VectorD]
  with DefaultSerializable:
```

32
VectorD includes methods for size, indices, set, copy, filter, select, concatenate, vector arithmetic, power, square, reciprocal, abs, sum, mean variance, rank, cumulate, normalize, dot, norm, max, min, mag, argmax, argmin, indexOf, indexWhere, count, contains, sort and swap.

Table 2.1: Vector Arithmetic Operations

<table>
<thead>
<tr>
<th>op</th>
<th>vector op vector</th>
<th>vector op scalar</th>
<th>vector element op scalar</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>def + (b: VectorD): VectorD</td>
<td>def + (s: Double): VectorD</td>
<td>def + (s: (Int, Double)): VectorD</td>
</tr>
<tr>
<td>+=</td>
<td>def += (b: VectorD): VectorD</td>
<td>def += (s: Double): VectorD</td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>def - (b: VectorD): VectorD</td>
<td>def - (s: Double): VectorD</td>
<td>def - (s: (Int, Double)): VectorD</td>
</tr>
<tr>
<td>-=</td>
<td>def -= (b: VectorD): VectorD</td>
<td>def -= (s: Double): VectorD</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>def * (b: VectorD): VectorD</td>
<td>def * (s: Double): VectorD</td>
<td>def * (s: (Int, Double)): VectorD</td>
</tr>
<tr>
<td>*=</td>
<td>def *= (b: VectorD): VectorD</td>
<td>def *= (s: Double): VectorD</td>
<td></td>
</tr>
<tr>
<td>/</td>
<td>def / (b: VectorD): VectorD</td>
<td>def / (s: Double): VectorD</td>
<td>def / (s: (Int, Double)): VectorD</td>
</tr>
<tr>
<td>/=</td>
<td>def /= (b: VectorD): VectorD</td>
<td>def /= (s: Double): VectorD</td>
<td></td>
</tr>
</tbody>
</table>

33
2.4 Vector Calculus

Data science uses optimization to fit parameters in models, where for example a quality of fit measure (e.g., sum of squared errors) is minimized. Typically, gradients are involved. In some cases, the gradient of the measure can be set to zero allowing the optimal parameters to be determined by matrix factorization. For complex models, this may not work, so an optimization algorithm that moves in the direction opposite to the gradient can be applied.

2.4.1 Gradient Vector

Consider the following function \( f : \mathbb{R}^2 \rightarrow \mathbb{R} \) of vector \( \mathbf{u} = [x, y] \)

\[
f(\mathbf{u}) = (x - 2)^2 + (y - 3)^2
\]

For example, the functional value at the point \([1, 1]\), \( f([1, 1]) = 1 + 4 = 5 \). The gradient of function \( f \) consists of a vector formed from the two partial derivatives and

\[
\nabla f(\mathbf{u}) = \left[ \frac{\partial f}{\partial x}, \frac{\partial f}{\partial y} \right]
\]

indicates the direction of steepest increase. For example, the gradient at the point \([1, 1]\), \( \nabla f([1, 1]) = [-2, -4] \). Its norm indicates the magnitude of the rate of change. By setting the gradient equal to zero in this case

\[
\frac{\partial f}{\partial x} = 2(x - 2)
\]
\[
\frac{\partial f}{\partial y} = 2(y - 3)
\]

one may find the vector that minimizes function \( f \), namely \( \mathbf{u} = [2, 3] \) where \( f = 0 \). For more complex functions, repeatedly moving in the opposite direction to the gradient, may lead to finding a minimal value.

In general, the gradient (or gradient vector) of function \( f : \mathbb{R}^n \rightarrow \mathbb{R} \) of vector \( \mathbf{x} \in \mathbb{R}^n \) is

\[
\nabla f(\mathbf{x}) = \frac{\partial f(\mathbf{x})}{\partial \mathbf{x}} = \left[ \frac{\partial f}{\partial x_0}, \ldots, \frac{\partial f}{\partial x_{n-1}} \right]
\] (2.8)

In data science, it is often convenient to take the gradient of a dot product of two functions of \( \mathbf{x} \), in which case the following product rule can be applied.

\[
\nabla (f(\mathbf{x}) \cdot g(\mathbf{x})) = \nabla f(\mathbf{x}) \cdot g(\mathbf{x}) + f(\mathbf{x}) \cdot \nabla g(\mathbf{x})
\] (2.9)

2.4.2 Jacobian Matrix

The Jacobian Matrix is an extension of the gradient vector to the case where the value of the function is multi-dimensional, i.e., \( f = [f_0, f_1, \ldots, f_{m-1}] \). In general, the Jacobian of function \( f : \mathbb{R}^n \rightarrow \mathbb{R}^m \) of vector \( \mathbf{x} \in \mathbb{R}^n \) is

\[
\mathcal{J} f(\mathbf{x}) = \left[ \frac{\partial f_j}{\partial x_i} \right]_{0 \leq i < m, 0 \leq j < n}
\] (2.10)
Consider the following function \( f : \mathbb{R}^2 \to \mathbb{R}^2 \) that maps vectors in \( \mathbb{R}^2 \) into other vectors in \( \mathbb{R}^2 \).

\[
f(\mathbf{x}) = [(x_0 - 2)^2 + (x_1 - 3)^2, (2x_0 - 6)^2 + (3x_1 - 6)^2]
\]

The Jacobian of the function, \( Jf(\mathbf{x}) \), is

\[
\begin{bmatrix}
\frac{\partial f_0}{\partial x_0}, & \frac{\partial f_0}{\partial x_1} \\
\frac{\partial f_1}{\partial x_0}, & \frac{\partial f_1}{\partial x_1}
\end{bmatrix}
\]

Taking the partial derivatives gives the following Jacobian matrix.

\[
\begin{bmatrix}
2x_0 - 4, & 2x_1 - 6 \\
4x_0 - 12, & 6x_1 - 12
\end{bmatrix}
\]

### 2.4.3 Hessian Matrix

While the gradient is a vector of first partial derivatives, the Hessian is a symmetric matrix of second partial derivatives. The Hessian Matrix of a scalar-valued function \( f : \mathbb{R}^n \to \mathbb{R} \) of vector \( \mathbf{x} \in \mathbb{R}^n \) is

\[
\mathbb{H} f(\mathbf{x}) = \left[ \frac{\partial^2 f}{\partial x_i \partial x_j} \right]_{0 \leq i < n, 0 \leq j < n} \tag{2.11}
\]

Consider the following function \( f : \mathbb{R}^2 \to \mathbb{R} \) that maps vectors in \( \mathbb{R}^2 \) into scalars in \( \mathbb{R} \).

\[
f(\mathbf{x}) = (2x_0 - 6)^2 + (3x_1 - 6)^2
\]

The Hessian of the function, \( \mathbb{H} f(\mathbf{x}) \), is

\[
\begin{bmatrix}
\frac{\partial^2 f}{\partial x_0^2}, & \frac{\partial^2 f}{\partial x_0 \partial x_1} \\
\frac{\partial^2 f}{\partial x_1 \partial x_0}, & \frac{\partial^2 f}{\partial x_1^2}
\end{bmatrix}
\]

Taking the second partial derivatives gives the following Hessian matrix.

\[
\begin{bmatrix}
4, & 0 \\
0, & 6
\end{bmatrix}
\]

Consider a differentiable function of \( n \) variables, \( f : \mathbb{R}^n \to \mathbb{R} \). The points at which its gradient vector \( \nabla f \) is zero are referred to as critical points. In particular, they may be local minima, local maxima or saddle points/inconclusive, depending on whether the Hessian matrix \( \mathbb{H} \) is positive definite, negative definite, or otherwise. A symmetric matrix \( A \) is positive definite if \( \mathbf{x}^\intercal A \mathbf{x} > 0 \) for all \( \mathbf{x} \neq 0 \) (alternatively, all of \( A \)'s eigenvalues are positive). Note: a positive/negative semidefinite Hessian matrix may or may not indicate an optimal (minimal/maximal) point.
2.5 Matrix

A matrix may be viewed as a collection of vectors, one for each row in the matrix. Matrices may be used to represent linear transformations

\[ f : \mathbb{R}^n \rightarrow \mathbb{R}^m \]  

that map vectors in \( \mathbb{R}^n \) to vectors in \( \mathbb{R}^m \). For example, in SCALATion an \( m \)-by-\( n \) matrix \( A \) with \( m = 3 \) rows and \( n = 2 \) columns may be created as follows:

```scala
val a = MatrixD ((3, 2), 1, 2,
                 3, 4,
                 5, 6)
```
to produce matrix \( A \).

\[
\begin{bmatrix}
1 & 2 \\
3 & 4 \\
5 & 6
\end{bmatrix}
\]

Matrix \( A \) will transform \( u \) vectors in \( \mathbb{R}^2 \) into \( v \) vectors in \( \mathbb{R}^3 \).

\[ Au = v \]  

For example,

\[
A \begin{bmatrix} 1 \\ 2 \end{bmatrix} = \begin{bmatrix} 5 \\ 11 \\ 17 \end{bmatrix}
\]

SCALATion supports retrieval of row vectors, column vectors and matrix elements. In particular, the following access operations are supported.

- \( A = a \) = matrix
- \( A = a() \) = underlying array
- \( a_i = a(i) \) = row vector \( i \)
- \( a_{j} = a.col(j) \) = column vector \( j \)
- \( a_{ij} = a(i, j) \) = the element at row \( i \) and column \( j \)
- \( A_{i:k,j:l} = a(i \text{ to } k, j \text{ to } l) \) = row and column matrix slice

Note that in Scala, \( i \text{ to } k \) is a Range that includes \( k \), while \( i \text{ until } k \) does not include \( k \). Common operations on matrices are supported as well.

2.5.1 Matrix Operation in SCALATion

Matrix operations in SCALATion are defined in the MatriD trait and implemented in multiple classes, such as MatrixD for dense matrices.
Matrix Addition and Subtraction

Matrix addition \( c = a + b \)

\[
    c_{ij} = a_{ij} + b_{ij}
\]

and matrix subtraction \( c = a - b \) are supported.

Matrix Multiplication

A frequently used operation in data science is matrix multiplication \( c = a \cdot b \).

\[
    c_{ij} = \sum_{k=0}^{n-1} a_{ik} b_{kj} = a_i \cdot b_j
\]

Mathematically, this is written as \( C = AB \). The \( ij \) element in matrix \( C \) is the vector dot product of the \( i^{th} \) row of \( A \) with the \( j^{th} \) column of \( B \).

Matrix Transpose

The transpose of matrix \( A \), written \( A^\top \) \( (\text{val } t = a.\text{transpose}) \), simply exchanges the roles of rows and columns.

```scala
def transpose: MatrixD =
    val a = Array.ofDim[Double] (dim2, dim)
    for j <- indices do
        val v_j = v(j)
        var i = 0
        cfor (i < dim2, i += 1) { a(i)(j) = v_j(i) }
    end for
end transpose
```

Matrix Determinant

The determinant of square \( (m = n) \) matrix \( A \), written \(|A| \) \( (\text{val } d = a.\text{det}) \), indicates whether a matrix is singular or not (and hence invertible), based on whether the determinant is zero or not.
Trace of a Matrix

The trace of matrix $A \in \mathbb{R}^{n \times n}$ is simply the sum of its diagonal elements.

$$\text{tr}(A) = \sum_{i=0}^{n-1} a_{ii} \quad (2.16)$$

In SCALATION, the trace is computed using the `trace` method (e.g., `a.trace`).

Matrix Dot Product

SCALATION provides several types of dot products on both vectors and matrices, three of which are shown below. The first method computes the usual dot product between two vectors, while the second method is between two matrices. The second method provides another (and sometimes more efficient) way to compute $A^T B = A \cdot B = a^\text{transpose} * b = a \cdot b$.

```scala
def dot (y: IndexedSeq[Double]): Double = { 
  var sum = 0.0 
  for (i <- v.indices) sum += v(i) * y(i) 
  sum 
}
end dot

def dot (y: MatrixD): MatrixD = { 
  if (dim2 != y.dim) 
    flaw ("dot", s"matrix dot matrix - incompatible cross dimensions: dim2 = $dim2, y.dim = $y.dim")
  val a = Array.ofDim[Double] (dim, y.dim) 
  for (ii <- 0 until dim by TSZ) 
    for (jj <- 0 until y.dim2 by TSZ) 
      for (kk <- 0 until dim2 by TSZ) 
        for (i <- ii until math.min(ii + TSZ, dim)) 
          val v_i = v(i); val a_i = a(i) 
          for (j <- jj until math.min(jj + TSZ, y.dim2)) 
            val y_j = y.v(j) 
            var sum = 0.0 
            var k = kk 
            cfor (k < k2, k += 1) { sum += v_i(k) * y_j(k) } 
            a_i(j) += sum 
    end for 
  end for 
  new MatrixD (dim, y.dim, a) 
}
end dot
```
2.6 Matrix Factorization

Many problems in data science involve matrix factorization to for example solve linear systems of equations or perform Ordinary Least Squares (OLS) estimation of parameters. ScalaTion supports several factorization techniques, including the techniques shown in table 2.2

<table>
<thead>
<tr>
<th>Factorization</th>
<th>Factors</th>
<th>Factor 1</th>
<th>Factor 2</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>LU</td>
<td>$A = LU$</td>
<td>lower left triangular</td>
<td>upper right triangular</td>
<td>Fac_LU</td>
</tr>
<tr>
<td>Cholesky</td>
<td>$A = LL^T$</td>
<td>lower left triangular</td>
<td>its transpose</td>
<td>Fac_Cholesky</td>
</tr>
<tr>
<td>QR</td>
<td>$A = QR$</td>
<td>orthogonal</td>
<td>upper right triangular</td>
<td>Fac_QR</td>
</tr>
<tr>
<td>SVD</td>
<td>$A = UΣV^T$</td>
<td>orthogonal</td>
<td>diagonal, orthogonal</td>
<td>Fac_SVD</td>
</tr>
<tr>
<td>Eigen</td>
<td>$A = QΛQ^{-1}$</td>
<td>eigenvectors</td>
<td>diagonal, inverse eigen</td>
<td>Fac_Eigen</td>
</tr>
</tbody>
</table>

These algorithms are faster or more numerically stable than algorithms for matrix inversion. See the Prediction chapter to see how matrix factorization is used in Ordinary Least Squares estimation.

2.6.1 Eigenvalues and Eigenvectors

Consider the following matrix $A$ and two vectors $x = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$ and $z = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$.

$$
\begin{bmatrix}
2 & 0 \\
0 & 3 \\
\end{bmatrix}
$$

Multiplying $A$ and $x$ yields $\begin{bmatrix} 2 \\ 0 \end{bmatrix}$, while multiplying $A$ and $z$ yields $\begin{bmatrix} 0 \\ 3 \end{bmatrix}$. Thus, letting $λ_0 = 2$ and $λ_1 = 3$, we see that $Ax = λ_0x$ and $Az = λ_1z$. In general, a matrix $A^{n×n}$ of rank $r$ will have $r$ non-zero eigenvalues $λ_i$ with corresponding eigenvector $x^{(i)}$ such that

$$Ax^{(i)} = λ_i x^{(i)} \quad (2.17)$$

In other words, there will be $r$ unit eigenvectors, for which multiplying by the matrix simply rescales the eigenvector $x^{(i)}$ by its eigenvalue $λ_i$. The same will happen for any non-zero vector in alignment with one of the $r$ unit eigenvectors.

Given an eigenvalue $λ_i$, an eigenvector may be found by noticing that

$$Ax^{(i)} - λ_i x^{(i)} = [A - λ_i] x^{(i)} = 0 \quad (2.18)$$

Any vector in the nullspace of the matrix $A - λ_i I$ is an eigenvector corresponding to $λ_i$. Note, if the above equation is transposed, it is called a left eigenvalue problem (see the section on Markov Chains).

In low dimensions, the eigenvalues may be found as roots of the characteristic polynomial/equation derived from taking the determinant of $A - λ_i I$. Software like ScalaTion, however, use iterative algorithms that convert a matrix into Hessenburg and tridiagonal forms.
2.7 Internal Representation

The current internal representation used for storing the elements in a dense matrix is `Array [Array [Double]]` in row major order (row-by-row). Depending on usage, operations may be more efficient using column major order (column-by-column). Also, using a one dimensional array `Array [Double]` mapping \((i, j)\) to the \(k^{th}\) location may be more efficient. Furthermore, having operations access through sub-matrices (blocks) may improve performance because of caching efficiency or improved performance for parallel and distributed versions.

The `mathstat` package provides several traits and classes implementing multiple types of vectors and matrices as shown in Table 2.3 including `VectorD` and `MatrixD`.

Table 2.3: Types of Vectors and Matrices: Implementing Classes

<table>
<thead>
<tr>
<th>trait</th>
<th>VectorD</th>
<th>MatrixD</th>
</tr>
</thead>
<tbody>
<tr>
<td>dense</td>
<td><code>VectorD</code></td>
<td><code>MatrixD</code></td>
</tr>
<tr>
<td>sparse</td>
<td><code>SparseVectorD</code></td>
<td><code>SparseMatrixD</code></td>
</tr>
<tr>
<td>compressed</td>
<td><code>RleVectorD</code></td>
<td><code>RleMatrixD</code></td>
</tr>
<tr>
<td>tridiagonal</td>
<td>-</td>
<td><code>SymTriMatrixD</code></td>
</tr>
<tr>
<td>bidiagonal</td>
<td>-</td>
<td><code>BidMatrixD</code></td>
</tr>
</tbody>
</table>


`Scalation` supports many operations involving matrices and vectors, including the following show in Table 2.4.

Table 2.4: Types of Vector and Matrix Products

<table>
<thead>
<tr>
<th>Product</th>
<th>Method</th>
<th>Example</th>
<th>in Math</th>
</tr>
</thead>
<tbody>
<tr>
<td>vector dot</td>
<td>def dot (y: VectorD): Double</td>
<td>x dot y</td>
<td>(x \cdot y)</td>
</tr>
<tr>
<td>vector elementwise</td>
<td>def * (y: VectorD): VectorD</td>
<td>x * y</td>
<td>(xy)</td>
</tr>
<tr>
<td>vector outer</td>
<td>def outer (y: VectorD): MatrixD</td>
<td>x outer y</td>
<td>(x \otimes y)</td>
</tr>
<tr>
<td>matrix mult</td>
<td>def * (y: MatrixD): MatrixD</td>
<td>x * y</td>
<td>(XY)</td>
</tr>
<tr>
<td>matrix mdot</td>
<td>def dot (y: MatrixD): MatrixD</td>
<td>x dot y</td>
<td>(X^T \cdot Y)</td>
</tr>
<tr>
<td>matrix vector</td>
<td>def * (y: VectorD): VectorD</td>
<td>x * y</td>
<td>(Xy)</td>
</tr>
<tr>
<td>matrix vector</td>
<td>def * (y: VectorD): MatrixD</td>
<td>x * y</td>
<td>(X \text{diag}(y))</td>
</tr>
</tbody>
</table>
2.8 Tensor

Loosely speaking, a tensor is a generalization of scalar, vector and matrix. The order of the tensor indicates the number dimensions. In this text, tensors are treated as *hypermatrices* and issues such as contravariate and covariate vectors, and the rules for index notation involving super and subscripts are ignored [59].

For data science, input into a model may be a vector (e.g., simple regression, univariate time-series), a matrix (e.g., multiple linear regression, neural networks), a tensor with three dimensions (e.g., monochromatic/greyscale images), and a tensor with four dimensions (e.g., color images).

<table>
<thead>
<tr>
<th>Order</th>
<th>Analog/Name</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>zeroth</td>
<td>scalar</td>
<td>FICA score</td>
</tr>
<tr>
<td>first</td>
<td>vector</td>
<td>customer financial record</td>
</tr>
<tr>
<td>second</td>
<td>matrix</td>
<td>collection of financial records</td>
</tr>
<tr>
<td>third</td>
<td>tensor</td>
<td>collection of grayscale images</td>
</tr>
<tr>
<td>fourth</td>
<td>tensor4</td>
<td>collection color images</td>
</tr>
</tbody>
</table>

2.8.1 Three Dimensional Tensors

In *ScalaTion*, tensors with three dimensions are supported by the *Tensor* class. The default names for the dimensions [59] were chosen to follow a common convention (row, column, slice). In data science, the first index usually indicates which instance, e.g., *i*\(^{th}\) element of a vector, *i*\(^{th}\) row of a matrix, *i*\(^{th}\) row of a tensor.

```scala
@constructor private [tenalgebra] var v: Array [Array [Array [Double]]] = null
```

A tensor \( T \) is stored in a triple array \( [t_{ijk}] \). Below is an example of a 2-by-2-by-2 tensor, \( T = [T_{i0}]_{ij} \)

\[
\begin{bmatrix}
  t_{000} & t_{001} \\
  t_{100} & t_{101}
\end{bmatrix}
\begin{bmatrix}
  t_{010} & t_{011} \\
  t_{110} & t_{111}
\end{bmatrix}
\]

where each slice \( T_{i:} \) is a 2-by-2 matrix.

Note, *ScalaTion* allows the default names for the dimensions to be changed, so they are more suggestive given the application, e.g., (row, column, channel) for one color image or (sheet, row, column) for spreadsheets.
2.9 Exercises

1. Draw two 2-dimensional non-zero vectors whose dot product is zero.

2. Given the matrix $X$ and the vector $y$, solve for the vector $b$ in the equation $y = Xb$ using matrix inversion and $LU$ factorization.

   ```scala
   import scalation.mathstat.{MatrixD, VectorD, Fac_LU}
   val x = new MatrixD ((2, 2), 1, 3,
                         2, 1)
   val y = VectorD (1, 7)
   println ("using inverse: b = X^-1 y = " + x.inverse * y)
   println ("using LU fact: Lb = Uy = " + { val lu = new Fac_LU (x); lu.factor ().solve (y) } )
   ```

   Modify the code to show the inverse matrix $X^{-1}$ and the factorization into the $L$ and $U$ matrices.

3. If $Q$ is an orthogonal matrix, then $Q^T Q$ becomes what type of matrix? What about $QQ^T$? Illustrate with an example 3-by-3 matrix. What is the inverse of $Q$?

4. Show that the Hessian matrix of a scalar-valued function $f : \mathbb{R}^n \rightarrow \mathbb{R}$ is the transpose of the Jacobian of the gradient, i.e.,

   $$\mathbb{H}f(x) = [\nabla f(x)]^T$$

5. Critical points for a function $f : \mathbb{R}^n \rightarrow \mathbb{R}$ occur when $\nabla f(x) = 0$. How can the Hessian Matrix be used to decide whether a particular critical point is a local minimum or maximum?

2.10 Further Reading


2. Matrix Computations [32]
Chapter 3

Probability

Probability is used to measure the likelihood of certain events occurring, such as flipping a coin and getting a head, rolling a pair of dice and getting a sum of 7, or getting a full house in five card draw. Given a random experiment, the sample space $\Omega$ is the set of all possible outcomes.

3.1 Probability Measure

Definition: A probability measure $P$ can be defined axiomatically as follows:

$$ P(A) \geq 0 \text{ for any event } A \subseteq \Omega $$

$$ P(\Omega) = 1 $$

$$ P(\bigcup A_i) = \sum P(A_i) \text{ for a countable collection of disjoint events} $$

(3.1)

Technically speaking, an event is a measurable subset of $\Omega$ (see [24] for a measure-theoretic definition). Letting $\mathcal{F}$ be the set of all possible events, one may define a probability space as follows:

Definition: A probability space is defined as a triple $(\Omega, \mathcal{F}, P)$.

Given an event $A \in \mathcal{F}$, the probability of its occurrence is restricted to the unit interval, $P(A) \in [0, 1]$. Thus, $P$ may be viewed as a function that maps events to the unit interval.

$$ P : \mathcal{F} \to [0, 1] $$

(3.2)

3.1.1 Joint Probability

Given two events $A$ and $B$, the joint probability of their co-occurrence is denoted by

$$ P(AB) = P(A \cap B) \in [0, \min(P(A), P(B))] $$

(3.3)

If events $A$ and $B$ are independent, simply take the product of the individual probabilities,

$$ P(AB) = P(A)P(B) $$

(3.4)
3.1.2 Conditional Probability

The conditional probability of the occurrence of event $A$, given it is known that event $B$ has occurred/will occur is

$$P(A|B) = \frac{P(AB)}{P(B)}$$  \hspace{1cm} (3.5)

If events $A$ and $B$ are independent, the conditional probability reduces to

$$P(A|B) = \frac{P(AB)}{P(B)} = \frac{P(A)P(B)}{P(B)} = P(A)$$  \hspace{1cm} (3.6)

In other words, the occurrence of event $B$ has no affect on the probability of event $A$ occurring. An important theorem involving conditional probability is Bayes Theorem.

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$  \hspace{1cm} (3.7)
3.2 Random Variable

Rather than just looking at individual events, e.g., $E_1$ or $E_2$, one is often more interested in the probability that random variables take on certain values.

Definition: A random variable $y$ is a function that maps outcomes in the sample space $\Omega$ into a set/domain of numeric values $D_y$.

$$y : \Omega \to D_y$$

(3.8)

Some commonly used domains are real numbers $\mathbb{R}$, integers $\mathbb{Z}$, natural numbers $\mathbb{N}$, or subsets thereof. An example of a mapping from outcomes to numeric values is tail $\rightarrow$ 0, head $\rightarrow$ 1. In other cases such as the roll of one dice, the map is the identity function.

One may think of a random variable $y$ (blue font) as taking on values from a given domain $D_y$. With a random variable, its value is uncertain, i.e., its value is only known probabilistically.

For $A \subseteq D_y$ one can measure the probability of the random variable $y$ taking on a value from the set $A$. This is denoted by

$$P(y \in A)$$

(3.9)

This really means the probability of event $E$ which maps to set $A$

$$E = y^{-1}(A)$$

(3.10)

$$P(E)$$

(3.11)

where $y^{-1}(A)$ is the inverse image of $A$.

3.2.1 Discrete Random Variable

A discrete random variable is defined on finite or countably infinite domains. For example, the probability of rolling a natural in dice (sum of 7 or 11 with two dice) is given by

$$P(y \in \{7, 11\}) = \frac{6}{36} + \frac{2}{36} = \frac{8}{36} = \frac{2}{9}$$

3.2.2 Continuous Random Variable

A continuous random variable is defined on uncountably infinite domains. For example, the probability of my tee shot on a par-3 golf hole ending up within 10 meters of the hole is 0.1 or 10 percent.

$$P(y \in [0, 10]) = 0.1$$
3.3 Probability Distribution

A random variable $y$ is characterized by how its probability is distributed over its domain $D_y$. This can be captured by functions that map $D_y$ to $\mathbb{R}^+$. 

3.3.1 Cumulative Distribution Function

The most straightforward way to do this is to examine the probability measure for a random variable in terms of a Cumulative Distribution Function (CDF).

$$F_y : D_y \rightarrow [0, 1]$$ (3.12) 

It measures the amount probability or mass accumulated over the domain up to and including the point $y$. The color highlighted symbol $y$ is the random variable, while $y$ simply represents a value.

$$F_y(y) = P(y \leq y)$$ (3.13) 

To illustrate the concept, let $x_1$ and $x_2$ be the number on dice 1 and dice 2, respectively. Let $y = x_1 + x_2$, then $F_y(6) = P(y \leq 6) = 5/12$. The entire CDF for the discrete random variable $y$ (roll of two dice), $F_y(y)$ is

$\{(2, 1/36), (3, 3/36), (4, 6/36), (5, 10/36), (6, 15/36), (7, 21/36), (8, 26/36), (9, 30/36), (10, 33/36), (11, 35/36), (12, 36/36)\}$

As another example, the CDF for a continuous random variable $y$ that is defined to be uniformly distributed on the interval $[0, 2]$ is

$$F_y(y) = \frac{y}{2} \quad \text{on} \quad [0, 2]$$

When random variable $y$ follows this CDF, we may say that $y$ is distributed as Uniform $(0, 2)$, symbolically, $y \sim \text{Uniform} \ (0, 2)$.

3.3.2 Probability Mass Function

While the CDF indicates accumulated probability or mass (totaling 1), examining probability or mass locally can be more informative. In case the random variable is discrete (i.e., $D_y$ is discrete), a probability mass function (pmf) may be defined.

$$p_y : D_y \rightarrow [0, 1]$$ (3.14) 

This function indicates the amount of mass/probability at point $y_i \in D_y$,

$$p_y(y_i) = F_y(y_i) - F_y(y_{i-1})$$ (3.15) 

It can be calculated as the first difference of the CDF, i.e., the amount of accumulated mass at point $y_i$ minus the amount of accumulated mass at the previous point $y_{i-1}$.

For one dice $x_1$, the pmf is

$\{(1, 1/6), (2, 1/6), (3, 1/6), (4, 1/6), (5, 1/6), (6, 1/6)\}$
A second dice \( x_2 \) will have the same pmf. Both random variables follow the Discrete Uniform Distribution, \( \text{Randi} (1, 6) \).

\[
p_x(x) = \frac{1}{6} \mathbb{1}_{\{1 \leq x \leq 6\}} \quad (3.16)
\]

where \( \mathbb{1}_{\{c\}} \) is the indicator function (if \( c \) then 1 else 0).

If the two random variables are added \( y = x_1 + x_2 \), the pmf for the random variable \( y \) (roll of two dice), \( p_y(y) \) is

\[
\{(2, 1/36), (3, 2/36), (4, 3/36), (5, 4/36), (6, 5/36), (7, 6/36), (8, 7/36), (9, 2/36), (10, 3/36), (11, 4/36), (12, 5/36)\}
\]

The random variable \( y \) follows the Discrete Triangular Distribution (that peaks in the middle) and not the flat Discrete Uniform Distribution.

\[
p_y(y) = \frac{\min(y - 1, 13 - y)}{36} \mathbb{1}_{\{2 \leq y \leq 12\}} \quad (3.17)
\]

### 3.3.3 Probability Density Function

Suppose \( y \) is defined on the continuous domain, e.g., \( D_y = [0, 2] \), and that mass/probability is uniformly spread amongst all the points in the domain. In such situations, it is not productive to consider the mass at one particular point. Rather one would like to consider the mass in a small interval and scale it by dividing by the length of the interval. In the limit this is the derivative which gives the density. For a continuous random variable, if the function \( F_y \) is differentiable, a probability density function (pdf) may be defined.

\[
f_y : D_y \rightarrow \mathbb{R}^+ \quad (3.18)
\]

It is calculated as the first derivative of the CDF, i.e.,

\[
f_y(y) = \frac{dF_y(y)}{dy} 
\]

For example, the pdf for a uniformly distributed random variable \( y \) on \( [0, 2] \) is

\[
f_y(y) = \frac{d}{dy} \frac{y}{2} = \frac{1}{2} \quad \text{on} \ [0, 2]
\]

The pdf for the Uniform Distribution is shown in the figure below.
Random variates of this type may be generated using ScalaTion’s \texttt{Uniform (0, 2)} class within the \texttt{scalation.random} package.

\begin{verbatim}
val rvg = Uniform (0, 2)
val yi = rvg.gen
\end{verbatim}

For another example, the pdf for an exponentially distributed random variable \( y \) on \([0, \infty)\) with rate parameter \( \lambda \) is

\[
f_y(y) = \lambda e^{-\lambda y} \quad \text{on \([0, \infty)\)}
\]

The pdf for the Exponential \((\lambda = 1)\) Distribution is shown in the figure below.

Going the other direction, the CDF \( F_y(y) \) can be computed by summing the pmf \( p_y(y) \)
\[ F_y(y) = \sum_{x_i < y} p_y(x_i) \quad (3.20) \]

or integrating the pdf \( f_y(y) \).

\[ F_y(y) = \int_{-\infty}^{y} f_y(x)dx \quad (3.21) \]
3.4 Empirical Distribution

An empirical distribution may be used to describe a dataset probabilistically. Consider a dataset \((X, y)\) where \(X \in \mathbb{R}^{m \times n}\) is the data matrix collected about the predictor variables and \(y \in \mathbb{R}^m\) is the data vector collected about the response variable. In other words, the dataset consists of \(m\) instances of an \(n\)-dimensional predictor vector \(x_i\) and a response value \(y_i\).

The joint empirical probability mass function (epmf) may be defined on the basis of a given dataset \((X, y)\).

\[
p_{\text{data}}(x, y) = \frac{\nu(x, y)}{m} = \frac{1}{m} \sum_{i=0}^{m-1} \mathbb{1}_{\{x_i = x, y_i = y\}} \tag{3.22}
\]

where \(\nu(x, y)\) is the frequency count and \(\mathbb{1}_{\{c\}}\) is the indicator function (if \(c\) then 1 else 0).

The corresponding Empirical Cumulative Distribution Function (ECDF) may be defined as follows:

\[
F_{\text{data}}(x, y) = \frac{1}{m} \sum_{i=0}^{m-1} \mathbb{1}_{\{x_i \leq x, y_i \leq y\}} \tag{3.23}
\]
3.5 Expectation

Using the definition of a CDF, one can determine the *expected value* (or *mean*) for random variable \( y \) using a Riemann-Stieltjes integral.

\[
\mathbb{E}[y] = \int_{D_y} y \, dF_y(y)
\]  

(3.24)

The mean specifies the center of mass, e.g., a two-meters rod with the mass evenly distributed throughout, would have a center of mass at 1 meter. Although it will not affect the center of mass calculation, since the total probability is 1, unit mass is assumed (one kilogram). The center of mass is the balance point in the middle of the bar.

### 3.5.1 Continuous Case

When \( y \) is a continuous random variable, we may write the mean as follows:

\[
\mathbb{E}[y] = \int_{D_y} y \, f_y(y) \, dy
\]  

(3.25)

The mean of \( y \sim \text{Uniform} (0, 2) \) is

\[
\mathbb{E}[y] = \int_{0}^{2} y \, \frac{1}{2} \, dy = 1.
\]

### 3.5.2 Discrete Case

When \( y \) is a discrete random variable, we may write

\[
\mathbb{E}[y] = \sum_{y \in D_y} y \, p_y(y)
\]  

(3.26)

The mean for rolling two dice is \( \mathbb{E}[y] = 7 \). One way to interpret this is to imagine winning \( y \) dollars by playing a game, e.g., two dollars for rolling a 2 and twelve dollars for rolling a 12, etc. The expected earnings when playing the game once is seven dollars. Also, by the *law of large numbers*, the average earnings for playing the game \( n \) times will converge to seven dollars as \( n \) gets large.

### 3.5.3 Variance

The *variance* of random variable \( y \) is given by

\[
\mathbb{V}[y] = \mathbb{E} [(y - \mathbb{E}[y])^2]
\]  

(3.27)

The variance specifies how the mass spreads out from the center of mass. For example, the variance of \( y \sim \text{Uniform} (0, 2) \) is

\[
\mathbb{V}[y] = \mathbb{E} [(y - 1)^2] = \int_{0}^{2} (y - 1)^2 \, \frac{1}{2} \, dy = \frac{1}{3}
\]
That is, the variance of the one kilogram, two-meter rod is \( \frac{1}{3} \) kilogram meter\(^2\). Again, for probability to be viewed as mass, unit mass (one kilogram) must be used, so the answer may also be given as \( \frac{1}{3} \) meter\(^2\). Similarly to interpreting the mean as the center of mass, the variance corresponds to the moment of inertia. The *standard deviation* is simply the square root of variance.

\[
\text{SD} [y] = \sqrt{V[y]} \tag{3.28}
\]

For the two-meter rod, the standard deviation is \( \frac{1}{\sqrt{3}} = 0.57735 \). The percentage of mass within one standard deviation unit of the center of mass is then 58%. Many distributions, such as the Normal (Gaussian) distribution concentrate mass closer to the center. For example, the *Standard Normal Distribution* has the following pdf.

\[
f_y(y) = \frac{1}{\sqrt{2\pi}} e^{-y^2/2} \tag{3.29}
\]

The mean for this distribution is 0, while the variance is 1. The percentage of mass within one standard deviation unit of the center of mass is 68%. The pdf for the Normal \((\mu = 0, \sigma^2 = 1)\) Distribution is shown in the figure below.

**pdf for Standard Normal Distribution**

---

### 3.5.4 Covariance

The *covariance* of two random variable \( x \) and \( y \) is given by

\[
C[x,y] = \mathbb{E}[(x - \mathbb{E}[x])(y - \mathbb{E}[y])] \tag{3.30}
\]

The covariance specifies whether the two random variables have similar tendencies. If the random variables are *independent*, the covariance will be zero, while similar tendencies show up as positive covariance and dissimilar tendencies as negative covariance. *Correlation* normalizes covariance to the domain \([-1,1]\). Covariance can be extended to more than two random variables. Let \( z \) be a vector of \( k \) random variables, then a *covariance matrix* is produced.
\[ C(z) = [C[z_i, z_j]_{0 \leq i, j < k} \] (3.31)
3.6 Algebra of Random Variables

When random variables $x_1$ and $x_2$ are added to create a new random variable $y$,

$$ y = x_1 + x_2 $$

how is $y$ described in terms of mean, variance and probability distribution? Also, what happens when a random variable is multiplied a constant?

$$ y = ax $$

3.6.1 Expectation is a Linear Operator

The expectation/mean of the sum is simply the sum of the means.

$$ E[y] = E[x_1] + E[x_2] \quad (3.32) $$

The expectation of a random variable multiplied by a constant, is the constant multiplied by the random variable’s expectation.

$$ E[ay] = aE[y] \quad (3.33) $$

The last two equations imply that expectation is a linear operator.

3.6.2 Variance is not a Linear Operator

The variance of the sum is the sum of variances plus twice the covariance.

$$ \text{V}[y] = \text{V}[x_1] + \text{V}[x_2] + 2 \text{C}[x_1, x_2] \quad (3.34) $$

When the random variable are independent, the covariance is zero, so the variance of sum is just the sum of variances.

$$ \text{V}[y] = \text{V}[x_1] + \text{V}[x_2] \quad (3.35) $$

The variance of a random variable multiplied by a constant, is the constant squared multiplied by the random variable’s variance.

$$ \text{V}[ay] = a^2\text{V}[y] \quad (3.36) $$

See the exercises for derivations.

3.6.3 Convolution of Probability Distributions

Determination the new probability distribution is more difficult. Assuming the random variables are independent and discrete, the pmf of the sum $p_y$ is the convolution of two pmfs $p_{x_1}$ and $p_{x_2}$.

$$ p_y = p_{x_1} * p_{x_2} \quad (3.37) $$
\[ p_y(y) = \sum_{x \in D_x} p_{x_1}(x) p_{x_2}(y-x) \quad (3.38) \]

For example, letting \( x_1, x_2 \sim \text{Bernoulli}(p) \), i.e., \( p_{x_1}(x) = p^x(1-p)^{1-x} \) on \( D_x = \{0, 1\} \), gives

\[
\begin{align*}
  p_y(0) &= \sum_{x \in D_x} p_{x_1}(x) p_{x_2}(0-x) = p^2 \\
  p_y(1) &= \sum_{x \in D_x} p_{x_1}(x) p_{x_2}(1-x) = 2p(1-p) \\
  p_y(2) &= \sum_{x \in D_x} p_{x_1}(x) p_{x_2}(2-x) = (1-p)^2
\end{align*}
\]

which indicates that \( y \sim \text{Binomial}(p, 2) \). The pmf for the Binomial\((p, n)\) distribution is

\[ p_y(y) = \binom{n}{y} p^y (1-p)^{n-y} \quad (3.39) \]

Now, assuming the random variables are independent and continuous, the pdf of the sum \( f_y \) is the \textit{convolution} of two pdfs \( f_{x_1} \) and \( f_{x_2} \).

\[ f_y = f_{x_1} * f_{x_2} \quad (3.40) \]

\[
 f_y(y) = \int_{D_x} f_{x_1}(x) f_{x_2}(y-x) \, dx \quad (3.41)
\]

For example, letting \( x_1, x_2 \sim \text{Uniform}(0, 1) \), i.e., \( f_{x_1}(x) = 1 \) on \( D_x = [0, 1] \), gives

\[
\begin{align*}
  \text{for } y \in [0, 1] & \quad f_y(y) = \int_{[0,y]} f_{x_1}(x) f_{x_2}(y-x) \, dx = y \\
  \text{for } y \in [1, 2] & \quad f_y(y) = \int_{[0,2-y]} f_{x_1}(x) f_{x_2}(y-x) \, dx = 2-y
\end{align*}
\]

which indicates that \( y \sim \text{Triangular}(0, 1, 2) \).

### 3.6.4 Central Limit Theorem

When several random variables are added (or averaged), interesting phenomena occurs, e.g., consider the distribution of \( y \) as the sum of \( m \) random variables.

\[ y = \sum_{i=0}^{m-1} x_i \]

When \( x_i \sim \text{Uniform}(0, 1) \) with mean \( \frac{1}{2} \) and variance \( \frac{1}{12} \), then for \( m \) large enough \( y \) will follow a Normal distribution

\[ y \sim \text{Normal}(\mu, \sigma^2) \]
where \( \mu = \frac{m}{2} \) and \( \sigma^2 = \frac{m}{12} \). The pdf for the Normal Distribution is

\[
f_y(y) = \frac{1}{\sqrt{2\pi \sigma}} e^{-\frac{1}{2} (\frac{y-\mu}{\sigma})^2}
\]

(3.42)

For most distributions, summed random variables will be approximately distributed as Normal, as indicated by the Central Limit Theorem (CLT); for proofs see \[28, 6\]. Suppose \( x_i \sim F \) with mean \( \mu_x \) and variance \( \sigma_x^2 < \infty \), then the sum of \( m \) independent and identically distributed (iid) random variables is distributed as follows:

\[
y = \sum_{i=0}^{m-1} x_i \sim N(m\mu_x, m\sigma_x^2) \quad \text{as } m \to \infty
\]

(3.43)

This is one simple form of the CLT. See the exercises for a visual illustration of the CLT.

Similarly, the sum of \( m \) independent and identically distributed random variables (with mean \( \mu_x \) and variance \( \sigma_x^2 \)) divided by \( m \) will also be Normally distributed for sufficiently large \( m \).

\[
y = \frac{1}{m} \sum_{i=0}^{m-1} x_i
\]

The expectation of \( y = \frac{1}{m} m \mu_x = \mu_x \), while variance is \( \sigma_x^2/m \), so

\[
y \sim \text{Normal}(\mu_x, \sigma_x^2/m)
\]

As, \( \mathbb{E}[y] = \mu_x \), \( y \) can serve as an unbiased estimator of \( \mu_x \). This can be transformed to the Standard Normal Distribution with the following transformation.

\[
z = \frac{y - \mu_x}{\sigma_x/\sqrt{m}} \sim \text{Normal}(0, 1)
\]

The Normal distribution is also referred to as the Gaussian distribution. See the exercises for related distributions: Chi-square, Student’s \( t \) and \( F \).
3.7 Median, Mode and Quantiles

As stated, the mean is the expected value, a probability weighted sum/integral of the values in the domain of the random variable. Other ways of characterizing a distribution are based more directly on the probability.

3.7.1 Median

Moving along the distribution, the place at which half of the mass is below you and half is above you is the **median**.

\[
P(y \leq \text{median}) \geq \frac{1}{2} \quad \text{and} \quad P(y \geq \text{median}) \geq \frac{1}{2}
\] (3.44)

Given equally likely values (1, 2, 3), the median is 2. Given equally likely values (1, 2, 3, 4), there are two common interpretations for the median: The smallest value satisfying the above equation (i.e., 2) or the average of the values satisfying the equation (i.e., 2.5) The median for two dice (with the numbers summed) which follow the Triangular distribution is 7.

3.7.2 Quantile

The **median** is also referred to as the half quantile.

\[
Q[y] = F^{-1}_y\left(\frac{1}{2}\right)
\] (3.45)

More generally, the \( p \in [0, 1] \) quantile is given by

\[
pQ[y] = F^{-1}_y(p)
\] (3.46)

where \( F^{-1}_y \) is the inverse CDF (iCDF). For example, recall the CDF for Uniform (0, 2) is

\[
p = F_y(y) = \frac{y}{2} \quad \text{on} \quad [0, 2]
\]

Taking the inverse yields the iCDF.

\[
F^{-1}_y(p) = 2p \quad \text{on} \quad [0, 1]
\]

Consequently, the median \( Q[y] = F^{-1}_y\left(\frac{1}{2}\right) = 1 \).

3.7.3 Mode

Similarly, one may be interested in the **mode**, which is the average of the points of maximal probability mass.

\[
M[y] = \arg\max_{y \in D_y} p_y(y)
\] (3.47)

The mode for rolling two dice is \( y = 7 \). For continuous random variables, it is the average of points of maximal probability density.

\[
M[y] = \arg\max_{y \in D_y} f_y(y)
\] (3.48)

For the two-meter rod, the mean, median and mode are all equal to 1.
3.8 Joint, Marginal and Conditional Distributions

Knowledge of one random variable may be useful in narrowing down the possibilities for another random variable. Therefore, it is important to understand how probability is distributed in multiple dimensions. There are three main concepts: joint, marginal and conditional.

In general, the joint CDF for two random variables $x$ and $y$ is

$$F_{xy}(x,y) = P(x \leq x, y \leq y) \tag{3.49}$$

3.8.1 Discrete Case: Joint and Marginal Mass

In the discrete case, the joint pmf for two random variables $x$ and $y$ is

$$p_{xy}(x_i, y_j) = F_{xy}(x_i, y_j) - [F_{xy}(x_{i-1}, y_j) + F_{xy}(x_i, y_{j-1}) - F_{xy}(x_{i-1}, y_{j-1})] \tag{3.50}$$

See the exercises to check this formula for the matrix shown below.

Imagine nine weights placed in a 3-by-3 grid with the number indicating the relative mass.

```java
val mat = MatrixD ((3, 3), 1, 2, 3,
                   4, 5, 6,
                   7, 8, 9)
```

Dividing the matrix by 45 or calling `toProbability` in the `scalation.mathstat.Probability` object yields a probability matrix representing a joint probability mass function (pmf), $p_{xy}(x_i, y_j)$,

```java
MatrixD(0.0222222, 0.0444444, 0.0666667,
        0.0888889, 0.111111, 0.133333,
        0.155556, 0.177778, 0.200000)
```

The marginal pmfs are computed as follows:

$$p_x(x_i) = \sum_{y_j \in D_y} p_{xy}(x_i, y_j) \quad \text{sum out } y \tag{3.51}$$

$$p_y(y_j) = \sum_{x_i \in D_x} p_{xy}(x_i, y_j) \quad \text{sum out } x \tag{3.52}$$

Carrying out the summations or calling `margProbX` (pxy) for $p_x(x_i)$ and `margProbY` (pxy) for $p_y(y_j)$ gives,

Marginal $X$: $px = VectorD(0.13333333333333333, 0.33333333333333337, 0.53333333333333334)$
Marginal $Y$: $py = VectorD(0.26666666666666666, 0.33333333333333337, 0.4)$

Scaling back by multiplying by 45 produces,

45*Marg $X$: $px = VectorI(6, 15, 24)$
45*Marg $Y$: $py = VectorI(12, 15, 18)$

It is now easy to see that $px$ is based on row sums, while $py$ is based on column sums.
3.8.2 Continuous Case: Joint and Marginal Density

In the continuous case, the joint pdf for two random variables $x$ and $y$ is

$$f_{xy}(x, y) = \frac{\partial^2 F_{xy}(x, y)}{\partial x \partial y}$$  \hspace{1cm} (3.53)

Consider the following joint pdf that specifies the distribution of one kilogram of mass (or probability) uniformly over a 2-by-3 meter plate.

$$f_{xy}(x, y) = \frac{1}{6} \text{ on } [0, 2] \times [0, 3]$$

The joint CDF is then a double integral,

$$F_{xy}(x, y) = \int_0^x \int_0^y \frac{1}{6} dydx = \frac{xy}{6}$$

There are two marginal pdfs that are single integrals: Think of the mass of the vertical red line being collected into the thick red bar at the bottom. Collecting all such lines creates the red bar at the bottom and its mass is distributed as follows:

$$f_x(x) = \int_0^3 \frac{1}{6} dy = \frac{3}{6} = \frac{1}{2} \text{ on } [0, 2] \text{ integrate out } y$$

Now think of the mass of the horizontal green line being collected into the thick green bar on the left. Collecting all such lines creates the green bar on the left and its mass is distributed as follows:

$$f_y(y) = \int_0^2 \frac{1}{6} dx = \frac{2}{6} = \frac{1}{3} \text{ on } [0, 3] \text{ integrate out } x$$

For more details and examples, see Class 7 of [72].
3.8.3 Discrete Case: Conditional Mass

Conditional probability can be examined locally. Given two discrete random variables $x$ and $y$, the conditional mass function of $x$ given $y$ is defined as follows:

$$p_{x|y}(x_i, y_j) = P(x = x_i | y = y_j) = \frac{p_{xy}(x_i, y_j)}{p_y(y_j)} \quad (3.54)$$

where $p_{xy}(x_i, y_j)$ is the joint mass function. Again, the marginal mass functions are

$$p_x(x_i) = \sum_{y_j \in D_y} p_{xy}(x_i, y_j)$$
$$p_y(y_j) = \sum_{x_i \in D_x} p_{xy}(x_i, y_j)$$

Consider the following example: Roll two dice. Let $x$ be the value on the first dice and $y$ be the sum of the two dice. Compute the conditional pmf for $x$ given that it is known that $y = 2$.

$$p_{x|y}(x_i, 2) = P(x = x_i | y = 2) = \frac{p_{xy}(x_i, 2)}{p_y(2)} \quad (3.55)$$

Try this problem for each possible value for $y$.

3.8.4 Continuous Case: Conditional Density

Similarly, for two continuous random variables $x$ and $y$, the conditional density function of $x$ given $y$ is defined as follows:

$$f_{x|y}(x, y) = \frac{f_{xy}(x, y)}{f_y(y)} \quad (3.56)$$

where $f_{xy}(x, y)$ is the joint density function. The marginal density functions are

$$f_x(x) = \int_{y \in D_y} f_{xy}(x, y)dy \quad (3.57)$$
$$f_y(y) = \int_{x \in D_x} f_{xy}(x, y)dx \quad (3.58)$$

The marginal density function in the $x$-dimension is the probability mass projected onto the $x$-axis from all other dimensions, e.g., for a bivariate distribution with mass distributed in the first $xy$ quadrant, all the mass will fall onto the $x$-axis.

Consider the example below where the random variable $x$ indicates how far down the centerline of a straight golf hole the golf ball was driven in units of 100 yards. The random variable $y$ indicates how far left (positive) or right (negative) the golf ball ends up from the center of the fairway. Let us call these random variable distance and dispersion. The golfer teed the ball up at location $[0, 0]$. For simplicity, assume the probability is uniformly distributed within the triangle.
As the area of the triangle is 3, the joint density function is

\[ f_{xy}(x, y) = \frac{1}{3} \quad \text{on} \quad x \in [0, 3], y \in [-x/3, x/3] \]

The distribution (density) of the driving distance down the centerline is given the marginal density for the random variable \( x \)

\[
f_x(x) = \int_{-x/3}^{x/3} f_{xy}(x, y) \, dy = \frac{1}{3} \left[ \frac{x^3}{3} \right]_{-x/3}^{x/3} = \frac{2x}{9}
\]

Therefore, the conditional density of dispersion \( y \) given distance \( x \) is given by

\[
f_{y|x}(x, y) = \frac{f_{xy}(x, y)}{f_x(x)} = \frac{1/3}{2x/9} = \frac{3}{2x}
\]

### 3.8.5 Independence

The two random variables \( x \) and \( y \) are said to \textit{independent} denoted \( x \perp y \) when the joint CDF (equivalently pmf/pdf) can be factored into the product of its marginal CDFs (equivalently pmf/pdf).

\[
F_{xy}(x, y) = F_x(x) F_y(y)
\]

For example, determine which of the following two joint density functions defined on \([0, 1]^2\) signify independence.

\[
f_{xy}(x, y) = 4xy
\]
\[
f_{xy}(x, y) = 8xy - x - y
\]

For the first joint density, the two marginal densities are the following:

\[
f_x(x) = \int_0^1 4xy \, dy = \frac{4xy^2}{2} \bigg|_0^1 = 2x
\]
The product of the marginal densities \( f_x(x) f_y(y) = 4xy \) is the joint density.

Compute the conditional density under the assumption that the random variables, \( x \) and \( y \), are independent.

\[
 f_{x|y}(x, y) = \frac{f_{xy}(x, y)}{f_y(y)}
\]

As the joint density can be factored, \( f_{xy}(x, y) = f_x(x) f_y(y) \), we obtain,

\[
 f_{x|y}(x, y) = \frac{f_x(x) f_y(y)}{f_y(y)} = f_x(x)
\]

showing that the value of random variable \( y \) has no effect on \( x \). See the exercises for a proof that independence implies zero covariance (and therefore zero correlation).

### 3.8.6 Conditional Expectation

The value of one random variable may influence the expected value of another random variable. The *conditional expectation* of random variable \( x \) given random variable \( y \) is defined as follows:

\[
 E[x|y = y] = \int_D x dF_{x|y}(x, y) \tag{3.63}
\]

When \( y \) is a discrete random variable, we may write

\[
 E[x|y = y] = \sum_{x \in D_x} x p_{x|y}(x, y) \tag{3.64}
\]

When \( y \) is a continuous random variable, we may write

\[
 E[x|y = y] = \int_{D_x} x f_{x|y}(x, y)dx \tag{3.65}
\]

Consider the previous example on the dispersion \( y \) of a golf ball conditioned on the driving distance \( y \). Compute the conditional mean and the conditional variance for \( y \) given \( x \).

\[
 \mu_{y|x} = E[y|x = x] = \int_{-x/3}^{x/3} y f_{y|x}(x, y)dy
\]

\[
 \sigma_{y|x}^2 = E[(y - \mu_{y|x})^2|x = x] = \int_{-x/3}^{x/3} (y - \mu_{y|x})^2 f_{y|x}(x, y)dy
\]
3.8.7 Conditional Independence

A wide class of modeling techniques are under the umbrella of probabilistic graphical models (e.g., Bayesian Networks and Markov Networks). They work by factoring a joint probability based on conditional independencies. Random variables $x$ and $y$ are conditionally independent given $z$, denoted

$$x \perp y \mid z$$

means that

$$F_{x,y|z}(x, y, z) = F_{x|z}(x, z) F_{y|z}(y, z)$$
3.9 Odds

Another way of looking a probability is odds. This is the ratio of probabilities of an event $A$ occurring over the event not occurring $S - A$.

$$\text{odds}(y \in A) = \frac{P(y \in A)}{P(y \in S - A)} = \frac{P(y \in A)}{1 - P(y \in A)} \quad (3.66)$$

For example, the odds of rolling a pair dice and getting natural is 8 to 28.

$$\text{odds}(y \in \{7,11\}) = \frac{8}{28} = \frac{2}{7} = .2857$$

Of the 36 individual outcomes, eight will be a natural and 28 will not. Odds can be easily calculated from probability.

$$\text{odds}(y \in \{7,11\}) = \frac{P(y \in \{7,11\})}{1 - P(y \in \{7,11\})} = \frac{2/9}{7/9} = \frac{2}{7} = .2857$$

Calculating probability from odds may be done as follows:

$$P(y \in \{7,11\}) = \frac{\text{odds}(y \in \{7,11\})}{1 + \text{odds}(y \in \{7,11\})} = \frac{2/7}{9/7} = \frac{2}{9} = .2222$$
3.10 Example Problems

Understanding of some of techniques to be discussed requires some background in conditional probability.

1. Consider the probability of rolling a natural (i.e., 7 or 11) with two dice where the random variable $y$ is the sum of the dice.

$$P(y \in \{7, 11\}) = \frac{1}{6} + \frac{1}{18} = \frac{2}{9}$$

If you knew you rolled a natural, what is the conditional probability that you rolled a 5 or 7?

$$P(y \in \{5, 7\} | y \in \{7, 11\}) = \frac{P(y \in \{5, 7\}, y \in \{7, 11\})}{P(y \in \{7, 11\})} = \frac{1/6}{2/9} = \frac{3}{4}$$

This is the conditional probability of rolling a 5 or 7 given that you rolled a natural.

More generally, the conditional probability that $y \in A$ given that $x \in B$ is the joint probability divided by the probability that $x \in B$.

$$P(y \in A | x \in B) = \frac{P(y \in A, x \in B)}{P(x \in B)}$$

where

$$P(y \in A, x \in B) = P(x \in B | y \in A) P(y \in A)$$

Therefore, the conditional probability of $y$ given $x$ is

$$P(y \in A | x \in B) = \frac{P(x \in B | y \in A) P(y \in A)}{P(x \in B)}$$

This is Bayes Theorem written using random variables, which provides an alternative way to compute conditional probabilities, i.e., $P(y \in \{5, 7\} | y \in \{7, 11\})$ is

$$\frac{P(y \in \{7, 11\} | y \in \{5, 7\}) P(y \in \{5, 7\})}{P(y \in \{7, 11\})} = \frac{(3/5) \cdot (5/18)}{2/9} = \frac{3}{4}$$

2. To illustrate the usefulness of Bayes Theorem, consider the following problem from John Allen Paulos that is hard to solve without it. Suppose you are given three coins, two fair and one counterfeit (always lands heads). Randomly select one of the coins. Let $x$ indicate whether the selected coin is fair (0) or counterfeit (1). What is the probability that you selected the counterfeit coin?

$$P(x = 1) = \frac{1}{3}$$

Obviously, the probability is 1/3, since the probability of picking any of the three coins is the same. This is the prior probability.

Not satisfied with this level of uncertainty, you conduct experiments. In particular, you flip the selected coin three times and get all heads. Let $y$ indicate the number of heads rolled. Using Bayes Theorem, we have,
\[ P(x = 1 \mid y = 3) = \frac{P(y = 3 \mid x = 1) P(x = 1)}{P(y = 3)} = \frac{1 \cdot (1/3)}{5/12} = 4/5 \]

where \( P(y = 3) = (1/3)(1) + (2/3)(1/8) = 5/12 \). After conducting the experiments (collecting evidence) the probability estimate may be improved. Now the posterior probability is 4/5.
3.11 Estimating Parameters from Samples

Given a model for predicting a response value for \( y \) from a feature/predictor vector \( x \),

\[
y = f(x; b) + \epsilon
\]
one needs to pick a functional form for \( f \) and collect a sample of data to estimate the parameters \( b \). The sample will consist of \( m \) instances \((y_i, x_i)\) that form the response/output vector \( y \) and the data/input matrix \( X \).

\[
y = f(X; b) + \epsilon
\]

There are multiple types of estimation procedures. The central ideas are to minimize error or maximize the likelihood that the model would generate data like the sample. A common way to minimize error is to minimize the Mean Squared Error (MSE). The error vector \( \epsilon \) is the difference between the actual response vector \( y \) and the predicted response vector \( \hat{y} \).

\[
\epsilon = y - \hat{y} = y - f(x; b)
\]

The mean squared error on the length (Euclidean norm) of the error vector \( \|\epsilon\| \) is given by

\[
E[\|\epsilon\|^2] = V[\|\epsilon\|] + E[\|\epsilon\|^2]
\]

where \( V[\|\epsilon\|] \) is error variance and \( E[\|\epsilon\|] \) is the error mean. If the model is unbiased the error mean will be zero, in which case the goal is to minimize the error variance.

3.11.1 Sample Mean

Suppose the speeds of cars on an interstate highway are Normally distributed with a mean at the speed limit of 70 mph (113 kph) and a standard deviation of 8 mph (13 kph), i.e., \( y \sim N(\mu, \sigma^2) \) in which case the model is

\[
y = \mu + \epsilon
\]

where \( \epsilon \sim N(0, \sigma^2) \). Create a sample of size \( m = 100 \) data points, using a Normal random variate generator. The population values for the mean \( \mu \) and standard deviation \( \sigma \) are typically unknown and need to be estimated from the sample, hence the names sample mean \( \hat{\mu} \) and sample standard deviation \( \hat{\sigma} \). Show the generated sample, by plotting the data points and displaying a histogram.

object SampleStats extends App:

```scala
val (mu, sig) = (70.0, 8.0) // pop. mean and standard deviation
val m = 100 // sample size
val rvg = Normal (mu, sig * sig) // Normal random variate generator
val sample = VectorD (for i <- 0 until m yield rvg.gen) // sample from Normal distribution
val (mu_, sig_) = (sample.mean, sample.stdev) // sample mean and standard deviation
println (s"(mu_, sig_) = ($mu_, $sig_)")
new Plot (null, sample)
```
3.11.2 Confidence Interval

Now that you have an estimate for the mean, you begin to wonder if it is correct or rather close enough. Generally, an estimate is considered close enough if its confidence interval contains the population mean. Collect an iid sample of values into a vector $y$. Then the sample mean is simply

$$\hat{\mu} = \frac{1}{m} \cdot \sum_{i=0}^{m-1} y_i$$

To create a confidence interval, we need to determine the variability or variance in the estimate $\hat{\mu}$.

$$\mathbb{V}[\hat{\mu}] = \mathbb{V} \left[ \frac{1}{m} \sum_{i=0}^{m-1} y_i \right] = \frac{1}{m^2} \sum_{i=0}^{m-1} \mathbb{V}[y_i] = \frac{\sigma^2}{m}$$

The difference between the estimate from the sample and the population mean is Normally distributed and centered at zero (show that $\hat{\mu}$ is an unbiased estimator for $\mu$, i.e., $\mathbb{E}[\hat{\mu}] = \mu$).

$$\hat{\mu} - \mu \sim N(0, \frac{\sigma^2}{m})$$

We would like to transform the difference so that the resulting expression follows a Standard Normal distribution. This can be done by dividing by $\frac{\sigma}{\sqrt{m}}$.

$$\frac{\hat{\mu} - \mu}{\sigma/\sqrt{m}} \sim N(0, 1)$$

Consequently, the probability that the expression is greater than $z$ is given by the CDF of the Standard Normal distribution, $F_N(z)$.

$$P \left( \frac{\hat{\mu} - \mu}{\sigma/\sqrt{m}} > z \right) = 1 - F_N(z)$$

One might consider that if $z = 2$, two standard deviation units, then the estimate is not close enough. The same problem can exist on the negative side, so we should require

$$\left| \frac{\hat{\mu} - \mu}{\sigma/\sqrt{m}} \right| \leq 2$$

In other words,

$$|\hat{\mu} - \mu| \leq \frac{2\sigma}{\sqrt{m}}$$

This condition implies that $\mu$ would likely be inside the following confidence interval.

$$\left[ \hat{\mu} - \frac{2\sigma}{\sqrt{m}}, \hat{\mu} + \frac{2\sigma}{\sqrt{m}} \right]$$
In this case it is easy to compute values for the lower and upper bounds of the confidence interval. The interval half width is simply $\frac{2}{10} = 1.6$, which is to be subtracted and added to the sample mean.

Use ScalaTion to determine the probability that $\mu$ is within such confidence intervals?

```scala
println (s"1 - F(2) = \${1 - normalCDF (2)}")
```

The probability is one minus twice this value. If 1.96 is used instead of 2, what is the probability, expressed as a percentage.

Typically, the population standard deviation is unlikely to be known. It would need to estimated by using the sample standard deviation, where the sample variance is

$$\hat{\sigma}^2 = \frac{1}{m-1} \sum_{i=0}^{m-1} (y_i - \hat{\mu})^2$$

(3.68)

Note, this textbook uses $\hat{\theta}$ to indicate an estimator for parameter $\theta$, regardless of whether it is a Maximum Likelihood (MLE) estimator. This substitution introduces more variability into the estimation of the confidence interval and results in the Standard Normal distribution ($z$-distribution)

$$\left[ \hat{\mu} - z^* \hat{\sigma} \sqrt{\frac{1}{m}}, \hat{\mu} + z^* \hat{\sigma} \sqrt{\frac{1}{m}} \right]$$

(3.69)

being replace by the Student’s $t$-distribution

$$\left[ \hat{\mu} - t^* \hat{\sigma} \sqrt{\frac{1}{m}}, \hat{\mu} + t^* \hat{\sigma} \sqrt{\frac{1}{m}} \right]$$

(3.70)

where $z^*$ and $t^*$ represent distances from zero, e.g., 1.96 or 2.09, that are large enough so that the analyst is comfortable with the probability that they may be wrong.

The numerators for the interval half widths (ihw) are calculated by the following top-level functions in Statistics.scala. The `z_sigma` function is used for the $z$-distribution.

```scala
def z_sigma (sig: Double, p: Double = .95): Double =
  val pp = 1.0 - (1.0 - p) / 2.0 // e.g., .95 --> .975 (two tails)
  val z = random.Quantile.normalInv (pp)
  z * sig
end z_sigma
```

The `t_sigma` function is used for the $t$-distribution.

```scala
def t_sigma (sig: Double, df: Int, p: Double = .95): Double =
  if df < 1 then { flaw ("interval", "must have at least 2 observations"); return 0.0 }
  val pp = 1.0 - (1.0 - p) / 2.0 // e.g., .95 --> .975 (two tails)
  val t = random.Quantile.studentTInv (pp, df)
  t * sig
end t_sigma
```

Does the probability you determined in the last example problem make any sense. Seemingly, if you took several samples, only a certain percentage of them would have the population mean within their confidence interval.
object ConfidenceIntervalTest extends App:

    val (mu, sig) = (70.0, 8.0) // pop. mean and standard deviation
    val m = 100 // sample size
    val rm = sqrt(m)
    val rvg = Normal(mu, sig * sig) // Normal random variate generator
    var count_z, count_t = 0

    for it <- 1 to 100 do // test several datasets
        val y = VectorD(for i <- 0 until m yield rvg.gen) // sample from Normal distribution
        val (mu_, sig_) = (y.mean, y.stdev) // sample mean and standard deviation
        val ihw_z = z_sigma(sig_) / rm // interval half width: z-distribution
        val ci_z = (mu_ - ihw_z, mu_ + ihw_z) // z-confidence interval
        println(s"mu = $mu in ci_z = $ci_z?")
        if mu in ci_z then count_z += 1

        val ihw_t = t_sigma(sig_, m-1) / rm // interval half width: t-distribution
        val ci_t = (mu_ - ihw_t, mu_ + ihw_t) // z-confidence interval
        println(s"mu = $mu in ci_t = $ci_t?")
        if mu in ci_t then count_t += 1
    end for

    println(s"mu inside $count_z % z-confidence intervals")
    println(s"mu inside $count_t % t-confidence intervals")
end ConfidenceIntervalTest

Imports: scalation., scalation.mathstat., scalation.random..

Try various values for m starting with m = 20. Compute percentages for both the t-distribution and the z-distribution. Given the default confidence level used by SCALATION is 0.95 (or 95%) what would you expect your percentages to be?

3.11.3 Estimation for Discrete Outcomes/Responses

Explain why the probability mass function (pmf) for flipping a coin n times with the experiment resulting in the discrete random variable y = k heads is given by the Binomial Distribution having unknown parameter p, the probability of getting a head for any particular coin flip,

\[ p_n(k) = P(y = k) = \binom{n}{k} p^k (1-p)^{n-k} \]

i.e., \( y \sim \text{Binomial}(n, p) \).

Now suppose an experiment is run and \( y = k \), a fixed number, e.g., \( n = 100 \) and \( k = 60 \). For various values of p, plot the following function.

\[ L(p) = \binom{n}{k} p^k (1-p)^{n-k} \]
What value of $p$ maximizes the function $L(p)$? The function $L(p)$ is called the *Likelihood function* and it is used in Maximum Likelihood Estimation (MLE) [13].

The `VectorD` class provides methods for computing statistics on vectors.
3.12 Entropy

The entropy of a discrete random variable \( y \) with probability mass function (pmf) \( p_y(y) \) is the negative of the expected value of the log of the probability.

\[
H(y) = H(p_y) = -E[\log_2 p_y] = -\sum_{y \in D_y} p_y(y) \log_2 p_y(y)
\] (3.71)

The following single loop is used in ScalaTion to compute entropy.

```scala
def entropy (px: VectorD): Double =
  var sum = 0.0
  for p <- px if p > 0.0 do sum -= p * log2 (p)
  sum
end entropy
```

For finite domains of size \( k = |D_y| \), entropy \( H(y) \) ranges from 0 to \( \log_2(k) \). Low entropy (close to 0) means that there is low uncertainty/risk in predicting an outcome of an experiment involving the random variable \( y \), while high entropy (close to \( \log_2 k \)) means that there is high uncertainty/risk in predicting an outcome of such an experiment. For binary classification \( k = 2 \), the upper bound on entropy is 1.

The entropy may be normalized by setting the base of the logarithm to the size of the domain \( k \), in which case, the entropy will be in the interval \([0, 1]\).

\[
H_k(y) = H_k(p_y) = -E[\log_k p_y] = -\sum_{y \in D_y} p_y(y) \log_k p_y(y)
\]

A random variable \( y \sim \text{Bernoulli}(p) \) may be used to model the flip of a single coin that has a probability of success/head (1) of \( p \). Its pmf is given by the following formula.

\[
p(y) = p^y(1-p)^{1-y}
\]

The pmf \( p_y \) can be captured in a probability vector \( p_y \)

\[
H(y) = H(p_y) = H(p_y) = H([p, 1-p]) = p \log_2 p + (1-p) \log_2 1 - p
\]

The figure below plots the entropy \( H([p, 1-p]) \) as probability of a head \( p \) ranges from 0 to 1.
A random variable \( y = z_1 + z_2 \) where \( z_1, z_2 \) are distributed as Bernoulli\((p)\) may be used to model the sum of flipping two coins.

\[
H(y) = H(p_y) = H([p^2, 2p(1-p), (1-p)^2])
\]

See the exercises for how to extend entropy to continuous random variables.

### 3.12.1 Joint Entropy

Entropy may be defined for multiple random variables as well. Given two discrete random variables, \( x, y \), with a joint pmf \( p_{x,y}(x,y) \) the joint entropy is defined as follows:

\[
H(x, y) = H(p_{x,y}) = -\mathbb{E} \left[ \log_2 p_{x,y} \right] = -\sum_{x \in D_x} \sum_{y \in D_y} p_{x,y}(x,y) \log_2 p_{x,y}(x,y)
\]  

(3.72)

### 3.12.2 Conditional Entropy

Replacing the joint pmf, with the conditional pmf gives conditional entropy.

\[
H(x|y) = H(p_{x|y}) = -\mathbb{E} \left[ \log_2 p_{x|y} \right] = -\sum_{x \in D_x} \sum_{y \in D_y} p_{x,y}(x,y) \log_2 p_{x,y}(x,y)
\]  

(3.73)

Suppose an experiment involves two random variables \( x \) and \( y \). Initially, the overall entropy is given by the joint entropy \( H(x, y) \). Now, partial evidence allows the value of \( y \) to be determined, so the overall entropy should decrease by \( y \)'s entropy.

\[
H(x|y) = H(x, y) - H(y)
\]  

(3.74)

When there is no dependency between \( x \) and \( y \) (i.e., they are independent), \( H(x, y) = H(x) + H(y) \), so

\[
H(x|y) = H(x)
\]  

(3.75)
At the other extreme, when there is full dependency (i.e., the value of $x$ can be determined from the value of $y$).

$$H(x|y) = 0$$ (3.76)

### 3.12.3 Relative Entropy

Relative entropy, also known as Kullback-Leibler (KL) divergence, measures the dissimilarity between two probability distributions.

#### Discrete Random Variable

Given a discrete random variable $y$ with two candidate probability mass functions (pmf)s $p_y(y)$ and $q_y(y)$ the relative entropy is defined as follows:

$$H(p_y||q_y) = E \left[ \log_2 \frac{p_y}{q_y} \right] = \sum_{y \in D_y} p_y(y) \log_2 \frac{p_y(y)}{q_y(y)}$$ (3.77)

One way to look at relative entropy is that it measures the uncertainty that is introduced by replacing the true/empirical distribution $p_y$ with an approximate/model distribution $q_y$. If the distributions are identical, then the relative entropy is 0, i.e., $H(p_y||p_y) = 0$. The larger the value of $H(p_y||q_y)$ the greater the dissimilarity between the distributions $p_y$ and $q_y$.

As an example, assume the true distribution for a coin is [.6, .4], but it is thought that the coin is fair [.5, .5]. The relative entropy is computed as follows:

$$H(p_y||q_y) = .6 \log_2 .6/.5 + .4 \log_2 .4/.5 = 0.029$$

#### Continuous Random Variable

Given a continuous random variable $y$ with two candidate probability density functions (pdf)s $f_y(y)$ and $g_y(y)$ the relative entropy is defined as follows:

$$H(f_y||g_y) = E \left[ \log_2 \frac{f_y}{g_y} \right] = \int_{D_y} f_y(y) \log_2 \frac{f_y(y)}{g_y(y)} dy$$ (3.78)

#### Maximum Likelihood Estimation

In this subsection, we examine the relationship between KL Divergence and Maximum Likelihood. Consider the dissimilarity of an empirical distribution $p_{data}(x, y)$ and a model generated distribution $p_{mod}(x, y, b)$.

$$H(p_{data}(x, y)||p_{mod}(x, y, b)) = E \left[ \log \frac{p_{data}(x, y)}{p_{mod}(x, y, b)} \right]$$ (3.79)

$$= \sum_{i=0}^{m-1} p_{data}(x_i, y_i) \log \frac{p_{data}(x_i, y_i)}{p_{mod}(x_i, y_i, b)}$$ (3.80)

Note, that $p_{data}(x_i, y_i)$ is unaffected by the choice of parameters $b$, so it represents a constant C.
\[
H(p_{\text{data}}(x,y)\|p_{\text{mod}}(x,y,b)) = C - \sum_{i=0}^{m-1} p_{\text{data}}(x_i, y_i) \log p_{\text{mod}}(x_i, y_i, b)
\] (3.81)

The probability for the \(i\)th data instance is \(\frac{1}{m}\), thus

\[
H(p_{\text{data}}(x,y)\|p_{\text{mod}}(x,y,b)) = C - \frac{1}{m} \sum_{i=0}^{m-1} \log p_{\text{mod}}(x_i, y_i, b)
\] (3.82)

The second term is the negative log-likelihood (the Chapter on Generalized Linear and Additive Models for details).

### 3.12.4 Cross Entropy

Relative entropy can be adjusted to capture overall entropy by adding the entropy of \(p_y\).

\[
H(p_y \times q_y) = H(p_y) + H(p_y\|q_y)
\] (3.83)

It is the sum of the entropy of the empirical distribution and the model distribution’s relative entropy to the empirical distribution. It can be calculated using the following formula (see exercises for details):

\[
H(p_y \times q_y) = -\sum_{y \in D_y} p_y(y) \log_2 q_y(y)
\] (3.84)

Since cross entropy is more efficient to calculate than relative entropy, it is a good candidate as a loss function for machine learning algorithms. The smaller the cross entropy, the more the model (e.g., Neural Network) agrees with the empirical distribution (dataset). The formula looks like the one for ordinary entropy with \(q_y\) substituted in as the argument for the log function. Hence the name cross entropy.

### 3.12.5 Mutual Information

Recall that if \(x\) and \(y\) are independent, then for all \(x \in D_x\) and \(y \in D_y\),

\[
p_{x,y}(x,y) = p_x(x) p_y(y)
\]

A possibly more interesting and practical question is to measure how close two random variables are to being independent. One approach is to look at the covariance (or correlation) between \(x\) and \(y\).

\[
\mathbb{C}[x,y] = \mathbb{E}[(x-\mu_x)(y-\mu_y)] = \sum_{x \in D_x} \sum_{y \in D_y} (x-\mu_x)(y-\mu_y)p_{x,y}(x,y)
\]

If \(x\) and \(y\) are independent, then

\[
\mathbb{C}[x,y] = \sum_{x \in D_x} \sum_{y \in D_y} [(x-\mu_x)p_x(x)][(y-\mu_y)p_y(y)] = 0
\]

An alternative is to look at the relative entropy of \(p_{x,y}\) and \(p_x p_y\).

\[
H(p_{x,y}\|p_x p_y) = \mathbb{E} \left[ \log_2 \frac{p_{x,y}}{p_x p_y} \right] = \sum_{x \in D_x} \sum_{y \in D_y} p_{x,y}(x,y) \log_2 \frac{p_{x,y}(x,y)}{p_x(x)p_y(y)}
\] (3.85)
The relative entropy (KL divergence) of the joint distribution to the product of the marginal distributions is referred to as mutual information.

\[ I(x; y) = H(p_{x,y} || p_x p_y) \] (3.86)

The following double loop is used in SCALATION to compute mutual information.

```scala
def muInfo (pxy: MatrixD, px: VectorD, py: VectorD): Double =
  var sum = 0.0
  for i <- pxy.indices; j <- pxy.indices2 do
    val p = pxy(i, j)
    if p > 0.0 then sum += p * log2 (p / (px(i) * py(j)))
  end for
  sum
end muInfo
```

As with covariance (or correlation) mutual information will be zero when \( x \) and \( y \) are independent. While independence implies zero covariance, independence is equivalent to zero mutual information. Mutual information is symmetric and non-negative. See the exercises for additional comparisons between covariance/correlation and mutual information.

While mutual information measures the dependence between two random variables, relative entropy (KL divergence) measures the dissimilarity of two distribution.

Mutual Information corresponds to Information Gain, i.e., the drop in entropy of one random variable due to knowledge of the value of the other random variable.

\[ I(x; y) = H(x) - H(x|y) = H(y) - H(y|x) \] (3.87)

The Probability object in the scalation.stat package provides methods to compute probabilities from frequencies, compute joint, marginal, conditional and log probabilities, as well as entropy, normalized entropy, relative entropy, cross entropy, and mutual information.

**Probability Object**

**Class Methods:**

```scala
object Probability:

  def isProbability (px: VectorD): Boolean = px.min >= 0.0 && abs (px.sum - 1.0) < EPSILON
  def isProbability (pxy: MatrixD): Boolean = pxy.min >= 0.0 && abs (pxy.sum - 1.0) < EPSILON
  def frequency (y: VectorI, k: Int, idx_: VectorI = null): VectorI =
  def frequency (x: VectorI, y: VectorI, k: Int, vl: Int, idx_: VectorI): (Double, VectorI) =
  def toProbability (nu: VectorI): VectorD = nu.toDouble / nu.sum.toDouble
  def toProbability (nu: VectorI, n: Int): VectorD = nu.toDouble / n.toDouble
  def toProbability (nu: MatrixD): MatrixD = nu / nu.sum
  def toProbability (nu: MatrixD, n: Int): MatrixD = nu / n.toDouble
```
def jointProbXY (px: VectorD, py: VectorD): MatrixD = outer (px, py)
def margProbX (pxy: MatrixD): VectorD =
def margProbY (pxy: MatrixD): VectorD =
def condProbY_X (pxy: MatrixD, px_: VectorD = null): MatrixD =
def condProbX_Y (pxy: MatrixD, py_: VectorD = null): MatrixD =
def logProb (px: VectorD): VectorD = px.map (- log2 (_))
def entropy (px: VectorD): Double =
def entropy (nu: VectorI): Double =
def entropy (px: VectorD, b: Int): Double =
def nentropy (px: VectorD): Double =
def centropy (px: VectorD, qx: VectorD): Double =
def entropy (pxy: MatrixD): Double =
def entropy (pxy: MatrixD, px_y: MatrixD): Double =
def muInfo (pxy: MatrixD, px: VectorD, py: VectorD): Double =
def muInfo (pxy: MatrixD): Double = muInfo (pxy, margProbX (pxy), margProbY (pxy))
3.13 Exercises

Several random number and random variate generators can be found in ScalaTion’s random package. Some of the following exercises will utilize these generators.

1. Let the random variable $h$ be the number heads when two coins are flipped. Determine the following conditional probability: $P(h = 2| h \geq 1)$.

2. Prove Bayes Theorem.

\[ P(A|B) = \frac{P(B|A)P(A)}{P(B)} \]

3. Compute the mean and variance for the Bernoulli Distribution with success probability $p$.

\[ p_y(y) = p^y (1-p)^{1-y} \text{ for } y \in \{0,1\} \]

4. Use the Randi random variate generator to run experiments to check the pmf and CDF for rolling two dice.

   ```scala
   import scalation.mathstat._
   import scalation.random.Randi

   object DiceTest extends App:
     val dice = Randi (1, 6)
     val x = VectorD.range (0, 13)
     val freq = new VectorD (13)
     for i <- 0 until 10000 do
       val sum = dice.igen + dice.igen
       freq(sum) += 1
     end for
     new Plot (x, freq)
   end DiceTest
   
   5. Show that the variance may be written as follows:

\[ \mathbb{V}[y] = \mathbb{E}[(y - \mathbb{E}[y])^2] = \mathbb{E}[y^2] - \mathbb{E}[y]^2 \]

6. Show that the covariance may be written as follows:

\[ \mathbb{C}[x,y] = \mathbb{E}[(x - \mathbb{E}[x])(y - \mathbb{E}[y])] = \mathbb{E}[xy] - \mathbb{E}[x]\mathbb{E}[y] \]

7. Show that the covariance of two independent, continuous random variables, $x$ and $y$, is zero.

\[ \mathbb{C}[x,y] = \mathbb{E}[(x - \mu_x)(y - \mu_y)] = \int_{D_x} \int_{D_y} (x - \mu_x)(y - \mu_y)f_{xy}(x,y)dxdy \]

where $\mu_x = \mathbb{E}[x]$ and $\mu_y = \mathbb{E}[y]$. 

78
8. Derive the formula for the expectation of the sum of random variables.

\[ E[x_1 + x_2] = E[x_1] + E[x_2] \]

9. Derive the formula for the variance of the sum of random variables.

\[ V[x_1 + x_2] = V[x_1] + V[x_2] + 2C[x_1, x_2] \]

Hint: use \( V[x_1 + x_2] = E[(x_1 + x_2)^2] - E[x_1 + x_2]^2 \)

10. Use the Uniform random variate generator and the Histogram class to run experiments illustrating the Central Limit Theorem (CLT).

```scala
import scalation.mathstat._
import scalation.random.Uniform

object CLTTest extends App:

  val rg = Uniform()
  val x = VectorD(for i <- 0 until 100000 yield rg.gen + rg.gen + rg.gen + rg.gen)
  new Histogram(x)

end CLTTest
```

Try with other distributions such as Exponential.

11. Chi-square distribution: Show that if \( z \sim \text{Normal}(0, 1) \), then

\[ z^2 \sim \chi^2_1 \]

12. Student’s t-distribution: Show that if \( z \sim \text{Normal}(0, 1) \) and \( v \sim \chi^2_k \), then

\[ \frac{z}{\sqrt{v/k}} \sim t_k \]

13. F distribution: Show that if \( u \sim \chi^2_{k_1} \) and \( v \sim \chi^2_{k_2} \), then

\[ \frac{u/k_1}{v/k_2} \sim F_{k_1,k_2} \]

14. Given three random variables such that \( x \perp y \mid z \), show that

\[ F_{x|y,z}(x, y, z) = F_{x|z}(x, z) \]

15. Show that formula for computing the joint probability mass function (pmf) for the 3-by-3 grid of weights is correct. Hint: Add/subtract rectangular regions of the grid and make sure nothing is double counted.
16. Show for \( k = 2 \) where \( \mathbf{p} = [p, 1-p] \), that \( H(\mathbf{p}) = p \log_2(p) + (1-p) \log_2(1-p) \). Plot the entropy \( H(\mathbf{p}) \) versus \( p \).

```scala
val p = VectorD.range (1, 100) / 100.0
val h = p.map (p => -p * log2 (p) - (1-p) * log2 (1-p))
new Plot (p, h)
```

17. Plot the entropy \( H \) and normalized entropy \( H_k \) for the first 16 Binomial(\( p, n \)) distributions, i.e., for the number of coins \( n = 1, \ldots, 16 \). Try with \( p = .6 \) and \( p = .5 \).

18. Entropy can be defined for continuous random variables. Take the definition for discrete random variables and replace the sum with an integral and the pmf with a pdf. Compute the entropy for \( y \sim \text{Uniform}(0,1) \).

19. Using the summation formulas for entropy, relative entropy and cross entropy, show that cross entropy is the sum of entropy and relative entropy.

20. Show that mutual information equals the sum of marginal entropies minus the joint entropy, i.e.,

\[
I(x;y) = H(x) + H(y) - H(x,y)
\]

21. Compare correlation and mutual information in terms of how well they measure dependence between random variables \( x \) and \( y \). Try various functional relationships: negative exponential, reciprocal, constant, logarithmic, square root, linear, right-arm quadratic, symmetric quadratic, cubic, exponential and trigonometric.

\[
y = f(x) + \epsilon
\]

Other types of relationships are also possible. Try various constrained mathematical relations: circle, ellipse and diamond.

\[
f(x, y) + \epsilon = c
\]

What happens as the noise \( \epsilon \) increases?

22. Consider an experiment involving the roll of two dice. Let \( x \) indicate the value of dice 1 and \( x_2 \) indicate of the value of dice 2. In order to examine dependency between random variables, define \( y = x + x_2 \). The joint pmf \( p_{x,y} \) can be recorded in a 6-by-11 matrix that can be computed from the following feasible occurrence matrix (0 \( \rightarrow \) cannot occur, 1 \( \rightarrow \) can occur), since all the non-zero probabilities are the same (equal likelihood).

```scala
// X - dice 1: 1, 2, 3, 4, 5, 6
// X2 - dice 2: 1, 2, 3, 4, 5, 6
// Y = X + X2: 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12
val nuxy = MatrixD ((6, 11), 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0,
                   0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0,
                   80)
Use methods in the Probability object to compute the joint, marginal and conditional probability distributions, as well as the joint, marginal, conditional and relative entropy, and mutual information. Explore the independence between random variables $x$ and $y$.

23. **Convolution.** The convolution operator may be applied to vectors as well as functions (including mass and density functions). Consider two vectors $c \in \mathbb{R}^m$ and $x \in \mathbb{R}^n$. Without loss of generality let $m \leq n$, then their convolution is defined as follows:

$$y = c \ast x = \begin{bmatrix} y_k \end{bmatrix} = \begin{bmatrix} \sum_{j=0}^{m-1} c_j x_{k-j} \end{bmatrix}_{k=0,m+n-2}$$  \hspace{1cm} (3.88)

Compute the ('full') convolution of $c$ and $x$.

$$y = c \ast x = [1, 2, 3, 4, 5] \ast [1, 2, 3, 4, 5, 6, 7]$$  \hspace{1cm} (3.89)

Note, there are also 'same' and 'valid' versions of convolution operators.

24. Consider a distribution with density on the interval $[0, 2]$. Let the probability density function (pdf) for this distribution be the following:

$$f_y(y) = \frac{y}{2} \quad \text{on } [0, 2]$$

(i) Draw/plot the pdf $f_y(y)$ vs. $y$ for the interval $[0, 2]$.

(ii) Determine the Cumulative Distribution Function (CDF), $F_y(y)$.

(iii) Draw/plot the CDF $F_y(y)$ vs $y$ for the interval $[0, 2]$.

(iv) Determine the expected value of the Random Variable (RV) $y$, i.e., $\mathbb{E}[y]$.

### 3.14 Further Reading

1. Probability and Mathematical Statistics [89].

2. Entropy, Relative Entropy and Mutual Information [20].
3.15 Notational Conventions

With respect to random variables, vectors and matrices, the following notational conventions shown in Table 3.1 will be used in this book.

<table>
<thead>
<tr>
<th>variable type</th>
<th>case</th>
<th>font</th>
<th>color</th>
</tr>
</thead>
<tbody>
<tr>
<td>scalar</td>
<td>lower</td>
<td>italics</td>
<td>black</td>
</tr>
<tr>
<td>vector</td>
<td>lower</td>
<td>bold</td>
<td>black</td>
</tr>
<tr>
<td>matrix</td>
<td>upper</td>
<td>italics</td>
<td>black</td>
</tr>
<tr>
<td>tensor</td>
<td>upper</td>
<td>bold</td>
<td>black</td>
</tr>
<tr>
<td>random scalar</td>
<td>lower</td>
<td>italics</td>
<td>blue</td>
</tr>
<tr>
<td>random vector</td>
<td>lower</td>
<td>bold</td>
<td>blue</td>
</tr>
</tbody>
</table>

Built on the Functional Programming features in Scala, ScalaTion support several function types:

```scala
type FunctionS2S = Double => Double // function of a scalar - Double
type FunctionV2S = VectorD => Double // function of a vector - VectorD
type FunctionV_2S = VectorD => Double // function of a vector - VectorD
type FunctionV2V = VectorD => VectorD // vector-valued function of a vector - VectorD
type FunctionV_2V = VectorD => VectorD // vector-valued function of a vector - VectorD
type FunctionM2M = MatrixD => MatrixD // matrix-valued function of a matrix - MatrixD
type FunctionM_2M = MatrixD => MatrixD // matrix-valued function of a matrix - MatrixD
```

These function types are defined in scalation.math and svalation.linalg. A scalar-valued function type ends in 'S', a vector-valued function type ends in 'V', and a matrix-valued function type ends in 'M'. Mathematically, the scalar-valued functions are denoted by a symbol, e.g., $f$.

$$S2S \text{ function } f : \mathbb{R} \rightarrow \mathbb{R}$$

$$V2S \text{ function } f : \mathbb{R}^n \rightarrow \mathbb{R}$$

$$V_2S \text{ function } f : \mathbb{R}^n \rightarrow \mathbb{R}$$

Mathematically, the vector-valued and matrix-valued functions are denoted by a bold symbol, e.g., $f$.

$$V2V \text{ function } f : \mathbb{R}^n \rightarrow \mathbb{R}^m$$

$$V_2V \text{ function } f : \mathbb{R}^n \rightarrow \mathbb{R}^m$$

$$M2M \text{ function } f : \mathbb{R}^{m \times n} \rightarrow \mathbb{R}^{p \times q}$$

$$M_2M \text{ function } f : \mathbb{R}^{m \times n} \rightarrow \mathbb{R}^{p \times q}$$
Models are about making predictions such as given certain properties of a car, predict the car’s mileage, given recent performance of a stock index fund, forecast its future value, or given a person’s credit report, classify them as either likely to repay or not likely to repay a loan. The thing that is being predicted, forecasted or classified is referred to the response/output variable, call it \( y \). In many cases, the “given something” is either captured by other input/feature variables collected into a vector, call it \( x \),

\[
y = f(x; b) + \epsilon
\]

or by previous values of \( y \). Some functional form \( f \) is chosen to map input vector \( x \) into a predicted value for response \( y \). The last term indicates the difference between actual and predicted values, i.e., the residuals \( \epsilon \). The function \( f \) is parameterized and often these parameters can be collected into a matrix \( b \).

If values for the parameter vector \( b \) are set randomly, the model is unlikely to produce accurate predictions. The model needs to be trained by collecting a dataset, i.e., several \( (m) \) instances of \((x_i, y_i)\), and optimizing the parameter vector \( b \) to minimize some loss function, such as mean squared error (mse),

\[
mse = \frac{1}{m} \|y - \hat{y}\|^2
\]

where \( y \) is the vector from all the response instances and \( \hat{y} = f(X; b) \) is the vector of predicted response values and \( X \) is the matrix formed from all the input/feature vector instances.

### Estimation Procedures

Although there are many types of parameter estimation procedures, this text only utilizes the three most commonly used procedures [8].

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Full Name</th>
<th>Inventor</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSE</td>
<td>Least Squares Estimation</td>
<td>Gauss</td>
</tr>
<tr>
<td>MoM</td>
<td>Method of Moments</td>
<td>Pearson</td>
</tr>
<tr>
<td>MLE</td>
<td>Maximum Likelihood Estimation</td>
<td>Fisher</td>
</tr>
</tbody>
</table>

The method of moments develops equations the relate the moments of a distribution to the parameters of the model, in order to create estimates for the parameters. Least Squares Estimation takes the sum of squared errors and sets the parameter values to minimize this sum. It has three main varieties: Ordinary Least Squares (OLS), Weighted Least Squares (WLS), and Generalized Least Squares (GLS). Finally, Maximum Likelihood Estimation sets the parameter values so that the observed data is likely to occur. The easiest way to think about this is to imagine that one wants to create a generative model (a model that generates data). One would want to set the parameters of the model so it generates data that looks like the given dataset.

Setting of parameters is done by solving a system of equations for the simpler models, or by using an optimization algorithm for more complex models.
Quality of Fit (QoF)

After a model is trained, its Quality of Fit (QoF) should be evaluated. One way to perform the evaluation is to train the model on the full dataset and test as well on the full dataset. For complex models with many parameters, overfitting will likely occur. Then its excellent evaluation is unlikely to be reproduced when the model is applied in the real-world. To avoid overly optimistic evaluations due to overfitting, it is common to divide a dataset \((X, y)\) into a training dataset and testing dataset where training is conducted on the training dataset \((X_r, y_r)\) and evaluation is done on the test dataset \((X_e, y_e)\). The conventions used in this book for the full, training and test datasets are shown in Table 3.3.

<table>
<thead>
<tr>
<th>Math Symbol</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(X)</td>
<td>x</td>
<td>full data/input matrix</td>
</tr>
<tr>
<td>(X_r)</td>
<td>x_r</td>
<td>training data/input matrix (maybe full)</td>
</tr>
<tr>
<td>(X_e)</td>
<td>x_e</td>
<td>test data/input matrix (maybe full)</td>
</tr>
<tr>
<td>(y)</td>
<td>y</td>
<td>full response/output vector</td>
</tr>
<tr>
<td>(y_r)</td>
<td>y_r</td>
<td>training response/output vector (maybe full)</td>
</tr>
<tr>
<td>(y_e)</td>
<td>y_e</td>
<td>test response/output vector (maybe full)</td>
</tr>
</tbody>
</table>

Note, when training and testing on the full dataset, the training and test dataset are actually the same, i.e., they are the full dataset. If a model has many parameters, the Quality of Fit (QoF) found from training and testing on the full dataset should be suspect. See the section on cross-validation for more details.

In ScalaTion, the Model trait serves as base trait for all the modeling techniques in the modeling package and its subpackages classifier, clusterer, fda, forecaster, and recommender.

Model Trait

Trait Methods:

```scala
trait Model {
  def modelName: String
  def train (x_: MatrixD, y_: VectorD): Unit
  def test (x_: MatrixD, y_: VectorD): (VectorD, VectorD)
  def predict (z: VectorD): Double
  def hparameter: HyperParameter
  def parameter: VectorD
  def report (ftVec: VectorD): String =
}
```

The train method will use a training or full dataset to train the model, i.e., optimize its parameter vector \(b\) to minimize a given loss function. After training, the quality of the model may be assessed using the test method. The evaluation may be performed on a test or full dataset. Finally, information about
the model may be extracted by the following three methods: (1) `hparameter` showing the hyper-parameters, (2) `parameter` showing the parameters, and (3) `report` showing the hyper-parameters, the parameter, and the Quality of Fit (QoF) of the model. Note, hyper-parameters are used by some modeling techniques to influence either the result or how the result is obtained.

Classes that implement (directly or indirectly) the `Model` trait should default `x_` and `x_e` to the full data/input matrix `x`, and `y_` and `y_e` to the full response/output vector `y` that are passed into the class constructor.

Implementations of the `train` method take a training data/input matrix `x_` and a training response/output vector `y_` and optimize the parameter vector `b` to, for example, minimize error or maximize likelihood. Implementations of the `test` method take a test data/input matrix `x_e` and the corresponding test response/output vector `y_e` to compute errors and evaluate the Quality of Fit (QoF). Note that with cross-validation (to be explained later), there will be multiple training and test datasets created from one full dataset. Implementations of the `hparameter` method simply return the hyper-parameter vector `hparam`, while implementations of the `parameter` method simply return the optimized parameter vector `b`. (The `fname_` and `technique` parameters for Regression are the feature names and the solution/optimization technique used to estimate the parameter vector, respectively.)
Chapter 4

Data Management

4.1 Introduction

Data Science relies on having large amounts of quality data. Collecting data and handling data quality issues are of utmost importance. Without support from a system or framework, this can be very time-consuming and error-prone. This chapter provides a quick overview of the support provided by ScalaTion for data management and preprocessing.

In the era of big data, a variety of database management technologies have been proposed, including those under the umbrella of Not-only-SQL (NoSQL). These technologies include the following:

- Key-value stores (e.g., ). When the purpose the data store is very rapid lookup and not advanced query capabilities, a key-value store may be ideal. They are often implemented as distributed hash tables.

- Document-oriented databases (e.g., MongoDB). These databases are intended for storage and retrieval of unstructured (e.g., text) and semi-structured (e.g., XML or JSON) data.

- Columnar databases (e.g., Vertica). Such databases are intended for structured data like traditional relational databases, but to better facilitate data compression and analytic operations. Data is stored in columns rather rows as in traditional relational databases.

- Graph databases (e.g., Neo4j). These make the implicit relationships (via foreign-key, primary-key pairs) in relational databases explicit. A tuple in a relational database is mapped to a node in a graph database, while an implicit relationship is mapped to edge in a graph database. The database then consists of a collection directed graphs, each consisting of nodes and edges connecting the nodes. These database are particularly suited to social networks.

The purpose of these database technologies is to provide enhanced performance over traditional, row-oriented relational database and each of the above are best suited to particular types of data.

Data management capabilities provided by ScalaTion include Columnar Databases and Graph Databases. Both include extensions making them suitable as a Time Series DataBase (TSDB).

Preprocessing of data should be done before applying analytics techniques to ensure they are working on quality data. ScalaTion provides a variety of preprocessing techniques.
4.1.1 Analytics Databases

In data science, it is convenient to collect data from multiple sources and store the data in a database. Analytics databases are organized to support efficient data analytics.

A database supporting data science should make it easy and efficient to view and select data to be fed into models. The structures supported by the database should make it easy to extract data to create vectors, matrices and tensors that are used by data science tools and packages.

Multiple systems, including SCALaTION’s TSDB, are built on top of columnar, main memory databases in order to provide high performance. SCALaTION’s TSDB is a Time Series DataBase that has built-in capabilities for handling time series data. It is able to store non-time series data as well. It provides two Application Programming Interfaces (APIs) for convenient access to the data [?].

4.2 Columnar Database Model

Of the NoSQL database management systems, columnar databases are closest to traditional relational databases. Rather than tuples/rows taking center stage, columns/vectors take center stage.

A columnar database is made up of the following components:

- **Element** - a value from a given Domain or Datatype (e.g., Int, Long, Double, Rational, Real, Complex, StrNum, TimeNum)
- **Column/Vector** - a collection of values from the same Datatype (e.g., forming VectorI, VectorL, VectorD, VectorQ, VectorR, VectorC, VectorS, VectorT)
- **Columnar Relation** - a heterogeneous collection of columns/vectors put into a table-like structure.
- **Columnar Database** - a collection of columnar relations.

Table 4.1 shows the first 10 rows (out of 392) for the well-known Auto+MPG dataset (see [https://archive.ics.uci.edu/ml/datasets/Auto+MPG](https://archive.ics.uci.edu/ml/datasets/Auto+MPG)).

Table 4.1: Example Columnar Relation: First 10 Rows of Auto+MPG Dataset

<table>
<thead>
<tr>
<th>mpg</th>
<th>cylinders</th>
<th>displacement</th>
<th>horsepower</th>
<th>weight</th>
<th>acceleration</th>
<th>model_year</th>
<th>origin</th>
<th>car_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>18.0</td>
<td>8</td>
<td>307.0</td>
<td>130.0</td>
<td>3504.0</td>
<td>12.0</td>
<td>70</td>
<td>1</td>
<td>&quot;chevrolet chevelle&quot;</td>
</tr>
<tr>
<td>15.0</td>
<td>8</td>
<td>350.0</td>
<td>165.0</td>
<td>3693.0</td>
<td>11.5</td>
<td>70</td>
<td>1</td>
<td>&quot;buick skylark 320&quot;</td>
</tr>
<tr>
<td>18.0</td>
<td>8</td>
<td>318.0</td>
<td>150.0</td>
<td>3436.0</td>
<td>11.0</td>
<td>70</td>
<td>1</td>
<td>&quot;plymouth satellite&quot;</td>
</tr>
<tr>
<td>16.0</td>
<td>8</td>
<td>304.0</td>
<td>150.0</td>
<td>3433.0</td>
<td>12.0</td>
<td>70</td>
<td>1</td>
<td>&quot;amc rebel sst&quot;</td>
</tr>
<tr>
<td>17.0</td>
<td>8</td>
<td>302.0</td>
<td>140.0</td>
<td>3449.0</td>
<td>10.5</td>
<td>70</td>
<td>1</td>
<td>&quot;ford torino&quot;</td>
</tr>
<tr>
<td>15.0</td>
<td>8</td>
<td>429.0</td>
<td>198.0</td>
<td>4341.0</td>
<td>10.0</td>
<td>70</td>
<td>1</td>
<td>&quot;ford galaxy 500&quot;</td>
</tr>
<tr>
<td>14.0</td>
<td>8</td>
<td>454.0</td>
<td>220.0</td>
<td>4354.0</td>
<td>9.0</td>
<td>70</td>
<td>1</td>
<td>&quot;chevrolet impala&quot;</td>
</tr>
<tr>
<td>14.0</td>
<td>8</td>
<td>440.0</td>
<td>215.0</td>
<td>4312.0</td>
<td>8.5</td>
<td>70</td>
<td>1</td>
<td>&quot;plymouth fury iii&quot;</td>
</tr>
<tr>
<td>14.0</td>
<td>8</td>
<td>455.0</td>
<td>225.0</td>
<td>4425.0</td>
<td>10.0</td>
<td>70</td>
<td>1</td>
<td>&quot;pontiac catalina&quot;</td>
</tr>
<tr>
<td>15.0</td>
<td>8</td>
<td>390.0</td>
<td>190.0</td>
<td>3850.0</td>
<td>8.5</td>
<td>70</td>
<td>1</td>
<td>&quot;amc ambassador dpl&quot;</td>
</tr>
</tbody>
</table>

88
Since each column is stored as a vector, they can be readily compressed. Due to the high repetition in the **cylinders** column it can be effectively compressed using Run Length Encoding (RLE) compression. In addition, a column can be efficiently extracted since it already stored as a vector in the database. These vectors can be used in aggregate operators or passed into analytic models.

Data files in various formats (e.g., comma separated values (csv)) can be loaded into the database.

```scala
val auto_mpg = Relation("auto_mpg", "auto_mpg.csv")
```

It is easy to create a Multiple Linear Regression model for this dataset. Simply pick the response column, in this case **mpg** and the predictor columns, in this case all other columns besides **car.name**. The connection between **car.name** and **mpg** is coincidental. The response column/variable goes into a vector.

```scala
val y = auto_mpg.toVectorD (0)
```

The predictor columns/variables goes into a matrix.

```scala
val x = auto_mpg.toMatrixD (1 to 7))
```

Then the matrix `x` and vector `y` can be passed into a `Regression` model constructor.

```scala
val rg = new Regression (x, y)
```

See the next chapter for how to train a model, evaluate the quality of fit and make predictions.
4.3 Columnar Relational Algebra API

The first API is a *Columnar Relational Algebra* that includes the standard operators of relational algebra plus those common to column-oriented databases. It consists of the `Table` trait and two implementing classes: `Relation` and `MM_Relation`. Persistence for `Relation` is provided by the `save` method, while `MM_Relation` utilizes memory-mapped files.

4.3.1 Relation Creation

A `Relation` object is created by invoking a constructor or factory apply function. For example, the following six `Relations` may be useful in a traffic forecasting study.

```scala
val sensor = Relation("sensor",
    Seq("sensorId", "model", "latitude", "longitude", "roadId"),
    Seq (), 0, "ISDDI")
val road = Relation("road",
    Seq("roadId", "rdName", "lat1", "long1", "lat2", "long2"),
    Seq (), 0, "ISDDDD")
val mroad = Relation("mroad",
    Seq("roadId", "rdName", "lanes", "lat1", "long1", "lat2", "long2"),
    Seq (), 0, "ISIDDDD")
val traffic = Relation("traffic",
    Seq("time", "sensorId", "count" "speed"),
    Seq (), Seq (0, 1), "TIID")
val wsensor = Relation("wsensor",
    Seq("sensorId", "model", "latitude", "longitude"),
    Seq (), 0, "ISDD")
val weather = Relation("weather",
    Seq("time", "sensorId", "precipitation" "wind"),
    Seq (), Seq (0, 1), "TIDD")
```

The name of the first relation is “sensor” and it stores information about traffic sensors.

- The first argument is the name of the relation (`name`).
- The second argument is the sequence of attribute/column names (`colName`).
- The third argument is the sequence of data, currently empty (`col`).
- The fourth argument is the column number for the primary key (`key`).
- The fifth argument, “ISDDI”, indicates the domains (`domain`) for the attributes (`Integer`, `String`, `Double`, `Double`, `Integer`).
- The sixth and optional argument can be used to define foreign keys (`fKeys`).
- The seventh and optional argument indicates whether to `enter` that relation is the system `Catalog`.  

90
The second relation `road` stores the Id, name, beginning and ending latitude-longitude coordinates.
The third relation `mroad` is for multi-lane roads.
The fourth relation `traffic` stores the data collected from traffic sensors. The primary key in this case is composite, Seq (0, 1), as both the time and the sensorId are required for unique identification.
The fifth relation `wsensor` stores information about weather sensors.
Finally, the sixth relation `weather` stores data collected from the weather sensors.

### 4.3.2 Relation Population

There are several ways to populate the `Relations`. A row/tuple can be added one at a time using `def add (tuple: Row)`. Population may also occur during relation construction (via a constructor or apply method). There are factory apply functions that take a file or URL as input.

For example to populate the `sensor` relation with information about Austin, Texas’ traffic sensors stored in the file `austin_traffic_sensors.csv` the following line of code may be used.

```scala
val sensor = Relation ("sensor", "austin_traffic_sensors.csv")
```

Data files are stored in subdirectories of SCALATION’s `data` directory.

### 4.3.3 Columnar Relational Algebra Operators

Table 4.2 shows the thirteen operators supported (the first six are considered fundamental). Operator names as well as Unicode symbols may be used interchangeably (e.g., `r union s` or `r ∪ s` compute the union of relations `r` and `s`). Note, the extended projection operator `eproject` (Π) provides a convenient mechanism for applying aggregate functions. It is often called after the `groupBy` operator, in which case multiple rows will be returned. Multiple columns may be specified in `eproject` as well. There are also several varieties of `join` operators. As an alternative to using the Unicode symbol when they are Greek letters, the letter may be written out in English (`pi`, `sigma`, `rho`, `gamma`, `epi`, `omega`, `zeta`, `unzeta`).

The subsections below present the columnar relational algebra operators, first showing the textbook notation followed by the syntax in SCALATION’s `column_db` package. To make the examples complex more concise, let `r = road`, `s = sensor`, `t = traffic`, `q = mroad`, `v = wsensor` and `w = weather`.

**Select Operator**

The `select` operator will return the rows that match the predicate, in this case `rdName == “I285”`.

\[
\sigma_{\text{rdName}==”I285”}(r)
\]

\[
r.\sigma(\text{rdName}, \_ == “I285”)
\]

**Project Operator**

The `project` operator will return the specified columns, in this case `rdName, lat1, long1`.

\[
\pi_{\text{rdName}, \text{lat1}, \text{long1}}(r)
\]
Union Operator

The \textit{union} operator will return the rows from \( r \) and \( s \) with no duplicates. For this operator the textbook syntax and \texttt{column.db} syntax are identical.

\[ r \cup s \]

Minus Operator

The \textit{minus} operator will return the rows from \( r \) that are not in \( s \). For this operator the textbook syntax and \texttt{column.db} syntax are identical.

\[ r - s \]

Cartesian Product Operator

The \textit{product} operator will return all combinations of rows in \( r \) with rows in \( s \). For this operator the textbook syntax and \texttt{column.db} syntax are identical.

\[ r \times s \]

Rename Operator

The \textit{rename} operator renames relation \( r \)'s name to \( r.2 \).

\[ r.\rho(“r.2”) \]

The above six operators form the fundamental operators for \textsc{SacalaTion}'s \texttt{column.db} package and are shown as the first group in Table \ref{tab:operators}.

Join Operators

In order to combine information from two relations, \textit{join} operators are preferred over products, as they are much more efficiently and only combine related rows. \textsc{SacalaTion}'s \texttt{column.db} package supports natural-join, equi-join, general theta join, left outer join, and right outer join, as shown below.

\[
\begin{align*}
  r & \bowtie s & \text{natural – join} \\
  r & \bowtie (“roadId”, “roadId”, s) & \text{equi – join} \\
  r & \bowtie [Int](s, (“roadId”, “roadId”, \_ == \_)) & \text{theta join} \\
  t & \bowtie (“time”, “time”, w) & \text{left outer join} \\
  t & \bowtie (“time”, “time”, w) & \text{right outer join}
\end{align*}
\]
Intersect Operator

The intersect operator will return the rows in $r$ that are also in $s$. For this operator the textbook syntax and column_db syntax are identical.

\[ r \cap s \]

GroupBy Operator

The groupBy operator forms groups among the relation based on the equality of attributes. The following example groups traffic data based in the value of the "sensorId" attribute.

\[ t.\gamma(\text{"sensorId"}) \]

Extended Projection Operator

The extended projection operator eproject applies aggregate operators on aggregation columns (first arguments) and regular project on the other columns (second arguments). Typically it is called after the groupBy operator.

\[ t.\gamma(\text{"sensorId"}).\Pi(\text{avg}, \text{"aount"}, \text{"count"})(\text{"sensorId"}) \]

OrderBy Operator

The orderBy operator effectively puts the rows into ascending (descending) order based on the given attributes.

\[ t.\omega(\text{"sensorId"}) \]
Compress Operator

The `compress` operator will compress the given columns of the relation.

\[ t.\zeta(\text{"count"}) \]

Uncompress Operator

The `uncompress` operator will uncompress the given columns of the relation.

\[ t.Z(\text{"count"}) \]

4.3.4 Example Queries

Several example queries for the traffic study are given below.

1. Retrieve the automobile mileage data for cars with 8 cylinders.

   ```scala
   auto_mpg.select ("cylinders", _ == 8)
   ```

   Note, select and \( \sigma \) may be use interchangeably

2. Retrieve the automobile mileage data for cars with 8 cylinders, returning the car name and mpg.

   ```scala
   auto_mpg.select ("cylinders", _ == 8).project ("car_name", "mpg")
   ```

   Note, project and \( \pi \) may be use interchangeably

3. Retrieve traffic data within a 100 kilometer-grid from the center of Austin, Texas. The latitude-longitude coordinates for Austin, Texas are (30.266667, -97.733333).

   ```scala
   val austin = latLong2UTMxy (LatitudeLongitude (30.266667, -97.733333))
   val alat = (austin._1 - 100000, austin._1 + 100000)
   val along = (austin._2 - 100000, austin._2 + 100000)
   traffic ⊙◁ sensor.σ [Double] ("latitude", _ ∈ alat).σ [Double] ("longitude" _ ∈ along)
   ```

Relation Class

Class Methods:

- **@param name** the name of the relation
- **@param colName** the names of columns
- **@param col** the Scala Vector of columns making up the columnar relation
- **@param key** the column number for the primary key (< 0 => no primary key)
- **@param domain** an optional string indicating domains for columns (e.g., 'SD' = 'StrNum', 'Double')
- **@param fKeys** an optional sequence of foreign keys
  - Seq (column name, ref table name, ref column position)
- **@param enter** whether to enter the newly created relation into the 'Catalog'
class Relation (val name: String, val colName: Seq[String], var col: Vector[Vec] = null,
val key: Int = 0, val domain: String = null,
var fKeys: Seq[(String, String, Int)] = null, enter: Boolean = true)
extends Table with Error with Serializable
4.4 SQL-Like API for Columnar Database

The SQL-Like API in SCALATION provides many of the language constructs of SQL in a functional style.

4.4.1 Relation Creation

A RelationSQL object is created by invoking a constructor or factory apply function. For example, the following six RelationSQLs may be useful in a traffic forecasting study.

```scala
val sensor = RelationSQL("sensor",
    Seq("sensorId", "model", "latitude", "longitude", "roadId"),
    null, 0, "ISDDI")
val road = RelationSQL("road",
    Seq("roadId", "rdName", "lat1", "long1", "lat2", "long2"),
    null, 0, "ISDDDD")
val mroad = RelationSQL("mroad",
    Seq("roadId", "rdName", "lanes", "lat1", "long1", "lat2", "long2"),
    null, 0, "ISIDDDD")
val traffic = RelationSQL("traffic",
    Seq("time", "sensorId", "count", "speed"),
    null, 0, "TIID")
val wsensor = RelationSQL("wsensor",
    Seq("sensorId", "model", "latitude", "longitude"),
    null, 0, "ISDD")
val weather = RelationSQL("weather",
    Seq("time", "sensorId", "precipitation", "wind"),
    null, 0, "TIDD")
```

4.4.2 Sample Queries

The SCALATION columnar database provides a functional SQL-like query language.

1. Retrieve the vehicle traffic counts over time from all sensors on the road with Id = 101.

   ```scala
   (traffic join sensor).where[Int] ("roadId", _ == 101)
   .select ("sensorId", "time", "count")
   ```

   In SQL, this would be written as follows:

   ```sql
   select sensorId, time, count
   from traffic natural join sensor
   where roadId == 101
   ```

2. Retrieve the vehicle traffic counts averaged over time from all sensors on the road with Id = 101.

   ```scala
   (traffic join sensor).where[Int] ("roadId", _ == 101)
   .groupBy("sensorId")
   .eselect ((avg, "acount", "count"))("sensorId")
   ```
RelationSQL Class

Class Methods:

@param name the name of the relation
@param colName the names of columns
@param col the Scala Vector of columns making up the columnar relation
@param key the column number for the primary key (< 0 => no primary key)
@param domain an optional string indicating domains for columns (e.g., 'SD' = 'StrNum', 'Double')
@param fKeys an optional sequence of foreign keys - Seq (column name, ref table name, ref column position)

class RelationSQL (name: String, colName: Seq[String], col: Vector[Vec],
key: Int = 0, domain: String = null, fKeys: Seq[(String, String, Int)] = null)

extends Tabular with Serializable

def repr: Relation = r
def this (r: Relation) = this (r.name, r.colName, r.col, r.key, r.domain, r.fKeys)
def select (cName: String*): RelationSQL =
def eselect (aggCol: AggColumn*)(cName: String*): RelationSQL =
def join (r2: RelationSQL): RelationSQL =
def join (cName1: String, cName2: String, r2: RelationSQL): RelationSQL =
def join (cName1: Seq[String], cName2: Seq[String], r2: RelationSQL): RelationSQL =
def where [T: ClassTag] (cName: String, p: T => Boolean): RelationSQL =
def where2 [T: ClassTag] (p: Predicate[T]*): RelationSQL =
def groupBy (cName: String*): RelationSQL =
def orderBy (cName: String*): RelationSQL =
def reverseOrderBy (cName: String*): RelationSQL =
def union (r2: RelationSQL): RelationSQL =
def intersect (r2: RelationSQL): RelationSQL =
def intersect2 (r2: RelationSQL): RelationSQL =
def minus (r2: RelationSQL): RelationSQL =
def minus2 (r2: RelationSQL): RelationSQL =
def stack (cName1: String, cName2: String): RelationSQL =
def insert (rows: Row*)
def materialize ()
def exists: Boolean = r.exists

def toMatriD (colPos: Seq[Int], kind: MatrixKind = DENSE): MatriD =
def toMatriDD (colPos: Seq[Int], colPosV: Int, kind: MatrixKind = DENSE): (MatriD, VectorD) =
def toMatriDI (colPos: Seq[Int], colPosV: Int, kind: MatrixKind = DENSE): (MatriD, VectorI) =
def toMatriI (colPos: Seq[Int], kind: MatrixKind = DENSE): MatriI =
def toMatriII (colPos: Seq[Int], colPosV: Int, kind: MatrixKind = DENSE): (MatriI, VectorI) =
def toVectorC (colPos: Int = 0): VectorC = r.toVectorC (colPos)
def toVectorC (colName: String): VectorC = r.toVectorC (colName)
def toVectorD (colPos: Int = 0): VectorD = r.toVectorD (colPos)
def toVectorD (colName: String): VectorD = r.toVectorD (colName)
def toVectorI (colPos: Int = 0): VectorI = r.toVectorI (colPos)
def toVectorI (colName: String): VectorI = r.toVectorI (colName)
def toVectorL (colPos: Int = 0): VectorL = r.toVectorL (colPos)
def toVectorL (colName: String): VectorL = r.toVectorL (colName)
def toVectorQ (colPos: Int = 0): VectorQ = r.toVectorQ (colPos)
def toVectorQ (colName: String): VectorQ = r.toVectorQ (colName)
def toVectorR (colPos: Int = 0): VectorR = r.toVectorR (colPos)
def toVectorR (colName: String): VectorR = r.toVectorR (colName)
def toVectorS (colPos: Int = 0): VectorS = r.toVectorS (colPos)
def toVectorS (colName: String): VectorS = r.toVectorS (colName)
def toVectorT (colPos: Int = 0): VectorT = r.toVectorT (colPos)
def toVectorT (colName: String): VectorT = r.toVectorT (colName)
def show (limit: Int = Int.MaxValue) { r.show (limit) }
def save () { r.save () }
def generateIndex (reset: Boolean = false) { r.generateIndex (reset) }
4.5 Exercises

1. Create the sensor schema using the RelationSQL class in the columnar_db package.

2. Populate the sensor database with sample data. See 
   https://data.austintexas.gov/Transportation-and-Mobility/Traffic-Count-Study-Area/cqdh-farx

3. Retrieve the sensors that are on I35.

4. Retrieve traffic data within a 100 kilometer-grid from the center of Austin, Texas. The latitude-longitude coordinates for Austin, Texas are (30.266667, -97.733333).
4.6 Graph Databases

Graph databases enhance traditional, row-oriented relational databases by making implicit relationships where a foreign key references a primary, explicit. For example, in the sensor relation rather having a foreign key roadId that references the primary key roadId in the road relation, the relationship is made explicit via an edge type.

Table 4.3 shows the correspondence between concepts in the relational database model and the graph database model.

Table 4.3: Mapping from Relational Database Concept to Graph Database Concept

<table>
<thead>
<tr>
<th>Relational</th>
<th>Graph</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Domain</td>
<td>Domain</td>
<td>a set of values or datatype</td>
</tr>
<tr>
<td>Value</td>
<td>Value</td>
<td>an individual/atomic data value</td>
</tr>
<tr>
<td>Attribute</td>
<td>Property</td>
<td>named components in a vertex/edge</td>
</tr>
<tr>
<td>Tuple</td>
<td>Vertex</td>
<td>aggregation of values related to an entity</td>
</tr>
<tr>
<td>Relation</td>
<td>Vertex Type</td>
<td>a collection of vertices with the same type</td>
</tr>
<tr>
<td>Primary Key</td>
<td>Primary Key</td>
<td>unique combination of attributes used a the main identifier</td>
</tr>
<tr>
<td>Foreign Key</td>
<td>Edge</td>
<td>a directed edge links a source vertex ( u ) to a target vertex ( v )</td>
</tr>
<tr>
<td></td>
<td>Edge Type</td>
<td>a collection of edges with the same type</td>
</tr>
<tr>
<td></td>
<td>PGraph</td>
<td>a collection of vertex types and edge types</td>
</tr>
<tr>
<td>Database</td>
<td>Database</td>
<td>a collection of graphs</td>
</tr>
</tbody>
</table>

The notion that all the vertices in a vertex type have the same type means that their properties have the same names and domains. This also implies that they will have the same arity. Furthermore, the notion that all edges in an edge type have the type means that their properties have the same names and domains as well as they connect vertices from the same vertex types. For example, edges in the takes edge type connect vertices from the student vertex type to the course vertex type.

The subsections below define several types of graphs that are useful for graph analytics and graph databases. Such graphs need information content that is provided by labels or properties. In some cases, the labels or properties may only be associated with vertices/nodes, but more generally, they are provided for both vertices and edges. Labels associate a single value, while properties associate multiple values. For graph analytics and some types of graph databases, labels may suffice, but properties are usually required for robust database applications.
4.7 Directed Graphs

The starting point for graph databases is the basic definition of a directed graph from *graph theory*. A *directed graph* consists of vertices (nodes) connected via one-way (directed) edges. More formally, it is a two-tuple $G(V, E)$ where

- $V$ = set of vertices/nodes
- $E \subseteq V \times V$ set of directed edges

The edge from vertex/node $u$ to vertex/node $v$ is an ordered pair $(u, v)$, also denoted as $u \rightarrow v$, or more concisely $uv$.

The definition of a graph captures the topological structure, but needs to be augmented to attach information. The simplest way to do this is to just add labels to nodes. A *vertex-labeled directed graph* is a four-tuple $G(V, E, L, l)$ where

- $V$ = set of vertices/nodes
- $E \subseteq V \times V$
- $L$ = set of labels
- $l : V \rightarrow L$ is a labelling function

Notice that if a graph has $n_v$ vertices/nodes, the number of edges may range from 0 to $n_v^2$. There are two general ways to represent the connectivity structure of a graph, an *adjacency matrix* and an *adjacency list*. When the graph is sparse, i.e., $n_e \ll n_v^2$, then an adjacency list will be more efficient.

Under this approach, a vertex/node needs to keep track of its outgoing edges (also known as children). The *Graph* class below has a *name* and a flag *inverse* indicating whether to also store the incoming edges (parents). The rest of the arguments to the constructor are Arrays, where each element of the array corresponds to a node in the graph. These arrays record the children $ch$, the node *label* and the vertex id $vid$. The children of a node are a set of integers. For example, node 1 may have children \{2, 4, 5\}.

Consider the graph shown in Figure 5.1. If Bob and Sue know each other, then the graph consists of 2 nodes and 2 edges.

![Figure 4.1: Example Directed Graph](image)

Graph Class

| Class Methods: | 101 |
The TLabel defined in the LabelType object indicates a basic datatype for all labels, e.g., Int, Double, or String using Scala type definition.

    type TLabel = String
4.8 Directed Multi-Graphs

Adding edge labels introduces a fundamental change to the graph. No longer is an edge uniquely identified by its source and target nodes. So long as the edge labels are different, multiple edges may connect the same source and target node. For example, if Bob and Sue know each other and Sue employs Bob, then the graph consists of 2 nodes and 3 edges as shown in Figure 4.2.

![Figure 4.2: Example Directed Multi-Graph](image)

Providing both edge and node labels allows rich information content to stored. A fully labeled directed multi-graph is a four-tuple $G(V, E, L, l)$ where

- $V =$ set of vertices/nodes
- $E \subseteq V \times L \times V \{ \text{source-node, edge-label, target-node} \}$
- $L =$ set of labels
- $l : V \rightarrow L$ is a node labeling function

Of course, it may also be useful to split the set of labels $L$ into $L_v =$ set of vertex/node labels and $L_e =$ set of edge labels.

The edge is no longer an ordered pair, as it becomes a triple where the 3 parts are known by various names as shown in Table 4.4

<table>
<thead>
<tr>
<th>$(v, l, u)$</th>
<th>source-node</th>
<th>edge-label</th>
<th>target-node</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(h, r, t)$</td>
<td>head</td>
<td>relation</td>
<td>tail</td>
</tr>
<tr>
<td>$(s, p, o)$</td>
<td>subject</td>
<td>predicate</td>
<td>object</td>
</tr>
</tbody>
</table>

Such a structure provides the basis for Resource Description Framework (RDF) graphs.

The MuGraph class below extends the Graph class, mainly adding edge labels elabel. The edge labels are stored in a Map for looking up the label(s) based on the source and target nodes. Notice that it can be multivalued. An optional schema specification is also provided to introduce a type structure. (This will not be discussed further here.)
MuGraph Class

Class Methods:

- @param ch the array of child (adjacency) vertex sets (outgoing edges)
- @param label the array of vertex labels: v -> vertex label
- @param elabel the map of edge labels: (u, v) -> edge label
- @param inverse whether to store inverse adjacency sets (parents)
- @param name the name of the multi-digraph
- @param id the array of vertex id's
- @param schema optional schema: map from label to label type

```java
class MuGraph [TLabel: ClassTag] (ch: Array[SET[Int]],
  label: Array[TLabel],
  elabel: Map[Pair, SET[TLabel]],
  inverse: Boolean = false,
  name: String = "g",
  id: Array[Int] = Array(),
  schema: Array[String] = Array())
  extends Graph[TLabel] (ch, label, inverse, name) with Cloneable
```

Example: Create a MuGraph

```java
val links = new MuGraph[String] (Array (SET (1), SET (0), SET (0, 1)),
  Array ("Bob", "Sue", "Joe"),
  Map ((0, 1) -> SET ("knows"),
    (1, 0) -> SET ("employs", "knows"),
    (2, 0) -> SET ("knows"),
    (2, 1) -> SET ("knows")),
  false,
  "links")
```

Draw the directed multi-graph called links.
4.9 Property Graphs

Property graphs enhance plain labeled directed multi-graphs by replacing labels with a list/map of properties. (Note some database models and systems keep labels and assign them a special purpose.) As such, information may be flexibly added to a property graph. The structure and type of a property graph is defined by specifying its structural organization (analogously to specifying relational schema).

4.9.1 Structure of a Property Graph

The following structural definitions specify datatypes and structural organization for property graphs as defined in the scala-graph package. In order to support rapid access to into collections VEC is used. Currently, the fastest such Scala 3 structure for immutable collections is Vector, while for mutable collections it is ArrayBuffer.

// import scala.collection.immutable.{Vector => VEC}
import scala.collection.mutable.{ArrayBuffer => VEC}

ScalaTion’s graph database supports property graphs and is organized as follows: The primary concepts are vertex and edge. Similar vertices and edges are collected into vertex types and edge types. A property graph consists of multiple vertex types and edge types. Element and element type are introduced as generalizations that are useful in defining graph algebra operators.

- The ValueType type is a Scala 3 union type for atomic database values.

  type ValueType = Int | Long | Double | String

- The Property type corresponds to the notion of an attribute in a relational database. It maps property names to property values (e.g., Map ("name" -> "Bob", "salary" -> 85000.0).

  type Property = Map [String, ValueType]

An example of a property is shown below.

  val knows: Property = Map ("type" -> "knows")

- The Element class serves as an abstract base class for both vertices and edges.

  abstract class Element (val prop: Property) extends Serializable

- The ElementType class serves as an abstract base class for both vertex types and edge types.

  abstract class ElementType (val schema: VEC [String]) extends Serializable

- The Vertex class specifies the form of vertices/nodes in the property graph. A vertex maintains properties for a vertex, e.g., a person. It is analogous to a tuple in a relational database.
class Vertex (prop: Property)
    extends Element (prop) with Serializable:

For example, three vertices can easily be created.

    val v0 = Vertex (Map ("name" -> "Bob", "salary" -> 85000.0))
    val v1 = Vertex (Map ("name" -> "Sue", "salary" -> 95000.0))
    val v2 = Vertex (Map ("name" -> "Joe", "salary" -> 99000.0))

• The **VertexType** class corresponds to the notion of an entity type in an Entity-Relationship Model. A vertex type collects vertices of the same type, e.g., a person vertex type. Its schema specification determines the types of properties that vertices in this collection are allowed to have. It is analogous to a relation with no foreign keys in a relational database.

    class VertexType (schema: VEC [String], val vs: VEC [Vertex])
        extends ElementType (schema) with Serializable:

The three vertices defined above may be collected into a vertex type.

    val vt0 = VertexType (VEC ("name", "salary"), VEC (v0, v1, v2))

The information content in a vertex type can be shown in a tabular format.

<table>
<thead>
<tr>
<th>vertex</th>
<th>name</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>v0</td>
<td>&quot;Bob&quot;</td>
<td>85000.0</td>
</tr>
<tr>
<td>v1</td>
<td>&quot;Sue&quot;</td>
<td>95000.0</td>
</tr>
<tr>
<td>v2</td>
<td>&quot;Joe&quot;</td>
<td>99000.0</td>
</tr>
</tbody>
</table>

• The **Edge** class allows explicit relationships to be formed between vertices. An edge connects a source vertex to a target vertex and may have its own properties. It may be thought of a triple (from, prop, to). It is roughly analogous to an implicit relationship manifest via foreign key-primary key pairs in a relational database.

    class Edge (val from: Vertex, prop: Property, val to: Vertex)
        extends Element (prop) with Serializable

For example, three edges can easily be created.

    val e0 = Edge (v0, knows, v1)
    val e1 = Edge (v1, knows, v0)
    val e2 = Edge (v2, knows, v0)
The **EdgeType** class corresponds to the notion of a relationship type in an Entity-Relationship Model. An edge type collects edges of the same type and has a source **VertexType** and a target **VertexType**. Its schema specification determines the type of properties that edges in this collection are allowed to have. An edge type is analogous to a relation with foreign keys in a relational database.

```scala
class EdgeType (val from: VertexType, schema: VEC [String], val to: VertexType, val es: VEC [Edge])
  extends ElementType (schema) with Serializable:
```

The three edges defined above may be collected into an edge type.

```scala
val et0 = EdgeType (vt0, VEC ("type"), vt0, VEC (e0, e1, e2))
```

The information content in an edge type can be shown in a tabular format.

<table>
<thead>
<tr>
<th>edge</th>
<th>from-vertex</th>
<th>type</th>
<th>to-vertex</th>
</tr>
</thead>
<tbody>
<tr>
<td>e0</td>
<td>v0</td>
<td>&quot;knows&quot;</td>
<td>v1</td>
</tr>
<tr>
<td>e1</td>
<td>v1</td>
<td>&quot;knows&quot;</td>
<td>v0</td>
</tr>
<tr>
<td>e2</td>
<td>v2</td>
<td>&quot;knows&quot;</td>
<td>v0</td>
</tr>
</tbody>
</table>

The **PGraph** class is used to store property graphs. A property graph has a name, zero or more **VertexTypes** and zero or more **EdgeTypes**. Each of the **EdgeTypes** can only reference **VertexTypes** specified in this **PGraph**.

```scala
class PGraph (val name: String, val vt: VEC [VertexType], val et: VEC [EdgeType])
  extends Serializable:
```

A **PGraph** with no **EdgeTypes** essentially allows a similar organization to a relational database. Simply add a property that acts as a foreign key and use a join operation.

Example: Create a **PGraph**

```scala
val knows: Property = Map ("type" -> "knows")
val employs: Property = Map ("type" -> "employs")

val v0 = Vertex (Map ("name" -> "Bob", "salary" -> 85000.0))
val v1 = Vertex (Map ("name" -> "Sue", "salary" -> 95000.0))
val v2 = Vertex (Map ("name" -> "Joe", "salary" -> 99000.0))
```
val vt0 = VertexType (VEC ("name", "salary"), VEC (v0, v1, v2))
println (s"check schema = ${vt0.check}")

val e0 = Edge (v0, knows, v1)
val e1 = Edge (v1, knows, v0)
val e2 = Edge (v2, knows, v0)
val et0 = EdgeType (vt0, VEC ("type"), vt0, VEC (e0, e1, e2))
println (s"check schema = ${et0.check}")

val e3 = Edge (v1, employs, v0)
val e4 = Edge (v2, employs, v1)
val et1 = EdgeType (vt0, VEC ("type"), vt0, VEC (e3, e4))
println (s"check schema = ${et1.check}")

val g = PGraph ("links", VEC (vt0), VEC (et0, et1))
g.vmap += "person" -> vt0
g.emap += "knows" -> et0
g.emap += "employs" -> et1
println (s"g = $g")

As an exercise, draw the property graph called links.

4.9.2 Native Storage

Achieving the performance potential for graph databases requires appropriate data structures and organizations, whether in main memory or in files. It is common for graph database engines to support index-free adjacency, although various hybrids approaches are also possible [?].

Index-Free Adjacency

Notice the basic structure for property graphs, makes it easy to find vertices connected to an edge. Some types of navigation will require finding edges that a vertex is connected to. Suppose there are vertex types student and course connected with edge type takes. The meta-graph for this is shown in the next subsection. Suppose a student named "Bob" wishes to know the courses he is taking. One would like to traverse from the "Bob" vertex to his course vertices. In general this will not work, since some queries may depend upon an edge property (e.g., the grade). In order to handle such cases, the "Bob" vertex should reference its outgoing edges in the takes edge type. Unfortunately, "Bob" may have taken many courses (i.e., the takes edge/relationship type is many-to-many). A storage solution is to have the vertex reference the first edge in the "Bob" group of courses and have this edge reference the next edge in the group, etc. The chain of edges may be organized as either a singly or doubly linked list, however, to make edge deletions efficient the list should be doubly linked. This is the approach utilized by Neo4j’s native graph storage engine, as described in Chapter 6 of [82].

To illustrate details about the storage, example vertex types and edge types will be shown for a simple course-enrollment database.
Meta-Graph for Course Enrollment Database

Property Graphs are rich enough to depict their own conceptual model. A meta-graph shows the metadata for a graph database. The meta-graph in Figure ?? shows three vertex types, student, course and professor. These are connected via the takes and taughtBy edge types. The takes edge/relationship type is many-to-many, while the taughtBy edge/relationship type is many-to-one.

![Meta-Graph for the Course Enrollment Database](image)

The arrow on the taughtBy edge type indicates a many-to-one relationship from course to professor. That is, a course is taught by one professor, while a professor may teach many courses.

Storage for Course Enrollment Database

The course and professor vertex types as well as the taughtBy edge type are shown in Figure 4.7 in tabular format. Using the index-free adjacency approach, each course vertex maintains a reference to its first outgoing taughtBy edge. Similarly, each professor vertex maintains a reference to its first incoming taughtBy edge.

To handle the case when there are multiple edges (e.g., a professor may teach multiple courses), an edge chain is maintained in the taughtBy Edge Type. Following the edge chain for “Dr. Bill”, gives the first edge as $e_0$, with the next edge $e_1$ and then $e_2$; a null terminates the edge chain.

The edge chains are shown here as singly-linked lists for simplicity, while in practice doubly-linked lists allow for more efficient maintenance of the edge chains.
### Table 4.7: Course Enrollment Database

(a) course Vertex Type

<table>
<thead>
<tr>
<th>vertex</th>
<th>cname</th>
<th>hours</th>
<th>taughtBy</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v_0$</td>
<td>“DB”</td>
<td>4</td>
<td>$e_0$</td>
</tr>
<tr>
<td>$v_1$</td>
<td>“SE”</td>
<td>4</td>
<td>$e_1$</td>
</tr>
<tr>
<td>$v_2$</td>
<td>“OS”</td>
<td>4</td>
<td>$e_2$</td>
</tr>
<tr>
<td>$v_3$</td>
<td>“AI”</td>
<td>3</td>
<td>$e_3$</td>
</tr>
<tr>
<td>$v_4$</td>
<td>“ML”</td>
<td>3</td>
<td>$e_4$</td>
</tr>
</tbody>
</table>

(b) professor Vertex Type

<table>
<thead>
<tr>
<th>vertex</th>
<th>pname</th>
<th>rank</th>
<th>taughtBy</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v_5$</td>
<td>“Dr. Bill”</td>
<td>AssocProf</td>
<td>$e_0$</td>
</tr>
<tr>
<td>$v_6$</td>
<td>“Dr. John”</td>
<td>Professor</td>
<td>$e_3$</td>
</tr>
</tbody>
</table>

(c) taughtBy Edge Type

<table>
<thead>
<tr>
<th>edge</th>
<th>from-vertex</th>
<th>type</th>
<th>to-vertex</th>
<th>next edge</th>
</tr>
</thead>
<tbody>
<tr>
<td>$e_0$</td>
<td>$v_0$</td>
<td>“taughtBy”</td>
<td>$v_5$</td>
<td>$e_1$</td>
</tr>
<tr>
<td>$e_1$</td>
<td>$v_1$</td>
<td>“taughtBy”</td>
<td>$v_5$</td>
<td>$e_2$</td>
</tr>
<tr>
<td>$e_2$</td>
<td>$v_2$</td>
<td>“taughtBy”</td>
<td>$v_5$</td>
<td>null</td>
</tr>
<tr>
<td>$e_3$</td>
<td>$v_3$</td>
<td>“taughtBy”</td>
<td>$v_6$</td>
<td>$e_4$</td>
</tr>
<tr>
<td>$e_4$</td>
<td>$v_4$</td>
<td>“taughtBy”</td>
<td>$v_6$</td>
<td>null</td>
</tr>
</tbody>
</table>

### 4.9.3 High-Level Query Language for Graph Databases

Examples of high-level query languages for graph databases include Cypher, Gremlin and GQL.

- Group Lectures 1: High Level Query Languages for Graph Databases, September 8, 10, 14

### 4.9.4 Graph Algebra

Often high-level query languages are implemented by translating to an algebraic language that consists of basic operators, such as select, project and join. Although graph algebra is analogous to relational algebra, it is richer and more complex.

- Group Lectures 2: Graph Algebra, September 15, 17, 21

#### Operators for Vertex Types

- Project 1: See Exercise 1 - due September 28
  For the discussion below, the person vertex type will be used.

  ```scala
  val person = g.vmap("person")
  ```
The **project** operator for vertex types will project onto the properties given in the specified *subschema*.

```scala
def project (subschema: VEC [String]): VertexType =  
    VertexType (subschema,  
        for v <- vs yield Vertex (v.prop.filter (subschema contains _._1)))
```

For example, to return the vertices with the properties trimmed down to just names the following two queries may be used. The corresponding Cypher query is also shown.

```scala
val q0 = person.project (VEC ("name"))  
val q2 = person.project ("name")
```

```cypher
MATCH (p: Person) RETURN p.name
```

The **select** operator for vertex types will return the subset of vertices that satisfy the predicate *pred*.

```scala
def select (pred: Property => Boolean): VertexType =  
    VertexType (schema,  
        for v <- vs if pred (v.prop) yield v)
```

For example, to return the vertices where the name property is “Sue”, the following two queries may be used (the second one is an abbreviated form of the first one). The corresponding Cypher query is also shown.

```scala
val q2 = person.select ((p: Property) => p("name") == "Sue")  
val q3 = person.select (_("name") == "Sue")
```

```cypher
MATCH (p: Person) WHERE p.name = 'Sue' RETURN p
```

The **unionAll** operator for vertex types will return the all vertices that in the two vertex type.

```scala
def unionAll (vt2: VertexType): VertexType =  
    VertexType (schema, vs ++ vt2.vs)
```

For example, the union of *person* and *android* will return person and android vertices. The corresponding Cypher query is also shown.

```scala
val q4 = person unionAll android
```

```cypher
MATCH (p: Person) RETURN p UNION ALL MATCH (a: Andoid) RETURN a
```

An issue to address is defining union-compatibility in a more flexible manner than in relational databases.

The **union** operator for vertex types will return the vertices that are in either of two vertex types, without duplication.

```scala
def union (vt2: VertexType): VertexType =  
    VertexType (schema, (vs ++ vt2.vs).distinct)
```
For example, the union of person and android will return person and android vertices. If any two vertices have exactly the same properties, the duplicate will be removed. The corresponding Cypher query is also shown.

```cypher
val q4 = person union android
MATCH (p: Person) RETURN p UNION MATCH (a: Android) RETURN a
```

- The minus operator for vertex types will return the vertices that are in either of two vertex types. The Cypher query language does not currently provide a MINUS operation (although a work-around can be used).

```cypher
def minus (vt2: VertexType): VertexType =
    VertexType (schema, vs diff vt2.vs)
```

For example, starting with the vertices in q4 and subtracting the vertices in android.

```cypher
val q5 = q4 minus android
```

Operators for Edge Types

- Project 2: See Exercise 2 - due October 12

Operators for Property Graphs

- Project 2: See Exercise 2 - due October 12

As Cypher and Gremlin are the two major query languages for graph databases, graph algebras have been developed for each: Cypher [65, 100], Gremlin [105].

4.9.5 Query Processing in Graph Databases

With a larger and more complex graph algebra as well as excellent potential for high-performance query processing, query processing in graph databases is challenging [78].
4.10 Special Types of Graph Databases

Graph databases attempt to strike the right balance between (i) rich and flexible database models, (ii) efficient query processing, and (iii) reduction in complexity. For example, compared to relational databases, graph databases trade off (iii) for gains in (i) and (ii). Different organizational structures for graph databases will be positioned differently among these three competing goals.

– Group Lectures 4: Special Types of Graph Databases, September 29, October 1, 5

4.10.1 Embedding Relationships in Vertex Types

The implicit relationships (foreign-key, primary-key) pairs in relational databases only support many-to-one relationships (i.e., the foreign-key references a single primary-key). In order to support many-to-many relationships, relational databases require such relationships to be translated in a new association relation that includes two foreign keys. This can be viewed a disadvantage of having a database model that is too simple. On the other hand, it allows tables to be connected using a single join. If the all relationships were stored in a separate table, two joins would be required. This is analogous to what is occurring with EdgeTypes. It is possible the replace EdgeTypes by adding child references to VertexTypes.

The graph_relation package in SCALATION adds a Map that takes edge names and maps them to another vertex. As in SCALATION’s graph package, a vertex has a property Map called prop that maps property names to property values, e.g., "name" -> "Sue". In graph_relation, the Vertex class also has a edge Map called edge that maps edge names to other vertices, e.g., "taughtBy" -> professor.verts(1).

```scala
@operator prop maps vertex’s property names to property values
@operator edge maps vertex’s edge names to other vertices

case class Vertex (prop: Property, edge: Reference = emptyRef)
  extends Serializable:
```

The VertexType class has a name, a schema for its properties, and edge schema for its edge references, as well the vertices contained in it.

```scala
@operator name the name of this vertex type
@operator schema the property names for this vertex type
@operator eschema the edge names for this vertex type
@operator verts the set of vertices having this vertex type (extension)

case class VertexType (name: String, schema: VEC [String], eschema: VEC [String], verts: VEC [Vertex])
  extends Flaw ("VertexType") with Serializable:
```

A course vertex type may have a reference taughtBy to a professor vertex type. At the instance level this would indicates that for example, the Database course is taught by professor 2.

```scala
val professor = VertexType ("professor",
  VEC ("pid", "name", "phone"),
  VEC ()
  VEC (Vertex (Map ("pid" -> 1, "name" -> "Bob", "phone" -> 1234567)),
    Vertex (Map ("pid" -> 2, "name" -> "Sue", "phone" -> 2345678)),
```
The graph relation package is similar to a traditional relational model, except that foreign key values are no longer stored in a column of a table whose correspondence with the primary key value facilitates a join. Rather a direct reference to the vertex containing that primary key value is used. The trade-off is a slight increase in the complexity of the database model, for the potential of faster joins. See the exercises for a comparison of property graphs, graph relations and relations.

4.10.2 Resource Description Framework (RDF) Graphs

Various restrictions/simplifications can be added to the above organizational structure to make query processing more efficient. For example, the edge properties may be replaced with an edge label. Similar to a graph database, an RDF store holds a collection of triples of the form (subject, predicate, object) or \((s, p, o)\). With certain restrictions on RDF triples, an RDF Graph may be built upon the concept of a Vertex-Edge-Labeled Directed Multi-Graph. RDF Graphs may be viewed as a refinement as follows:

The vertices are partitioned into entity nodes and literal nodes. An entity node's label is an International Resource Identifier (IRI), while a literal node's label is a primitive value (e.g., number or string). An edge represents a triple of the form (subject, predicate, object) or \((s, p, o)\). The subject is the from-vertex, the predicate may be viewed as an edge label, and the object is the to-vertex.

Resource Description Framework Schema (RDFS) Graphs

On the flip side, various extensions can be added to the above organizational structure. For example, type hierarchies can be imposed on vertex types as well as edge types. To handle certain use cases in RDF and RDFS, more complicated graph forms are needed, including bipartite graphs, hypergraphs and Labeled Directed Multigraph with Triple Nodes (LDM-3N) \cite{70}. There are also efforts underway to provide interoperability between RDF and Property Graphs \cite{4}.

4.10.3 Knowledge Graphs

Some view knowledge graphs as RDF/RDFS graphs and their extensions such as using the Web Ontology Language (OWL) to specify type hierarchies and constraints. Another viewpoint is that they are graph databases, primarily property graphs, used to store knowledge in addition to data.

Knowledge graphs go beyond the structural issues related to basing them on RDF graphs versus property graphs, by including the following deeper issues: (1) the type of logic used to express constraints and make...
inferences, (2) methodologies for adding vertices/nodes and edges/relationships, and (3) their synergy with machine learning.
4.11 Exercises

1. Use Scala 3/Dotty to complete the implementation of `VertexType` in the `scalation.graph` package. It should provide operators like those in the Neo4j graph algebra.

2. Use Scala 3/Dotty to complete the implementation of `EdgeType` and `PGraph` in the `scalation.graph` package. Again, it should provide operators like those the Neo4j graph algebra. The exiting operators in these classes may need to renamed and/or modified.

3. Using your implementation for `PGraph` translate the road sensor schema.

4. Create the same schema using Neo4j.

5. Populate the database with sample data.

6. Using Neo4j’s Cypher Query language, retrieve the sensors that are on I35.

7. Retrieve traffic data within a 100 kilometer-grid from the center of Austin, Texas. The latitude-longitude coordinates for Austin, Texas are (30.266667, -97.733333).

8. **Project.** Compare the complexity and performance of the following three database models and four database system:
   - Properties Graphs - using Neo4j and `scalation.graph`
   - Graph Relations - using `scalation.graph.relation`
   - Relations - using MySQL

   Plot the performance of various types of queries executed with each of the four database systems. For each plot, the x-axis will be the number of tuples/vertices, while the y-axis will be the time in milliseconds (using `nanoTime`) to execute the query. Be sure to skip the timing results for the first iteration and average the next five to get reliable timing results. Note, the JIT compiler is used during the first iteration, so it tends to be slow. The performance evaluation will require a large number of tuples/vertices, so a tuple/vertex generator will be needed. Make sure to test path queries common in social networking applications.
Chapter 5

Data Preprocessing

5.1 Basic Operations

Using the SCALATION TSDB, data scientists may write queries that extract data from one or more columnar relations. These data are used to create vectors and matrices that may be passed to various analytics techniques. Before the vectors and matrices are created the data need to be preprocessed to improve data quality and transform the data into a form more suitable for analytics.

5.1.1 Remove Identifiers

Any column that is unique (e.g., a primary key) with arbitrary values should be removed before applying a modeling/analytics technique. For example, an employee ID in a Neural Network analysis to predict salary could result in a perfect fit. Upon knowing the employee ID, the salary is a known. As the ID itself (e.g., ID = 1234567) is arbitrary, such a model has little value.

5.1.2 Convert String Columns to Numeric Columns

In SCALATION, columns with strings (of type StrNum) should be converted to integers. For displaying final results, however, is often useful to convert the integers back to the original strings. The capabilities are provided by the mapToInt function in the scalation.linalgebra.Converter object.

5.1.3 Identify Missing Values

Missing Values are common in real datasets. For some datasets, a question mark character ‘?’ is used to indicate that a value is missing. In Comma Separated Value (CSV) files, repeated commas may indicate missing values, e.g., 10.1, 11.2,,9.8. If zero or negative numbers are not valid for the application, these may be used to indicate missing values.

5.2 Detect Outliers

Data points that are considered outliers may happen because of errors or highly unusual occurrences. For example, suppose a dataset records the times for members of a football team to run a 100-yard dash and
one of the recorded values is 3.2 seconds. This is an outlier. Some analytics techniques are less sensitive to outliers, e.g., $\ell_1$ Regression, while others, e.g., $\ell_2$ Regression, are more sensitive. Detection of outliers suffers from the obvious problems of being too strict (in which case good data may be thrown away) or too lenient (in which case outliers are passed to an analytics technique). One may choose to handle outliers separately, or turn them into missing values, so that both outliers and missing values may be handled together.

**SCALATION** currently provides the following techniques for outlier detection: so many standard deviation units from the mean, **DistanceOutlier**; the smallest and largest percent values, **QuantileOutlier**; and an expansion multiplier beyond the middle two quartiles, **QuartileXOutlier**. For example, the following function will turn outliers in missing values, by reassigning the outliers to `noDouble`, **SCALATION**’s indicator of a missing value of type `Double`.

```scala
DistanceOutlier.rmOutlier (traffic.column ("speed"))
```

### 5.3 Imputation Techniques

The two main ways to handle missing values are (1) throw them away, or (2) use imputation to replace them with reasonable guesses. When there is a gap in time series data, imputation may be used for short gaps, but is unlikely to be useful for long gaps. This is especially true when imputation techniques are simple. The alternative could be to use an advanced modeling technique like SARIMA for imputation, but then results of a modeling study using SARIMA are likely to be biased. Imputation implementations are based on the **Imputation** trait in the **scalation.modeling** package.

#### 5.3.1 Imputation Trait

**Trait Methods:**

```scala
trait Imputation

  def setMissVal (missVal_ : Double) { missVal = missVal_ }
  def setDist (dist_ : Int) { dist = dist_ }
  def imputeAt (x: VectoD, i: Int): Double
  def impute (x: VectoD, i: Int = 0): (Int, Double) = findMissing (x, i)
  def imputeAll (x: VectoD): VectoD =
  def impute (x: MatriD): MatriD =
  def imputeCol (c: Vec, i: Int = 0): (Int, Any) =
```

**SCALATION** currently supports the following imputation techniques:

1. **object ImputeRegression extends Imputation**: Use `SimpleRegression` on the instance index to estimate the next missing value.

2. **object ImputeForward extends Imputation**: Use the previous value and slope to estimate the next missing value.
3. **object ImputeBackward extends Imputation**: Use the subsequent value and slope to estimate the previous missing value.

4. **object ImputeMean extends Imputation**: Use the filtered mean to estimate the next missing value.

5. **object ImputeMovingAvg extends Imputation**: Use the moving average of the last 'dist' values to estimate the next missing value.

6. **object ImputeNormal extends Imputation**: Use the median of three Normally distributed, based on filtered mean and variance, random values to estimate the next missing value.

7. **object ImputeNormalWin extends Imputation**: Same as ImputeNormal except mean and variance are recomputed over a sliding window.

### 5.4 Preliminary Feature Selection

Before selecting a modeling/analytics technique, certain columns may be thrown away. Examples include columns with too many missing values or columns with near zero variance.

EXPAND

### 5.5 Align Multiple Time Series

When the data include multiple time series, there are likely to be time alignment problems. The frequency and/or phase may not be in agreement. For example, traffic count data may be recorded every 15 minutes and phased on the hour, while weather precipitation data may be collected every 30 minutes and phased to 10 minutes past the hour.

**ScalaTion** supports the following alignments techniques: (1) approximate left outer join and (2) dynamic time warping. The first operator will perform a left outer join between two relations based on their time (TimeNum) columns. Rather than the usual matching based on equality, approximately equal times are considered sufficient for alignment. For example, to align traffic data with the weather data, the following approximate left outer join may be used.

```
traffic ⋉ (0.01)(“time”, “time”, weather) approximate left outer join
```

The second operator ...

### 5.6 Creating Vectors and Matrices

Once the data have been preprocessed, columns may be projected out to create a matrix that may be passed to analytics/modeling techniques.

```
val mat = π “time”, “count” (traffic).toMatriD
```
This matrix may then be passed into multiple modeling techniques: (1) a Multiple Linear Regression, (2) a AutoRegressive, Integrated, Moving Average (ARIMA) model.

```scala
val model1 = Regression (mat)
val model2 = ARIMA (mat)
```

By default in SCALATlon the rightmost columns are the response/output variables. As many of the modeling techniques have a single response variable, it will be assumed to in the last column. There are also constructors and factory apply functions that take explicit vector and matrix parameters, e.g., a matrix of predictor variables and a response vector.

### 5.7 Exercises

1. Load the `auto_mpg.csv` dataset into an `auto_mpg` relation. Perform the preprocessing steps above to create a cleaned-up relation `auto_mpg2` and produce a data matrix called `auto_mat` from this relation. Print out the correlation matrix for `auto_mat`. Which columns have the highest correlation? To predict the miles per gallon `mpg` which columns are likely to be the best predictors.

Part II

Modeling
Chapter 6

Prediction

As the name predictive analytics indicates, the purpose of techniques that fall in this category is to develop models to predict outcomes. For example, the distance a golf ball travels $y$ when hit by a driver depends on several factors or inputs $x$ such as club head speed, barometric pressure, and smash factor (how square the impact is). The models can be developed using a combination of data (e.g., from experiments) and knowledge (e.g., Newton’s Second Law). The modeling techniques discussed in this technical report tend to emphasize the use of data more than knowledge, while those in the simulation modeling technical report emphasize knowledge.

Abstractly, a predictive model can generally be formulated using a prediction function $f$ as follows:

$$ y = f(x, t; b) + \epsilon $$

where

- $y$ is a response/output scalar,
- $x$ is a predictor/input vector,
- $t$ is a scalar representing time,
- $b$ is the vector of parameters of the function, and
- $\epsilon$ represents remaining residuals/errors.

Both the response $y$ and residuals/errors $\epsilon$ are treated as random variables, while the predictor/feature variables $x$ may be treated as either random or deterministic depending on context. Depending on the goals of the study as well as whether the data are the product of controlled/designed experiments, the random or deterministic view may be more suitable.

The parameters $b$ can be adjusted so that the predictive model matches the available data. Note, in the definition of a function, the arguments appear before the “;”, while the parameters appear after. The residuals/errors are typically additive as shown above, but may also be multiplicative. Of course, the formulation could be generalized by turning the output/response into a vector $y$ and the parameters into a matrix $B$.

When a model is time-independent or time can be treated as just another dimension within the $x$ vectors, prediction functions can be represented as follows:
\[ y = f(x; \ b) + \epsilon \]  

Another way to look at such models, is that we are trying to estimate the conditional expectation of \( y \) given \( x \).

\[ y = \mathbb{E}[y|x] + \epsilon \]

\[ \epsilon = y - f(x; \ b) \]

Given a dataset (\( m \) instances of data), each instance contributes to an overall residual/error vector \( \epsilon \). One of the simpler ways to estimate the parameters \( \mathbf{b} \) is to minimize the size of the residual/error vector, e.g., its Euclidean norm. The square of this norm is the sum of squared errors (\( sse \))

\[ sse = \|\epsilon\|^2 = \epsilon \cdot \epsilon \]  

This corresponds to minimizing the raw mean square error (\( mse = sse/m \)). See the section on Generalized Linear Models for further development along these lines.

In SCALATION, data are passed to the `train` function to train the model/fit the parameters \( \mathbf{b} \). In the case of prediction, the `predict` function is used to predict values for the scalar response \( y \).

A key question to address is the possible functional forms that \( f \) may take, such as the importance of time, the linearity of the function, the domains for \( y \) and \( x \), etc. We consider several cases in the subsections below.
6.1 Predictor

The Predictor trait provides a common framework for several predictor classes such as SimpleRegression or Regression. All of the modeling techniques discussed in this chapter extend either the Predictor trait, or an abstract class extending the Predictor trait, namely PredictorMat. These PredictorMat also extend the Fit class to enable Quality of Fit (QoF) evaluation.

Predictor Trait

Trait Methods:

trait Predictor extends Model

  def getX: MatriD
  def getY: VectoD
  def residual: VectoD
  def analyze (x_: MatriD, y_: VectoD, x_e: MatriD, y_e: VectoD): Predictor
  def predict (z: VectoD): Double
  def predict (z: VectoI): Double = predict (z.toDouble)
  def predict (z: MatriD): VectoD
  def forwardSel (cols: Set[Int], index_q: Int): (Int, Predictor)
  def corrMatrix (xx: MatriD): MatriD = corr (xx.asInstanceOf[MatrixD])
  def test (modelName: String, doPlot: Boolean = true)

The Predictor trait inherits the five methods from the Model trait: train, eval, hparameter, parameter and report. In addition, it defines the nine methods shown above (eight of which are to implemented by classes extending from the Predictor trait). The getX and getY methods simply return the data matrix and response vector that are used internally by the modeling technique (some techniques expand the original x and y passed into the constructor). The residual method returns the error vector (difference between actual and predicted values). The analyze method trains on the training dataset and evaluates on the test dataset. The predict method take a data vector (e.g., a new data instance) and predict its response. An overloaded predict method takes a matrix as input (with each row being an instance) and makes predictions for each row. The forwardSel method performs forward selection by adding the next most predictive variable/feature to an existing model. The corrMatrix method computes the correlation matrix for the column vectors in a matrix. The test method analyzes, reports and plots the actual response versus the predicted response.

6.1.1 Fit

The related Fit class provides a common framework for computing Quality of Fit (QoF) measures. The dataset for many models comes in the form of an m-by-n data matrix X and an m response vector y. After the parameters b (an n vector) have been fit/estimated, the error vector e may be calculated. The basic QoF measures involve taking either $\ell_1$ (Manhattan) or $\ell_2$ (Euclidean) norms of the error vector as indicated in Table 6.1.
### Table 6.1: Quality of Fit

<table>
<thead>
<tr>
<th>error/residual</th>
<th>absolute</th>
<th>( \ell_1 ) norm</th>
<th>squared</th>
<th>( \ell_2 ) norm</th>
</tr>
</thead>
<tbody>
<tr>
<td>sum</td>
<td>sum of absolute errors</td>
<td>( sae = | \epsilon |_1 )</td>
<td>sum of squared errors</td>
<td>( sse = | \epsilon |_2^2 )</td>
</tr>
<tr>
<td>mean</td>
<td>mean absolute error</td>
<td>( mae^0 = sae/m )</td>
<td>mean squared error</td>
<td>( mse^0 = sse/m )</td>
</tr>
<tr>
<td>unbiased mean</td>
<td>mean absolute error</td>
<td>( mae = sae/df )</td>
<td>mean squared error</td>
<td>( mse = sse/df )</td>
</tr>
</tbody>
</table>

Typically, if a model has \( m \) instances/rows in the dataset and \( n \) parameters to fit, the error vector will live in an \( m - n \) dimensional space (ignoring issues related to the rank the data matrix). Note, if \( n = m \), there may be a unique solution for the parameter vector \( b \), in which case \( \epsilon = 0 \), i.e., the error vector lives in a 0-dimensional space. The *degrees of freedom* (for error) is the dimensionality of the space that the error vector lives in, namely, \( df = m - n \).

### Fit Class

**Class Methods:**

```scala
@param y the values in the n-dimensional response vector
@param n the number of parameters (b.dim)
@param df the degrees of freedom (df._1, df._2) for (model/regression, error)

class Fit (y: VectoD, n: Int, private var df: PairD = (0.0, 0.0))
extends QoF with Error

def resetDF (df_update: PairD)
def mse_ : Double = mse
def diagnose (e: VectoD, yy: VectoD, w: VectoD = null, ym_ : Double = noDouble)
def ll (ms: Double = mse0, s2: Double = sig2e): Double = m * (log (.2pi) + log (s2) + ms / s2)
def fit: VectoD = VectorD (rSq, rBarSq, sst, sse, mse0, rmse, mae, df._1, df._2, fStat, aic, bic, smape)
def fitLabel: Seq [String] = Fit.fitLabel
def help: String = Fit.help
```

The `Fit` class as well as other similar classes extend the basic `QoF` trait.

### QoF Trait

The `QoF` trait defines methods to determine basic Quality of Fit (QoF) measures.

**Class Methods:**
For modeling, a user chooses one of the classes (directly or indirectly) extending the trait `Predictor` (e.g., `Regression`) to instantiate an object. Next the `train` method would be typically called, followed by the `eval` method, which computes the residual/error vector and calls the `diagnose` method. Then the `fitMap` method would be called to return quality of fit statistics computed by the `diagnose` method. The quality of fit measures compute by the `diagnose` method in the `Fit` class are shown below.

```scala
@param e the m-dimensional error/residual vector (yy - yp)
@param yy the actual response/output vector to use (test/full)
@param yp the predicted response/output vector (test/full)
@param w the weights on the instances (defaults to null)
@param ym_ the mean of the actual response/output vector to use (training/full)

def diagnose (e: VectoD, yy: VectoD, yp: VectoD, w: VectoD = null, ym_: Double = noDouble)
{
  m = yy.dim // size of response vector (test/full)
  if (m < 1) flaw ("diagnose", s"no responses to evaluate m = $m")
  if (e.dim != m) flaw ("diagnose", s"e.dim = ${e.dim} != yy.dim = $m")
  if (yp.dim != m) flaw ("diagnose", s"yp.dim = ${yp.dim} != yy.dim = $m")

  val ln_m = log (m) // natural log of m (ln(m))
  val mu = yy.mean // mean of yy (may be zero)
  val ym = if (ym_ == noDouble) { if (DEBUG) println ("diagnose: test mean"); mu }
  else { if (DEBUG) println ("diagnose: train mean"); ym_ }

  sse = e.normSq // sum of squares for error
  if (w == null) {
    sst = (yy - ym).normSq // sum of squares total (ssr + sse)
    ssr = sst - sse // sum of squares regression/model
  } else {
    ssr = (w * (yp - (w * yp / w.sum).sum))~^2).sum // regression sum of squares
    sst = ssr + sse
  } // if

  mse0 = sse / m // raw/MLE mean squared error
  rmse = sqrt (mse0) // root mean squared error (RMSE)
  nrmse = rmse / mu // normalized RMSE
```
mae = e.norm1 / m // mean absolute error
rSq = ssr / sst // coefficient of determination
if (df._1 <= 0 || df._2 <= 0) flaw ("diagnose", s"degrees of freedom df = $df <= 0")
mse = sse / df._2 // mean of squares for error
msr = ssr / df._1 // mean of squares for regression/model
rse = sqrt (mse) // residual standard error
rBarSq = 1 - (1-rSq) * r_df // adjusted R-squared
fStat = msr / mse // F statistic (quality of fit)
p_fS = 1.0 - fisherCDF (fStat, df._1.toInt, df._2.toInt) // p-value for fStat
if (p_fS.isNaN) p_fS = 0.0 // NaN => check error from 'fisherCDF'
if (sig2e == -1.0) sig2e = e.pvariance
aic = ll() + 2 * (df._1 + 1) // Akaike Information Criterion
bic = aic + (df._1 + 1) * (ln_m - 2) // Bayesian Information Criterion
smape = (e.abs / (yy.abs + yp.abs)).sum / m // symmetric mean abs. percentage error / 100
// nmae = mae / mu // normalized MAE (MAD/Mean Ratio)
} // diagnose

Note, ^ is the exponentiation operator provided in SCALATION, where the first character is ^ to give the operator higher precedence than multiplication (*).

One may look at the sum of squared errors (sse) as an indicator of model quality.

\[ sse = \epsilon \cdot \epsilon \] (6.4)

In particular, sse can be compared to the sum of squares total (sst), which measures the total variability of the response \( y \),

\[ sst = \| y - \mu_y \|^2 = y \cdot y - m \mu_y^2 = y \cdot y - \frac{1}{m} [\sum y_i]^2 \] (6.5)

while the sum of squares regression (ssr = sst – sse) measures the variability captured by the model, so the coefficient of determination measures the fraction of the variability captured by the model.

\[ R^2 = \frac{ssr}{sst} = 1 - \frac{sse}{sst} \leq 1 \] (6.6)

Values for \( R^2 \) would be non-negative, unless the proposed model is so bad (worse than the Null Model that simply predicts the mean) that the proposed model actually adds variability.

6.1.2 PredictorMat

Many modeling techniques utilize several predictor/input variables to predict a value for a response/output variable, e.g., given values for \([x_0, x_1, x_2] \) predict a value for \( y \). The datasets fed into such modeling techniques will collect multiple instances of the predictor variables into a matrix \( x \) and multiple instances of the response variable into a vector \( y \). The Predictor-Matrix (or PredictorMat) abstract class takes datasets of this form. Also, using an apply method in PredictorMat’s companion object both \( x \) and \( y \) may be passed together in a combined data matrix \( xy \). As the Predictor trait and the Fit class are used to together in most models, the PredictorMat abstract class merges them.

128
PredictorMat Abstract Class

Class Methods:

@tparam x the data/input m-by-n matrix
   (augment with a first column of ones to include intercept in model)
@tparam y the response/output m-vector
@tparam fname the feature/variable names
@tparam hparam the hyper-parameters for the model

abstract class PredictorMat (protected val x: MatriD, protected val y: VectoD, protected var fname: Strings, hparam: HyperParameter)
  extends Fit (y, x.dim2, (x.dim2 - 1, x.dim1 - x.dim2)) with Predictor
  // if not using an intercept df = (x.dim2, x.dim1-x.dim2)
  // correct by calling 'resetDF' method from 'Fit'

def getX: MatriD = x
def getY: VectoD = y
def train (x_: MatriD = x, y_: VectoD = y): PredictorMat
def train2 (x_: MatriD = x, y_: VectoD = y): PredictorMat =
def eval (x_e: MatriD = x, y_e: VectoD = y): PredictorMat =
def eval (ym: Double, y_e: VectoD, yp: VectoD): PredictorMat = ???
def analyze (x_: MatriD = x, y_: VectoD = y, hparameter: HyperParameter = hparam
def parameter: VectoD = b

def report: String =
def summary: String =
def residual: VectorD = e

def predict (z: VectorD): Double = b dot z

def predict (z: MatrixD = x): VectoD = VectorD (for i <- z.indices yield predict (z(i)))
def buildModel (x_cols: MatrixD): PredictorMat
def forwardSel (cols: Set[Int], index_q: Int = index_rSqBar): (Int, PredictorMat) =
def forwardSelAll (index_q: Int = index_rSqBar, cross: Boolean = true): (Set[Int], MatriD) =
def backwardElim (cols: Set[Int], index_q: Int = index_rSqBar, first: Int = 1): (Int, PredictorMat) =
def backwardElimAll (index_q: Int = index_rSqBar, first: Int = 1, cross: Boolean = true): (Set[Int], MatriD) =

override def corrMatrix (xx: MatriD = x): MatriD = corr (x.asInstanceOf [MatrixD])
def vif (skip: Int = 1): VectoD =
def crossValidate (k: Int = 10, rando: Boolean = true): Array[Statistic] =

A brief description of all these methods follows:

1. The `getX` method returns the actual data/input matrix used by the model. Some complex models expanded the columns in an initial data matrix to add for example quadratic or cross terms.
2. The `getY` method returns the actual response/output vector used by the model. Some complex models transform the initial response vector.

3. The `train` method takes the dataset passed into the model (either the full dataset or a training dataset) and optimizes the model parameters $b$.

4. The `train2` method takes the dataset passed into the model (either the full dataset or a training dataset) and optimizes the model parameters $b$. It also optimizes the hyper-parameters.

5. The `eval` method evaluates the Quality of Fit (QoF) either on the full dataset or a designated test dataset that is passed into the `eval` method.

6. The `parameter` method returns the estimated parameters for the model.

7. The `hparameter` method returns the hyper-parameters for the model. Many simple models have none, but more sophisticated modeling techniques such as `RidgeRegression` and `LassoRegression` have them (e.g., a shrinkage hyper-parameter).

8. The `report` method return a basic report on the model’s hyper-parameters, parameters and overall quality of fit.

9. The `summary` method return a statistical summary for each of the model parameters/coefficients including $t$ and $p$ values for the variable/parameter, where low values for $p$ indicate a strong contribution of the variable/parameter to the model.

10. The `residual` method returns the difference between the actual and predicted response vectors. The residual indicates what the model has left to explain/account for (e.g., an ideal model will only leave the noise in the data unaccounted for).

11. The two `predict` methods use the parameter vector that results from training to predict the response/output for a new instance/input. One version takes a single instance in the form of vector, while the other overloaded method takes multiple instances in the form of a matrix.

12. The `buildModel` method build a sub-model that is restricted to given columns of the data matrix.

13. The `forwardSel` method is used for forward selection of variables/features for inclusion into the model. At each step the variable that increases the predictive power of the model the most is selected. This method is called repeatedly in `forwardSelAll` to find “best” combination of features. Not guaranteed to find the optimal combination.

14. The `backwardElim` method is used for backward elimination of variables/features from the model. At each step the variable that contributes the least to the predictor power of the model is eliminated. This method is called repeatedly in `backwardElimAll` to find “best” combination of features. Not guaranteed to find the optimal combination.

15. The `corrMatrix` method returns the correlation matrix for the data/input matrix. High correlation between column vectors in the matrix may indicate **collinearity**.

16. The `vif` method returns the Variance Inflation Factors (VIFs) for each of the columns in the data/input matrix. High VIF scores may indicate **multi-collinearity**.
17. The `crossValidate` method implements $k$-fold cross-validation, where a dataset is divided into a training dataset and a test dataset. The training dataset is used by the `train` method, while the test dataset is used by the `eval` method. This is repeated $k$ times.
6.2 Null Model

The `NullModel` class implements the simplest type of predictive modeling technique. If all else fails it may be reasonable to simply guess that \( y \) will take on its expected value or mean.

\[
y = \mathbb{E}[y] + \epsilon
\]

(6.7)

This could happen if the predictors \( x \) are not relevant, not collected in a useful range or the relationship is too complex for the modeling techniques you have applied.

6.2.1 Model Equation

Ignoring the predictor variables \( x \) gives the following simple model equation.

\[
y = b_0 + \epsilon
\]

(6.8)

This intercept-only model is just a constant term plus the error/residual term.

6.2.2 Training

The training dataset in this case only consists of a response vector \( y \). The error vector in this case is

\[
\epsilon = y - \hat{y} = y - b_01
\]

(6.9)

For Least Squares Estimation (LSE), the loss function \( h(b) \) can be set to half the sum of squared errors.

\[
h(b) = \frac{1}{2} \text{sse} = \frac{1}{2} \epsilon \cdot \epsilon
\]

(6.10)

Substituting for \( \epsilon \) gives

\[
h(b) = \frac{1}{2} (y - b_01) \cdot (y - b_01)
\]

(6.11)

6.2.3 Optimization - Derivative

A function can be optimized using Calculus by taking the first derivative and setting it equal to zero. If the second derivative is positive (negative) it will be minimal (maximal).

In particular, the derivative product rule (for dot products) may be used.

\[
(f \cdot g)' = f' \cdot g + f \cdot g'
\]

(6.12)

\[
(f \cdot f)' = 2f' \cdot f
\]

(6.13)

Taking the derivative w.r.t. \( b_0 \), \( \frac{dh}{db_0} \), using the derivative product rule and setting it equal to zero yields the following equation.

\[
\frac{dh}{db_0} = -1 \cdot (y - b_01) = 0
\]

Therefore, the optimal value for the parameter \( b_0 \) is
This shows that the optimal value for the parameter is the mean of the response vector.

In SCALATION this requires just one line of code inside the train method.

```scala
def train (x_null: Matrix, y_: Vector): NullModel =
{
  b = Vector (y_.mean) // parameter vector [b0]
  this
} // train
```

After values for the model parameters are determined, it is important to assess the Quality of Fit (QoF). The eval method will compute the residual/error vector $\epsilon$ and then call the diagnose method.

```scala
def eval (x_null: Matrix, y_e: Vector): NullModel =
{
  val yp = Vector.fill (y_e.dim)(b(0)) // y predicted for (test/full)
  e = y_e - yp // compute residual/error vector e
  diagnose (e, y_e, yp) // compute diagnostics
} // eval
```

The coefficient of determination $R^2$ for the null regression model is always 0, i.e., none of variance in the random variable $y$ is explained by the model. A more sophisticated model should only be used if it is better than the null model, that is when its $R^2$ is strictly greater than zero. Also, a model can have a negative $R^2$ if its predictions are worse than guessing the mean.

Finally, the predict method is simply.

```scala
def predict (z: Vector): Double = b(0)
```

### 6.2.4 Example Calculation

For the training data shown below, the optimal value for the intercept parameter $b_0 = \mu_y = \frac{11}{4} = 2.75$. The table below shows the values of $x$, $y$, $\hat{y}$, $\epsilon$, and $\epsilon^2$. for the Null Model,

$$y = 2.75 + \epsilon \quad (6.15)$$

<table>
<thead>
<tr>
<th>$x$</th>
<th>$y$</th>
<th>$\hat{y}$</th>
<th>$\epsilon$</th>
<th>$\epsilon^2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11</td>
<td>11.25</td>
<td>-0.25</td>
<td>0.0625</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>3.25</td>
<td>-0.25</td>
<td>0.0625</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3.25</td>
<td>-0.25</td>
<td>0.0625</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4.25</td>
<td>-0.25</td>
<td>0.0625</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>11</td>
<td>0</td>
<td>0.0625</td>
</tr>
</tbody>
</table>
The sum of squared errors (sse) is given in the lower, right corner of the table. The sum of squares total for this dataset is 4.75, so

\[ R^2 = 1 - \frac{sse}{sst} = 1 - \frac{4.75}{4.75} = 0 \]

The plot below illustrates how the Null Model attempts to fit the four given data points.

Null Model Line vs. Data Points

NullModel Class

Class Methods:

@param y the response/output vector

class NullModel (y: VectoD)
    extends Fit (y, 1, (1, y.dim)) with Predictor with NoFeatureSelection

    def getX: MatriD = null
    def getY: VectoD = y
    def train (x_null: MatriD, y_: VectoD): NullModel =
    def eval (x_null: MatriD, y_e: VectoD): NullModel =
    def analyze (x_: MatriD = null, y_: VectoD = y,
    def hparameter: HyperParameter = null
    def parameter: VectoD = b
    def report: String =
    def residual: VectoD = e
    def predict (z: VectoD): Double = b(0)
    def predict (z: MatriD = null): VectoD = VectorD.fill (y.dim)(b(0))
6.2.5 Exercises

1. Determine the value of the second derivative of the loss function

\[
\frac{d^2 h}{db_0^2} = \text{?}
\]

at the critical point \( b_0 = \mu_y \). What kind of critical point is this?

2. Let the response vector \( y \) be

\[
\text{val } y = \text{VectorD (1, 3, 3, 4)}
\]

and execute the NullModel.

For context, assume the corresponding predictor vector \( y \) is

\[
\text{val } x = \text{VectorD (1, 2, 3, 4)}
\]

Draw an \( xy \) plot of the data points. Give the value for the parameter vector \( b \). Show the error distance for each point in the plot. Compare the sum of squared errors \( sse \) with the sum of squares total \( sst \). What is the value for the coefficient of determination \( R^2 \)?

3. Using ScalADion, analyze the NullModel for the following response vector \( y \).

\[
\text{val y = VectorD (2.0, 3.0, 5.0, 4.0, 6.0) // response vector y}
\]

\[
\text{println (s"y = $y")}
\]

\[
\text{val rg = new NullModel (y) // create a NullModel}
\]

\[
\text{rg.analyze () // train on data and evaluate}
\]

\[
\text{println (rg.report) // parameter values and QoF}
\]

\[
\text{val z = VectorD (5.0) // predict \( y \) for one point}
\]

\[
\text{val yp = rg.predict (z) // yp (y-predicted or y-hat)}
\]

\[
\text{println (s"predict ($z) = $yp")}
\]

4. Execute the NullModel on the Auto MPG dataset. See scalation.analytics.Auto_MPG_Regression. What is the quality of the fit (e.g., \( R^2 \) or \( r^2 \))? Is this value expected? Is it possible for a model to perform worse than this?
6.3 Simpler Regression

The SimplerRegression class supports simpler linear regression. In this case, the predictor vector \( \mathbf{x} \) consists of a single variable \( x_0 \), i.e., \( \mathbf{x} = [x_0] \) and there is only a single parameter that is the coefficient for \( x_0 \) in the model.

6.3.1 Model Equation

The goal is to fit the parameter vector \( \mathbf{b} = [b_0] \) in the following model/regression equation,

\[
y = \mathbf{b} \cdot \mathbf{x} + \epsilon = b_0 x_0 + \epsilon
\]  

(6.16)

where \( \epsilon \) represents the residuals/errors (the part not explained by the model).

6.3.2 Training

A dataset may be collected for providing an estimate for parameter \( b_0 \). Given \( m \) data points, stored in an \( m \)-dimensional vector \( \mathbf{x}_0 \) and \( m \) response values, stored in an \( m \)-dimensional vector \( \mathbf{y} \), we may obtain the following vector equation.

\[
\mathbf{y} = b_0 \mathbf{x}_0 + \epsilon
\]  

(6.17)

One way to find a value for parameter \( b_0 \) is to minimize the norm of residual/error vector \( \epsilon \).

\[
\min_{b_0} \| \epsilon \| \tag{6.18}
\]

Since \( \epsilon = \mathbf{y} - b_0 \mathbf{x}_0 \), we may solve the following optimization problem:

\[
\min_{b_0} \| \mathbf{y} - b_0 \mathbf{x}_0 \| \tag{6.19}
\]

This is equivalent to minimizing half the dot product \( \frac{1}{2} \| \epsilon \|^2 = \frac{1}{2} \epsilon \cdot \epsilon = \frac{1}{2} \text{sse} \). Thus the loss function is

\[
h(\mathbf{b}) = \frac{1}{2} (\mathbf{y} - b_0 \mathbf{x}_0) \cdot (\mathbf{y} - b_0 \mathbf{x}_0) \tag{6.20}
\]

6.3.3 Optimization - Derivative

Again, a function can be optimized using Calculus by taking the first derivative and setting it equal to zero. If the second derivative is positive (negative) it will be minimal (maximal). Taking the derivative w.r.t. \( b_0 \), \( \frac{dh}{db_0} \), using the derivative product rule (for dot products) gives

\[
(f \cdot f)' = 2 f' \cdot f
\]  

(6.21)

and setting it equal to zero yields the following equation.

\[
\frac{dh}{db_0} = - \mathbf{x}_0 \cdot (\mathbf{y} - b_0 \mathbf{x}_0) = 0
\]

Therefore, the optimal value for the parameter \( b_0 \) is

\[
b_0 = \frac{\mathbf{x}_0 \cdot \mathbf{y}}{\mathbf{x}_0 \cdot \mathbf{x}_0}
\]  

(6.22)
6.3.4 Example Calculation

Consider the following data points \{(1,1), (2,3), (3,3), (3,4)\} and solve for the parameter (slope) \(b_0\).

\[
b_0 = \frac{[1,2,3,4] \cdot [1,3,3,4]}{[1,2,3,4] \cdot [1,2,3,4]} = \frac{32}{30} = \frac{16}{15}
\]

Using this optimal value for the parameter \(b_0 = \frac{16}{15}\), we may obtain predicted values for each of the \(x\)-values.

\[
\hat{y} = \hat{y} = \text{predict}(x_0) = b_0x_0 = [1.067, 2.133, 3.200, 4.267]
\]

Therefore, the error/residual vector is

\[
\epsilon = y - \hat{y} = [1,3,3,4] - [1.067, 2.133, 3.200, 4.267] = [-0.067, 0.867, -0.2, -0.267]
\]

The table below shows the values of \(x\), \(y\), \(\hat{y}\), \(\epsilon\), and \(\epsilon^2\) for the Simpler Regression Model,

\[
y = \left[\frac{16}{15}\right] \cdot [x] + \epsilon = \frac{16}{15}x + \epsilon
\]

<table>
<thead>
<tr>
<th>(x)</th>
<th>(y)</th>
<th>(\hat{y})</th>
<th>(\epsilon)</th>
<th>(\epsilon^2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>(\frac{16}{15})</td>
<td>(-\frac{1}{15})</td>
<td>(\frac{1}{225})</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>(\frac{32}{15})</td>
<td>(\frac{13}{15})</td>
<td>(\frac{169}{225})</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>(\frac{48}{15})</td>
<td>(-\frac{3}{15})</td>
<td>(\frac{9}{225})</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>(\frac{64}{15})</td>
<td>(-\frac{4}{15})</td>
<td>(\frac{16}{225})</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>(\frac{160}{15})</td>
<td>(\frac{5}{15})</td>
<td>(\frac{13}{15} = 0.867)</td>
</tr>
</tbody>
</table>

The sum of squared errors (sse) is given in the lower, right corner of the table. The sum of squares total for this dataset is 4.75, so

\[
R^2 = 1 - \frac{sse}{sst} = 1 - \frac{0.867}{4.75} = 0.813
\]

The plot below illustrates how the Simpler Regression Model attempts to fit the four given data points.
Note, that this model has no intercept. This makes the solution for the parameter very easy, but may make the model less accurate. This is remedied in the next section. Since no intercept really means the intercept is zero, the regression line will go through the origin.

**SimplerRegression Class**

**Class Methods:**

- `@param x the data/input matrix`
- `@param y the response/output vector`
- `@param fname_ the feature/variable names`
- `@param hparam the hyper-parameters (currently has none)`

```scala
class SimplerRegression (x: MatriD, y: VectoD, fname_ : Strings = null)
  hparam: HyperParameter = null
  extends PredictorMat (x, y, fname_, hparam) with NoFeatureSelectionMat

  def train (x_ : MatriD, y_ : VectoD): SimplerRegression =
  override def vif (skip: Int = 1): VectoD = VectorD (1.0) // not a problem for this model
```

### 6.3.5 Exercises

1. For $x_0 = [1, 2, 3, 4]$ and $y = [1, 3, 3, 4]$, try various values for the parameter $b_0$. Plot the sum of squared errors ($sse$) vs. $b_0$.

```scala
import scalation.linalgebra.VectorD
import scalation.plot.Plot
```
object SimplerRegression_exer_1 extends App {
  val x0 = VectorD (1, 2, 3, 4)
  val y = VectorD (1, 3, 3, 4)
  val b0 = VectorD.range (0, 50) / 25.0
  val sse = new VectorD (b0.dim)
  for (i <- b0.range) {
    val e = ?
    sse(i) = e dot e
  } // for
  new Plot (b0, sse, lines = true)
} // SimplerRegression_exer_1 object

Where do you think the minimum occurs?

Note, to run your code you should make a separate project directory outside of SCALATion. See https://alvinalexander.com/scala/how-to-create-sbt-project-directory-structure-scala for the directory structure. Your lib directory must access SCALATion’s jar files in its lib directory. Create a file called SimplerRegression_exer_1.scala in your src/main/scala directory. In your project’s base directory, type sbt. Within sbt type compile and then run.

2. From the X matrix and y vector, plot the set of data points \{(x_i, y_i)\} \(0 \leq i < m\} \) and draw the line that is nearest to these points. What is the slope of this line. Pass the X matrix and y vector as arguments to the SimplerRegression class to obtain the \(\mathbf{b} = [b_0]\) vector.

    // 4 data points: x0
    val x = new MatrixD ((4, 1), 1.0, // x 4-by-1 matrix
                         2.0,
                         3.0,
                         4.0)
    val y = VectorD (1.0, 3.0, 3.0, 4.0) // y vector

    val rg = new SimplerRegression (x, y) // create a SimplerRegression
    rg.analyze () // train and evaluate
    println (rg.report)

    val yp = rg.predict ()
    new Plot (x.col(0), y, yp, lines = true) // black for y and red for yp

An alternative to using the above constructor new SimplerRegression is to use a factory function SimplerRegression. Substitute in the following lines of code to do this.

    val x = VectorD (1, 2, 3, 4)
    val rg = SimplerRegression (x, y, drp._1, drp._2)
    new Plot (x, y, yp, lines = true)

Note, drp stands for default remaining parameters, giving null values for fname and hparam.
3. From the $X$ matrix and $y$ vector, plot the set of data points \{(x_i, y_i) | 0 \leq i < m\} and draw the line that is nearest to these points and intersects the origin \{0, 0\}. What is the slope of this line? Pass the $X$ matrix and $y$ vector as arguments to the SimplerRegression class to obtain the $b = [b_0]$ vector.

```scala
// 5 data points:
val x = new MatrixD ((5, 1), 0.0, // x 5-by-1 matrix
  1.0,
  2.0,
  3.0,
  4.0)
val y = VectorD (2.0, 3.0, 5.0, 4.0, 6.0) // y vector

val rg = new SimplerRegression (x, y) // create a SimplerRegression
rg.analyze () // train and evaluate
println (rg.report)

val z = VectorD (5.0) // predict y for one point
val yp = rg.predict (z) // y-predicted
println (s"predict ($z) = $yp")
```

4. Execute the SimplerRegression on the Auto MPG dataset. See scalation.analytics.ExampleAuto MPG. What is the quality of the fit (e.g., $R^2$ or $rSq$)? Is this value expected? What does it say about this model? Try using different columns for the predictor variable.

5. Compute the second derivative of the loss function w.r.t. $b_0$, $\frac{d^2 h}{db_0^2}$. Under what conditions will it be positive?
6.4 Simple Regression

The SimpleRegression class supports simple linear regression. In this case, the predictor vector \( x \in \mathbb{R}^2 \) consists of the constant one and a single variable \( x_1 \), i.e., \([1, x_1]\), so there are now two parameters \( b = [b_0, b_1] \in \mathbb{R}^2 \) in the model.

6.4.1 Model Equation

The goal is to fit the parameter vector \( b \) in the model/regression equation,

\[
y = b \cdot x + \epsilon = [b_0, b_1] \cdot [1, x_1] + \epsilon = b_0 + b_1 x_1 + \epsilon
\]  
(6.23)

where \( \epsilon \) represents the residuals (the part not explained by the model).

6.4.2 Training

The model is trained on a dataset consisting of \( m \) data points/vectors, stored row-wise in an \( m \)-by-2 matrix \( X \in \mathbb{R}^{m \times 2} \) and \( m \) response values, stored in an \( m \) dimensional vector \( y \in \mathbb{R}^m \).

\[
y = Xb + \epsilon
\]  
(6.24)

The parameter vector \( b \) may be determined by solving the following optimization problem:

\[
\min_b \| \epsilon \|
\]  
(6.25)

Substituting \( \epsilon = y - \hat{y} = y - Xb \) yields

\[
\min_b \| y - Xb \|
\]  
(6.26)

Using the fact that the matrix \( X \) consists of two column vectors \( 1 \) and \( x_1 \), it can be rewritten,

\[
\min_{[b_0, b_1]} \| y - [1 x_1] \begin{bmatrix} b_0 \\ b_1 \end{bmatrix} \|
\]

\[
\min_{[b_0, b_1]} \| y - (b_0 1 + b_1 x_1) \|
\]  
(6.26)

This is equivalent to minimizing the dot product \( \| \epsilon \|^2 = \epsilon \cdot \epsilon = sse \)

\[
(y - (b_0 1 + b_1 x_1)) \cdot (y - (b_0 1 + b_1 x_1))
\]  
(6.27)

Since \( x_0 \) is just \( 1 \), for simplicity we drop the subscript on \( x_1 \). Thus the loss function \( \frac{1}{2}sse \) is

\[
h(b) = \frac{1}{2} (y - (b_0 1 + b_1 x)) \cdot (y - (b_0 1 + b_1 x))
\]  
(6.28)
6.4.3 Optimization - Gradient

A function of several variables can be optimized using Vector Calculus by setting its gradient equal to zero and solving the resulting system of equations. When the system of equations are linear, matrix factorization may be used, otherwise techniques from Nonlinear Optimization may be needed.

Taking the gradient \( \nabla = \left[ \frac{\partial}{\partial b_0}, \frac{\partial}{\partial b_1} \right] \) of the loss function \( h \) using the derivative product rule for dot products ((\( f \cdot f \))' = 2f' \cdot f) and setting it equal to zero (0 = [0, 0]) yields two equations.

\[ \nabla h(b) = \left[ \frac{\partial h}{\partial b_0}, \frac{\partial h}{\partial b_1} \right] = 0 \quad (6.29) \]

**Partial Derivative w.r.t. \( b_0 \)**

The first equation results from setting \( \frac{\partial}{\partial b_0} \) of \( h \) to zero.

\[-1 \cdot (y - (b_0 1 + b_1 x)) = 0 \]
\[1 \cdot y - 1 \cdot (b_0 1 + b_1 x) = 0 \]
\[b_0 1 = 1 \cdot y - b_1 1 \cdot x \]

Since \( 1 \cdot 1 = m \), \( b_0 \) may be expressed as

\[ b_0 = \frac{1 \cdot y - b_1 1 \cdot x}{m} \quad (6.30) \]

**Partial Derivative w.r.t. \( b_1 \)**

Similarly, the second equation results from setting \( \frac{\partial}{\partial b_1} \) of \( h \) to zero.

\[-x \cdot (y - (b_0 1 + b_1 x)) = 0 \]
\[x \cdot y - x \cdot (b_0 1 + b_1 x) = 0 \]
\[b_0 1 \cdot x + b_1 x \cdot x = x \cdot y \]

Multiplying by both sides by \( m \) produces

\[ m b_0 1 \cdot x + m b_1 x \cdot x = m x \cdot y \quad (6.31) \]

Substituting for \( m b_0 = 1 \cdot y - b_1 1 \cdot x \) yields

\[ [1 \cdot y - b_1 1 \cdot x] 1 \cdot x + m b_1 x \cdot x = m x \cdot y \]
\[ b_1 [m x \cdot x - (1 \cdot x)^2] = m x \cdot y - (1 \cdot x)(1 \cdot y) \]

Solving for \( b_1 \) gives

\[ b_1 = \frac{m x \cdot y - (1 \cdot x)(1 \cdot y)}{m x \cdot x - (1 \cdot x)^2} \quad (6.32) \]

The \( b_0 \) parameter gives the *intercept*, while the \( b_1 \) parameter gives the *slope* of the line that best fits the data points.
6.4.4 Example Calculation

Consider again the problem from the last section where the data points are \{(1, 1), (2, 3), (3, 3), (3, 4)\} and solve for the two parameters, (intercept) \(b_0\) and (slope) \(b_1\).

\[
b_1 = \frac{4[1, 2, 3, 4] \cdot [1, 3, 3, 4] - (1 \cdot [1, 2, 3, 4])(1 \cdot [1, 3, 3, 4])}{4[1, 2, 3, 4] \cdot [1, 2, 3, 4] - (1 \cdot [1, 2, 3, 4])^2} = \frac{128 - 110}{120 - 100} = \frac{18}{20} = 0.9
\]

\[
b_0 = \frac{1 \cdot [1, 3, 3, 4] - 0.9(1 \cdot [1, 2, 3, 4])}{4} = \frac{11 - 0.9 \cdot 10}{4} = 0.5
\]

The table below shows the values of \(x\), \(y\), \(\hat{y}\), \(\epsilon\), and \(\epsilon^2\) for the Simple Regression Model,

\[
y = [0.5, 0.9] \cdot [1, x] + \epsilon = 0.5 + 0.9x + \epsilon
\]

Table 6.4: Simple Regression Model: Example Training Data

<table>
<thead>
<tr>
<th>(x)</th>
<th>(y)</th>
<th>(\hat{y})</th>
<th>(\epsilon)</th>
<th>(\epsilon^2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1.4</td>
<td>-0.4</td>
<td>0.16</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>2.3</td>
<td>0.7</td>
<td>0.49</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3.2</td>
<td>-0.2</td>
<td>0.04</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4.1</td>
<td>-0.1</td>
<td>0.01</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>11</td>
<td>0</td>
<td>0.7</td>
</tr>
</tbody>
</table>

The sum of squared errors (sse) is given in the lower, right corner of the table. The sum of squares total for this dataset is 4.75, so the Coefficient of Determination,

\[
R^2 = 1 - \frac{sse}{sst} = 1 - \frac{0.7}{4.75} = 0.853
\]

The plot below illustrates how the Simpler Regression Model attempts to fit the four given data points.

Simple Regression Model Line vs. Data Points
Concise Formulas for the Parameters

More concise and intuitive formulas for the parameters $b_0$ and $b_1$ may be derived.

- Using the definition for mean from chapter 2 for $\mu_x$ and $\mu_y$, it can be shown that the expression for $b_0$ shortens to

\[ b_0 = \mu_y - b_1\mu_x \]  

(6.33)

Draw a line through the following two points $[0, b_0]$ (the intercept) and $[\mu_x, \mu_y]$ (the center of mass). How does this line compare to the regression line.

- Now, using the definitions for covariance $\sigma_{x,y}$ and variance $\sigma_x^2$ from chapter 2, it can be shown that the expression for $b_1$ shortens to

\[ b_1 = \frac{\sigma_{x,y}}{\sigma_x^2} \]  

(6.34)

If the slope of the regression line is simply the ratio of the covariance to the variance, what would the slope be if $y = x$.

6.4.5 Exploratory Data Analysis

As discussed in Chapter 1, Exploratory Data Analysis (EDA) should be performed after preprocessing the dataset. Once the response variable $y$ is selected, a null model should be created to see in a plot where the data points lie compared to the mean. The code below shows how to do this for the AutoMPG dataset.

```java
import ExampleAutoMPG.{t, x, y}

// Plot response y vs. row index t
val nm = new NullModel (y)
nm.analyze ()
println (nm.report)
val yp = nm.predict ()
new Plot (t, y, yp, "EDA: y and yp (red) vs. t", lines = true)
```

Next the relationships between the predictor variable $x_j$ (the columns in input/data matrix $X$) should be compared. If two of the predictor variables are highly correlated, their individual effects on the response variable $y$ may be indistinguishable. The correlations between the predictor variable, may be seen by examining the correlation matrix.

```java
// Correlation Matrix for columns of x
println (nm.corrMatrix (x))
```

Although Simple Regression may be too simple for many problems/datasets, it should be used in Exploratory Data Analysis (EDA). A simple regression model should be created for each predictor variable $x_j$. The data points and the best fitting line should be plotted with $y$ on the vertical axis and $x_j$ on the horizontal axis. The data scientist should look for patterns/tendencies of $y$ versus $x_j$, such as linear, quadratic, etc. When there is no relationship, the points will appear to be randomly and uniformly positioned in the plane.
for (j <- x.range2) {
    banner (s"Plot response y vs. predictor variable x$j")
    val xj = x.col(j)
    val rg = SimpleRegression (xj, y, drp._1, drp._2)
    rg.analyze()
    println (rg.report)
    val yp = rg.predict()
    new Plot (xj, y, yp, s"EDA: y and yp (red) vs. x$j", lines = true)
} // for

SimpleRegression Class

Class Methods:

@param x the data/input matrix augmented with a first column of ones
@param y the response/output vector
@param fname_ the feature/variable names
@param hparam the hyper-parameters (currently has none)

class SimpleRegression (x: MatrixD, y: VectorD, fname_ : Strings = null,
    hparam: HyperParameter = null)
        extends PredictorMat (x, y, fname_, hparam) with NoFeatureSelectionMat

def train (x_ : MatrixD, y_ : VectorD): SimpleRegression =
    override def vif (skip: Int = 1): VectorD = VectorD (1.0) // not a problem for this model

6.4.6 Exercises

1. From the $X$ matrix and $y$ vector, plot the set of data points $\{(x_i,y_i)\mid 0 \leq i < m\}$ and draw the line that is nearest to these points (i.e., that minimize $\|\epsilon\|$). Using the formulas from developed in this section, what are the intercept and slope $[b_0,b_1]$ of this line.

Also, pass the $X$ matrix and $y$ vector as arguments to the SimpleRegression class to obtain the $b$ vector.

// 4 data points: constant x1
val x = new MatrixD ((4, 2), 1, 1, // x 4-by-2 matrix
                     1, 2,
                     1, 3,
                     1, 4)

val y = VectorD (1, 3, 3, 4) // y vector

val rg = new SimpleRegression (x, y) // create a SimpleRegression
rg.analyze () // train and evaluate
println (rg.report)
2. For more complex models, setting the gradient to zero and solving a system of simultaneous equation may not work, in which case more general optimization techniques may be applied. Two simple optimization techniques are grid search and gradient descent.

For grid search, in a spreadsheet set up a 5-by-5 grid around the optimal point for $b$, found in the previous problem. Compute values for the loss function $h = \frac{1}{2} \text{sse}$ for each point in the grid. Plot $h$ versus $b_0$ across the optimal point. Do the same for $b_1$. Make a 3D plot of the surface $h$ as a function $b_0$ and $b_1$.

For gradient descent, pick a starting point $b^0$, compute the gradient $\nabla h$ and move $-\eta \nabla h$ from $b^0$ where $\eta$ is the learning rate (e.g., 0.1). Repeat for a few iterations. What is happening to the value of the loss function $h = \frac{1}{2} \text{sse}$.

$$\nabla h = \begin{bmatrix} -1 \cdot (y - (b_0 1 + b_1 x)), & -x \cdot (y - (b_0 1 + b_1 x)) \end{bmatrix}$$

Substituting $\epsilon = y - (b_0 1 + b_1 x)$, $\nabla h$ may be written as

$$[-1 \cdot \epsilon, -x \cdot \epsilon]$$

3. From the $X$ matrix and $y$ vector, plot the set of data points $\{(x_i, y_i) | 0 \leq i < m \}$ and draw the line that is nearest to these points. What are the intercept and slope of this line. Pass the $X$ matrix and $y$ vector as arguments to the SimpleRegression class to obtain the $b$ vector.

```scala
// 5 data points: constant x1
val x = new MatrixD ((5, 2), 1.0, 0.0, // x 5-by-2 matrix
  1.0, 1.0,
  1.0, 2.0,
  1.0, 3.0,
  1.0, 4.0)
val y = VectorD (2.0, 3.0, 5.0, 4.0, 6.0) // y vector
val rg = new SimpleRegression (x, y) // create a SimpleRegression
rg.analyze () // train and evaluate
println (rg.report)
val z = VectorD (1.0, 5.0) // predict y for one point
val yp = rg.predict (z) // y-predicted
println (s"predict ($z) = $yp")
```

4. Execute the SimpleRegression on the Auto MPG dataset. See scalation.analytics.ExampleAuto MPG. What is the quality of the fit (e.g., $R^2$ or $r^2$)? Is this value expected? Try using different columns for the predictor variable.
6.5 Regression

The Regression class supports multiple linear regression where multiple input/predictor variables are used to predict a value for the response/output variable. In this case, the predictor vector $\mathbf{x}$ is multi-dimensional $[1, x_1, ... x_k] \in \mathbb{R}^n$, so the parameter vector $\mathbf{b} = [b_0, b_1, \ldots, b_k]$ has the same dimension as $\mathbf{x}$, while response $y$ is a scalar.

![Figure 6.1: Multiple Linear Regression](image)

The intercept can be provided by fixing $x_0$ to one, making $b_0$ the intercept. Alternatively, $x_0$ can be used as a regular input variable by introducing another parameter $\beta$ for the intercept. In Neural Networks, $\beta$ is referred to as bias and $b_j$ is referred to as the edge weight connecting input vertex/node $j$ to the output node as shown in Figure 6.1. Note, if an activation function $f_a$ is added to the model, the Multiple Linear Regression model becomes a Perceptron model.

6.5.1 Model Equation

The goal is to fit the parameter vector $\mathbf{b}$ in the model/regression equation,

$$y = \mathbf{b} \cdot \mathbf{x} + \epsilon = b_0 + b_1 x_1 + \ldots + b_k x_k + \epsilon$$

(6.35)

where $\epsilon$ represents the residuals (the part not explained by the model).

6.5.2 Training

Using several data samples as a training set $(\mathbf{X}, \mathbf{y})$, the Regression class in SCALATion can be used to estimate the parameter vector $\mathbf{b}$. Each sample pairs an $\mathbf{x}$ input vector with a $y$ response value. The $\mathbf{x}$ vectors are placed into a data/input matrix $\mathbf{X} \in \mathbb{R}^{m \times n}$ row-by-row with a column of ones as the first column in $\mathbf{X}$. The individual response values taken together form the response vector $\mathbf{y} \in \mathbb{R}^m$.

The training diagram shown in Figure 6.2 illustrates how the $i^{th}$ instance/row flows through the diagram computing the predicted response $\hat{y} = \mathbf{b} \cdot \mathbf{x}$ and the error $\epsilon = y - \hat{y}$.
The matrix-vector product $Xb$ provides an estimate for the response vector $\hat{y}$.

$$y = Xb + \epsilon$$  \hfill (6.36)

The goal is to minimize the distance between $y$ and its estimate $\hat{y}$, i.e., minimize the norm of residual/error vector.

$$\min_b \| \epsilon \|$$  \hfill (6.37)

Substituting $\epsilon = y - \hat{y} = y - Xb$ yields

$$\min_b \| y - Xb \|$$  \hfill (6.38)

This is equivalent to minimizing half the dot product of the error vector with itself ($\frac{1}{2}||\epsilon||^2 = \frac{1}{2}\epsilon \cdot \epsilon = \frac{1}{2}sse$)

Thus, the loss function is

$$h(b) = \frac{1}{2}(y - Xb) \cdot (y - Xb)$$  \hfill (6.39)

### 6.5.3 Optimization - Gradient

The gradient of the loss function $\nabla h$ with respect to the parameter vector $b$ is the vector of partial derivatives.

$$\nabla h = \begin{bmatrix} \frac{\partial h}{\partial b_0} \\ \frac{\partial h}{\partial b_1} \\ \vdots \\ \frac{\partial h}{\partial b_k} \end{bmatrix}$$  \hfill (6.40)

where the $j^{th}$ partial derivative (again using the product rule for dot products $(f \cdot f)' = 2f' \cdot f$) is

$$\frac{\partial h}{\partial b_j} = -x_j \cdot (y - Xb) = -x_j^\top(y - Xb)$$  \hfill (6.41)

as the parameter $b_j$ is only multiplied by column $x_j$ in the matrix-vector product $Xb$. The dot product is equivalent a transpose operation followed by matrix multiplication. The gradient is formed by collecting all these partial derivatives together.
\[ \nabla h = -X^T(y - Xb) \]

(6.42)

Now, setting the gradient equal to the zero vector \( 0 \in \mathbb{R}^n \) yields

\[-X^T(y - Xb) = 0\]

\[-X^Ty + (X^TX)b = 0\]

A more detailed derivation of this equation is given in section 3.4 of “Matrix Calculus: Derivation and Simple Application” [44]. Moving the term involving \( b \) to the left side, results in the \textit{Normal Equations}.

\[(X^TX)b = X^Ty\]  

(6.43)

Note: equivalent to minimizing the distance between \( y \) and \( Xb \) is minimizing the sum of the squared residuals/errors (Least Squares method).

	extsc{ScalaTion} provides five techniques for solving for the parameter vector \( b \) based on the Normal Equations: Matrix Inversion, LU Factorization, Cholesky Factorization, QR Factorization and SVD Factorization.

### 6.5.4 Matrix Inversion Technique

Starting with the Normal Equations

\[(X^TX)b = X^Ty\]  

(6.44)

a simple technique is Matrix Inversion, which involves computing the inverse of \( X^TX \) and using it to multiply both sides of the Normal Equations.

\[b = (X^TX)^{-1}X^Ty\]  

(6.45)

where \((X^TX)^{-1}\) is an \( n \)-by-\( n \) matrix, \( X^T \) is an \( n \)-by-\( m \) matrix and \( y \) is an \( m \)-vector. When \( X \) is full rank, the expression above involving the \( X \) matrix may be referred to as the pseudo-inverse \( X^+ \).

\[X^+ = (X^TX)^{-1}X^T\]

When \( X \) is not full rank, Singular Value Decomposition may be applied to compute \( X^+ \). Using the pseudo-inverse, the parameter vector \( b \) may be solved for as follows:

\[b = X^+y\]  

(6.46)

The pseudo-inverse can be computed by first multiplying \( X \) by its transpose. Gaussian Elimination can be used to compute the inverse of this, which can be then multiplied by the transpose of \( X \). In \textsc{ScalaTion}, the computation for the pseudo-inverse \((x.pinv)\) looks similar to the math.

\[val x.pinv = (x.t * x).inverse * x.t\]

A more robust \textsc{ScalaTion} implementation is provided by the \texttt{Fac.Inv} class.
val fac = new Fac_Inv (x.t * x).factor ()
val x_pinv = fac.factors._2 * x.t

For efficiency, the code in Regression does not calculate $x_{\text{pinv}}$, rather is directly solves for the parameters $b$.

val b = fac.solve (x.t * y)

The Hat Matrix

Starting with using the pseudo-inverse to determine the optimal parameter vector $b$

$$b = X^+ y$$  \hspace{1cm} (6.47)

One can substitute the rhs into the equation for $\hat{y} = Xb$

$$\hat{y} = XX^+ y = X(X^T X)^{-1} X^T y = Hy$$  \hspace{1cm} (6.48)

where $H = X(X^T X)^{-1} X^T$ is the hat matrix (puts a hat on $y$). The hat matrix may be viewed as a projection matrix.

6.5.5 LU Factorization Technique

Lower, Upper Factorization (Decomposition) works like Matrix Inversion, except that is just reduces the matrix to zeroes below the diagonal, so it tends to be faster and less prone to numerical instability. First the product $X^T X$, an $n$-by-$n$ matrix, is factored

$$X^T X = LU$$

where $L$ is a lower left triangular $n$-by-$n$ matrix and $U$ is an upper right triangular $n$-by-$n$ matrix. Then the normal equations may be rewritten

$$LUb = X^T y$$

Letting $w = Ub$ allows the problem to solved in two steps. The first is solved by forward substitution to determine the vector $w$.

$$Lw = X^T y$$

Finally, the parameter vector $b$ is determined by backward substitution.

$$Ub = w$$
Example Calculation

Consider the example where the input/data matrix \( X \) and output/response vector \( y \) are as follows:

\[
X = \begin{bmatrix}
1 & 1 \\
1 & 2 \\
1 & 3 \\
1 & 4
\end{bmatrix}, \quad y = \begin{bmatrix}
1 \\
3 \\
3 \\
4
\end{bmatrix}
\]

Putting these values into the Normal Equations \( (X^TX)b = X^Ty \) yields

\[
\begin{bmatrix}
4 & 10 \\
10 & 30
\end{bmatrix}
\]

Multiply the first row by -2.5 and add it to the second row,

\[
\begin{bmatrix}
4 & 10 \\
0 & 5
\end{bmatrix}
\]

This results in the following optimal parameter vector \( b = [5, .9] \). Note, the product of \( L \) and \( U \) gives \( X^TX \).

\[
\begin{bmatrix}
1 & 0 \\
2.5 & 1
\end{bmatrix}
\begin{bmatrix}
4 & 10 \\
0 & 5
\end{bmatrix}
= \begin{bmatrix}
4 & 10 \\
10 & 30
\end{bmatrix}
\]

6.5.6 Cholesky Factorization Technique

A faster and slightly more stable technique is to use Cholesky Factorization. Since the product \( X^TX \) is a positive definite, symmetric matrix, it may factored using Cholesky Factorization into

\[
X^TX = LL^T
\]

where \( L \) is a lower triangular \( n \)-by-\( n \) matrix. Then the normal equations may be rewritten

\[
LL^Tb = X^Ty
\]

Letting \( w = L^Tb \), we may solve for \( w \) using forward substitution

\[
Lw = X^Ty
\]

and then solve for \( b \) using backward substitution.

\[
L^Tb = w
\]

As an example, the product of \( L \) and its transpose \( L^T \) gives \( X^TX \).

\[
\begin{bmatrix}
2 & 0 \\
5 & \sqrt{5}
\end{bmatrix}
\begin{bmatrix}
2 & 5 \\
0 & \sqrt{5}
\end{bmatrix}
= \begin{bmatrix}
4 & 10 \\
10 & 30
\end{bmatrix}
\]
Therefore, \( \mathbf{w} \) can be determined by forward substitution and \( \mathbf{b} \) by backward substitution.

\[
\begin{bmatrix}
2 & 0 \\
5 & \sqrt{5}
\end{bmatrix}
\begin{bmatrix}
w
\end{bmatrix}
=
\begin{bmatrix}
11 \\
32
\end{bmatrix},
\begin{bmatrix}
2 & 5 \\
0 & \sqrt{5}
\end{bmatrix}
\begin{bmatrix}
b
\end{bmatrix}
=
\begin{bmatrix}
w
\end{bmatrix}
\]

### 6.5.7 QR Factorization Technique

A slightly slower, but even more robust technique is to use QR Factorization. Using this technique, the \( m \)-by-\( n \) \( \mathbf{X} \) matrix can be factored directly, which increases the stability of the technique.

\[
\mathbf{X} = \mathbf{QR}
\]

where \( \mathbf{Q} \) is an orthogonal \( m \)-by-\( n \) matrix and \( \mathbf{R} \) matrix is a right upper triangular \( n \)-by-\( n \) matrix. Starting again with the Normal Equations,

\[
(\mathbf{X}^\top \mathbf{X}) \mathbf{b} = \mathbf{X}^\top \mathbf{y}
\]

simply substitute \( \mathbf{QR} \) for \( \mathbf{X} \).

\[
(\mathbf{QR})^\top \mathbf{QR} \mathbf{b} = (\mathbf{QR})^\top \mathbf{y}
\]

Taking the transpose gives

\[
\mathbf{R}^\top \mathbf{Q}^\top \mathbf{QR} \mathbf{b} = \mathbf{R}^\top \mathbf{Q}^\top \mathbf{y}
\]

and using the fact that \( \mathbf{Q}^\top \mathbf{Q} = \mathbf{I} \), we obtain the following:

\[
\mathbf{R}^\top \mathbf{R} \mathbf{b} = \mathbf{R}^\top \mathbf{Q}^\top \mathbf{y}
\]

Multiply both sides by \( (\mathbf{R}^\top)^{-1} \) yields

\[
\mathbf{R} \mathbf{b} = \mathbf{Q}^\top \mathbf{y}
\]

Since \( \mathbf{R} \) is an upper triangular matrix, the parameter vector \( \mathbf{b} \) can be determined by backward substitution. Alternatively, the pseudo-inverse may be computed as follows:

\[
\mathbf{X}^+ = \mathbf{R}^{-1} \mathbf{Q}^\top
\]

SCALaTION uses Householder Orthogonalization (alternately Modified Gram-Schmidt Orthogonalization) to factor \( \mathbf{X} \) into the product of \( \mathbf{Q} \) and \( \mathbf{R} \).

### 6.5.8 Use of Factorization in Regression

By default, SCALaTION uses QR Factorization to compute the pseudo-inverse \( \mathbf{X}^+ \). The other techniques may be selected by using the third argument (technique) in the constructor, setting it to Cholesky, SVD, LU or Inverse. For more information see [http://see.stanford.edu/materials/lsoldsee263/05-1s.pdf](http://see.stanford.edu/materials/lsoldsee263/05-1s.pdf).
object RegTechnique extends Enumeration
{
  type RegTechnique = Value
  val QR, Cholesky, SVD, LU, Inverse = Value
  val techniques = Array (QR, Cholesky, SVD, LU, Inverse)
}

Based on the selected technique, the appropriate type of matrix factorization is performed. The first part of
the code below constructs and returns a factorization object.

private def solver (x_: MatriD): Factorization =
{
  technique match {
    case QR => new Fac_QR (x_, false) // QR Factorization
    case Cholesky => new Fac_Cholesky (x_.t * x_) // Cholesky Factorization
    case SVD => new SVD (x_) // Singular Value Decomposition
    case LU => new Fac_LU (x_.t * x_) // LU Factorization
    case _ => new Fac_Inv (x_.t * x_) // Inverse Factorization
  } // match
} // solver

The `train` method below computes parameter/coefficient vector \( b \) by calling the `solve` method provided
by the factorization classes.

def train (x_: MatriD = x, y_: VectoD = y): Regression =
{
  val fac = solver (x_)
  fac.factor () // factor the matrix, either X or X.t * X
  b = technique match {
    case QR => fac.solve (y_) // R * b = Q.t * y
    case Cholesky => fac.solve (x_.t * y_) // L * L.t * b = X.t * y
    case SVD => fac.solve (y_) // b = V * S^-1 * U.t * y
    case LU => fac.solve (x_.t * y_) // b = (X.t * X) \ X.t * y
    case _ => fac.solve (x_.t * y_) // b = (X.t * X)^-1 * X.t * y
  } // match
  if (b(0).isNaN) flaw ("train", s"parameter b = $b")
  if (DEBUG) (s"train: parameter b = $b")
  this
} // train

After training, the `eval` method overridden for efficiency from `PredictorMat` does two things: First, the
residual/error vector \( \epsilon \) is computed. Second, several quality of fit measures are computed by calling the
diagnose method.

override def eval (x_e: MatriD = x, y_e: VectoD = y): Regression =
val yp = x_e * b          // y predicted for x_e (test/full)
e = y_e - yp              // compute residual/error vector e
diagnose (e, y_e, yp)     // compute diagnostics
this
}  // eval

To see how the train and eval methods work in a Regression model see the Collinearity Test and Texas Temperatures examples in subsequent subsections.

6.5.9 Model Assessment

The quality of fit measures includes the coefficient of determination $R^2$ as well as several others. Given $m$ instances (data points) and $n$ parameters in the regression model, the degrees of freedom captured by the regression model is $df_r$ and left for error is $df$.

$$df_r = n - 1 = k$$
$$df = m - n$$

If the model is without an intercept, $df_r = n$. The ratio of total degrees of freedom to degrees of freedom for error is

$$r_{df} = \frac{df_r + df}{df}$$

SimplerRegression is at one extreme of model complexity, where $df = m-1$ and $df_r = 1$, so $r_{df} = m/(m-1)$ is close to one. For a more complicated model, say with $n = m/2$, $r_{df}$ will be close to 2. This ratio can be used to adjust the Coefficient of Determination $R^2$ to reduce it with increasing number of parameters. This is called the Adjusted Coefficient of Determination $\bar{R}^2$

$$\bar{R}^2 = 1 - r_{df}(1 - R^2)$$

Suppose $m = 121$, $n = 21$ and $R^2 = 0.9$, as an exercise, show that $r_{df} = 1.2$ and $\bar{R}^2 = 0.88$.

Dividing $sse$ and $ssr$ by their respective degrees of freedom gives the mean square error and regression, respectively

$$mse = sse / df$$
$$msr = ssr / df_r$$

The mean square error $mse$ follows a Chi-square distribution with $df$ degrees of freedom, while the mean square regression $msr$ follows a Chi-square distribution with $df_r$ degrees of freedom. Consequently, the ratio

$$\frac{msr}{mse} \sim F_{df_r, df}$$

(6.49)

that is, it follows an $F$-distribution with $(df_r, df)$ degrees of freedom. If this number exceeds the critical value, one can claim that the parameter vector $b$ is not zero, implying the model is useful. More general quality of fit measures useful for comparing models are the Akaike Information Criterion (AIC) and Bayesian Information Criterion (BIC).

In ScalaTion the several Quality of Fit (QoF) measures are computed by the diagnose method in the Fit class, as described in section 1 of this chapter.
def diagnose (e: VectoD, yy: VectoD, yp: VectoD = null, w: VectoD = null)

6.5.10 Model Validation

Data are needed to two purposes: First, the characteristics or patterns of the data need to be investigated to select an appropriate modeling technique, features for a model and finally to estimate the parameters and probabilities used by the model. Data Scientists assisted by tools do the first part of this process, while the latter part is called training. Hence the `train` method is part of all modeling techniques provided by `ScalaTion`. Second, data are needed to test the quality of the trained model.

One approach would be to train the model using all the available data. This makes sense, since the more data used for training, the better the model. In this case, the testing data would need to be same as the training leading to whole dataset evaluation. Now the difficult issue is how to guard against over-fitting. With enough flexibility and parameters to fit, modeling techniques can push quality measures like $R^2$ to perfection ($R^2 = 1$) by fitting the noise in the data. Doing so tends to make a model worse in practice than a simple model that just captures the signal. That is where quality measures like $\bar{R}^2$ come into play, but computations of $\bar{R}^2$ require determination of degrees of freedom ($df$), which may be difficult for some modeling techniques. Furthermore, the amount of penalty introduced by such quality measures is somewhat arbitrary.

Would not it be better to measure quality in way in which models fitting noise are downgraded because they perform more poorly on data they have not seen? Is it really a test, if the model has already seen the data? The answers to these questions are obvious, but the solution of the underlying problem is a bit tricky. The first thought would be to divide a dataset in half, but then only half of the data are available for training. Also, picking a different half may result in substantially different quality measures.

This leads to two guiding principles: First, the majority of the data should be used for training. Second, multiple testing should be done. In general, conducting real-world tests of a model can be difficult. There are, however, strategies that attempt to approximate such testing. Two simple and commonly used strategies are the following: Leave-One-Out and Cross-Validation. In both cases, a dataset is divided into a training dataset and a testing dataset.

**Leave-One-Out**

When fitting the parameters $b$ the more data available in the training set, in all likelihood, the better the fit. The Leave-One-Out strategy takes this to the extreme, by splitting the dataset into a training set of size $m - 1$ and test set of size 1 (e.g., row $t$ in data matrix $X$). From this, a test error can be computed $y_t - b \cdot x_t$. This can be repeated by iteratively letting $t$ range from the first to the last row of data matrix $X$. For certain predictive analytics techniques such as Multiple Linear Regression, there are efficient ways to compute the test $sse$ based on the leverage each point in the training set has [46].

**k-Fold Cross-Validation**

A more generally applicable strategy is called cross-validation, where a dataset is divided into $k$ test datasets. For each test dataset, the corresponding training dataset is all the instances not chosen for that test dataset. A simple way to do this is to let the first test dataset be first $m/k$ rows of matrix $X$, the second be the second $m/k$ rows, etc.
val tsize = m / k // test dataset size
for (l <- 0 until k) {
    x_e = x.slice (l * tsize, ((l+1) * tsize)) // l-th test dataset
    x_ = x.sliceEx (l * tsize, ((l+1) * tsize)) // l-th training dataset
} // for

The model is trained $k$ times using each of the training datasets. The corresponding test dataset is then used to estimate the test sse (or other quality measure such as mse). From each of these samples, a mean, standard deviation and confidence interval may be computed for the test sse. Due to patterns that may exist in the dataset, it is more robust to randomly select each of the test datasets.

Typically, training QoF measures such as $R^2$ will be better than testing QoF measures such as $R^2_{cv}$. Adjusted measures such as $\bar{R}^2$ are intending to more closely follow $R^2_{cv}$ than $R^2$.

### 6.5.11 Collinearity

Consider the matrix-vector equation used for estimating the parameters $\mathbf{b}$ via the minimization of $||\epsilon||$.

$$ y = X\mathbf{b} + \epsilon $$

The parameter/coefficient vector $\mathbf{b} = [b_0, b_1, \ldots, b_k]$ may be viewed as weights on the column vectors in the data/predictor matrix $X$.

$$ y = b_0 \mathbf{1} + b_1 \mathbf{x}_1 + \ldots + b_k \mathbf{x}_k + \epsilon $$

A question arises when two of these column vectors are nearly the same. They will affect and may obfuscate each others’ parameter values.

First, we will examine ways of detecting such problems and then give some remedies. A simple check is to compute the correlation matrix for the column vectors in matrix $X$. High (positive or negative) correlation indicates collinearity.

#### Example Problem

Consider the following data/input matrix $X$ and response vector $y$. This is the same example used for SimpleRegression with new variable $x_2$ added (i.e., $y = b_0 + b_1 x_1 + b_2 x_2 + \epsilon$). The CollinearityTest object allows one to see the effects of increasing the collinearity of features/variables $x_1$ and $x_2$.

```scala
package <your-package>

import scalation.analytics.Regression
import scalation.linalgebra.{MatrixD, VectorD}
import scalation.stat.StatVector.corr

object CollinearityTest extends App
{
    // c x1 x2
    val x = new MatrixD ((4, 3), 1, 1, 1, 
                         1, 2, 2, 
                         1, 3, 3, 
                         // input/data matrix
                         1, 1, 1, 
                         1, 2, 2, 
                         1, 3, 3,

```
val y = VectorD (1, 3, 3, 4) // output/response vector
println (s"x = $x")
println (s"corr (x) = ${corr (x)}") // show correlation matrix
val rg = new Regression (x, y) // create a regression model
rg.train ().eval () // train and then evaluate
println (rg.report) // show brief report
println (rg.summary) // show R-langauge-like summary

} // CollinearityTest

Try changing the value of element $x_{32}$ from 0 to 4 by .5 and observe what happens to the correlation matrix. What effect do these changes have on the parameter vector $b = [b_0, b_1, b_2]$ and how do the first two parameters compare to the regression where the last column of $X$ is removed giving the parameter vector $b = [b_0, b_1]$.

The `corr` function is provided by the `scalation.stat.StatVector` object. For this function, if either column vector has zero variance, when the column vectors are the same, it returns 1.0, otherwise -0.0 (indicating undefined).

### Multi-Collinearity

Even if no particular entry in the correlation matrix is high, a column in the matrix may still be nearly a linear combination of other columns. This is the problem of multi-collinearity. This can be checked by computing the Variance Inflation Factor (VIF) function (or vif in SCALATION). For a particular parameter $b_j$ for the variable/predictor $x_j$, the function is evaluated as follows:

$$vif(b_j) = \frac{1}{1 - R^2(x_j)}$$

where $R^2(x_j)$ is $R^2$ for the regression of variable $x_j$ onto the rest of the predictors. It measures how well the variable $x_j$ (or its column vector $x_{j\cdot}$) can be predicted by all $x_l$ for $l \neq j$. Values above 20 ($R^2(x_j) = 0.95$) are considered by some to be problematic. In particular, the value for parameter $b_j$ may be suspect, since its variance is inflated by $vif(b_j)$.

$$\hat{\sigma}^2(b_j) = \frac{mse}{k \hat{\sigma}^2(x_j)} \cdot vif(b_j)$$

See the exercises for details. Both `corr` and `vif` may be tested in SCALATION using `RegressionTest4`.

One remedy to reduce collinearity/multi-collinearity is to eliminate the variable with the highest $corr/vif$ value. Another is to use regularized regression such as `RidgeRegression` or `LassoRegression`.

### 6.5.12 Feature Selection

There may be predictor variables (features) in the model that contribute little in terms of their contributions to the model’s ability to make predictions. The improvement to $R^2$ may be small and may make $R^2$ or other quality of fit measures worse. An easy way to get a basic understanding is to compute the correlation of each predictor variable $x_j$ ($j^{th}$ column of matrix $X$) with the response vector $y$. A more intuitive way to do this would be to plot the response vector $y$ versus each predictor variable $x_j$. See the exercises for an example.
Ideally, one would like to pick a subset of the $k$ variables that would optimize a selected quality measure. Unfortunately, there are $2^k$ possible subsets to test. Two simple techniques for selecting features are forward selection and backward elimination.

**Forward Selection**

The `forwardSel` method, coded in the `PredictorMat` abstract class, performs *forward selection* by adding the most predictive variable to the existing model, returning the variable to be added and a reference to the new model with the added variable/feature.

```scala
@param cols     the columns of matrix x currently included in the existing model
@param index_q  index of Quality of Fit (QoF) to use for comparing quality

def forwardSel (cols: Set[Int], index_q: Int = index_rSqBar): (Int, PredictorMat) =
```

Selecting the most predictive variable to add boils down to comparing on the basis of a Quality of Fit (QoF) measure. The default is the Adjusted Coefficient of Determination $\overline{R^2}$. The optional argument `index_q` indicates which QoF measure to use (defaults to `index_rSqBar`). To start with a minimal model, set `cols = Set(0)` for an intercept-only model. The method will consider every variable/column `x.range2` not already in `cols` and pick the best one for inclusion.

```scala
for (j <- x.range2 if ! (cols contains j)) {
```

To find the best model, the `forwardSel` method should be called repeatedly while the quality of fit measure is sufficiently improving. This process is automated in the `forwardSelAll` method.

```scala
@param index_q  index of Quality of Fit (QoF) to use for comparing quality
@param cross     whether to include the cross-validation QoF measure

def forwardSelAll (index_q: Int = index_rSqBar, cross: Boolean = true): (Set[Int], Matrix) =
```

The `forwardSelAll` method takes the QoF measure to use as the selection criterion and whether to apply cross-validation as inputs and returns the best collection of features/columns to include in the model as well as the QoF measures for all steps.

To see how $R^2$, $\overline{R^2}$ and $R^2_{cv}$ change with the number of parameters added to the model by `forwardSelAll` method, run the following test code from the `scalation_modeling` module.

```scala
sbt> runMain scalation.analytics.RegressionTest11
```

From the `scalation_1.6` base directory, change directory (cd) to `scalation_modeling` before starting the Simple Build Tool (sbt).

**Backward Elimination**

The `backwardElim` method, coded in the `PredictorMat` abstract class, performs *backward elimination* by removing the least predictive variable from the existing model, returning the variable to eliminate, the new parameter vector and a reference to the new model with the removed variable/feature.
def backwardElim(cols: Set[Int], index_q: Int = index_rSqBar, first: Int = 1):
    (Int, PredictorMat) =

To start with a maximal model, set cols = Set(0, 1, ..., k) for a full model. As with forwardSel, the
index_q optional argument allows one to choose from among the QoF measures. The last parameter first
provides immunity from elimination for any variable/parameter that is less than first (e.g., to ensure that
models include an intercept \(b_0\), set first to one). The method will consider every variable/column from
first until \(x\).dim2 in cols and pick the worst one for elimination.

for (j <- first until x.dim2 if cols contains j) {
    To find the best model, the backwardElim method should be called repeatedly until the quality of fit measure
sufficiently decreases. This process is automated in the backwardElimAll method.

def backwardElimAll(index_q: Int = index_rSqBar, first: Int = 1, cross: Boolean = true):
    (Set[Int], MatriD) =

The backwardElimAll method takes the QoF measure to use as the selection criterion, the index of the first
variable to consider for elimination, and whether to apply cross-validation as inputs and returns the best
collection of features/columns to include in the model as well as the QoF measures for all steps.

An improvement over forward selection and backward elimination, stepwise regression, is presented in
the exercises.

More advanced feature selection techniques include using genetic algorithms to find near optimal subsets
of variables as well as techniques that select variables as part of the parameter estimation process, e.g.,
LassoRegression.

Categorical Variables/Features

For Regression, the variables/features are treated as continuous/ordinal. Some of the variable may be
categorical, where there is no ordering of the values for a categorical variable. In such cases it may be useful
to replace a categorical variable with multiple dummy, binary variables. For details on how to do this, see
the section on ANCOVA.

6.5.13 Regression Problem: Texas Temperatures

Solving a regression problem in SCALATION simply involves creating a data/input matrix \(X \in \mathbb{R}^{m \times n}\) and
a response/output vector \(y \in \mathbb{R}^m\) and then creating a Regression object upon which analyze (i.e., train
and eval) and report methods are called. The Texas Temperature dataset below from [http://www.stat.]
The purpose of the model is to predict average January high temperatures for 16 Texas county weather stations based on their Latitude, Elevation and Longitude.

```scala
// 16 data points: Constant x1 x2 x3
// Lat Elev Long County
val x = new MatrixD((16, 4), 1.0, 29.767, 41.0, 95.367, // Harris
1.0, 32.850, 440.0, 96.850, // Dallas
1.0, 26.933, 25.0, 97.800, // Kennedy
1.0, 31.950, 2851.0, 102.183, // Midland
1.0, 34.800, 3840.0, 102.467, // Deaf Smith
1.0, 33.450, 1461.0, 99.633, // Knox
1.0, 28.700, 815.0, 100.483, // Maverick
1.0, 32.450, 2380.0, 100.533, // Nolan
1.0, 31.800, 3918.0, 106.400, // El Paso
1.0, 34.850, 2040.0, 100.217, // Collington
1.0, 30.867, 3000.0, 102.900, // Pecos
1.0, 36.350, 3693.0, 102.083, // Sherman
1.0, 30.300, 597.0, 97.700, // Travis
1.0, 26.900, 315.0, 99.283, // Zapata
1.0, 28.450, 459.0, 99.217, // Lasalle
1.0, 25.900, 19.0, 97.433) // Cameron

val y = VectorD(56, 48, 60, 46, 38, 46, 53, 46, 44, 41, 47, 36, 52, 60, 56, 62)

val rg = new Regression(x, y) // create a Regression model
rg.analyze() // train and evaluate
println(rg.report) // brief report
println("yp = "+rg.predict()) // predict for all given values
```

The report Method

The **report** method returns the following basic information: (1) the values of the hyper-parameters \( hp \) (used for controlling the model/optimizer), (2) the values of the parameters \( b \), and (3) several Quality of Fit measures \( qof \).

```
REPORT

hparameter hp = null
parameter b = VectorD(151.298, -1.99323, -0.000955478, -0.384710)
fitMap qof = Map(rSq -> 0.99192, rBarSq -> 0.98990, sst -> 941.93750, sse -> 7.60949,
 mse0 -> 0.47559, rmse -> 0.68963, mae -> 0.53135, dfm -> 3.00000,
df -> 12.00000, fStat -> 491.13802, aic -> -8.75748, bic -> -5.66713,
mape -> 1.09599, smape -> 1.09478)
```

The plot below shows the results from running the **SCALATiON** Regression Model in terms of actual \((y)\) vs. predicted \((yp)\) response vectors.
The summary Method

More details about the parameters/coefficients including standard errors, $t$-values, $p$-values, and Variance Inflation Factors (VIFs) are shown by the summary method.

```
println (rg.summary)
```

For the Texas Temperatures dataset it provides the following information: The Estimate is the value assigned to the parameter for the given Var. The Std. Error, $t$-value, $p$-value and VIF are also given.

```
fname = Array(x0 = intercept, x1 = Lat, x2 = Elev, x3 = Long)
SUMMARY

Parameters/Coefficients:
| Var | Estimate   | Std. Error | $t$ value | Pr($>|t|)$ | VIF  |
|-----|------------|------------|-----------|------------|------|
| x0  | 151.297616 | 25.133361  | 6.019792  | 0.000060   | NA   |
| x1  | -1.993228  | 0.136390   | -14.614194| 0.000000   | 4.228079 |
| x2  | -0.000955  | 0.000568   | -1.683440 | 0.118102   | 16.481808 |
| x3  | -0.384710  | 0.228584   | -1.683018 | 0.118185   | 9.432463 |

Residual standard error: 0.796319 on 12.0 degrees of freedom
Multiple R-squared: 0.991921, Adjusted R-squared: 0.989902
F-statistic: 491.138015 on 3.0 and 12.0 DF, p-value: 8.089084957418891E-13
```

Given the following assumptions: (1) $\epsilon \sim D(0, \sigma^2 I)$ for some distribution $D$ and (2) for each column $j$, $\epsilon$ and $x_j$ are independent, the covariance matrix of the parameter vector $b$ is

$$
\mathbb{C}[b] = \sigma^2 (X^T X)^{-1}
$$

(6.52)

See [85] for a derivation. Using $\hat{\sigma}^2$ as an estimate for $\sigma^2$, 161
\[ \hat{\sigma}^2 = \frac{\varepsilon \cdot \varepsilon}{df} = mse \]  

(6.53)

the standard deviation (or standard error) of the \( j^{th} \) parameter/coefficient may be given as the square root of the \( j^{th} \) diagonal element of the covariance matrix.

\[ \hat{\sigma}_{b_j} = \hat{\sigma} \sqrt{(X^T X)^{-1}}_{jj} \]  

(6.54)

The corresponding \( t \)-value is simply the parameter value divided by its standard error, which indicates how many standard deviation units it is away from zero. The farther way from zero the more significant (or more important to the model) the parameter is.

\[ t(b_j) = \frac{b_j}{\hat{\sigma}_{b_j}} \]  

(6.55)

When the error distribution is Normal, then \( t(b_j) \) follows the Student’s \( t \) Distribution. For example, the pdf for the Student’s \( t \) Distribution with \( df = \nu = 2 \) degrees of freedom is shown in the figure below (the \( t \) Distribution approaches the Normal Distribution as \( \nu \) increases).

\[
f_y(y) = \frac{\Gamma(\frac{\nu+1}{2})}{\sqrt{\nu\pi}}\left(1 + \frac{y^2}{\nu}\right)^{-\frac{\nu+1}{2}} = \frac{\Gamma(3/2)}{\sqrt{2\pi}1} \left(1 + \frac{y^2}{2}\right)^{-3/2} = \frac{1}{2\sqrt{2}} \left(1 + \frac{y^2}{2}\right)^{-3/2}
\]  

(6.56)

pdf for Student’s \( t \) Distribution (blue) vs. Normal (green)

The corresponding \( p \)-value \( P(|y| > t) \) measures how significant the \( t \)-value is, e.g.,

\[
F_y(-1.683018) = 0.0590926 \\
P(|y| > -1.683018) = 2F_y(-1.683018)) = 0.118185 \quad \text{for} \quad \nu = df = 12
\]

Typically, the \( t \)-value is only considered significant if is in the tails of the Student’s \( t \) distribution. The farther out in the tails, the less likely for the parameter to be non-zero (and hence be part of the model) simply by chance. The \( p \)-value measures the risk (chance of being wrong) in including parameter \( b_j \) and therefore variable \( x_j \) in the model.
The predict Method

Finally, a given new data vector $z$, the predict method may be used to predict its response value.

```scala
val z = VectorD (1.0, 30.0, 1000.0, 100.0)
println (s"predict ($z) = ${rg.predict (z)}")
```

Feature Selection

Feature selection (or Variable Selection) may be carried out by using either forwardSel or backwardElim. These methods add or remove one variable at a time. To iteratively add or remove, the following methods may be called.

```scala
rg.forwardSelAll (cross = false)
rg.backwardElimAll (cross = false)
```

The default criterion for choosing which variable to add/remove is Adjusted $R^2$. It may be changed via the index.q parameter to the methods (see the Fit class for the possible values for this parameter). Note: The cross-validation is turned off (cross = false) due to the small size of the dataset.


Regression Class

Class Methods:

```scala
class Regression (x: MatriD, y: VectoD, fname_, Strings = null,
                    hparam: HyperParameter = null, technique: RegTechnique = QR)
                  extends PredictorMat (x, y, fname_, hparam)

    def train (x_: MatriD = x, y_: VectoD = y): Regression =
    override def eval (x_e: MatriD = x, y_e: VectoD = y): Regression =
    override def predict (z: MatriD = x): VectoD = z * b
    def buildModel (x_cols: MatriD): Regression =
```
6.5.14 Exercises

1. For Exercise 1 from the last section, compute \( A = X^T X \) and \( z = X^T y \). Now solve the following linear systems of equations for \( b \).

\[
Ab = z
\]

2. Gradient descent can be used for Multiple Linear Regression as well. For gradient descent, pick a starting point \( b^0 \), compute the gradient of the loss function \( \nabla h \) and move \(-\eta \nabla h\) from \( b^0 \) where \( \eta \) is the learning rate. Write a Scala program that repeats this for several iterations for the above data. What is happening to the value of the loss function \( h \).

\[
\nabla h = -X^T (y - Xb)
\]

Substituting \( e = y - Xb \), allows \( \nabla h \) to be written as

\[-X^T e\]

Starting with data matrix \( x \), response vector \( y \) and parameter vector \( b \), in SCALATION, the calculations become

```scala
val yp = x * b // y predicted
val e = y - yp // error
val g = x.t * e // - gradient
b += g * eta // update parameter b
val h = 0.5 * (e dot e) // half the sum of squared errors
```

Unless the dataset is normalized, finding an appropriate learning rate \( \eta \) may be difficult. See the MatrixTransform object for details. Do this for the Blood Pressure ExampleBPressure dataset. Try using another dataset.

3. Consider the relationships between the predictor variables and the response variable in the AutoMPG dataset. This is a well know dataset that is available at multiple websites including the UCI Machine Learning Repository [http://archive.ics.uci.edu/ml/datasets/Auto+MPG](http://archive.ics.uci.edu/ml/datasets/Auto+MPG). The response variable is the miles per gallon (\( mpg \): continuous) while the predictor variables are cylinders: multi-valued discrete, displacement: continuous, horsepower: continuous, weight: continuous, acceleration: continuous, model_year: multi-valued discrete, origin: multi-valued discrete, and car_name: string (unique for each instance). Since the car_name is unique and obviously not causal, this variable is eliminated, leaving seven predictor variables. First compute the correlations between \( mpg \) (vector \( y \)) and the seven predictor variables (each column vector \( x_j \) in matrix \( X \)).

```scala
val correlation = y corr x_j
```

and then plot \( mpg \) versus each of the predictor variables. The source code for this example is at [http://www.cs.uga.edu/~jam/scalation_1.6/src/main/scala/scalation/analytics/ExampleAutoMPG.scala](http://www.cs.uga.edu/~jam/scalation_1.6/src/main/scala/scalation/analytics/ExampleAutoMPG.scala).
Alternatively, a .csv file containing the AutoMPG dataset may be read into a relation called `auto_tab` from which data matrix `x` and response vector `y` may be produced. If the dataset has missing values, they may be replaced using a spreadsheet or using the techniques discusses in the Data Preprocessing Chapter.

```scala
val auto_tab = Relation(BASE_DIR + "auto-mpg.csv", "auto_mpg", null, -1)
val (x, y) = auto_tab.toMatriDD (1 to 6, 0)
println (s"x = $x")
println (s"y = $y")
```

4. Apply Regression analysis on the AutoMPG dataset. Compare with results of applying the `NullModel`, `SimplerRegression` and `SimpleRegression`. Try using `SimplerRegression` and `SimpleRegression` with different predictor variables for these models. How does their $R^2$ values compare to the correlation analysis done in the previous exercise?

5. Examine the collinearity and multi-collinearity of the column vectors in the AutoMPG dataset.

6. For the AutoMPG dataset, repeatedly call the `backwardElim` method to remove the predictor variable that contributes the least to the model. Show how the various quality of fit (QoF) measures change as variables are eliminated. Do the same for the `forwardSel` method. Using $R^2$, select the best models from the forward and backward approaches. Are they the same?

7. Compare model assessment and model validation. Compute $sse$, $mse$ and $R^2$ for the full and best AutoMPG models trained on the entire data set. Compare this with the results of Leave-One-Out, 5-fold Cross-Validation and 10-fold Cross-Validation.

8. The variance of the estimate of parameter $b_j$ may be estimated as follows:

$$\hat{\sigma}^2(b_j) = \frac{mse}{k\hat{\sigma}^2(x_j)} \cdot vif(b_j)$$

Derive this formula. The standard error is the square root of this value. Use the estimate for $b_j$ and its standard error to compute a $t$-value and $p$-value for the estimate. Run the AutoMPG model and explain these values produced by the `summary` method.

9. **Singular Value Decomposition Technique.** In cases where the rank of the data/input matrix $X$ is not full or its multi-collinearity is high, a useful technique to solve for the parameters of the model is Singular Value Decomposition (SVD). Based on the derivation given in [http://www.ime.unicamp.br/~marianar/MI602/material/20extra/svd-regression-analysis.pdf](http://www.ime.unicamp.br/~marianar/MI602/material/20extra/svd-regression-analysis.pdf), we start with the equation estimating $y$ as the product of the data matrix $X$ and the parameter vector $b$.

$$y = Xb$$

We then perform a singular value decomposition on the $m$-by-$n$ matrix $X$

$$X = U\Sigma V^T$$
where in the full-rank case, $U$ is an $m$-by-$n$ orthogonal matrix, $\Sigma$ is an $n$-by-$n$ diagonal matrix of singular values, and $V^T$ is an $n$-by-$n$ orthogonal matrix. The $r = \text{rank}(A)$ equals the number of nonzero singular values in $\Sigma$, so in general, $U$ is $m$-by-$r$, $\Sigma$ is $r$-by-$r$, and $V^T$ is $r$-by-$n$. The singular values are the square roots of the nonzero eigenvalues of $X^TX$. Substituting for $X$ yields

$$y = U\Sigma V^T b$$

Defining $d = \Sigma V^T b$, we may write

$$y = Ud$$

This can be viewed as a estimating equation where $X$ is replaced with $U$ and $b$ is replaced with $d$. Consequently, a least squares solution for the alternate parameter vector $d$ is given by

$$d = (U^T U)^{-1} U^T y$$

Since $U^T U = I$, this reduces to

$$d = U^T y$$

If $\text{rank}(A) = n$ (full-rank), then the conventional parameters $b$ may be obtained as follows:

$$b = V \Sigma^{-1} d$$

where $\Sigma^{-1}$ is a diagonal matrix where elements on the main diagonal are the reciprocals of the singular values.

10. **Stepwise Regression.** An improvement over Forward Selection and Backward Elimination is Stepwise Regression. Start with no variables in the model and add one variable that improves the selection criterion the most. Add the second best variable for step two. After the second step determine whether it is better to add or remove a variable. Continue in this fashion until no improvement in the selection criterion is found. For Forward Selection and Backward Elimination it may instructive to continue all the way to the end (all variables for forward/no variables for backward).

Stepwise regression may lead to coincidental relationships being included in the model, particularly if a $t$-test is the basis of inclusion or a penalty-free QoF measure such as $R^2$ is used. Typically, this approach is used when there a penalty for having extra variables/parameters, e.g., $R^2$ adjusted $\bar{R}^2$, $R^2$ cross-validation $R^2_{cv}$ or Akaike Information Criterion (AIC). See the section on Maximum Likelihood Estimation for a definition of AIC. Alternatives to Stepwise Regression include Lasso Regression ($\ell_1$ regularization) and to a lesser extent Ridge Regression ($\ell_2$ regularization).

Write ScalaTion code for a `stepRegression` method and `stepRegressionAll` method and redo exercise 6 using all three: Forward Selection, Backward Elimination, and Stepwise Regression with all four criteria: $R^2$, $\bar{R}^2$, $R^2_{cv}$, and AIC. Plot the curve for each criterion, determine the best number of variables and what these variables are. Compare the four criteria.
As part of a larger project compare this form of feature selection with that provided by Ridge Regression and Lasso Regression. See the next two sections.

Now add features including quadratic terms and dummy variables to the model using QuadRegression, QuadXRegression, CubicRegression, CubicXRegression, and ANCOVA. See the subsequent sections.

In addition to the AutoMPG dataset, use the Concrete dataset and three more datasets from UCI Machine Learning Repository. The UCI datasets should have more instances \( m \) and variables \( n \) than the first two datasets. The testing should also be done in R or Python.

11. **Regression as Projection.** Consider the following six vectors/points in 3D space where the response variable \( z \) is a linear function of predictor variables \( x \) and \( y \).

\[
\begin{array}{c|c|c|c}
\hline
x & y & z \\
\hline
1 & 1 & 2.8 \\
1 & 2 & 4.2 \\
1 & 3 & 4.8 \\
2 & 1 & 5.3 \\
2 & 2 & 5.5 \\
2 & 3 & 6.5 \\
\hline
\end{array}
\]

Consider a regression model equation with no intercept.

\[
z = b_0x + b_1y + \epsilon
\]

Determine the plane (response surface) that these six points are projected onto.

\[
\hat{z} = b_0x + b_1y
\]

For this problem, the number of instances \( m = 6 \) and the number of parameters/predictor variables \( n = 2 \). Determine the number of degrees of freedom for the model \( df_m \) and the number of degrees of freedom for the residuals/errors \( df \).

12. State and explain the conditions required for the Ordinary Least Squares (OLS) estimate of parameter vector \( \mathbf{b} \) for multiple linear regression to be B.L.U.E. See the Gauss-Markov Theorem.
6.5.15 Further Reading

1. Introduction to Linear Regression Analysis, 5th Edition [68]
6.6 Ridge Regression

The RidgeRegression class supports multiple linear ridge regression. As with Regression, the predictor variables \( x \) are multi-dimensional \([x_1, \ldots, x_k]\), as are the parameters \( b = [b_1, \ldots, b_k] \). Ridge regression adds a penalty based on the \( \ell_2 \) norm of the parameters \( b \) to reduce the chance of them taking on large values that may lead to less robust models.

The penalty holds down the values of the parameters and this may result in several advantages: (1) better out-of-sample (e.g., cross-validation) quality of fit, (2) reduced impact of multi-collinearity, (3) turn singular matrices, non-singular, and to a limited extent (4) eliminate features/predictor variables from the model.

The penalty is not to be included on the intercept parameter \( b_0 \), as this would shift predictions in a way that would adversely affect the quality of the model. See the exercise on scale invariance.

6.6.1 Model Equation

Centering of the data allows the intercept to be removed from the model. The combined centering on both the predictor variables and the response variable takes care of the intercept, so it is not included in the model. Thus, the goal is to fit the parameter vector \( b \) in the model/regression equation,

\[
y = b \cdot x + \epsilon = b_1 x_1 + \cdots + b_k x_k + \epsilon
\]

where \( \epsilon \) represents the residuals (the part not explained by the model).

6.6.2 Training

Centering the dataset \((X, y)\) has the following effects: First, when the \( X \) matrix is centered, the intercept \( b_0 = \mu_y \). Second, when \( y \) is centered, \( \mu_y \) becomes zero, implying \( b_0 = 0 \). To rescale back to the original response values, \( \mu_y \) can be added back during prediction. Therefore, both the data/input matrix \( X \) and the response/output vector \( y \) should be *centered* (zero mean).

The regularization of the model adds an \( \ell_2 \)-penalty on the parameters \( b \). The objective function to minimize is now the loss function \( h(b) = \frac{1}{2} \text{sse} \) plus the \( \ell_2 \)-penalty.

\[
f_{\text{obj}} = h(b) + \frac{1}{2} \lambda \|b\|^2 = \frac{1}{2} \epsilon \cdot \epsilon + \frac{1}{2} \lambda b \cdot b
\]

where \( \lambda \) is the shrinkage parameter. A large value for \( \lambda \) will drive the parameters \( b \) toward zero, while a small value can help stabilize the model (e.g., for nearly singular matrices or high multi-collinearity).

\[
f_{\text{obj}} = \frac{1}{2} (y - Xb) \cdot (y - Xb) + \frac{1}{2} \lambda b \cdot b
\]

6.6.3 Optimization

Fortunately, the quadratic nature of the penalty function allows it to be combined easily with the quadratic error terms, so that matrix factorization can still be used for finding optimal values for parameters.

Taking the gradient of the objective function \( f_{\text{obj}} \) with respect to \( b \) and then setting it equal to zero yields
\[-X^\top (y - Xb) + \lambda b = 0\]
\[-X^\top y + (X^\top X)b + \lambda b = 0\]
\[(X^\top X)b + \lambda b = X^\top y\]

Since \( \lambda b = \lambda Ib \) where \( I \) is the \( n \)-by-\( n \) identity matrix, we may write

\[(X^\top X + \lambda I) b = X^\top y\]  \hspace{1cm} (6.60)

Matrix factorization may now be used to solve for the parameters \( b \) in the modified Normal Equations. For example, use of matrix inversion yields,

\[b = (X^\top X + \lambda I)^{-1} X^\top y\]  \hspace{1cm} (6.61)

For Cholesky factorization, one may compute \( X^\top X \) and simply add \( \lambda \) to each of the diagonal elements (i.e, along the ridge). QR and SVD factorizations require similar, but slightly more complicated, modifications. Note, use of SVD can improve the efficiency of searching for an optimal value for \( \lambda \).

### 6.6.4 Centering

Before creating a RidgeRegression model, the \( X \) data matrix and the \( y \) response vector should be centered. This is accomplished by subtracting the means (vector of column means for \( X \) and a mean value for \( y \)).

```scala
val mu_x = x.mean // columnwise mean of x
val mu_y = y.mean // mean of y
val x_c = x - mu_x // centered x (columnwise)
val y_c = y - mu_y // centered y
```

The centered matrix \( x_c \) and center vector \( y_c \) are then passed into the RidgeRegression constructor.

```scala
val rrg = new RidgeRegression (x_c, y_c)
rrg.analyze ()
println (rrg.report)
```

Now, when making predictions, the new data vector \( z \) needs to be centered by subtracting \( \mu_x \). Then the predict method is called, after which the mean of \( y \) is added.

```scala
val z_c = z - mu_x // center z first
yp = rrg.predict (z_c) + mu_y // predict z_c and add y's mean
println (s"predict ($z) = $yp")
```

### 6.6.5 The \( \lambda \) Hyper-parameter

The value for \( \lambda \) can be user specified (typically a small value) or chosen by a method like findLambda. It finds a roughly optimal value for the shrinkage parameter \( \lambda \) based on the cross-validated sum of squared errors \( \text{sse}_{cv} \). The search starts with the low default value for \( \lambda \) and then doubles it with each iteration, returning the minimizing \( \lambda \) and its corresponding cross-validated \( \text{sse} \). A more precise search could be used to provide a better value for \( \lambda \).
```scala
def findLambda: (Double, Double) =
{
  var l = lambda // start with the small default value
  var l_best = l
  var sse = Double.MaxValue
  for (i <- 0 to 20) {
    RidgeRegression.hp("lambda") = l
    val rrg = new RidgeRegression (x, y)
    val stats = rrg.crossValidate ()
    val sse2 = stats(Fit.index_sse).mean
    banner (s"RidgeRegression with lambda = \${rrg.lambda_} has sse = \$sse2")
    if (sse2 < sse) { sse = sse2; l_best = l }
    l *= 2
  }
  (l_best, sse) // best lambda and its sse_cv
}

// 5 data points: x_0 x_1
val x = new MatrixD ((5, 2), 36.0, 66.0, // 5-by-2 matrix data matrix
                    37.0, 68.0,
                    47.0, 64.0,
                    32.0, 53.0,
                    1.0, 101.0)
val y = VectorD (745.0, 895.0, 442.0, 440.0, 1598.0) // 5-dim response vector

6.6.6 Comparing RidgeRegression with Regression

This subsection compares the results of RidgeRegression with those of Regression by examining the estimated parameter vectors, the quality of fit, predictions made, and comparing the summary information.

First, create a Regression model with an intercept and produce a summary.

  banner ("Regression")
  val ox = VectorD.one (y.dim) +^: x // prepend a column of all 1’s
  val rg = new Regression (ox, y) // create a Regression model
  println (rg.analyze ().report) // analyze and report

Second, create a RidgeRegression model using the centered data

  banner ("RidgeRegression")
  val mu_x = x.mean // columnwise mean of x
  val mu_y = y.mean // mean of y
  val x_c = x - mu_x // centered x (columnwise)
  val y_c = y - mu_y // centered y
  val rrg = new RidgeRegression (x_c, y_c) // create a Ridge Regression model
  println (rrg.analyze ().report) // analyze and report

Third, predict a value for new input vector z using each model.
```
val z = VectorD(20.0, 80.0) // new instance to predict
val _1z = VectorD.++(1.0, z) // prepend 1 to z
val z_c = z - mu_x // center z
println(s"rg.predict ($z) = ${rg.predict(_1z)}") // predict using _1z
println(s"rrg.predict ($z) = ${rrg.predict(z_c + mu_y)}") // predict using z_c and add y's mean

The summary information for Regression is shown below.

println(rg.summary)

SUMMARY
Parameters/Coefficients:
| Var | Estimate | Std. Error | t value | Pr(>|t|) | VIF |
|-----|----------|------------|---------|---------|-----|
| x0  | -281.426985 | 835.349154 | -0.336897 | 0.768262 | NA  |
| x1  | -7.611030 | 8.722908 | -0.872534 | 0.474922 | 3.653976 |
| x2  | 19.010291 | 8.423716 | 2.256758 | 0.152633 | 3.653976 |

Residual standard error: 159.206002 on 2.0 degrees of freedom
Multiple R-squared: 0.943907, Adjusted R-squared: 0.887815
F-statistic: 16.827632701228243 on 2.0 and 2.0 DF, p-value: 0.05609269703717268

The summary information for RidgeRegression is shown below.

println(rrg.summary)

SUMMARY
Parameters/Coefficients:
| Var | Estimate | Std. Error | t value | Pr(>|t|) | VIF |
|-----|----------|------------|---------|---------|-----|
| x0  | -7.611271 | 8.722908 | -0.872561 | 0.474910 | NA  |
| x1  | 19.009947 | 8.423716 | 2.256717 | 0.152638 | 3.653976 |

Residual standard error: 159.206002 on 2.0 degrees of freedom
Multiple R-squared: 0.943907, Adjusted R-squared: 0.887815
F-statistic: 16.827632684676626 on 2.0 and 2.0 DF, p-value: 0.056092697089250576

Notice there is very little difference between the two models. Try increasing the value of the shrinkage hyper-parameter $\lambda$ beyond its default value of 0.01. This example can be run as follows:

$ cd scalation_1.6/scalation_modeling
$ sbt
sbt> runMain scalation.analytics.RidgeRegressionTest
RidgeRegression Class

Class Methods:

@param x  the centered data m-by-n matrix, NOT augmented with a column of 1's
@param y  the centered response m-vector
@param fname_ the feature/variable names
@param hparam the shrinkage hyper-parameter, lambda (0 => OLS) in penalty
@param technique the technique used to solve for b in (x.t*x + lambda*I)*b = x.t*y

class RidgeRegression (x: MatriD, y: VectoD, fname_ : Strings = null,
  hparam: HyperParameter = RidgeRegression.hp,
  technique: RegTechnique = Cholesky)
  extends PredictorMat (x, y, fname_, hparam)

resetDF (x.dim2, x.dim1 - x.dim2 - 1) // no intercept => correct Degrees of Freedom (DoF)

def train (x_ : MatriD, y_ : VectoD): RidgeRegression =
def gcv (xx: MatriD = x, yy: VectoD = y): Double =
def buildModel (x_cols: MatriD): RidgeRegression =

6.6.7 Exercises

1. Based on the example given in this section, try increasing the value of the hyper-parameter $\lambda$ and examine its effect on the parameter vector $b$, the quality of fit and predictions made.

    import RidgeRegression.hp

    println (s"hp = $hp")
    val hp2 = hp.updateReturn ("lambda", 1.0)
    println (s"hp2 = $hp2")

Alternatively,

    hp("lambda") = 1.0

See the HyperParameter class in the scalation.analytics package for details.

2. For the AutoMPG dataset, use the findLambda method find a value for $\lambda$ that roughly minimizes out-of-sample $sse_cv$ based on using the crossValidate method. Plot $sse_cv$ vs. $\lambda$.

3. Why is it important to center (zero mean) both the data matrix $X$ and the response vector $y$? What is scale invariance and how does it relate to centering the data?
4. The Degrees of Freedom (DoF) used in SCALATion’s RidgeRegression class is approximate. As the shrinkage parameter $\lambda$ increases the effective DoF should be used instead. A general definition of effective DoF is the trace $\text{tr}$ of the hat matrix $H = X(X^T X + \lambda I)^{-1} X^T$

$$dfeff = \text{tr}(H)$$

Read [17] and explain the difference between DoF and effective DoF for Ridge Regression.

5. A matrix that close to singularity is said to be ill-conditioned. The condition number $\kappa$ of a matrix $A$ (e.g., $A = X^T X$) is defined as follows:

$$\kappa = \|A\| \|A^{-1}\| \geq 1$$

When $\kappa$ becomes large the matrix is considered to be ill-conditioned. In such cases, it is recommended to use QR or SVD Factorization for Least-Squares Regression [22]. Compute the condition number for of $X^T X$ for various datasets.

6. For over-parameterized (or under-determined) regression where the $n > m$ (number of parameters > number of instances) it is common to seek a min-norm solution.

$$\min_b \{\|b\|_2^2 \mid y = Xb\}$$

Use [http://people.csail.mit.edu/bkph/articles/Pseudo_Inverse.pdf](http://people.csail.mit.edu/bkph/articles/Pseudo_Inverse.pdf) to derive a solution for the parameters $b$.

$$b = X^T(XX^T)^{-1}y$$

Compare the use of regression and ridge regression of such problems.
6.7 Lasso Regression

The LassoRegression class supports multiple linear regression using the Least absolute shrinkage and selection operator (Lasso) that constrains the values of the $b$ parameters and effectively sets those with low impact to zero (thereby deselecting such variables/features). Rather than using an $\ell_2$-penalty (Euclidean norm) like RidgeRegression, it uses an $\ell_1$-penalty (Manhattan norm). In RidgeRegression when $b_j$ approaches zero, $b_j^2$ becomes very small and has little effect on the penalty. For LassoRegression, the effect based on $|b_j|$ will be larger, so it is more likely to set parameters to zero. See section 6.2.2 in [46] for a more detailed explanation on how LassoRegression can eliminate a variable/feature by setting its parameter/coefficient to zero.

6.7.1 Model Equation

As with Regression, the goal is to fit the parameter vector $b \in \mathbb{R}^n (k = n - 1)$ in the model/regression equation,

$$y = b \cdot x + \epsilon = b_0 + b_1 x_1 + \ldots + b_k x_k + \epsilon$$  \hspace{1cm} (6.62)

where $\epsilon$ represents the residuals (the part not explained by the model). See the exercise that considers whether to include the intercept $b_0$ in the shrinkage.

6.7.2 Training

The regularization of the model adds an $\ell_1$-penalty on the parameters $b$. The objective function to minimize is now the loss function $h(b) = \frac{1}{2} sse$ plus the penalty.

$$f_{obj} = \frac{1}{2} sse + \lambda \|b\|_1 = \frac{1}{2} \|\epsilon\|_2^2 + \lambda \|b\|_1$$  \hspace{1cm} (6.63)

where $\lambda$ is the shrinkage parameter. Substituting $\epsilon = y - Xb$ yields,

$$f_{obj} = \frac{1}{2} \|y - Xb\|_2^2 + \lambda \|b\|_1$$  \hspace{1cm} (6.64)

Replacing the norms with dot products gives,

$$f_{obj} = \frac{1}{2} (y - Xb) \cdot (y - Xb) + \lambda 1 \cdot |b|$$  \hspace{1cm} (6.65)

Although similar to the $\ell_2$-penalty used in Ridge Regression, it may often be more effective. Still, the $\ell_1$-penalty for Lasso has a disadvantage that the absolute values in the $\ell_1$ norm make the objective function non-differentiable.

$$\lambda 1 \cdot |b| = \lambda \sum_{j=0}^{k} |b_j|$$  \hspace{1cm} (6.66)

Therefore, the straightforward strategy of setting the gradient equal to zero to develop appropriate modified Normal Equations that allow the parameters to be determined by matrix factorization will no longer works. Instead, the objective function needs to be minimized using a search based optimization algorithm.
6.7.3 Optimization Strategies

There are multiple optimization algorithms that can be applied for parameter estimation in Lasso Regression.

Coordinate Descent

Coordinate Descent is analogous to gradient descent, except that after the gradient is computed, movement is only in single direction at a time. ...

Alternating Direction Method of Multipliers

SCALATION also uses the Alternating Direction Method of Multipliers (ADMM) [12] algorithm to optimize the \( b \) parameter vector. The algorithm for using ADMM for Lasso Regression is outlined in section 6.4 of Boyd [12] (https://stanford.edu/~boyd/papers/pdf/admm_distr_stats.pdf). Optimization problems in ADMM form separate the objective function into two parts \( f \) and \( g \).

\[
\min f(b) + g(z) \quad \text{subject to} \quad b - z = 0 \tag{6.67}
\]

For Lasso Regression, the \( f \) function will capture the loss function \( \frac{1}{2} \text{sse} \), while the \( g \) function will capture the \( \ell_1 \) regularization, i.e.,

\[
f(b) = \frac{1}{2} \|y - Xb\|_2^2, \quad g(z) = \lambda \|z\|_1 \tag{6.68}
\]

Introducing \( z \) allows the functions to be separated, while the constraint keeps \( z \) and \( b \) close. Therefore, the iterative step in the ADMM optimization algorithm becomes

\[
\begin{align*}
b &= (X^T X + \rho I)^{-1}(X^T y + \rho(z - u)) \\
z &= S_{\lambda/\rho}(b + u) \\
u &= u + b - z
\end{align*}
\]

where \( u \) is the vector of Lagrange multipliers and \( S_\lambda \) is the soft thresholding function.

\[
S_\lambda(\zeta) = \text{sign}(\zeta) (|\zeta| - \lambda)_+ \tag{6.69}
\]

Note \((a)_+ = \max(a, 0)\).

The details of ADMM for Lasso Regression are left as an exercise. In addition to Boyd, see [39] and see scalation.minima.LassoAdmm for coding details.

6.7.4 The \( \lambda \) Hyper-parameter

The shrinkage parameter \( \lambda \) can be tuned to control feature selection. The larger the value of \( \lambda \), the more features (predictor variables) whose parameters/coefficients will be set to zero. The \texttt{findLambda} method may be used to find a value \texttt{lambda} that improves the cross-validated sum of squared errors \texttt{sse_cv}.

```scala
def findLambda: (Double, Double) = {
  var l = lambda
  var l_best = l
  var sse = Double.MaxValue
  ...
```

176
for (i <- 0 to 20) {
    LassoRegression.hp("lambda") = 1
    val rrg = new LassoRegression (x, y)
    val stats = rrg.crossValidate ()
    val sse2 = stats(Fit.index_sse).mean
    banner (s"LassoRegression with lambda = \$\{rrg.lambda\}_ has sse = \$sse2")
    if (sse2 < sse) { sse = sse2; l_best = l }
    if (DEBUG) showQofStatTable (stats)
    l *= 2
} // for
1 **= 2
} // findLambda

As the default value for the shrinkage/penalty parameter $\lambda$ is very small, the optimal solution will be close to the Ordinary Least Squares (OLS) solution shown in green at $b = [b_1, b_2] = [3, 1]$ in Figure 6.3. Increasing the penalty parameter will pull the optimal $b$ towards the origin. At any given point in the plane, the objective function is the sum of the loss function $h(b)$ and the penalty function $p(b)$. The contours in blue show points of equal height for the penalty function, while those in black show the same for the loss function. Suppose the point $[2, 0]$ is this penalized optimum. This would mean that moving toward the origin would be non-productive, as the increase in the loss would exceed the drop in the penalty. On the other hand, moving toward $[3, 1]$ would be non-productive as the increase in the penalty would exceed the drop in the loss. Notice in this case, that the penalty has pulled the $b_1$ parameter to zero (an example of feature selection). Ridge regression will be less likely to pull a parameter to zero, as its contours are circles rather than diamonds. Lasso regression’s contours have sharp points on the axis which thereby increase the chance of intersecting a loss contour on an axis.

Figure 6.3: Contour Lines for Lasso Regression
6.7.5 Regularized and Robust Regression

Regularized and Robust Regression are useful in many cases including high-dimensional data, correlated data, non-normal data and data with outliers [37]. These techniques work by adding a $\ell_1$ and/or $\ell_2$-penalty terms to shrink the parameters and/or changing from an $\ell_2$ to $\ell_1$ loss function. Modeling techniques include Ridge, Lasso, Elastic Nets, Least Absolute Deviation (LAD) and Adaptive LAD [37].

LassoRegression Class

Class Methods:

@param x the data/input m-by-n matrix
@param y the response/output m-vector
@param fname_ the feature/variable names
@param hparam the shrinkage hyper-parameter, lambda (0 => OLS) in the penalty term

'velambda * b dot b'

class LassoRegression (x: MatriD, y: VectoD, fname_ : Strings = null,
                        hparam: HyperParameter = LassoRegression.hp)
                      extends PredictorMat (x, y, fname_, hparam)

def train (x_: MatriD = x, y_: VectoD = y): LassoRegression =
def buildModel (x_cols: MatriD): LassoRegression =

6.7.6 Exercises

1. Compare the results of LassoRegression with those of Regression and RidgeRegression. Examine the parameter vectors, quality of fit and predictions made.

   // 5 data points: one x_0 x_1
   val x = new MatrixD ((5, 3), 1.0, 36.0, 66.0, // 5-by-3 matrix
                        1.0, 37.0, 68.0,
                        1.0, 47.0, 64.0,
                        1.0, 32.0, 53.0,
                        1.0, 1.0, 101.0)

   val y = VectorD (745.0, 895.0, 442.0, 440.0, 1598.0)
   val z = VectorD (1.0, 20.0, 80.0)

   // Create a LassoRegression model

   val lrg = new LassoRegression (x, y)

   // Predict a value for new input vector z using each model.
2. Based on the last exercise, try increasing the value of the hyper-parameter $\lambda$ and examine its effect on the parameter vector $\mathbf{b}$, the quality of fit and predictions made.

```scala
import LassoRegression.hp

println (s"hp = $hp")
val hp2 = hp.updateReturn ("lambda", 1.0)
println (s"hp2 = $hp2")
```

3. Using the above dataset and the AutoMPG dataset, determine the effects of (a) **centering** the data $(\mu = 0)$, (b) **standardizing** the data $(\mu = 0, \sigma = 1)$.

```scala
import MatrixTransforms._

val x_n = normalize (x, (mu_x, sig_x))
val y_n = y.standardize
```

4. Explain how the ADMM Optimization Algorithm works.

5. Compare **LassoRegression** the with **Regression** that uses forward selection or backward elimination for feature selection. What are the advantages and disadvantages of each for feature selection.

6. Compare **LassoRegression** the with **Regression** on the AutoMPG dataset. Specifically, compare the quality of fit measures as well as how well feature selection works.

7. **Elastic Nets** combine both $\ell_2$ and $\ell_1$ penalties to try to combine the best features of both **RidgeRegression** and **LassoRegression**. Elastic Nets naturally includes two shrinkage parameters, $\lambda_1$ and $\lambda_2$. Is the additional complexity worth the benefits?

### 6.7.7 Further Reading

1. Distributed Optimization and Statistical Learning via the Alternating Direction Method of Multipliers [12]
2. The Application of Alternating Direction Method of Multipliers on $\ell_1$-norms Problems [39]
3. Feature Selection Using LASSO [29]
6.8 Quadratic Regression

The QuadRegression class extends the Regression class by automatically adding quadratic terms into the model. It can often be the case that the response variable \( y \) will have a nonlinear relationship with one more of the predictor variable \( x_j \). The simplest such nonlinear relationship is a quadratic relationship. Looking at a plot of \( y \) vs. \( x_j \), it may be evident that a bending curve will fit the data much better than a straight line.

The QuadRegression class achieves this simply by expanding the data matrix. From the dataset (initial data matrix), all columns will have another column added that contains the values of the original column squared. It is important that the initial data matrix has no intercept. The expansion will add an intercept column (column of all ones) and the index calculations will be thrown off if the initial data matrix has one.

6.8.1 Model Equation

In two dimensions (2D) where \( \mathbf{x} = [x_1, x_2] \), the quadratic model/regression equation is the following:

\[
y = \mathbf{b} \cdot \mathbf{x}' + \epsilon = b_0 + b_1x_1 + b_2x_1^2 + b_3x_2 + b_4x_2^2 + \epsilon
\]  

(6.70)

where \( \mathbf{x}' = [1, x_1, x_2, x_1^2, x_2^2] \), \( \mathbf{b} = [b_0, b_1, b_2, b_3, b_4] \), and \( \epsilon \) represents the residuals (the part not explained by the model).

The number of terms \( (nt) \) in the model increases linearly with the dimensionality of the space \( (n) \) according to the following formula:

\[
nt = 2n + 1 \quad \text{e.g., } nt = 5 \quad \text{for } n = 2
\]  

(6.71)

Each column in the initial data matrix is expanded into two in the expanded data matrix and an intercept column is added.

The addition of squared columns is performed by functions in the companion object. For example, the function `forms` will take an unexpanded vector \( v \), the number of variables, and the number of terms for the expanded form and will make a new vector where the zero-th element is 1.0, the odd elements are original values and the rest of the even elements are the squared values.

```scala
@param v the vector/point (i-th row of x) for creating forms/terms
@param k number of features/predictor variables (not counting intercept) [not used]
@param nt the number of terms

override def forms (v: VectorD, k: Int, nt: Int): VectorD =
{
    VectorD (for (j <- 0 until nt) yield
        if (j == 0) 1.0 // intercept term
        else if (j % 2 == 1) v(j/2) // linear terms
        else v((j-1)/2)^2 // quadratic terms
    ) // for
} // qForms
```

The allForms function in the ModelFactory object calls this function for each row of the data matrix \( X \) to create the expanded matrix.
6.8.2 Comparison of QuadRegression and Regression

This subsection compares the QuadRegression class to the Regression class.

```scala
// 8 data points: x  y
val xy = new MatrixD((8, 2), 1, 2, // 8-by-2 combined matrix
  2, 5,
  3, 10,
  4, 15,
  5, 20,
  6, 30,
  7, 50,
  8, 60)
val y = xy.col(1) // response in last column
val oxy = VectorD.one(xy.dim1) :^: xy // prepend a column of all ones
val rg = Regression(oxy) // create a regression model
println(rg.analyze().report) // analyze and report
println(rg.summary) // show summary
val qrg = QuadRegression(xy) // create a quadratic regression model
println(qrg.analyze().report) // analyze and report
println(qrg.summary) // show summary
```

Now compare their summary results.

**SUMMARY**

| Parameters/Coefficients: | Var | Estimate | Std. Error | t value | Pr(>|t|) | VIF |
|--------------------------|-----|----------|------------|---------|----------|-----|
|                          | x0  | -13.2857 | 5.15458    | -2.5774 | 0.0419   | NA  |
|                          | x1  |  8.2857  | 1.02076    |  8.1172 | 0.0001   | 1.00|
| Residual standard error: |     | 6.615278 | 6.0 degrees of freedom |
| Multiple R-squared:      |     | 0.916538 | Adjusted R-squared: 0.902628 |
| F-statistic:             |     | 65.8890  | on 1.0 and 6.0 DF, p-value: 1.8767258045970792E-4 |

**SUMMARY**

| Parameters/Coefficients: | Var | Estimate | Std. Error | t value | Pr(>|t|) | VIF |
|--------------------------|-----|----------|------------|---------|----------|-----|
|                          | x0  |  4.0357  | 3.87376    |  1.0418 | 0.3452   | NA  |
|                          | x1  | -2.1071  | 1.9750     | -1.0669 | 0.3348   | 21.25|
|                          | x2  |  1.1547  | 0.2142     |  5.3906 | 0.0029   | 21.25|
| Residual standard error: |     | 2.776603 | 5.0 degrees of freedom |
| Multiple R-squared:      |     | 0.987747 | Adjusted R-squared: 0.982846 |
| F-statistic:             |     | 201.5333 | on 2.0 and 5.0 DF, p-value: 4.872837579850131E-5 |
```
The summary results for QuadRegression highlight a couple of issues: (1) moderately high $p$-values and (2) borderline high VIF values. Try eliminating $x_1$ to see if these two improve without much of a drop in Adjusted R-squared $\bar{R}^2$. Note, eliminating $x_1$ makes the model non-hierarchical (see the exercises). Figure 6.4 shows the predictions ($y_p$) of the Regression and QuadRegression models.

![Figure 6.4: Actual y (red) vs. Regression yp (green) vs. QuadRegression yp (blue)](image)

QuadRegression Class

Class Methods:

- **@param x_** the initial data/input matrix (before quadratic term expansion) must not include an intercept column of all ones
- **@param y** the response/output vector
- **@param fname_** the feature/variable names
- **@param hparam** the hyper-parameters
- **@param technique** the technique used to solve for $b$ in $x^t x b = x^t y$

```python
class QuadRegression (x_: MatriD, y: VectoD, fname_: Strings = null, hparam: HyperParameter = null, 
technique: RegTechnique = QR)
    extends Regression (QuadRegression.allForms (x_), y, fname_, hparam, technique)
    with ExpandableForms

def expand (z: VectoD): VectoD = QuadRegression.forms (z, n0, nt)
def predict_ex (z: VectoD): Double = predict (expand (z))
```

182
The next few modeling techniques described in subsequent sections support the development of low-order
multi-dimensional polynomial regression models. Higher order polynomial regression models are typically
restricted to one-dimensional problems (see the \texttt{PolyRegression} class).

6.8.3 Exercises

1. Enter the \texttt{xy} combined data-response matrix from the example given in this section and use it to create
a \texttt{QuadRegression} model. Show the expanded input/data matrix and the response vector using the
following two print statements.

\begin{verbatim}
val qrg = QuadRegression (xy)
println (s"expanded x = \${qrg.getX}")
println (s"y = \${qrg.getY}")
\end{verbatim}

2. Perform Quadratic Regression on the \texttt{ExampleBPressure} dataset using the first two columns of its
data matrix \texttt{x}.

\begin{verbatim}
import ExampleBPressure.{x01 => x, y}
\end{verbatim}

3. Perform both forward selection and backward elimination to find out which of the terms have the most
impact on predicting the response. Which feature selection approach (forward selection or backward
elimination) finds a model with the highest $R^2$?

4. Generate a dataset with data matrix \texttt{x} and response vector \texttt{y} using the following loop where
\texttt{noise = new Normal (0, 10 * m * m)}.

\begin{verbatim}
for (i <- x.range1) {
  x(i, 0) = i
  y(i) = i*i + i + noise.gen
} // for
\end{verbatim}

Compare the results of \texttt{Regression} vs. \texttt{QuadRegression}. Compare the Quality of Fit and the parameter values. What correspondence do the parameters have with the coefficients used to generate the
data? Plot \( y \) vs. \( x \), \( y_p \) and \( y \) vs. \( t \) for both \texttt{Regression} and \texttt{QuadRegression}. Also plot the residuals \( e \) vs. \( x \) for both. Note, \( t \) is the index vector \texttt{VectorD.range (0, m)}.

5. Generate a dataset with data matrix \texttt{x} and response vector \texttt{y} using the following loop where
\texttt{noise = new Normal (0, 10 * s * s)} and \texttt{grid = 1 to s}.

\begin{verbatim}
var k = 0
for (i <- grid; j <- grid) {
  x(k) = VectorD (i, j)
  y(k) = x(k, 0)^2 + 2 * x(k, 1) + noise.gen
  k += 1
} // for
\end{verbatim}
Compare the results of Regression vs. QuadRegression. Try modifying the equation for the response and see how Quality of Fit changes.

6. The QuadRegression class as well as its more complex cousins (QuadXRegression, CubicRegression, and CubicXRegression) may have issues with having high multi-collinearity or high VIF values. Although high VIF values may not be a problem for prediction accuracy, they can make interpretation and inferencing difficult. For the problem given in this section, rather than adding \( x^2 \) to the existing Regression model, find a second order polynomial that could be added without causing high VIF values. VIF values are the lowest when column vectors are orthogonal. See the section on Polynomial Regression for more details.

7. Extrapolation far from the training data can be risky for many types of models. Show how having higher order polynomial terms in the model can increase this risk.

8. A polynomial regression model is said to be hierarchical if it contains all terms up to \( x^k \), e.g., a model with \( x, x^2, x^3 \) is hierarchical, while a model with \( x, x^3 \) is not. Show that hierarchical models are invariant under linear transformations.

Hint: Consider the following two models where \( x \) is the distance on I-70 West in miles from the center of Denver (junction with I-25) and \( y \) is the elevation in miles above sea level.

\[
\hat{y} = b_0 + b_1 x + b_2 x^2 \\
\hat{y} = b_0 + b_2 x^2
\]

The first model is hierarchical, while the second is not. A second study is conducted, but now the distance \( z \) is from the junction of I-70 and I-76. A linear transformation can be used to resolve the problem.

\[
x = z + 7
\]

Putting \( z \) into the second model (assuming the first study indicated a linear term is not needed) gives,

\[
\hat{y} = b_0 + b_2(z + 7)^2 = (b_0 + 49b_2) + 14b_2 z + b_2 z^2
\]

but now the linear term is back in the model.
6.9 Quadratic Regression with Cross Terms

The QuadXRegression class extends Regression and adds cross/interaction terms in addition to the quadratic terms added by QuadRegression. A cross term such as the one based on the product $x_1x_2$ indicates a combined effect of two predictor variables on the response variable $y$.

6.9.1 Model Equation

In two dimensions (2D) where $x = [x_1, x_2]$, the quadratic cross model/regression equation is the following:

$$
y = b \cdot x' + \epsilon = b_0 + b_1x_1 + b_2x_1^2 + b_3x_1x_2 + b_4x_2 + b_5x_2^2 + \epsilon \tag{6.72}
$$

where the components of the model equation are defined as follows:

- $x' = [1, x_1, x_1^2, x_1x_2, x_2, x_2^2]$ expanded input vector
- $b = [b_0, b_1, b_2, b_3, b_4, b_5]$ parameter/coefficient vector
- $\epsilon = y - b \cdot x'$ error/residual

The number of terms ($nt$) in the model increases quadratically with the dimensionality of the space ($n$) according to the formula for triangular numbers shifted by ($n \rightarrow n + 1$).

$$
nt = \binom{n+2}{2} = \frac{(n+2)(n+1)}{2} \quad \text{e.g., } nt = 6 \text{ for } n = 2 \tag{6.73}
$$

This result may derived by summing the number of constant terms (1), linear terms ($n$), quadratic terms ($n^2$), and cross terms ($\binom{n}{2}$).

Such models generalize QuadRegression by introducing cross terms, e.g., $x_1x_2$. Adding cross terms makes the number of terms increase quadratically rather than linearly with the dimensionality. Consequently, multi-collinearity problems may be intensified and the need for feature selection, therefore, increases.

The addition of squared and cross term columns is performed by functions in the companion object. The function forms will take an unexpanded vector $v$, the number of variables, and the number of terms for the expanded form and will make a new expanded vector.

```scala
override def forms(v: VectoD, k: Int, nt: Int): VectoD =
{
  val q = one (1) ++ v // augmented vector: [ 1., v(0), ..., v(k-1) ]
  val z = new VectorD (nt) // vector of all forms/terms
  var l = 0
  for (i <- 0 to k; j <- i to k) { z(l) = q(i) * q(j); l += 1 }
  z
} // forms
```

185
6.9.2 Response Surface

One may think of a QuadXRegression model as well as more complex models as approximating a response surface in multiple dimensions.

\[ y = f(x_1, x_2) + \epsilon \]  \hspace{1cm} (6.74)

For example, a model with two predictor variables and one response variable may be displayed in three dimensions. Such a response surface can also be shown in two dimensions using contour plots where a contour/curve shows points of equal height. Figure 6.5 shows three types of contours that represent the types of terms in QuadXRegression regression (1) linear terms, (2) quadratic terms, and cross terms.

![QuadXRegression Contours](image)

Figure 6.5: QuadXRegression Contours: \( x_1 + x_2 \) (green), \( x_1^2 + x_2^2 \) (blue), \( x_1x_2 \) (red)

A constant term simply moves the whole response surface up or down. The coefficients for each of terms can rotate and stretch these curves.

QuadXRegression Class

Class Methods:

- `@param x_` the m-by-n data/input matrix (original un-expanded)
- `@param y` the m response/output vector
- `@param fname_` the feature/variable names
- `@param hparam` the hyper-parameters
- `@param technique` the technique used to solve for \( b \) in \( x^t \times b = x^t \times y \)

```python
class QuadXRegression (x_: MatriD, y: VectoD, fname_: Strings = null, hparam: HyperParameter = null, technique: RegTechnique = QR)
```

186
extends Regression (QuadXRegression.allForms (x_), y, fname_, hparam, technique)
with ExpandableForms

def expand (z: VectoD): VectoD = QuadXRegression.forms (z, n0, nt)
def predict_ex (z: VectoD): Double = predict (expand (z))

6.9.3 Exercises

1. Perform Quadratic and QuadraticX Regression on the ExampleBPressure dataset using the first two columns of its data matrix x.

   import ExampleBPressure.{x01 => x, y}

2. Perform both forward selection and backward elimination to find out which of the terms have the most impact on predicting the response. Which feature selection approach (forward selection or backward elimination) finds a model with the highest $\bar{R}^2$?

3. Generate a dataset with data matrix x and response vector y using the following loop where noise = new Normal (0, 10 * s * s) and grid = 1 to s.

   var k = 0
   for (i <- grid; j <- grid) {
     x(k) = VectorD (i, j)
     y(k) = x(k, 0)\^2 + 2 * x(k, 1) + x(k, 0) * x(k, 1) + noise.gen
     k += 1
   } // for

   Compare the results of Regression, QuadRegression and QuadXRegression.

4. Prove that the number of terms for a quadratic function $f(x)$ in n dimensions is $\binom{n+2}{2}$, by decomposing the function into its quadratic (both squared and cross), linear and constant terms,

   $f(x) = x^T A x + b^T x + c$

   where $A$ in an n-by-n matrix, $b$ is an n-dimensional column vector and $c$ is a scalar. Hint: $A$ is symmetric, but the main diagonal is not repeated, and we are looking for unique terms (e.g., $x_1 x_2$ and $x_2 x_1$ are treated as the same). Note, when $n = 1$, $A$ and $b$ become scalars, yielding the usual quadratic function $ax^2 + bx + c$. 

187
6.10 Cubic Regression

The CubicRegression class extends Regression and adds cubic terms in addition to the quadratic terms and cross terms added by QuadXRegression. Linear terms in a model allow for slopes and quadratic terms allow for curvature. If the curvature changes substantially or there is an inflection point (curvature changes sign), then cubic terms may be useful.

6.10.1 Model Equation

In two dimensions (2D) where \( x = [x_1, x_2] \), the cubic regression equation is the following:

\[
y = b \cdot x' + \epsilon = b_0 + b_1 x_1 + b_2 x_1^2 + b_3 x_1 x_2 + b_4 x_2 + b_5 x_2^2 + b_6 x_1^3 + b_7 x_2^3 + \epsilon \quad (6.75)
\]

where the components of the model equation are defined as follows:

- \( x' = [1, x_1, x_1^2, x_1 x_2, x_2, x_2^2] \) expanded input vector
- \( b = [b_0, b_1, b_2, b_3, b_4, b_5, b_6, b_7] \) parameter/coefficient vector
- \( \epsilon = y - b \cdot x' \) error/residual

The number of terms (\( nt \)) in the model still increases quadratically with the dimensionality of the space (\( n \)) according to the formula for triangular numbers shifted by \( (n \rightarrow n + 1) \) plus \( n \) for the cubic terms.

\[
nt = \left( \frac{n+2}{2} \right) + n = \frac{(n+2)(n+1)}{2} + n \quad \text{e.g.,} \quad nt = 8 \quad \text{for} \quad n = 2 \quad (6.76)
\]

When \( n = 10 \), the number of terms and corresponding parameters \( nt = 76 \), whereas for Regression, QuadRegression and QuadXRegression and order 2, it would 11, 21 and 66, respectively. Issues related to negative degrees of freedom, overfitting and multi-collinearity will need careful attention.

The addition of squared, cross term and cubic columns is performed by functions in the companion object. The function forms will take an unexpanded vector \( v \), the number of variables, and the number of terms for the expanded form and will make a new expanded vector.

```scala
override def forms (v: VectoD, k: Int, nt: Int): VectoD =
{
  val z = new VectorD (k)
  for (i <- z.range) z(i) = v(i) ~^3
  QuadXRegression.forms (v, k, QuadXRegression.numTerms (k)) ++ z
} // forms
```

6.10.2 Comparison of CubicRegression, QuadRegression and Regression

This subsection compares the CubicRegression class to the QuadRegression class and the Regression class.
val xy = new MatrixD ((8, 2), 1, 2, // 8-by-2 combined matrix
    2, 11,
    3, 25,
    4, 28,
    5, 30,
    6, 26,
    7, 42,
    8, 60)
val y = xy.col(1) // response in last column
val oxy = VectorD.one (xy.dim1) +^: xy // prepend a column of all ones
val rg = Regression (oxy) // create a regression model
println (rg.analyze ().report) // analyze and report
println (rg.summary) // show summary
val qrg = QuadRegression (xy) // create a quadratic regression model
println (qrg.analyze ().report) // analyze and report
println (qrg.summary) // show summary
val qrg = CubicRegression (xy) // create a cubic regression model
println (qrg.analyze ().report) // analyze and report
println (qrg.summary) // show summary

Figure 6.6 shows the predictions (yp) of the Regression, QuadRegression and CubicRegression models.

Figure 6.6: Actual y (red) vs. Regression (green) vs. QuadRegression (blue) vs. CubicRegression (black)

Notice the quadratic curve follows the linear curve (line), while the cubic curve more closely follows the data.
CubicRegression Class

Class Methods:

@param $x_*$ the input vectors/points
@param $y$ the response vector
@param $fname_*$ the feature/variable names
@param $hparam$ the hyper-parameters
@param $technique$ the technique used to solve for $b$ in $x^t*x*b = x^t*y$

class CubicRegression ($x_*$ : Matrid, $y$: VectorD, $fname_*$ : Strings = null, $hparam$: HyperParameter = null, 
  $technique$: RegTechnique = QR) 
  extends Regression (CubicRegression.allForms ($x_*$), $y$, $fname_*$, $hparam$, $technique$) 
  with ExpandableForms

def expand ($z$: VectorD): VectorD = CubicRegression.forms ($z$, $n_0$, $n_t$)
def predict_ex ($z$: VectorD): Double = predict (expand ($z$))

6.10.3 Exercises

1. Generate and compare the model summaries produced by the three models (Regression, QuadRegression and CubicRegression) applied to the dataset given in this section.

2. Many laws in science involve quadratic and cubic terms as well as the inverses of these terms (e.g., inverse square laws). Find such a law and an open dataset to test the law.
### 6.11 Cubic Regression with Cross Terms

The `CubicXRegression` class extends `Regression` and adds cubic cross terms (e.g., $x_1^3 x_2$) in addition to the quadratic terms, cross terms and cubic terms added by `CubicRegression`.

#### 6.11.1 Model Equation

In two dimensions (2D) where $\mathbf{x} = [x_1, x_2]$, the cubic model/regression equation is the following:

$$ y = \mathbf{b} \cdot \mathbf{x}' + \epsilon = b_0 + b_1 x_1 + b_2 x_1^2 + b_3 x_1^3 + b_4 x_1 x_2 + b_5 x_1^2 x_2 + b_6 x_1 x_2^2 + b_7 x_2 + b_8 x_2^2 + b_9 x_2^3 + \epsilon \quad (6.77) $$

where the components of the model equation are defined as follows:

- $\mathbf{x}' = [1, x_1, x_1^2, x_1 x_2, x_1^2 x_2, x_2, x_2^2, x_2^3]$ expanded input vector
- $\mathbf{b} = [b_0, b_1, b_2, b_3, b_4, b_5, b_6, b_7, b_8, b_9]$ parameter/coefficient vector
- $\epsilon = y - \mathbf{b} \cdot \mathbf{x}'$ error/residual

Naturally, the number of terms in the model increases cubically with the dimensionality of the space ($n$) according to the formula for tetrahedral numbers shifted by ($n \to n+1$).

$$ nt = \binom{n + 3}{3} = \frac{(n + 3)(n + 2)(n + 1)}{6} \quad \text{e.g., } nt = 10 \text{ for } n = 2 \quad (6.78) $$

When $n = 10$, the number of terms and corresponding parameters $nt = 286$, whereas for `Regression`, `QuadRegression`, `QuaXdRegression` and `CubicRegression` and order 2, it would 11, 21, 66 and 76, respectively. Issues related negative degrees of freedom, overfitting and multi-collinearity will need careful attention.

The addition of squared, cross term, cubic and cubic cross term columns is performed by functions in the companion object. The function `forms` will take an unexpanded vector v, the number of variables, and the number of terms for the expanded form and will make a new expanded vector.

```scala
@override def forms (v: VectorD, k: Int, nt: Int): VectorD = {
  val q = one (1) ++ v // augmented vector: [ 1., v(0), ..., v(k-1) ]
  val z = new VectorD (nt) // vector of all forms/terms
  var l = 0
  for (i <- 0 to k; j <- i to k; h <- j to k) { z(l) = q(i) * q(j) * q(h); l += 1 }
  z
} // forms
```

If polynomials of higher degree are needed, SCALATiON provides a couple of means to deal with it. First, when the data matrix consists of single column and $\mathbf{x}$ is one dimensional, the `PolyRegression` class may be used. If one or two variables need higher degree terms, the caller may add these columns themselves as additional columns in the data matrix input into the `Regression` class.
CubicXRegression Class

Class Methods:

@param x_ the input vectors/points
@param y the response vector
@param fname_ the feature/variable names
@param hparam the hyper-parameters
@param technique the technique used to solve for b in \(x^t x b = x^t y\)

class CubicXRegression (x_: MatriD, y: VectoD, fname_: Strings = null, hparam: HyperParameter = null,
                              technique: RegTechnique = QR)
    extends Regression (CubicXRegression.allForms (x_), y, fname_, hparam, technique)
    with ExpandableForms

def expand (z: VectoD): VectoD = CubicXRegression.forms (z, n0, nt)

def predict_ex (z: VectoD): Double = predict (expand (z))

6.11.2 Exercises

1. Perform Cubic and CubicX Regression on the ExampleBPressure dataset using the first two columns of its data matrix \(x\).

   import ExampleBPressure.{x01 => x, y}

2. Perform both forward selection and backward elimination to find out which of the terms have the most impact on predicting the response. Which feature selection approach (forward selection or backward elimination) finds a model with the highest \(\bar{R}^2\)?

3. Generate a dataset with data matrix \(x\) and response vector \(y\) using the following loop where \(\text{noise} = \text{new Normal}(0, 10 \times s \times s)\) and \(\text{grid} = 1\) to \(s\).

   var k = 0
   for (i <- grid; j <- grid) {
      x(k) = VectorD (i, j)
      y(k) = x(k, 0)~^2 + 2 * x(k, 1) + x(k, 0) * x(k, 1) + noise.gen
      k += 1
   }

   // for

Compare the results of Regression, QuadRegression, QuadXRegression, CubicRegression and CubicXRegression. Try modifying the equation for the response and see how Quality of Fit changes.
6.12 Transformed Regression

The `TranRegression` class supports transformed multiple linear regression and hence, the predictor vector \( \mathbf{x} \) is multi-dimensional \([1, x_1, ..., x_k]\). In certain cases, the relationship between the response scalar \( y \) and the predictor vector \( \mathbf{x} \) is not linear. There are many possible functional relationships that could apply [76], but five obvious choices are the following:

1. The response grows exponentially versus a linear combination of the predictor variable.
2. The response grows quadratically versus a linear combination of the predictor variable.
3. The response grows as the square root of a linear combination of the predictor variable.
4. The response grows logarithmically versus a linear combination of the predictor variable.
5. The response grows inversely (as the reciprocal) versus a linear combination of the predictor variable.

The capability can be easily implemented by introducing a transform (transformation function) into `Regression`. The transformation function and its inverse are passed into the `TranRegression` class which extends the `Regression` class.

\[
f_t : \mathbb{R} \rightarrow \mathbb{R} \quad \text{transformation function} \tag{6.79}
\]

\[
f_a : \mathbb{R} \rightarrow \mathbb{R} \quad \text{activation function} \tag{6.80}
\]

The transform \( f_t \) (tran) and its inverse transform \( f_a = f_t^{-1} \) (itran) for the five cases are as follows:

\((\log, \exp), (\sqrt{\text{r}}, \text{sq}), (\text{sq}, \sqrt{\text{r}}), (\exp, \log), (\text{recip}, \text{recip})\)

Transformed Regression models extend the reach of linear models while maintaining their simplicity and highly efficient parameter estimation techniques. Beyond these models lay Generalized Linear Models and Nonlinear Models.

6.12.1 Model Equation

The goal then is to fit the parameter vector \( \mathbf{b} \) in the transformed model/regression equation

\[
f_t(y) = \mathbf{b} \cdot \mathbf{x} + \epsilon = b_0 + b_1 x_1 + ... b_k x_k + \epsilon \tag{6.81}
\]

where \( \epsilon \) represents the residuals (the part not explained by the model) and \( f_t \) (tran) is the function (defaults to log) used to transform the response \( y \). For example, for a log transformation, the equation becomes the following:

\[
\log(y) = \mathbf{b} \cdot \mathbf{x} + \epsilon = b_0 + b_1 x_1 + ... b_k x_k + \epsilon \tag{6.82}
\]

The transform \( f_t \) (tran) is implemented in the `TranRegression` class by transforming \( y \) and passing it to the `Regression` superclass (multiple linear regression).

\[
\text{Regression}(x, y.\text{map}(\text{tran}), \text{technique})
\]

The inverse transform \( f_a \) (itran) is then applied in the `predict` method.

\[
\text{override def predict}(z: \text{VectoD}): \text{Double} = \text{itran}(\mathbf{b} \cdot \mathbf{z})
\]

193
### 6.12.2 Training

Using several data samples as a training set \((X, y)\), a loss function \(h(b)\) can be minimized to find an optimal solution for the parameter vector \(b\).

The training diagram shown in Figure 6.7 illustrates how the \(i^{th}\) instance/row flows through the diagram computing the transformed response \(z = f_t(y)\), the predicted transformed response \(\hat{z} = b \cdot x\) and the transformed error \(e = z - \hat{z}\).

![Training Diagram for Transformed Regression](image)

Note, the actual (untransformed) error is \(\epsilon = y - \hat{y} = f_a(z) - f_a(\hat{z})\).

The loss function is based on the transformed errors,

\[
h(b) = \frac{1}{2} \|e\|^2 = \frac{1}{2} \|z - \hat{z}\|^2 = \frac{1}{2} \|f_t(y) - Xb\|^2
\]

where the transformed error vector \(e = z - \hat{z}\), \(z = f_t(y)\), and \(\hat{z} = Xb\). Note, \(f_t : \mathbb{R}^m \rightarrow \mathbb{R}^m\) is the vectorized version of \(f_t\). See the exercises for a loss function based the actual (untransformed) errors.

Taking the gradient of the loss function and setting it to zero,

\[
\nabla h(b) = X^T[f_t(y) - Xb] = 0
\]

yields \((X^TX)b = X^Tf_t(y)\).

### 6.12.3 Square Root Transformation

The square root transformation takes the square root of the response variable and regresses it onto a linear model.

\[
f_t(y) = \sqrt{y} = b \cdot x + \epsilon
\]
As an example in 1D, the `TranRegression` class is compared to the `QuadRegression` class and the `Regression` class.

\[
y = b_0 + b_1 x + \epsilon \quad \text{regression}
\]
\[
y = b_0 + b_1 x + b_2 x^2 + \epsilon \quad \text{quadratic regression}
\]
\[
\sqrt{y} = b_0 + b_1 x + \epsilon \quad \text{sqrt transformed regression}
\]

The ScalaTion code shown below compares these three models.

```java
import scala.math._
import scalation.analytics._
import scalationanalytics.TranRegression._
import scalation.math._

// 8 data points: x y
val xy = new MatrixD ((8, 2), 1, 2, // 8-by-2 combined matrix
    2, 5,
    3, 10,
    4, 15,
    5, 20,
    6, 30,
    7, 50,
    8, 60)
val y = xy.col(1) // response in last column
val oxy = VectorD.one (xy.dim1) +^: xy // prepend a column of all ones
val rg = Regression (oxy) // create a regression model
println (rg.analyze ().report) // analyze and report
println (rg.summary) // show summary
val qrg = QuadRegression (xy) // create a quadratic regression model
println (qrg.analyze ().report) // analyze and report
println (qrg.summary) // show summary
TranRegression.rescaleOff ()
val trg = TranRegression (oxy, drp._1, drp._2, sqrt _, sq _, drp._5, drp._6)
println (trg.analyze ().report) // analyze and report
println (trg.summary) // show summary
```

Figure 6.8 shows the predictions \((yp)\) of the `Regression`, `QuadRegression` and `TranRegression` models.

Notice that the sqrt transformed regression model closely follows the quadratic regression model, yet has one fewer parameter. The square root transformation can model quadratic effects and stabilize error variance, but it makes interpretation of coefficients less direct. See the exercises for a comparison.
6.12.4 Log Transformation

The log transformation takes the logarithm (defaults to ln, the natural log) of the response variable and regresses it onto a linear model.

\[ f_t(y) = \log(y) = b \cdot x + \epsilon \]  

(6.86)

Imagine a system where the rate of change of the response variable \( y \) with the predictor variable \( x \) (e.g., time) is proportional to its current value \( y \) and is \( y_0 \) when \( x = 0 \).

\[ \frac{dy}{dx} = gy \]

This differential equation can be solved by direct integration to obtain

\[ \int \frac{dy}{y} = \int g dx \]

As the integral of \( \frac{1}{y} \) is \( \ln(y) \), integrating both sides gives

\[ \ln(y) = gx + C \]

Solving for the constant gives \( C = \ln(y_0) \), and then taking the exp function of both sides produces (ignoring noise/error)

\[ \ln(y) = gx + \ln(y_0) \]

\[ y = y_0 e^{gx} \]

When the growth factor \( g \) is positive, the system exhibits exponential growth, while when it is negative, it exhibits exponential decay. So far we have ignored noise. For previous modeling techniques, we have assumed that noise is additive and typically normally distributed. For phenomena exhibiting exponential growth or decay, this may not be the case. When the error is multiplicative, we may collect it into the exponent.
\[ y = y_0 e^{gx + \epsilon} \]

Now applying a log transformation, will yield

\[ \log(y) = \log(y_0) + gx + \epsilon = b_0 + b_1 x + \epsilon \]

An alternative to using `TranRegression` is to use Exponential Regression `ExpRegression`, a form of Nonlinear Regression Model (see the exercises for a comparison).

### 6.12.5 Reciprocal Transformation

The reciprocal (or inverse) transformation takes the reciprocal of the response variable and regresses it onto a linear model.

\[
 f_1(y) = y^{-1} = b \cdot x + \epsilon 
\]  

(6.87)

It may be the case that rates, such as Miles-Per-Gallon (MPG), are inversely related to other quantities. This can be tested using the AutoMPG dataset as follows:

```scala
import scala.math._
import scalation.analytics._
import scalation.analytics.ExampleAutoMPG._
import scalationanalytics.TranRegression._
import scalation.math._
import scalation.plot._

def recip (y: Double): Double = 1.0 / y

val f = (recip _, recip _, "recip")
// val f = (log _, exp _, "log")
// val f = (sqrt _, sq _, "sqrt")
// val f = (sq _, sqrt _, "sq")
// val f = (exp _, log _, "exp")
// TranRegression.setLambda (0.2); val f = (box_cox _, cox_box _, "box_cox")

println (s"TranRegression with \${f._3} transform")
rescaleOff ()
val trg = TranRegression (ox, y, drp._1, drp._2, f._1, f._2, drp._5, drp._5)
println (trg.analyze ().report)
println (trg.summary)
val (cols, rSq) = trg.forwardSelAll () // R^2, R^2 bar, R^2 cv
val k = cols.size
println (s"k = $k, n = $({x.dim2})")
val t = VectorD.range (1, k) // instance index
new PlotM (t, rSq.t, Array ("R^2", "R^2 bar", "R^2 cv"),
```
Be sure to try all the commented out transformations as well as various values for $\lambda$ for the Box-Cox transformation (see the next subsection).

### 6.12.6 Box-Cox Transformation

As should be apparent from the last section, there are many transformations that could be applied. In order to provide a more systematic approach for transforming the response variable, Box-Cox transformations were developed. These transformations may be used to (1) improve prediction accuracy, (2) stabilize variance (reduce heteroscedasticity), and (3) improve normality.

Consider the following family of transformation functions.

$$ f_t(y) = \frac{y^\lambda - 1}{\lambda} \quad (6.88) $$

where $\lambda \neq 0$ determines the power function on $y$, e.g., 0.5 for $\sqrt{}$ and 2.0 for $\text{sq}$. The following simplified version of the original form given by Box and Cox may be used as well.

$$ f_t(y) = y^\lambda \quad (6.89) $$

The inverse transform is $f^{-1}_t(y) = y^{1/\lambda}$. When $\lambda = 0$, the log transformation is used. One way to find suitable values for $\lambda$ ($\text{lambda}$) is to perform grid search over the interval $[-3, 3]$, possibly as large as $[-5, 5]$.

**TranRegression (x, y, lambda)**

Box-Cox transformations are provided in the class’ companion object.

Note, although the focus has been on transforming the response variable, the predictor variables could be transformed as well.

### 6.12.7 Quality of Fit

For a fair comparison with other modeling techniques, the Quality of Fit (QoF) or overall diagnostics are based on the original response values, as provided by the usual `eval` method. It may be useful to study the Quality of Fit for the transformed response vector $y$.map (tran) as well via the `eval0` method.

**TranRegression Class**

Class Methods:

- `tparam x` the data/input matrix
- `tparam y` the response/output vector
- `tparam fname` the feature/variable names
- `tparam hparam` the hyper-parameters (currently none)
- `tparam tran` the transformation function (defaults to log)
- `tparam itran` the inverse transformation function to rescale predictions to original y scale
(defaults to exp)

```scala
@Param technique the technique used to solve for b in x.t*x*b = x.t*y

class TranRegression (x: MatriD, y: VectoD, fname_ : Strings = null,
  hparam: HyperParameter = null,
  tran: FunctionS2S = log, itran: FunctionS2S = exp,
  technique: RegTechnique = QR)
  extends Regression (x, y.map (tran), fname_, hparam, technique)

def eval0 (x_e: MatriD = x, y_e: VectoD = getY): TranRegression =
override def eval (x_e: MatriD = x, y_e: VectoD = y): TranRegression =
override def analyze (x_ : MatriD = x, y_ : VectoD = getY,
  x_e: MatriD = x, y_e: VectoD = y): PredictorMat =
override def predict (z: VectoD): Double = itran (b dot z)
override def predict (z: MatriD = x): VectoD = (z * b).map (itran)
```

### 6.12.8 Exercises

1. Use the following code to generate a dataset. You will need to import from `scalation.math.sq` and `scalation.random`.

   ```scala
   val cap = 30
   val rng = 0 until cap
   val (m, n) = (cap * cap, 3)
   val err = Normal (0, cap)
   val x = new MatrixD (m, n)
   val y = new VectorD (m)
   for (i <- rng; j <- rng) x(cap * i + j) = VectorD (1, i, j)
   for (k <- y.range) y(k) = sq (10 + 2 * x(k, 1) + err.gen)
   // for (k <- y.range) y(k) = sq (10 + 2 * x(k, 1) + 0.3 * x(k, 2) + err.gen)
   val t = VectorD.range (0, y.dim)
   ```

Notice that it uses a linear model inside and takes the square for the response variable y. Use `Regression` to create a predictive model. Ideally, the model should approximately recapture the equations used to generate the data. What correspondence do the parameters b have to these equations? Next, examine the relationship between the response y and predicted response yp, as well as the residuals (or remaining error) from the model.

```scala
val rg = new Regression (x, y)
rg.analyze ()
println (rg.report)
println (rg.summary)
val yp = rg.predict ()
```
val e = y - yp

new Plot (t, y, yp, "Original Regression y and yp vs. t")
new Plot (t, e, null, "Original e vs. t")

Are there discernible patterns in the residuals?

2. Transform the response y to a transformed response y2 that is the square root of the former.

val y2 = y.map (sqrt _)

Redo the regression as before, but now using the transformed response y2, i.e., new Regression (x, y2). Compute and plot the corresponding y2, yp2 and e2 vectors. What do the residuals look like now? How can predictions be made on the original scale?

3. Now transform yp2 to yp3 in order to match the actual response y, by using the inverse transformation function sq. Now, compute and plot the corresponding y, yp3 and e3 vectors. How well does yp3 predict the original response y? Compute the Coefficient of Determination $R^2$. What is the difference between the residuals e2 and e3? Finally, use PlotM to compare Regression vs. Transformed Regression.

val ys2 = MatrixD (y2, yp2)
val ys3 = MatrixD (y, yp3, yp)
new PlotM (t, ys2.t, null, "Transformed")
new PlotM (t, ys3.t, null, "Tran-back")

4. The TranRegression class provides direct support for making transformations. Compare the quality of fit resulting from Regression versus TranRegression.

banner ("Regression")
val rg = new Regression (x, y)
rg.analyze ()
println (rg.report)
println (rg.summary)

banner ("TranRegression")
val trg = new TranRegression (x, y, sqrt _, sq _)
trg.analyze ()
println (rg.report)
println (rg.summary)

5. Extend the previous exercise to include QuadRegression in the comparison.

6. Compare SimpleRegression, TranRegression and ExpRegression on the beer foam dataset [www.tf.uni-kiel.de/matwis/amat/iss/kap_2/articles/beer_article.pdf](http://www.tf.uni-kiel.de/matwis/amat/iss/kap_2/articles/beer_article.pdf). The last two are similar, but TranRegression assumes multiplicative noise, while ExpRegression assumes additive noise, so they produce different predictions. Plot and compare the three predictions.
val \( x_1 \) = VectorD (0, 15, 30, 45, 60, 75, 90, 105, 120, 150, 180, 210, 240, 300, 360) \\
val \( y \) = VectorD (14.0, 12.1, 10.9, 10.0, 9.3, 8.6, 8.0, 7.5, 
\quad 7.0, 6.2, 5.5, 4.5, 3.5, 2.0, 0.9) \\
val _1 = VectorD.one (x1.dim) \\
val x = MatrixD (_1, x1)

7. Compare the following loss function based on actual (untransformed) errors \( \epsilon \) with the one based on transformed errors \( e \).

\[
h(b) = \frac{1}{2} \| \epsilon \|^2 = \frac{1}{2} \| y - \hat{y} \|^2 = \frac{1}{2} \| y - f_n(Xb) \|^2
\]
6.13 Weighted Least Squares Regression

The `Regression.WLS` class supports weighted multiple linear regression. In this case, the predictor vector \( \mathbf{x} \) is multi-dimensional \([1, x_1, \ldots, x_k]\).

### 6.13.1 Model Equation

As before the model/regression equation is

\[
\mathbf{y} = \mathbf{b} \cdot \mathbf{x} + \epsilon = b_0 + b_1 x_1 + \ldots + b_k x_k + \epsilon
\]  

(6.90)

where \( \epsilon \) represents the residuals (the part not explained by the model). Under multiple linear regression, the parameter vector \( \mathbf{b} \) is estimated using matrix factorization with the Normal Equations.

\[
(X^\top X)\mathbf{b} = X^\top \mathbf{y}
\]

Let us look at the error vector \( \epsilon = \mathbf{y} - X\mathbf{b} \) in more detail. A basic assumption is that \( \epsilon \sim N(0, \sigma^2 I) \). i.e., each error is Normally distributed with mean 0 and variance \( \sigma^2 \). If this is violated substantially, the estimate for the parameters \( \mathbf{b} \) may be less accurate than desired. One way this can happen is that the variance changes \( \epsilon_i \sim N(0, \sigma_i^2) \). This is called heteroscedasticity (or heteroskedasticity) and it would imply that certain instances (data points) would have greater influence \( \mathbf{b} \) than they should. The problem can be corrected by weighting each instance by the inverse of its residual/error variance.

\[
w_i = \frac{1}{\sigma_i^2}
\]  

(6.91)

This begs the question on how to estimate the residual/error variance. This can be done by performing unweighted regression of \( \mathbf{y} \) onto \( X \) to obtain the error vector \( \epsilon \). It is used to compute a Root Absolute Deviation (RAD) vector \( \mathbf{r} \).

\[
\mathbf{r} = \sqrt{|\epsilon|}
\]  

(6.92)

A simple approach would be to make the weight \( w_i \) inversely proportional to \( r_i \).

\[
w_i = \frac{n}{r_i}
\]

More commonly, a second unweighted regression is performed, regressing \( \mathbf{r} \) onto \( X \) to obtain the predictions \( \hat{r} \). See Exercise 1 for a comparison or the two methods `setWeights0` and `setWeights`.

\[
w_i = \frac{n}{\hat{r}_i}
\]  

(6.93)

See [48] for additional discussion concerning how to set weights. These weights can be used to build a diagonal weight matrix \( W \) that factors into the Normal Equations

\[
X^\top WX\mathbf{b} = X^\top Wy
\]

(6.94)

In SCALATION, this is accomplished by computing a weight vector \( \mathbf{w} \) and taking its square root \( \mathbf{\omega} = \sqrt{\mathbf{w}} \). The data matrix \( X \) is then reweighted by premultiplying it by \( \mathbf{\omega} \) (\( r\mathbf{\omega} \) in the code), as if it is a diagonal matrix.
The response vector \( y \) is reweighted using vector multiplication \( rtW * y \). The reweighted matrix and vector are passed into the `Regression` class, which solves for the parameter vector \( b \).

In summary, Weighted Least-Squares (WLS) is accomplished by reweighting and then using Ordinary Least Squares (OLS). See [http://en.wikipedia.org/wiki/Least_squares#Weighted_least_squares](http://en.wikipedia.org/wiki/Least_squares#Weighted_least_squares).

**Regression_WLS Class**

**Class Methods:**

```scala
@param xx the data/input m-by-n matrix
   (augment with a first column of ones to include intercept in model)
@param yy the response/output m vector
@param fname_ the feature/variable names
@param technique the technique used to solve for b in x.t*w*x*b = x.t*w*y
@param w the weight vector (if null, computed in companion object)

class Regression_WLS (xx: MatrixD, yy: VectorD, technique: RegTechnique = QR,
private var w: VectorD = null)
   extends Regression ({ setWeights (xx, yy, technique, w); reweightX (xx, w) },
reweightY (yy, w), technique)

weights: VectorD = w
override def diagnose (e: VectorD, w_: VectorD, yp: VectorD, y_: VectorD = null)
override def crossVal (k: Int = 10, rando: Boolean = true): Array [Statistic]
```

### 6.13.2 Exercises

1. The `setWeights0` method used actual RAD’s rather than predicted RAD’s used by the `setWeights` method. Compare the two methods of setting the weights on the following dataset.

```scala
// 5 data points: constant term, x_1 coordinate, x_2 coordinate
val x = new MatrixD ((5, 3), 1.0, 36.0, 66.0, // 5-by-3 matrix
   1.0, 37.0, 68.0,
   1.0, 47.0, 64.0,
   1.0, 32.0, 53.0,
   1.0, 1.0, 101.0)
val y = VectorD (745.0, 895.0, 442.0, 440.0, 1598.0)
val z = VectorD (1.0, 20.0, 80.0)
```

Try the two methods on other datasets and discuss the advantages and disadvantages.

2. Prove that reweighting the data matrix \( X \) and the response vector \( y \) and solving for the parameter vector \( b \) in the standard Normal Equations \( (X^T X)b = X^T y \) gives the same result as not reweighting and solving for the parameter vector \( b \) in the Weighted Normal Equations \( X^T W X b = X^T W y \).
3. Given an error vector $\epsilon$, what does its covariance matrix $C[\epsilon]$ represent? How can it be estimated? What are its diagonal elements?

4. When the non-diagonal elements are non-zero, it may be useful to consider using Generalized Least Squares (GLS). What are the trade-offs of using this more complex technique?
6.14 Polynomial Regression

The PolyRegression class supports polynomial regression. In this case, \( x \) is formed from powers of a single parameter \( t, [1, t, t^2, \ldots, t^k] \).

6.14.1 Model Equation

The goal is to fit the parameter vector \( b \) in the model/regression equation

\[
y = b \cdot x + \epsilon = b_0 + b_1 t + b_2 t^2 + \ldots + b_k t^k + \epsilon
\]

where \( \epsilon \) represents the residuals (the part not explained by the model). Such models are useful when there is a nonlinear relationship between a response and a predictor variable, e.g., \( y \) may vary quadratically with \( t \).

A training set now consists of two vectors, one for the \( m \)-vector \( t \) and one for the \( m \)-vector \( y \). An easy way to implement polynomial regression is to expand each \( t \) value into an \( x \) vector to form a data/input matrix \( X \) and pass it to the Regression class (multiple linear regression). The columns of data matrix \( X \) represent powers of the vector \( t \).

\[
X = [1, t, t^2, \ldots, t^k]
\]

In ScalaTion the vector \( t \) is expanded into a matrix \( X \) before calling Regression. The number of columns in matrix \( X \) is the order \( k \) plus 1 for the intercept.

\[
\text{val x = new MatrixD (t.dim, 1 + k)}
\]

\[
\text{for (i <- t.range) x(i) = expand (t(i))}
\]

\[
\text{val rg = new Regression (x, y, technique)}
\]

Unfortunately, when the order of the polynomial \( k \) get moderately large, the multi-collinearity problem can become severe. In such cases it is better to use orthogonal polynomials rather than raw polynomials. This is done in ScalaTion by changing the raw flag to false.

PolyRegression Class

Class Methods:

- @see www.ams.sunysb.edu/~zhu/ams57213/Team3.pptx
- @param t the initial data/input \( m \)-by-1 matrix: \( t_i \) expands to \( x_i = [1, t_i, t_i^2, \ldots, t_i^k] \)
- @param y the response/output vector
- @param ord the order \( (k) \) of the polynomial (max degree)
- @param fname_ the feature/variable names
- @param hparam the hyper-parameters
- @param technique the technique used to solve for \( b \) in \( x.t*x*b = x.t*y \)

```scala
class PolyRegression (t: MatrixD, y: VectorD, ord: Int, fname_: Strings = null, hparam: HyperParameter = null, technique: RegTechnique = Cholesky)
    extends Regression (PolyRegression.allForms (t, ord), y, fname_, hparam, technique)
```

205
with ExpandableForms

def expand (z: VectoD): VectoD = PolyRegression.forms (z, n0, nt)
def predict (z: Double): Double = predict_ex (VectorD (z))
def predict_ex (z: VectoD): Double = predict (expand (z))

6.14.2 Exercises

1. Generate two vectors $t$ and $y$ as follows.

   ```scala
   val noise = Normal (0.0, 100.0)
   val t = VectorD.range (0, 100)
   val y = new VectorD (t.dim)
   for (i <- 0 until 100) y(i) = 10.0 - 10.0 * i + i^2 + i * noise.gen
   ```

   Test new PolyRegression ($t$, $y$, order, technique) for various orders and factorization techniques. Test for multi-collinearity using the correlation matrix and vif.

2. Test new PolyRegression ($t$, $y$, order, technique, false) for various orders and factorization techniques. Setting the raw flag to false will cause orthogonal polynomials to be used instead or raw polynomials. Again, test for multi-collinearity using the correlation matrix and vif.
6.15 Trigonometric Regression

The TrigRegression class supports trigonometric regression. In this case, \( x \) is formed from trigonometric functions of a single parameter \( t \), \([1, \sin(\omega t), \cos(\omega t), \ldots, \sin(k\omega t), \cos(k\omega t)]\).

A periodic function can be expressed as linear combination of trigonometric functions (sine and cosine functions) of increasing frequencies. Consequently, if the data points have a periodic nature, a trigonometric regression model may be superior to alternatives.

6.15.1 Model Equation

The goal is to fit the parameter vector \( b \) in the model/regression equation

\[
y = b \cdot x + \epsilon = b_0 + b_1 \sin(\omega t) + b_2 \cos(\omega t) + \ldots + b_{2k-1} \sin(k\omega t) + b_{2k} \cos(k\omega t) + \epsilon
\]

where \( \omega \) is the base angular displacement in radians (e.g., \( \pi \)) and \( \epsilon \) represents the residuals (the part not explained by the model).

A training set now consists of two vectors, one for the \( m \)-vector \( t \) and one for the \( m \)-vector \( y \). As was done for polynomial regression, an easy way to implement trigonometric regression is to expand each \( t \) value into an \( x \) vector to form a data/input matrix \( X \) and pass it to the Regression class (multiple linear regression). The columns of data matrix \( X \) represent sines and cosines at at multiple harmonic frequencies of the vector \( t \).

\[
X = [1, \sin(\omega t), \cos(\omega t), \sin(2\omega t), \cos(2\omega t), \ldots, \sin(k\omega t), \cos(k\omega t)]
\]

For a model with \( k \) harmonics (maximum multiplier of \( \omega t \)), the data matrix can be formed as follows:

```scala
def x = new MatrixD(t.dim, 1 + 2 * k) for (i <- t.range) x(i) = expand(t(i))
val rg = new Regression(x, y, technique)
```

TrigRegression Class

Class Methods:

- `@param t` the initial data/input m-by-1 matrix: \( t_i \) expands to \( x_i \)
- `@param y` the response/output vector
- `@param ord` the order (\( k \)), maximum multiplier in the trig function (\( k\omega t \))
- `@param fname_` the feature/variable names
- `@param hparam` the hyper-parameters
- `@param technique` the technique used to solve for \( b \) in \( x^t x b = x^t y \)

```scala
class TrigRegression(t: MatrixD, y: VectorD, ord: Int, fname_: Strings = null, hparam: HyperParameter = null, technique: RegTechnique = QR) extends Regression(TrigRegression.allForms(t, ord), y, fname_, hparam, technique) with ExpandableForms
```

207
def expand (z: VectoD): VectoD = TrigRegression.forms (z, n0, nt, w)
def predict (z: Double): Double = predict_ex (VectorD (z))
def predict_ex (z: VectoD): Double = predict (expand (z))

6.15.2 Exercises

1. Create a noisy cubic function and test how well TrigRegression can fit the data for various values of \( k \) (harmonics) generated from this function.

    val noise = Normal (0.0, 10000.0)
    val t = VectorD.range (0, 100)
    val y = new VectorD (t.dim)
    for (i <- 0 until 100) {
        val x = (i - 40)/2.0
        y(i) = 1000.0 + x + x*x + x*x*x + noise.gen
    } // for

2. Make the noisy cubic function periodic and test how well TrigRegression can fit the data for various values of \( k \) (harmonics) generated from this function.

    val noise = Normal (0.0, 10.0)
    val t = VectorD.range (0, 200)
    val y = new VectorD (t.dim)
    for (i <- 0 until 5) {
        for (j <- 0 until 20) {
            val x = j - 4
            y(40*i+j) = 100.0 + x + x*x + x*x*x + noise.gen
        } // for
        for (j <- 0 until 20) {
            val x = 16 - j
            y(40*i+20+j) = 100.0 + x + x*x + x*x*x + noise.gen
        } // for
    } // for

3. Is the problem of multi-collinearity an issue for Trigonometric Regression?

4. How does Trigonometric Regression relate to Fourier Series?
6.16 ANCOVA

An ANalysis of COVAriance (ANCOVA) model may be developed using the ANCOVA class. This type of model comes into play when input variables are mixed, i.e., some are (i) continuous/ordinal, while others are (ii) categorical/binary. The main difference between the two types of variables is type (i) variables define the notion of less than ($<$), while variables of type (ii) do not. Also, the expected value means much less for type (ii) variables, e.g., what is the expected value of English, French and Spanish? If we encode a language variable $x_j$ as 0, 1 or 2 for English, French and Spanish, respectively, and half of a group speaks English with the rest speaking Spanish, then the expected value would be French. Worse, if the encoding changes, so does the expected value.

6.16.1 Handling Categorical Variables

Binary Variables

In the binary case, when a variable $x_j$ may take on only two distinct values, e.g., Red or Black, then it may simply be encoded as 0 for Red and 1 for Black. Therefore, a single zero-one, encoded/dummy variable $x_j$, can be used to distinguish the two cases. For example, when $x_j \in \{\text{Red, Black}\}$, it would be replaced by one encoded/dummy variable, $x_{j0}$ as shown in Table 6.6.

<table>
<thead>
<tr>
<th>$x_j$</th>
<th>encoded $x_j$</th>
<th>dummy $x_{j0}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Red</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Black</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Categorical Variables

For the more general categorical case, when the number distinct values for a variable $x_j$ is greater than two, simply encoding the $j^{th}$ column may not be ideal. Instead multiple dummy variables should be used. The number of dummy variables required is one less than the number of distinct values $n_{dv}$. In one hot encoding, the number of dummy variables may be equal to the $n_{dv}$, however, this will produce a singular expanded data matrix $X$ (i.e., perfect multi-collinearity).

First, the categorical variable $x_j$ may be encoded using integer values as follows:

$$\text{encoded } x_j = 0, 1, \ldots, n_{dv} - 1$$

Next, for categorical variable $x_j$, create $n_{dv} - 1$ dummy variables $\{x_{jk}|k = 0, \ldots, n_{dv} - 2\}$ and use the following loop to set the value for each dummy variable.

$$\text{for (} k \leftarrow 0 \text{ until } n_{dv} - 1 \text{) } x_{jk} = 1_{x_j = k+1}$$

where $1_c$ is the indicator function that returns 1 when the condition $c$ evaluates to true and 0 otherwise. In this way, $x_j \in \{\text{English, French, German, Spanish}\}$ would be replaced by three dummy variables, $x_{j0}$, $x_{j1}$ and $x_{j2}$, as shown in Table 6.6.
Unfortunately, for the conventional encoding of a categorical variable, a dummy variable column will be identical to its square, which will result in singular matrix for QuadRegression. One solution is to exclude dummy variables in the column expansion done by QuadRegression. Alternatively, a more robust encoding such as the one given in Table 6.7 may be used.

<table>
<thead>
<tr>
<th></th>
<th>encoded $x_j$</th>
<th>dummy $x_j0$, $x_j1$, $x_j2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>English</td>
<td>0</td>
<td>0, 0, 0</td>
</tr>
<tr>
<td>French</td>
<td>1</td>
<td>1, 0, 0</td>
</tr>
<tr>
<td>German</td>
<td>2</td>
<td>0, 1, 0</td>
</tr>
<tr>
<td>Spanish</td>
<td>3</td>
<td>0, 0, 1</td>
</tr>
</tbody>
</table>

Conversion from strings to an integer encoding can be accomplished using the map2Int function in the Converter object within the scalation.linalgebra package. The vector of encoded integers $x_e$ can be made into a matrix using MatrixI ($x_e$). To produce the dummy variable columns the dummyVars function within the ANCOVA companion object may be called. See the first exercise for an example.

Multi-column expansion may done by the caller in cases where there are few categorical variables, by expanding the input data matrix before passing it to the Regression class. The expansion occurs automatically when the ANCOVA class is called. This class performs the expansion and then delegates to the work to the Regression class.

Before continuing the discussion of the ANCOVA class, a restricted form is briefly discussed.

### 6.16.2 ANOVA

An Aanalysis Of Variance (ANOVA) model may be developed using the ANOVA1 class. This type of model comes into play when all input/predictor variables are categorical/binary. One-way Analysis of Variance allows only one binary/categorical treatment variable and is framed in SCALATION using General Linear Model notation and supports the use of one binary/categorical treatment variable $t$. For example, the treatment variable $t$ could indicate the type of fertilizer applied to a field.

The ANOVA1 class in SCALATION only supports one categorical variable, so in general, $x$ consists of $n_{dv} - 1$ dummy variables $d_k$ for $k \in \{1, n_{dv} - 1\}$

$$ y = b \cdot x + \epsilon = b_0 + b_1 d_1 + \ldots + b_l d_l + \epsilon $$

210
where \( l = n_{dv} - 1 \) and \( \epsilon \) represents the residuals (the part not explained by the model). The dummy variables are binary and are used to determine the level/type of a categorical variable. See [http://psych.colorado.edu/~carey/Courses/PSYC5741/handouts/GLM%20Theory.pdf](http://psych.colorado.edu/~carey/Courses/PSYC5741/handouts/GLM%20Theory.pdf).

In ScalaTION, the ANOVA1 class is implemented using regular multiple linear regression. A data/input matrix \( X \) is built from columns corresponding to levels/types for the treatment vector \( t \). As with multiple linear regression, the \( y \) vector holds the response values. Multi-way Analysis of Variance may be performed using the more general ANCOVA class. Also, a more traditional implementation called Anova, not following the General Linear Model approach, is also provided in the stat package.

ANOVA1 Class

Class Methods:

```scala
@param t the treatment/categorical variable vector
@param y the response/output vector
@param fname_ the feature/variable names
@param technique the technique used to solve for \( b \) in \( x.t * x * b = x.t * y \)

class ANOVA1 (t: VectorI, y: VectorD, fname_ : Strings = null, technique: RegTechnique = QR)
    extends Regression (VectorD.one (t.dim) +^: Variable.dummyVars (t), y, fname_, null, technique)
    with ExpandableVariable

    def expand (t: VectorD, nCat: Int = 1): VectorD =
    def predict_ex (zt: VectorD): Double = predict (expand (zt))
```

6.16.3 ANCOVA Implementation

When there is only one categorical/binary variable, \( x \) consists of the usual \( k = n - 1 \) continuous variables \( x_j \). Assuming there is a single categorical variable, call it \( t \), it will need to be expanded into \( n_{dv} - 1 \) additional dummy variables.

\[
t \quad \text{expands to} \quad d = [d_0, \ldots, d_l] \quad \text{where} \quad l = n_{dv} - 2
\]

Therefore, the regression equation becomes the following:

\[
y = b \cdot x + \epsilon = b_0 + b_1 x_1 + \ldots + b_k x_k + b_{k+1} d_0 + \ldots + b_{k+l} d_l + \epsilon \\
\quad \text{(6.95)}
\]

The dummy variables are binary and are used to determine the level of a categorical variable. See [http://www.ams.sunysb.edu/~zhu/ams57213/Team3.pptx](http://www.ams.sunysb.edu/~zhu/ams57213/Team3.pptx).

In general, there may be multiple categorical variables and an expansion will be done for each such variable. Then the data for continuous variable are collected into matrix \( X \), and the values for the categorical variables are collected into matrix \( T \).
In ScalaTion, ANCOVA is implemented using regular multiple linear regression. An augmented data/input matrix $X$ is build from $X_c$ corresponding to the continuous variables with additional columns corresponding to the multiple levels for columns in the treatment matrix $T$. As with multiple linear regression, the $y$ vector holds the response values.

**ANCova Class**

**Class Methods:**

- `@param x_ the data/input matrix of continuous variables`
- `@param t the treatment/categorical variable matrix`
- `@param y the response/output vector`
- `@param fname_ the feature/variable names`
- `@param technique the technique used to solve for b in x.t*x*b = x.t*y`

```scala
class ANCOVA (x_: MatrixD, t: MatrixI, y: VectorD, fname_: Strings = null, technique: RegTechnique = QR) 
extends Regression (x_ ++^ ANCOVA.dummyVars (t), y, fname_, null, technique) 
with ExpandableVariable

def expand (zt: VectorD, nCat: Int = nCatVar): VectorD = 
def predict_ex (zt: VectorD): Double = predict (expand (zt))
```

### 6.16.4 Exercises

1. Use the `map2Int` function in the `Converter` object within the `scalation.linalgebra` package to convert the given strings into encoded integers. Turn this vector into a matrix and pass it into the `dummyVars` function to produce the dummy variable columns. Print out the values $x_e$, $x_m$ and $x_d$.

```scala
val x1 = VectorS ("English", "French", "German", "Spanish")
val (xe, map) = Converter.map2Int (x1) // map strings to integers
val xm = MatrixI (xe) // form a matrix from vector
val xd = ANCOVA.dummyVars (xm) // make dummy variable columns

Add code to recover the string values from the encoded integers using the returned `map`.

2. Use the `ANCOVAl` class to predict responses based on treatment levels trained using the following treatment $t$ and response $y$ vectors. Plot the given versus predicted responses.

```scala
val t = VectorI (1, 1, 1, 2, 2, 2, 3, 3, 3) // treatment level data
val y = VectorD (755.0, 865.0, 815.0, 
442.0, 420.0, 401.0, 
282.0, 250.0, 227.0)

val levels = 3
```
val arg = new ANOVA1 (t, y, levels)
arg.analyze ()
println (arg.report)

banner ("test predictions")
val yp = new VectorD (y.dim)
for (i <- yp.range) yp(i) = arg.predict (t(i))
println (s" y = $y \n yp = $yp")
new Plot (t.toDouble, y, yp, "ANOVA1")

3. Compare the results of using the ANCOVA class versus the Regression class for the following dataset.

// 6 data points: constant term, x_1 coordinate, x_2 coordinate
val x = new MatrixD ((6, 3), 1.0, 36.0, 66.0, // 6-by-3 matrix
                      1.0, 37.0, 68.0,
                      1.0, 47.0, 64.0,
                      1.0, 32.0, 53.0,
                      1.0, 42.0, 83.0,
                      1.0, 1.0, 101.0)

val t = new MatrixI ((6, 1), 0, 0, 1, 1, 2, 2) // treatments levels
val y = VectorD (745.0, 895.0, 442.0, 440.0, 643.0, 1598.0) // response vector
val z = VectorD (1.0, 20.0, 80.0, 1) // new instance
val ze = VectorD (1.0, 20.0, 80.0, 2, 1) // encoded and expanded

val xt = x ++^ t.toDouble
val rg = new Regression (xt, y) // treated as ordinal
val anc = new ANCOVA (x, t, y) // treated as categorical
Chapter 7

Classification

When the output/response $y$ is defined on small domains (categorical response), e.g., $\mathbb{B}$ or $\mathbb{Z}_k = \{0, 1, \ldots, k-1\}$, then the problem shifts from prediction to classification. This facilitates giving the response meaningful class names, e.g., low-risk, medium-risk and high-risk. When the response is discrete, but unbounded (e.g., Poisson Regression), the problem is considered to be a prediction problem.

$$y = f(x; \mathbf{b}) + \epsilon$$

As with Regression in continuous domains, some of the modeling techniques in this chapter will focus on estimating the conditional expectation of $y$ given $x$.

$$y = \mathbb{E}[y|x] + \epsilon$$

(7.1)

Others will focus on maximizing the conditional probability of $y$ given $x$, i.e., finding the conditional mode.

$$y^* = \text{argmax } P(y|x) = \mathbb{M}[y|x]$$

(7.2)

Rather than find a real number that is the best predictor, one of a set of distinct given values (e.g., 0 (false), 1 (true); negative (-1), positive (1); or low (0), medium (1), high (2)) is chosen. Abstractly, we can label the classes $C_0, C_1, \ldots, C_{k-1}$. In the case of classification, the train function is still used, but the classify method replaces the predict method.

Let us briefly contrast the two approaches based on the two equations (5.1 and 5.2). For simplicity, a selection (not classification) problem is used. Suppose that the goal is to select one of three actors ($y \in \{0, 1, 2\}$) such that they have been successful in similar films, based on characteristics (features) of the films (captured in variables $x$). From the data, the frequency of success for the actors in similar films has been 20, 0 and 30, respectively. Consequently, the expected value is 1.2 and one might be tempted to select actor 1 (the worst choice). Instead selecting the actor with maximum frequency (and therefore probability) will produce the best choice (actor 2).
7.1 Classifier

The Classifier trait provides a common framework for several classifiers such as NaiveBayes.

Trait Methods:

```scala
trait Classifier

  def size: Int // typically = m
  def train (itest: IndexedSeq[Int]): Classifier
  def train (testStart: Int, testEnd: Int): Classifier = train (testStart until testEnd)
  def train (): Classifier = train (0, 0)
  def classify (z: VectorI): (Int, String, Double)
  def classify (z: VectorD): (Int, String, Double)
  def test (itest: IndexedSeq[Int]): Double
  def test (testStart: Int, testEnd: Int): Double = test (testStart until testEnd)
  def crossValidate (nx: Int = 10, show: Boolean = false): Double =
  def crossValidateRand (nx: Int = 10, show: Boolean = false): Double =
  def fit (y: VectorI, yp: VectorI, k: Int = 2): VectorD =
  def fitLabel: Seq[String] = Seq("acc", "prec", "recall", "kappa")
  def fitMap (y: VectorI, yp: VectorI, k: Int = 2): Map[String, String] =
  def reset ()
```

For modeling, a user chooses one the of classes extending the trait Classifier (e.g., DecisionTreeID3) to instantiate an object. Next the train method would be typically called. While the modeling techniques in the last chapter focused on minimizing errors, the focus in this chapter will be on minimizing incorrect classifications. Generally, this is done by dividing a dataset up into a training dataset and test dataset. A way to utilize one dataset to produce multiple training and test datasets is called cross-validation.

As discussed in the Model Validation section in the Prediction chapter, k-fold cross-validation is a useful general purpose strategy for examining the quality of a model. The first cross-validation method takes the number of folds k (nx in the software) and a show flag. It performs k iterations of training (train method) and testing (test method).

```scala
def crossValidate (nx: Int = 10, show: Boolean = false): Double =
{
  val testSize = size / nx // number of instances in test set
  var sum = 0.0
  for (it <- 0 until nx) {
    val testStart = it * testSize // test set start index (inclusive)
    val testEnd = testStart + testSize // test set end index (exclusive)
    train (testStart, testEnd) // train on opposite instances
    val acc = test (testStart, testEnd) // test on test set
    sum += acc // accumulate accuracy
    if (show) println (s"crossValidate: for it = $it, acc = $acc")
  }
  sum
}
```
The second cross-validation method is more complicated, but usually preferred, since it randomizes the instances selected for the test dataset, so that patterns coincidental to the index are broken up.

```
def crossValidateRand(nx: Int = 10, show: Boolean = false): Double =
```

The `crossValidateRand` method calls the following methods:

- `train (itest: IndexedSeq[Int])`
- `test (itest: IndexedSeq[Int])`

while the `crossValidate` method calls the following methods:

- `train (testStart: Int, testEnd: Int)`
- `test (testStart: Int, testEnd: Int)`

Once a model/classifier has been sufficiently trained and tested, it is ready to be put into practice on new data via the `classify` method.
7.2 ClassifierInt

The ClassifierInt abstract class provides a common foundation for several classifiers that operate on integer-valued data.

Class Methods:

- @param x the integer-valued data vectors stored as rows of a matrix
- @param y the integer-valued classification vector, where y_i = class for row i of matrix x
- @param fn the names of the features/variables
- @param k the number of classes
- @param cn the names for all classes
- @param hparam the hyper-parameters

abstract class ClassifierInt (x: MatrixI, y: VectorI, protected var fn: Strings = null, k: Int, protected var cn: Strings = null, hparam: HyperParameter) extends ConfusionFit (y, k) with Classifier

def size: Int = m
def vc_default: Array[Int] = Array.fill (n)(2)
def vc_fromData: Array[Int] = (for (j <- x.range2) yield x.col(j).max() + 1).toArray
def vc_fromData2 (rg: Range): Array[Int] = (for (j <- rg) yield x.col(j).max() + 1).toArray
def shiftToZero () { x -= VectorI (for (j <- x.range2) yield x.col(j).min()) }
def test (itest: Ints): Double =
def test (xx: MatrixI, yy: VectorI): Double =
def eval (xx: MatrixD, yy: VectorD = null): ClassifierInt =
def crossValidate (nx: Int = 10, show: Boolean = false): Array[Statistic] =
def crossValidateRand (nx: Int = 10, show: Boolean = false): Array[Statistic] =
def hparameter: HyperParameter = hparam
def report: String =
def classify (z: VectorD): (Int, String, Double) = classify (roundVec (z))
def classify (xx: MatrixI = x): VectorI =
def calcCorrelation: MatrixD =
def calcCorrelation2 (zrg: Range, xrg: Range): MatrixD =
def featureSelection (TOL: Double = 0.01)

ClassifierInt provides methods to determine the value count (vc) for the features/variables. A method to shift values in a vector toward zero by subtracting the minimum value. It has base implementations for test methods and methods for calculating correlations. Finally, the featureSelection method will eliminate features that have little positive impact on the quality of the model. Rather than considering all n features/variables, a proper subset fset ⊂ {0, 1, ..., n−1} of the features is selected. Various algorithms can be used to search for an optimal feature set fset. SCALATION currently uses a simple backward elimination algorithm that removes the least significant feature, in terms of cross-validation accuracy, in each round.

218
7.3 Confusion Matrix

The `ConfusionMat` class provides methods to produce a confusion matrix and associated quality metrics. In Scala when \( k = 2 \), the confusion matrix \( C \) is configured as follows:

\[
\begin{bmatrix}
c_{00} = tn & c_{01} = fp \\
c_{10} = fn & c_{11} = tp
\end{bmatrix}
\]

The first column indicates the classification is negative (no or 0), while the second column indicates it is positive (yes or 1). The first letter indicates whether the classification is correct (true) or not (false). The row \((0, 1)\) indicates the actual class label, while the column \((0, 1)\) indicates the response of the classifier.

Class Methods:

@param \( y \) the actual class labels
@param \( yp \) the predicted class labels
@param \( k \) the number class values

```scala
class ConfusionMat (y: Vector[Int], yp: Vector[Int], k: Int = 2)

def confusion: Matrix = conf

def pos_neg (con: Matrix = conf): (Double, Double, Double, Double) =

def accuracy: Double = con.trace / con.sum.toDouble

def prec_recl: (Vector[Double], Vector[Double], Double, Double) =

def f1_measure (prec: Double, recl: Double): Double = 2.0 * prec * recl / (prec + recl)

def kappa: Double =
```

219
7.4 Bayes Classifier

The `BayesClassifier` abstract class provides common methods for several Bayes classifiers.

Class Methods:

- **@param x** the integer-valued data vectors stored as rows of a matrix
- **@param y** the class vector, where y(l) = class for row l of the matrix x, x(l)
- **@param fn_** the names for all features/variables
- **@param k** the number of classes
- **@param cn_** the names for all classes

```scala
abstract class BayesClassifier (x: Matrix[Int], y: Vector[Int], fn_: Strings = null, k: Int = 2,
      cn_: Strings = null)
     extends ClassifierInt (x, y, fn_, k, cn_) with BayesMetrics

def toggleSmooth () { smooth = ! smooth }
def calcCMI (idx: IndexedSeq[Int], vca: Array[Int]): MatrixD =
def cmiJoint (p_y: Vector[Double], p_Xy: HMatrix3[Double], p_XZy: HMatrix5[Double]): MatrixD =
def getParent: Any = null
protected def updateFreq (i: Int) {}
def printClassProb () { println (s"ClassProb = $p_y") }
```
7.5 Null Model

The NullModel class implements a simple Classifier suitable for discrete input data. Corresponding to the Null Model in the Prediction chapter, one could imagine estimating probabilities for outcomes of a random variable \( y \). Given an instance, this random variable indicates the classification or decision to be made. For example, it may be used for a decision on whether or not to grant a loan request. The model may be trained by collecting a training dataset. Probabilities may be estimated from data stored in an \( m \)-dimensional response/classification vector \( y \) within the training dataset. These probabilities are estimated based on the frequency \( \nu \) (nu in the code) with which each class value occurs.

\[
\nu(y = c) = |\{i \mid y_i = c\}| = m_c
\]

The right hand side is simply the size of the set containing the instance/row indices where \( y_i = c \) for \( c = 0, \ldots, k - 1 \). The probability that random variable \( y \) equals \( c \) can be estimated by the number of elements in the vector \( y \) where \( y_i \) equals \( c \) divided by the total number of elements.

\[
P(y = c) = \frac{\nu(y = c)}{m} = \frac{m_c}{m}
\] (7.3)

Exercise 1 below is the well-known toy classification problem on whether to play tennis (\( y = 1 \)) or not (\( y = 0 \)) based on weather conditions. Of the 14 days (\( m = 14 \)), tennis was not played on 5 days and was played on 9 days, i.e.,

\[
P(y = 0) = \frac{5}{14} \quad \text{and} \quad P(y = 1) = \frac{9}{14}
\]

This information, class frequencies and class probabilities, can be placed into a Class Frequency Vector (CFV) as shown in Table 7.1 and

<table>
<thead>
<tr>
<th>Table 7.1: Class Frequency Vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>( y )</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

a Class Probability Vector (CPV) as shown in Table 7.8

<table>
<thead>
<tr>
<th>Table 7.2: Class Probability Vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>( y )</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Picking the maximum probability case, one should always predict that tennis will be played, i.e., \( y^* = 1 \).

This modeling technique should outperform purely random guessing, since it factors in the relative frequency with which tennis is played. As with the NullModel for prediction, more sophisticated modeling techniques should perform better than this NullModel for classification. If they are unable to provide higher accuracy, they are of questionable value.
NullModel Class

Class Methods:

@param y the class vector, where y(i) = class for instance i
@param k the number of classes
@param cn_ the names for all classes

class NullModel (y: VectorI, k: Int = 2, cn_ : Strings = null)
  extends ClassifierInt (null, y, null, k, cn_)

def train (itest: IndexedSeq [Int]): NullModel =
def classify (z: VectorI): (Int, String, Double) =
override def classify (xx: Matrix): VectorI = VectorI.fill (xx.dim1)(p_y.argmax ()
override def test (itest: IndexedSeq [Int]): Double =
def reset () { /* NA */ }

The train method for this modeling technique is very simple. It takes the parameter itest as input that indicates which instance/row indices make up the test dataset. The training dataset is made up of the rest on the instances.

def train (itest: IndexedSeq [Int]): NullModel =
{
  val idx = 0 until m diff itest // training dataset - opposite of testing
  nu_y = frequency (y, k, idx) // frequency vector for y
  p_y = toProbability (nu_y, idx.size) // probability vector for y
  if (DEBUG) println (s" nu_y = $nu_y \n p_y = $p_y")
  this
} // train

Typically, one dataset is divided into a training dataset and testing dataset. For example, 80% may be used for training (estimating probabilities) with the remaining 20% used for testing the accuracy of the model. Furthermore, this is often done repeatedly as part of a cross-validation procedure.

The frequency and toProbability are functions from the Probability object in the scalation.analytics package.

def frequency (x: VectorI, k: Int, idx_ : IndexedSeq [Int] = null): VectorI =
{
  val idx = if (idx_ == null) IndexedSeq.range (0, x.dim) else idx_
  val nu = new VectorI (k)
  for (i <- idx) nu(x(i)) += 1
  nu
} // frequency
def toProbability (nu: VectoI, n: Int): VectoD =
{
  val nd = n.toDouble
  VectorD (for (i <- nu.range) yield nu(i) / nd)
} // toProbability

7.5.1 Exercises

1. The NullModel classifier can be used to solve problems such as the one below. Given the Outlook, Temperature, Humidity, and Wind determine whether it is more likely that someone will (1) or will not (0) play tennis. The data set is widely available on the Web. It is also available in scalation.analytics.classifier.ExampleTennis. Use the NullModel for classification and evaluate its effectiveness using cross-validation.

//:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
/** The 'ExampleTennis' object is used to test all integer based classifiers.
 * This is the well-known classification problem on whether to play tennis
 * based on given weather conditions. Applications may need to slice 'xy'.
 * val x = xy.sliceCol (0, 4) // columns 0, 1, 2, 3
 * val y = xy.col (4) // column 4
 * @see euclid.nmu.edu/~mkowalcz/cs495f09/slides/lesson004.pdf
 */
object ExampleTennis
{
  // dataset ----------------------------------------------------------------
  // x0: Outlook: Rain (0), Overcast (1), Sunny (2)
  // x1: Temperature: Cold (0), Mild (1), Hot (2)
  // x2: Humidity: Normal (0), High (1)
  // x3: Wind: Weak (0), Strong (1)
  // y: the response/classification decision
  // variables/features: x0 x1 x2 x3 y // combined data matrix
  val xy = new MatrixI ((14, 5), 2, 2, 1, 0, 0, // day 1
    2, 2, 1, 1, 0, // day 2
    1, 2, 1, 0, 1, // day 3
    0, 1, 1, 0, 1, // day 4
    0, 0, 0, 1, 0, // day 5
    0, 0, 0, 1, 1, // day 6
    1, 0, 0, 1, 1, // day 7
    2, 1, 1, 0, 0, // day 8
    2, 0, 0, 0, 1, // day 9
    0, 1, 0, 0, 1, // day 10
    2, 1, 0, 1, 1, // day 11
    1, 1, 1, 1, 1, // day 12
    1, 2, 0, 0, 1, // day 13
    0, 1, 1, 1, 0) // day 14
}
val fn = Array ("Outlook", "Temp", "Humidity", "Wind")   // feature names
val cn = Array ("No", "Yes")                // class names for y
val k = cn.size                           // number of classes
} // ExampleTennis object

7.6 Naive Bayes

The NaiveBayes class implements a Naive Bayes (NB) Classifier suitable for discrete input data. A Bayesian Classifier is a special case of a Bayesian Network where one of the random variables is distinguished as the basis for making decisions, call it random variable $y$, the class variable. The NullModel ignores weather conditions which are the whole point of the ExampleTennis exercise. For Naive Bayes, weather conditions (or other data relevant to decision making) are captured in an $n$-dimensional vector of random variables.

$$\mathbf{x} = [x_0, \ldots, x_{n-1}],$$

For the exercise, $n = 4$ where $x_0$ is Outlook, $x_1$ is Temperature, $x_2$ is Humidity, and $x_3$ is Wind. The decision should be conditioned on the weather, i.e., rather than computing $P(y)$, we should compute $P(y|x)$. Bayesian classifiers are designed to find the class (value for random variable $y$) that maximizes the conditional probability of $y$ given $x$.

It may be complex and less robust to estimate $P(y|x)$ directly. Often it is easier to examine the conditional probability of $x$ given $y$. This answers the question of how likely it is that the input data comes from a certain class $y$. Flipping the perspective can be done using Bayes Theorem.

$$P(y|x) = \frac{P(x|y)P(y)}{P(x)}$$

Since the denominator is the same for all $y$, it is sufficient to maximize the right hand side of the following proportionality statement.

$$P(y|x) \propto P(x|y)P(y)$$

Notice that the right hand side is the joint probability of all the random variables.

$$P(x, y) = P(x|y)P(y) \tag{7.4}$$

One could in principle represent the joint probability $P(x, y)$ or the conditional probability $P(x|y)$ in a matrix. Unfortunately, with 30 binary random variables, the matrix would have over one billion rows and exhibit issues with sparsity. Bayesian classifiers will factor the probability and use multiple matrices to represent the probabilities.

7.6.1 Factoring the Probability

A Bayesian classifier is said to be na"ive, when it is assumed that the $x_j$’s are sufficiently uncorrelated to factor $P(x|y)$ into the product of their conditional probabilities (independence rule).

$$P(x|y) = \prod_{j=0}^{n-1} P(x_j|y)$$

Research has shown that even though the assumption that given response/class variable $y$, the $x$-variables are independent is often violated by a dataset, Naive Bayes still tends to perform well [116]. Substituting this factorization in equation 5.4 yields

$$P(x, y) = P(y) \prod_{j=0}^{n-1} P(x_j|y) \tag{7.5}$$
The classification problem then is to find the class value for $y$ that maximizes this probability, i.e., let $y^*$ be the argmax of the product of the class probability $P(y)$ and all the conditional probabilities $P(x_j | y)$. The argmax is the value in the domain $D_y = \{0, \ldots, k - 1\}$ that maximizes the probability.

$$y^* = \arg \max_{y \in \{0, \ldots, k - 1\}} P(y) \prod_{j=0}^{n-1} P(x_j | y)$$

(7.6)

### 7.6.2 Estimating Conditional Probabilities

For Integer-based classifiers $x_j \in \{0, 1, \ldots, v_{c_j} - 1\}$ where $v_{c_j}$ is the value count for the $j^{th}$ variable/feature (i.e., the number of distinct values). The Integer-based Naïve Bayes classifier is trained using an $m$-by-$n$ data matrix $X$ and an $m$-dimensional classification vector $y$. Each data vector/row in the matrix is classified into one of $k$ classes numbered $0, 1, \ldots, k - 1$. The frequency or number of instances where column vector $x_j = h$ and vector $y = c$ is as follows:

$$\nu(x_j = h, y = c) = |\{i \mid x_{ij} = h, y_i = c\}|$$

The conditional probability for random variable $x_j$ given random variable $y$ can be estimated as the ratio of two frequencies.

$$P(x_j = h \mid y = c) = \frac{\nu(x_j = h, y = c)}{\nu(y = c)}$$

(7.7)

In other words, the conditional probability is the ratio of the joint frequency count for a given $h$ and $c$ divided by the class frequency count for a given $c$. These frequency counts can be collected into Joint Frequency Matrices/Tables (JFTs) and a Class Frequency Vector (CFV). From these, it is straightforward to compute Conditional Probability Matrices/Tables (CPTs) and a Class Probability Vector (CPV).

### Example Tennis Problem

For the Example Tennis problem, the Joint Frequency Matrix/Table (JFT) for Outlook random variable $x_0$ is shown in Table 7.3.

$$\nu(x_0 = h, y = c) \text{ for } h \in \{0, 1, 2\}, \ c \in \{0, 1\}$$

Table 7.3: JFT for $x_0$

<table>
<thead>
<tr>
<th>$x_0 \setminus y$</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

The column sums in the above matrix are 5 and 9, respectively. The corresponding Conditional Probability Matrix/Table (CPT) for random variable $x_0$, i.e., $P(x_0 = h \mid y = c)$, is computed by dividing each entry in the joint frequency matrix by its column sum.
Table 7.4: CPT for $x_0$

<table>
<thead>
<tr>
<th>$x_0 \setminus y$</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2/5</td>
<td>3/9</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>4/9</td>
</tr>
<tr>
<td>2</td>
<td>3/5</td>
<td>2/9</td>
</tr>
</tbody>
</table>

Continuing with the ExampleTennis problem, the Joint Frequency Matrix/Table for Wind random variable $x_3$ is shown in Table 7.5.

$$\nu(x_3 = h, y = c) \quad \text{for} \quad h \in \{0, 1\}, \ c \in \{0, 1\}$$

Table 7.5: JFT for $x_3$

<table>
<thead>
<tr>
<th>$x_3 \setminus y$</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

As expected, the column sums in the above matrix are again 5 and 9, respectively. The corresponding Conditional Probability Matrix/Table for random variable $x_0$, i.e., $P(x_0 = h \mid y = c)$, is computed by dividing each entry in the joint frequency matrix by its column sum as shown in table 7.6.

Table 7.6: CPT for $x_3$

<table>
<thead>
<tr>
<th>$x_3 \setminus y$</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2/5</td>
<td>6/9</td>
</tr>
<tr>
<td>1</td>
<td>3/5</td>
<td>3/9</td>
</tr>
</tbody>
</table>

Similar matrices/tables can be created for the other random variables: Temperature $x_1$ and Humidity $x_2$.

### 7.6.3 Laplace Smoothing

When there are several possible class values, a dataset may exhibit zero instances for a particular class. This will result in a zero in the CFV vector and cause a divide-by-zero error when computing CPTs. One way to avoid the divide-by-zero, is to add one ($m_e = 1$) fake instance for each class, guaranteeing no zeros in the CFV vector. If m-estimates are used, the conditional probability is adjusted slightly as follows:

$$P(x_j = h \mid y = c) = \frac{\nu(x_j = h, y = c) + m_e / vc_j}{\nu(y = c) + m_e}$$

where $m_e$ is the parameter used for the m-estimate. The term added to the numerator, takes the one (or $m_e$) instance(s) and adds uniform probability for each possible values for $x_j$ of which there are $vc_j$ of them.
Table 7.7 shows the result of adding 1/3 in the numerator and 1 in the denominator, (e.g., for \( h = 0 \) and \( c = 0 \), \( (2 + 1/3)/(5 + 1) = 7/18 \)).

Table 7.7: CPT for \( x_0 \) with \( m_e = 1 \)

<table>
<thead>
<tr>
<th>( x_0 ) ( \backslash ) ( y )</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>7/18</td>
<td>10/30</td>
</tr>
<tr>
<td>1</td>
<td>1/18</td>
<td>13/30</td>
</tr>
<tr>
<td>2</td>
<td>10/18</td>
<td>7/30</td>
</tr>
</tbody>
</table>

Another problem is when a conditional probability in a CPT is zero. If any CPT has a zero element, the corresponding product for the column (where the CPV and CPTs are multiplied) will be zero no matter how high the other probabilities may be. This happens when the frequency count is zero in the corresponding JFT (see element (1, 0) in Table 7.3). The question now is whether this is due to the combination of \( x_0 = 1 \) and \( y = 0 \) being highly unlikely, or that the dataset is not large enough to exhibit this combination. Laplace smoothing guards against this problem as well.

Other values may be used for \( m_e \) as well. SCALATION uses a small value for the default \( m_e \) to reduce the distortion of the CPTs.

### 7.6.4 Hypermatrices

The values within the class probability table and the conditional probability tables are assigned by the `train` method. In SCALATION, vectors and third level hypermatrices are used for storing frequencies (\( \text{nu} \)) and probabilities (\( \text{p} \)).

```scala
val nu_y = new VectorI (k) // frequency of \( y \) with classes 0, \ldots, \( k-1 \)
val nu_Xy = new HMatrix3[Int] (k, n, vc) // joint frequency of features \( x_j \)’s and class \( y \)
val p_y = new VectorD (k) // probability of \( y \) with classes 0, \ldots, \( k-1 \)
val p_Xy = new HMatrix3[Double] (k, n, vc) // conditional probability of features \( x_j \)’s
                                                   // given class \( y \)
```

where \( k \) is the number of class values, \( n \) is the number of \( x \)-random variables (features) and \( vc \) is the value count per feature. Note, one third level hypermatrix is able to store multiple matrices.

For the `ExampleTennis` problem where \( k = 2 \) and \( n = 4 \), the frequency counters (\( \text{nu} \)) would be defined as follows:

```scala
val nu_y = new VectorI (2) // Class Frequency Vector (CFV)
val nu_Xy = new HMatrix3[Int] (2, 4, Array (3, 3, 2, 2)) // all Joint Frequency Tables (JFTs)
```

The dimensionality of the hypermatrix \( \text{nu}_Xy \) could have been 2-by-4-by-3, but this would in general be wasteful of space. Each variable only needs space for the values it allows, as indicated by \( \text{Array} \ (3, 3, 2, 2) \) for the value counts \( vc \). The user may specify the optional \( vc \) parameter in the constructor call. If the \( vc \) parameter is unspecified, then SCALATION uses the `vc_fromData` method to determine the value counts from the training data. In some cases, the test data may include a value unseen in the training data. Currently, SCALATION requires the user to pass \( vc \) into the constructor in such cases.
7.6.5 The classify Method

A new instance can now be classified by simply matching its values with those in the class probability table and conditional probability tables and multiplying all the entries. This is done for all $k$ class values and the class with the highest product is chosen.

```scala
def classify (z: Vector[Int]): (Int, String, Double) =
{
  val prob = new VectorD (p_y)
  for (c <- 0 until k; j <- 0 until n) prob(c) *= p_Xy(c, j, z(j)) // P(x_j = z_j | y = c)
  val best = prob.argmax () // class with the highest relative probability
  (best, cn(best), prob(best)) // return the best class and its name
}
```

In situations where there are many variables/features the product calculation may underflow. An alternative calculation would be to take the log of the probability.

$$\log P(x, y) = P(y) + \sum_{j=0}^{n-1} P(x_j|y)$$

7.6.6 Feature Selection

Suppose that $x_1$ and $x_2$ are not considered useful for classifying a day as to its suitability for playing tennis. For $z = [2, 1]$, i.e., $z_0 = 2$ and $z_3 = 1$, the two relative probabilities are the following:

<table>
<thead>
<tr>
<th>$z$</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y$</td>
<td>5/14</td>
<td>9/14</td>
</tr>
<tr>
<td>$z_0$</td>
<td>3/5</td>
<td>2/9</td>
</tr>
<tr>
<td>$z_3$</td>
<td>3/5</td>
<td>3/9</td>
</tr>
<tr>
<td>$z, y$</td>
<td>9/70</td>
<td>1/21</td>
</tr>
</tbody>
</table>

The two probabilities are approximately 0.129 for $c = 0$ (Do not Play) and 0.0476 for $c = 1$ (Play). The higher probability is for $c = 0$.

To perform feature selection in a systematic way SCALATION provides an fset array that indicates the features/variables to be kept in the model. This array is assigned by calling the featureSelection method in the ClassifierInt abstract class.

7.6.7 Efficient Cross-Validation

There are actually two classes NaiveBayes0 and NaiveBayes. The former uses conventional “additive” cross-validation where frequency counters are reset to zero and are incremented for each fold. The latter uses a more efficient “subtractive” cross-validation where frequency counters are reset to the counts for the entire dataset and are decremented for each fold.
NaiveBayes0 Class

Class Methods:

@param x the integer-valued data vectors stored as rows of a matrix
@param y the class vector, where y(l) = class for row l of the matrix x, x(l)
@param fn_ the names for all features/variables
@param k the number of classes
@param cn_ the names for all classes
@param vc_ the value count (number of distinct values) for each feature
@param me use m-estimates (me == 0 => regular MLE estimates)

class NaiveBayes0 (x: MatriI, y: VectoI, fn_: Strings = null, k: Int = 2, cn_: Strings = null, vc_: Array[Int] = null, me: Double = me_default)
extends BayesClassifier (x, y, fn, k, cn)

protected var vc: Array[Int] = null
protected def frequencies (idx: IndexedSeq[Int]): Unit = NaiveBayes0 =
protected def updateFreq (i: Int) =
protected def lclassify (z: VectoI): (Int, String, Double) =
def reset ()
def printConditionalProb ()
def classify (z: VectoI): (Int, String, Double) =
def frequenciesAll ()

NaiveBayes Class

This class is the same as the one above, but uses an optimized cross-validation technique.

Class Methods:

@param x the integer-valued data vectors stored as rows of a matrix
@param y the class vector, where y(l) = class for row l of the matrix x, x(l)
@param fn_ the names for all features/variables
@param k the number of classes
@param cn_ the names for all classes
@param vc_ the value count (number of distinct values) for each feature
@param me use m-estimates (me == 0 => regular MLE estimates)

class NaiveBayes (x: MatriI, y: VectoI, fn_: Strings = null, k: Int = 2, cn_: Strings = null, vc_: Array[Int] = null, me: Float = me_default)
extends NaiveBayes0 (x, y, fn, k, cn, vc_, me)
def frequenciesAll ()
7.6.8 Exercises

1. Complete the ExampleTennis problem given in this section by creating CPTs for random variables $x_1$ and $x_2$ and then computing the relative probabilities for $z = [2, 2, 1, 1]$.

2. Use SCALATion’s Integer-based NaiveBayes class to build a classifier for the ExampleTennis problem.

   ```scala
   import scalation.analytics.classifier.ExampleTennis._
   println("Tennis Example")
   println("xy = " + xy) // combined matrix [x | y]
   val nb = NaiveBayes(xy, fn, k, cn, null, 0) // create a classifier
   nb.train() // train the classifier
   val z = VectorI(2, 2, 1, 1) // new data vector
   println(s"classify ($z) = ${nb.classify(z)}") // classify z
   ```

3. Compare the confusion matrix, accuracy, precision and recall of NaiveBayes on the full dataset to that of NullModel.

   ```scala
   val x = xy.sliceCol(0, xy.dim2 - 1) // data matrix
   val y = xy.col(xy.dim2 - 1) // response/class label vector
   val yp = new VectorI(xy.dim1) // predicted class label vector
   for (i <- x.range1) {
     yp(i) = nb.classify(x(i))._1
     println(s"Use nb to classify ($x(i)) = ${yp(i)}")
   } // for
   val cm = new ConfusionMat(y, yp, k) // confusion matrix
   println("Confusion Matrix = " + cm.confusion)
   println("accuracy = " + cm.accuracy)
   println("prec-recall = " + cm.prec_recl)
   ```

4. Compare the accuracy of NaiveBayes using 10-fold cross-validation (cv) to that of NullModel.

   ```scala
   println("nb cv accu = " + nb.crossValidateRand(10, true)) // 10-fold cross-validation
   ```

5. Compare the confusion matrix, accuracy, precision and recall of RoundRegression on the full dataset to that of NullModel.

6. Perform feature selection on the ExampleTennis problem. Which feature/variable is removed from the model, first, second and third. Explain the basis for the featureSelection method’s decision to remove a feature.

7. Use the Integer-based NaiveBayes class to build a classifier for the Breast Cancer problem (data in breast-cancer.arff file). Compare its accuracy to that of NullModel.
7.7 Tree Augmented Naïve Bayes

The TANBayes class implements a Tree Augmented Naïve (TAN) Bayes Classifier suitable for discrete input data. Unlike Naïve Bayes, a TAN model can capture more, yet limited dependencies between variables/features. In general, \( x_j \) can be dependent on the class \( y \) as well as one other variable \( x_{p(j)} \). Representing the dependency pattern graphically, \( y \) becomes a root node of a Directed Acyclic Graph (DAG), where each node/variable has at most two parents.

Starting with the joint probability given in equation 5.5,
\[
P(x, y) = P(x|y) P(y)
\]
we can obtain a better factored approximation (better than Naïve Bayes) by keeping the most important dependencies amongst the random variables. Each \( x_j \), except a selected \( x \)-root, \( x_r \), will have one \( x \)-parent \( (x_{p(j)}) \) in addition to its \( y \)-parent. The dependency pattern among the \( x \) random variables forms a tree and this tree augments the Naïve Bayes structure where each \( x \) random variable has \( y \) as its parent.

\[
P(x, y) = P(y) \prod_{j=0}^{n-1} P(x_j|x_{p(j)}, y)
\]
Since the root \( x_r \), has no \( x \)-parent, it can be factored out as special case.

\[
P(x, y) = P(y) P(x_r|y) \prod_{j \neq r} P(x_j|x_{p(j)}, y)
\]
As with Naïve Bayes, the goal is to find an optimal value for the random variable \( y \) that maximizes the probability.

\[
y^* = \arg\max_{y \in D} P(y) P(x_r|y) \prod_{j=0}^{n-1} P(x_j|x_{p(j)}, y)
\]

7.7.1 Structure Learning

Naïve Bayes has a very simple structure that does not require any structural learning. TAN Bayes, on the other hand, requires the tree structure among the \( x \) random variables/nodes to be learned. Various algorithms can be used to select the best parent \( x_{p(j)} \) for each \( x_j \). SCALATION does this by constructing a maximum spanning tree where the edge weights are Conditional Mutual Information (alternatively correlation).

The Mutual Information (MI) between two random variables \( x_j \) and \( x_l \) is
\[
I(x_j; x_l) = \sum_{x_j} \sum_{x_l} p(x_j, x_l) \log \frac{p(x_j, x_l)}{p(x_j)p(x_l)}
\]
The Conditional Mutual Information (CMI) between two random variables \( x_j \) and \( x_l \) given a third random variable \( y \) is
\[
I(x_j; x_l|y) = \]
The steps involved in the structure learning algorithm for TAN Bayes are the following:

1. Compute the CMI \( I(x_j; x_l|y) \) for all combinations of random variables, \( j \neq l \).
2. Build a complete undirected graph with a node for each $x_j$ random variable. The weight on undirected edge $\{x_j, x_l\}$ is its CMI value.

3. Apply a Maximum Spanning Tree algorithm (e.g., Prim or Kruskal) to the undirected graphs to create a maximum spanning tree (those $n - 1$ edges that (a) connect all the nodes, (b) form a tree, and (c) have maximum cumulative edge weights). Note, SCALaTIon’s MinSpanningTree in the scalation.graph.db package can be used with parameter min = false.

4. Pick one of the random variables to be the root node $x_r$.

5. To build the directed tree, start with root node $x_r$ and traverse from there giving each edge directionality as you go outward from the root.

### 7.7.2 Conditional Probability Tables

For the ExampleTennis problem limited to two variables, $x_0$ and $x_3$, suppose that structure learning algorithm found the $x$-parents as shown in Table 7.9.

Table 7.9: Parent Table

<table>
<thead>
<tr>
<th>$x_j$</th>
<th>$x_{p(j)}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_0$</td>
<td>$x_3$</td>
</tr>
<tr>
<td>$x_3$</td>
<td>null</td>
</tr>
</tbody>
</table>

In this case, the only modification to the CPV and CPTs from the Naïve Bayes solution, is that the JFT and CPT for $x_0$ are extended. The extended Joint Frequency Table (JFT) for $x_0$ is shown in Table 7.10.

Table 7.10: Extended JFT for $x_0$

<table>
<thead>
<tr>
<th>$x_0 \setminus x_3, y$</th>
<th>0, 0</th>
<th>0, 1</th>
<th>1, 0</th>
<th>1, 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

The column sums are 2, 6, 3, 3, respectively. Again they must add up to same total of 14. Dividing each element in the JFT by its column sum yields the extended Conditional Probability Table (CPT) shown in Table 7.11.

In general for TANBayes, the $x$-root will have a regular CPT, while all other $x$-variables will have an extended CPT, i.e., the extended CPT for $x_j$ is calculated as follows:

$$P(x_j = h \mid x_p = l, y = c) = \frac{\nu(x_j = h, x_p = l, y = c)}{\nu(x_p = l, y = c)}$$  \hspace{1cm} (7.11)
Table 7.11: Extended CPT for \( x_{0,0} \)

<table>
<thead>
<tr>
<th>( x_{0} )</th>
<th>( x_{0,0} )</th>
<th>( 0, 0 )</th>
<th>( 0, 1 )</th>
<th>( 1, 0 )</th>
<th>( 1, 1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1/2</td>
<td>2/3</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1/3</td>
<td>0</td>
<td>2/3</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1/6</td>
<td>1/3</td>
<td>1/3</td>
<td></td>
</tr>
</tbody>
</table>

### 7.7.3 Smoothing

The analog of Laplace smoothing used in Naïve Bayes is the following.

\[
P(x_j = h | x_p = l, y = c) = \frac{\nu(x_j = h, x_p = l, y = c) + m_e/\nu_j}{\nu(x_p = l, y = c) + m_e}
\]

In Friedman’s paper [30], he suggests using the marginal distribution rather than uniform (as shown above), which results in the following formula.

\[
P(x_j = h | x_p = l, y = c) = \frac{\nu(x_j = h, x_p = l, y = c) + m_e * mp_j}{\nu(x_p = l, y = c) + m_e}
\]

where

\[mp_j = \frac{\nu(x_j)}{m}\]

### 7.7.4 The classify Method

As with NaiveBayes, the classify simply multiplies entries in the CPV and CPTs (all except the root are extended). Again the class with the highest product is chosen.

```scala
def classify (z: VectoI): (Int, String, Double) =
{
  val prob = new VectorD (p_y)
  for (i <- 0 until k; j <- 0 until n if fset(j)) {
    prob(i) *= (if (parent(j) > -1) p_XyP(i, j, z(j), z(parent(j)))
                 else p_XyP(i, j, z(j), 0))
  } // for
  val best = prob.argmax ()
  (best, cn(best), prob(best))
} // classify
```

### 7.7.5 Cross-Validation

Again there are two classes: \( \text{TANBayes0} \) that uses conventional “additive” cross-validation and \( \text{TANBayes} \) that uses more efficient “subtractive” cross-validation.

234
TANBayes0 Class

Class Methods:

@param x the integer-valued data vectors stored as rows of a matrix
@param y the class vector, where $y(l)$ = class for row $l$ of the matrix, $x(l)$
@param fn_ the names for all features/variables
@param k the number of classes
@param cn_ the names for all classes
@param me use $m$-estimates ($me = 0$ => regular MLE estimates)
@param vc the value count (number of distinct values) for each feature

class TANBayes0 (x: Matrix, y: Vector, fn_: Strings = null, k: Int = 2, cn_: Strings = null,
               me: Double = me_default, protected var vc: Array[Int] = null)
   extends BayesClassifier (x, y, fn_, k, cn_)

def train (itest: IndexedSeq[Int]): TANBayes0 =
def computeParent (idx: IndexedSeq[Int])
override def getParent: Vector = parent
protected def updateFreq (i: Int)
def maxSpanningTree (ch: Array[Set[Int]], elabel: Map[Pair, Double]): MinSpanningTree =
def computeVcp ()
def classify (z: Vector): (Int, String, Double) =
def reset ()
def printConditionalProb ()

The TANBayes class is similar, but uses a more efficient cross-validation method.

7.7.6 Exercises

1. Use the Integer-based TANBayes to build classifiers for (a) the ExampleTennis problem and (b) the Breast Cancer problem (data in breast-cancer.arff file). Compare its accuracy to that of NullModel and NaiveBayes.

2. Re-engineer TANBayes to use correlation instead of conditional mutual information. Compare the results with the current TANBayes implementation.
7.8 Forest Augmented Naïve Bayes

The `FANBayes` class implements a Forest Augmented Naïve (FAN) Bayes Classifier suitable for discrete input data.
7.9 Network Augmented Naïve Bayes

The `TwoNANBayes` class implements a Network Augmented Naïve (NAN) Bayes Classifier suitable for discrete input data, that is restricted to at most two x-parents. It is a special case of a general Network Augmented Naïve (NAN) Bayes Classifier, also known as a Bayesian Network Classifier.

7.9.1 Bayesian Network Classifier

A Bayesian Network Classifier \[9\] is used to classify a discrete input data vector \(x\) by determining which of \(k\) classes has the highest Joint Probability of \(x\) and the response/outcome \(y\) (i.e., one of the \(k\) classes) of occurring.

\[P(y, x_0, x_1, \ldots, x_{n-1})\]

Using the Chain Rule of Probability, the Joint Probability calculation can factored into multiple calculations of conditional probabilities as well as the class probability of the response. For example, given three variables, the joint probability may be factored as follows:

\[P(x_0, x_1, x_2) = P(x_0)P(x_1|x_0)P(x_2|x_0, x_1)\]

Conditional dependencies are specified using a Directed Acyclic Graph (DAG). A feature/variable represented by a node in the network is conditionally dependent on its parents only,

\[y^* = \arg\max_{y \in D_y} P(y) \prod_{j=0}^{n-1} P(x_j|x_{p(j)}, y)\]

where \(x_{p(j)}\) is the vector of features/variables that \(x_j\) is dependent on, i.e., its parents. In our model, each variable has dependency with the response variable \(y\) (a defacto parent). Note, some more general BN formulations do not distinguish one of the variables to be the response \(y\) as we do.

Conditional probabilities are recorded in tables referred to as Conditional Probability Tables (CPTs). Each variable will have a CPT and the number of columns in the table is governed by the number of other variables it is dependent upon. If this number is large, the CPT may become prohibitively large.

7.9.2 Structure Learning

For `TwoNANBayes` the parents of variable \(x_j\) are recoded in a vector \(x_{p(j)}\) of length 0, 1 or 2. Although the restriction to at most 2 parents might seem limiting, the problem of finding the optimal structure is still NP-hard \[16\].

7.9.3 Conditional Probability Tables

Example Problem:

Class Methods:
@param dag the directed acyclic graph specifying conditional dependencies
@param table the array of tables recording conditional probabilities
@param k the number of classes

class BayesNetwork (dag: DAG, table: Array [Map [Int, Double]], k: Int)
    extends Classifier with Error

    def jp (x: VectoI): Double =
    def cp (i: Int, key: VectoI): Double =
    def train ()
    override def classify (z: VectoI): Int =
    def classify (z: VectoD): Int =

238
7.10 Markov Network

A Markov Network is a probabilistic graphical model where directionality/causality between random variables is not considered, only their bidirectional relationships. In general, let $\mathbf{x}$ be an $n$-dimensional vector of random variables.

$$\mathbf{x} = [x_0, \ldots, x_{n-1}]$$

Given a data instance $\mathbf{x}$, its likelihood of occurrence is given by the joint probability.

$$P(\mathbf{x} = \mathbf{x})$$

In order to compute the joint probability, it needs to be factored based on conditional independencies. These conditional independencies may be illustrated graphically, by creating a vertex for each random variable $x_i$ and letting the structure of the graph reflect the conditional independencies,

$$x_i \perp x_k \mid \{x_j\}$$

such that removal of the vertices in the set $\{x_j\}$ will disconnect $x_i$ and $x_k$ in the graph. These conditional independencies may be exploited to factor the joint probability, e.g.,

$$P(x_i, x_k \mid \{x_j\}) = P(x_i \mid \{x_j\}) P(x_k \mid \{x_j\})$$

When two random variables are directly connected by an undirected edge (denoted $x_i - x_j$) they cannot be separated by removal of other vertices. Together they form an Undirected Graph $G(\mathbf{x}, E)$ where the vertex-set is the set of random variables $\mathbf{x}$ and the edge-set is defined as follows:

$$E = \{x_i - x_j \mid x_i \text{ and } x_j \text{ are not conditionally independent}\}$$

When the random variables are distributed in space, the Markov Network may form a grid, in which case the network is often referred to as a Markov Random Field (MRF).

7.10.1 Markov Blanket

A vertex in the graph $x_i$ will be conditionally independent of all other vertices, except those in its Markov Blanket. The Markov Blanket for random variable $x_i$ is simply the immediate neighbors of $x_i$ in $G$:

$$B(x_i) = \{x_j \mid x_i - x_j \in E\}$$  \hspace{1cm} (7.12)$$

The edges $E$ are selected so that random variable $x_i$ will be conditionally independent of any other ($k \neq i$) random variable $x_k$ that is not in its Markov blanket.

$$x_i \perp x_k \mid B(x_i)$$  \hspace{1cm} (7.13)$$
7.10.2 Factoring the Joint Probability

Factorization of the joint probability is based on the graphical structure of $G$ that reflects the conditional independencies. It has been shown (see the Hammersley-Clifford Theorem) that $P(x)$ may be factored according the set of maximal cliques $\text{Cl}$ in graph $G$.

$$P(x) = \frac{1}{Z} \prod_{c \in \text{Cl}} \phi_c(x_c)$$  \hspace{1cm} (7.14)

For each clique $c$ in the set $\text{Cl}$, a potential function $\phi_c(x_c)$ is defined. (Potential functions are non-negative functions that are used in place of marginal/conditional probabilities and need not sum to one; hence the normalizing constant $Z$).

Suppose a graph $G([x_0, x_1, x_2, x_3, x_4], E)$ has two maximal cliques, $\text{Cl} = \{[x_0, x_1, x_2], [x_2, x_3, x_4]\}$ then

$$P(x) = \frac{1}{Z} \phi_0(x_0, x_1, x_2) \phi_1(x_2, x_3, x_4)$$

7.10.3 Exercises

1. Consider the random vector $x = [x_0, x_1, x_2]$ with conditional independency

$$x_0 \perp x_1 \mid x_2$$

show that

$$P(x_0, x_1, x_2) = P(x_2) P(x_0 \mid x_2) P(x_1 \mid x_2)$$

\footnote{A clique is a set of vertices that are fully connected}
7.11 Decision Tree ID3

A Decision Tree (or Classification Tree) classifier \[88, 84\] will take an input vector \(x\) and classify it, i.e., give one of \(k\) class values to \(y\) by applying a set of decision rules configured into a tree. Abstractly, the decision rules may be viewed as a function \(f\).

\[
y = f(x) = f(x_0, x_1, \ldots, x_{n-1}) \quad (7.15)
\]

The DecisionTreeID3 \[77\] class implements a Decision Tree classifier using the Iterative Dichotomiser 3 (ID3) algorithm. The classifier is trained using an \(m\)-by-\(n\) data matrix \(X\) and a classification vector \(y\). Each data vector in the matrix is classified into one of \(k\) classes numbered \(0, 1, \ldots, k - 1\). Each column in the matrix represents a \(x\)-variable/feature (e.g., Humidity). The value count \(vc\) vector gives the number of distinct values per feature (e.g., 2 for Humidity).

7.11.1 Entropy

In decision trees, the goal is to reduce the disorder in decision making. Assume the decision is of the yes(1)/no(0) variety and consider the following decision/classification vectors: \(y = (1, 1, \ldots, 1, 1)\) or \(y' = (1, 0, \ldots, 1, 0)\). In the first case all the decisions are yes, while in the second, three are an equal number of yes and no decisions. One way to measure the level of disorder is Shannon entropy. To compute the entropy, first convert the \(m\)-dimensional decision/classification vector \(y\) into a \(k\)-dimensional probability vector \(p\). The \texttt{frequency} and \texttt{toProbability} functions in the \texttt{Probability} object may be used for this task (see \texttt{NullModel} from the last chapter).

For the two cases, \(p = (1, 0)\) and \(p' = (.5, .5)\), so computing the Shannon entropy \(H(p)\) (see Chapter 3), we obtain \(H(p) = 0\) and \(H(p') = 1\). These indicate that there is no disorder in the first case and maximum disorder in the second case.

Entropy is used as measure of the impurity of a node (e.g., to what degree is it a mixture of ‘-’ and ‘+’). For a discussion of additional measures see \[84\].

7.11.2 Example Problem

Let us consider the Tennis example from \texttt{NullModel} and \texttt{NaiveBayes} and compute the entropy level for the decision of whether to play tennis. There are 14 days worth of training data see Table 7.12 which indicate that for 9 of the days the decision was yes (play tennis) and for 5 it was no (do not play). Therefore, the entropy (if no features/variables are considered) is

\[
H(p) = H\left(\frac{5}{14}, \frac{9}{14}\right) = -\frac{5}{14} \log_2\left(\frac{5}{14}\right) - \frac{9}{14} \log_2\left(\frac{9}{14}\right) = 0.9403
\]

Recall that the features are Outlook \(x_0\), Temp \(x_1\), Humidity \(x_2\), and Wind \(x_3\). To reduce entropy, find the feature/variable that has the greatest impact on reducing disorder. If feature/variable \(j\) is factored into the decision making, entropy is now calculated as follows:

\[
\sum_{v=0}^{vc_j-1} \frac{\nu(x_j = v)}{m} H(p_{x_j=v})
\]

where \(\nu(x_j = v)\) is the frequency count of value \(v\) for column vector \(x_j\) in matrix \(X\). The sum is the weighted average of the entropy over all possible \(vc_j\) values for variable \(j\).
Table 7.12: Tennis Example

<table>
<thead>
<tr>
<th>Day</th>
<th>x−0</th>
<th>x−1</th>
<th>x−2</th>
<th>x−3</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>14</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

To see how this works, let us compute new entropy values assuming each feature/variable is used, in turn, as the principal feature for decision making. Starting with feature $j = 0$ (Outlook) with values of Rain ($0$), Overcast ($1$) and Sunny ($2$), compute the probability vector and entropy for each value and weight them by how often that value occurs.

$$\sum_{v=0}^{2} \frac{\nu(x_{-0} = v)}{m} H(p_{x_{-0}=v})$$

For $v = 0$, we have 2 no (0) cases and 3 yes (1) cases ($2−, 3+$), for $v = 1$, we have ($0−, 4+$) and for $v = 2$, we have ($3−, 2+$).

$$\frac{\nu(x_{-0} = 0)}{14} H(p_{x_{-0}=0}) + \frac{\nu(x_{-0} = 1)}{14} H(p_{x_{-0}=1}) + \frac{\nu(x_{-0} = 2)}{14} H(p_{x_{-0}=2})$$

$$\frac{5}{14} H(p_{x_{-0}=0}) + \frac{4}{14} H(p_{x_{-0}=1}) + \frac{5}{14} H(p_{x_{-0}=2})$$

We are left with computing three entropy values:

$$H(p_{x_{-0}=0}) = H\left(\frac{2}{5}, \frac{3}{5}\right) = -\frac{2}{5} \log_2\left(\frac{2}{5}\right) - \frac{3}{5} \log_2\left(\frac{3}{5}\right) = 0.9710$$

$$H(p_{x_{-0}=1}) = H\left(\frac{0}{4}, \frac{4}{4}\right) = -\frac{0}{4} \log_2\left(\frac{0}{4}\right) - \frac{4}{4} \log_2\left(\frac{4}{4}\right) = 0.0000$$

$$H(p_{x_{-0}=2}) = H\left(\frac{3}{5}, \frac{2}{5}\right) = -\frac{3}{5} \log_2\left(\frac{3}{5}\right) - \frac{2}{5} \log_2\left(\frac{2}{5}\right) = 0.9710$$

The weighted average is then 0.6936, so that the drop in entropy (also called information gain) is 0.9403 - 0.6936 = 0.2467. As shown in Table 7.13, the other entropy drops are 0.0292 for Temperature (1), 0.1518 for Humidity (2) and 0.0481 for Wind (3).
Table 7.13: Choices for Principal Feature

<table>
<thead>
<tr>
<th>$j$</th>
<th>Variable/Feature</th>
<th>Entropy</th>
<th>Entropy Drop</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Outlook</td>
<td>0.6936</td>
<td>0.2467</td>
</tr>
<tr>
<td>1</td>
<td>Temperature</td>
<td>0.9111</td>
<td>0.0292</td>
</tr>
<tr>
<td>2</td>
<td>Humidity</td>
<td>0.7885</td>
<td>0.1518</td>
</tr>
<tr>
<td>3</td>
<td>Wind</td>
<td>0.8922</td>
<td>0.0481</td>
</tr>
</tbody>
</table>

Hence, Outlook ($j = 0$) should be chosen as the principal feature for decision making. As the entropy is too high, make a tree with Outlook (0) as the root and make a branch for each value of Outlook: Rain (0), Overcast (1), Sunny (2). Each branch defines a sub-problem.

The resulting tree is shown in Figure ?? where the node is designated by the variable (in this case $x_0$). The edges indicate the values that this variable can take on, while the two numbers $n^-p^+$ indicate the number of negative and positive cases.

![Decision Tree](image-url)

**Figure 7.1: Decision Tree for “Play Tennis?” Example**

**Sub-problem $x_0 = 0$**

The sub-problem for Outlook: Rain (0) see Table [7.14] is defined as follows: Take all five cases/rows in the data matrix $X$ for which $x_{-0} = 0$.

If we select Wind ($j = 3$) as the next variable, we obtain the following cases: For $v = 0$, we have $(0-, 3+)$, so the probability vector and entropy are

$$\mathbf{p}_{x_{-3}=0} = \left( \frac{0}{5}, \frac{3}{5} \right) \quad H(\mathbf{p}_{x_{-3}=0}) = 0$$

For $v = 1$, we have $(2-, 0+)$, so the probability vector and entropy are

$$\mathbf{p}_{x_{-3}=1} = \left( \frac{2}{5}, \frac{0}{5} \right) \quad H(\mathbf{p}_{x_{-3}=1}) = 0$$

If we stop expanding the tree at this point, we have the following rules.
Table 7.14: Sub-problem for node \( x_0 \) and branch 0

<table>
<thead>
<tr>
<th>Day</th>
<th>( x_{-1} )</th>
<th>( x_{-2} )</th>
<th>( x_{-3} )</th>
<th>( y )</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>14</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

if \( x_0 = 0 \) then
  if \( x_3 = 0 \) then yes
  if \( x_3 = 1 \) then no
if \( x_0 = 1 \) then yes
if \( x_0 = 2 \) then no

The overall entropy can be calculated as the weighted average of all the leaf nodes.

\[
\frac{3}{14} \cdot 0 + \frac{2}{14} \cdot 0 + \frac{4}{14} \cdot 0 + \frac{5}{14} \cdot .9710 = .3468
\]

**Sub-problem \( x_0 = 2 \)**

Note that if \( x_0 = 1 \), the entropy for this case is already zero, so this node need not be split and remains as a leaf node. There is still some uncertainty left when \( x_0 = 2 \), so this node may be split. The sub-problem for Outlook: Rain (2) see Table 8.1 is defined as follows: Take all five cases/rows in the data matrix \( X \) for which \( x_{-3} = 2 \).

Table 7.15: Sub-problem for node \( x_0 \) and branch 2

<table>
<thead>
<tr>
<th>Day</th>
<th>( x_{-1} )</th>
<th>( x_{-2} )</th>
<th>( x_{-3} )</th>
<th>( y )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

It should be obvious that \( y = 1 - x_{-2} \). For \( v = 0 \), we have \((0-, 2+)\), so the probability vector and entropy are

\[
p_{x_{-2}=0} = \left(\frac{0}{5}, \frac{2}{5}\right) \quad H(p_{x_{-2}=0}) = 0
\]

For \( v = 1 \), we have \((3-, 0+)\), so the probability vector and entropy are

\[
p_{x_{-2}=1} = \left(\frac{2}{5}, \frac{3}{5}\right) \quad H(p_{x_{-2}=0}) = 0
\]
At this point, the overall entropy is zero and the decision tree is the following (shown as a pre-order traversal from ScalaTion):

Decision Tree:
[ Node[0] b-1 : f = x0 ( 5-, 9+ ) ]
[ Node[1] b0 : f = x3 ( 2-, 3+ ) ]
  [ Leaf[2] b0 : y = 1 ( 0-, 3+ ) ]
  [ Leaf[3] b1 : y = 0 ( 2-, 0+ ) ]
[ Leaf[4] b1 : y = 1 ( 0-, 4+ ) ]
[ Node[5] b2 : f = x2 ( 3-, 2+ ) ]
  [ Leaf[6] b0 : y = 1 ( 0-, 2+ ) ]
  [ Leaf[7] b1 : y = 0 ( 3-, 0+ ) ]

The above process of creating the decision tree is done by a recursive, greedy algorithm. As with many greedy algorithms, it does not guarantee an optimal solution.

7.11.3 Early Termination
Producing a complex decision tree with zero entropy may suggest overfitting, so that a simpler tree may be more robust. One approach would be to terminate once entropy decreases to a certain level. One problem with this is that expanding a different branch could have led to a lower entropy with a tree of no greater complexity. Another approach is simply to limit the depth of the tree. Simple decision trees with limited depth are commonly used in Random Forests, a more advanced technique discussed later.

7.11.4 Pruning
An alternative to early termination is to build a complex tree and then prune the tree. Pruning involves selecting a node whose children are all leaves and undoing the split that created the children. Compared to early termination, pruning will take more time to come up with the solution. For the tennis example, pruning could be used to turn node 5 into a leaf node (pruning away two nodes) where the decision would be the majority decision $y = 1$. The entropy for this has already been calculated to be .3468. Instead node 1 could be turned into a leaf (pruning away two nodes). This case is symmetric to the other one, so the entropy would be .3468, but the decision would be $y = 0$. The original ID3 algorithm did not use pruning, but its follow on algorithm C4.5 does (see Chapter 7). The ScalaTion implementation of ID3 does support pruning.

DecisionTreeID3 Class

Class Methods:

@param x the data vectors stored as rows of a matrix
@param y the class array, where y_i = class for row i of the matrix x
@param fn_ the names for all features/variables
@param k the number of classes
@param cn_ the names for all classes
@param vc the value count array indicating number of distinct values per feature
@param td the maximum tree depth to allow

class DecisionTreeID3 (x: Matrix, y: Vector, fn_: Strings = null, k: Int = 2, cn_: Strings = null,
private var vc: Array[Int] = null, td: Int = -1)
extends ClassifierInt (x, y, fn_, k, cn_)

def frequency (dset: Array[(Int, Int)], value: Int): (Double, Vector) =
def gain (f: Int, path: List[(Int, Int)]: (Double, Vector) =
def train (itest: IndexedSeq[Int]): DecisionTreeID3 =
def calcEntropy (listOfLeaves: ArrayBuffer[LeafNode]): Double =
def buildTree (path: List[(Int, Int)], depth: Int): FeatureNode =
def prune (threshold: Double, fold: Int = 5): DecisionTreeID3 =
def compareModel (folds: Int, threshold: Double) =
def printTree ()
def classify (z: Vector): (Int, String, Double) =
def reset ()

7.11.5 Exercises

1. The Tennis example (see NaiveBayes) can also be analyzed using decision trees.

   val id3 = new DecisionTreeID3 (x, y, fn, k, cn, vc) // create the classifier
   id3.train ()
   val z = Vector(2, 2, 1, 1) // new vector to classify
   println (s"classify ($z) = ${id3.classify (z)}")

   Use DecisionTreeID3 to build classifiers for the ExampleTennis problem. Compare its accuracy to
   that of NullModel, NaiveBayes and TANBayes.

2. Do the same for the Breast Cancer problem (data in breast-cancer.arff file).

3. For the Breast Cancer problem, evaluate the effectiveness of the prune method.

4. Again for the Breast Cancer problem, explore the results for various limitations to the maximum tree
   depth via the td parameter.

7.12 Hidden Markov Model

A Hidden Markov Model (HMM) provides a natural way to study a system with an internal state and
external observations. One could image looking at a flame and judging the temperature (internal state) of
the flame by its color (external observation). When this is treated as a discrete problem, an HMM may
be used; whereas, as a continuous problem, a Kalman Filter may be used (see the chapter on State Space
Models). For HMMs, we assume that the internal state is unknown (hidden), but may be predicted by from the observations.

Consider two discrete-valued, discrete-time stochastic processes. The first process represents the internal state of a system

\[ \{ x_t : t \in \{0, \ldots T-1\} \} \]

while the second process represents corresponding observations of the system

\[ \{ y_t : t \in \{0, \ldots T-1\} \} \]

The internal state influences the observations. In a deterministic setting, one might imagine

\[ y_t = f(x_t) \]

Unfortunately, since both \( x_t \) and \( y_t \) are both stochastic processes, their trajectories need to be described probabilistically. For tractability and because it often suffices, the assumption is made that the state \( x_t \) is only significantly influenced by its previous state \( x_{t-1} \).

\[ P(x_t|x_{t-1}, x_{t-2}, x_0) = P(x_t|x_{t-1}) \]

In other words, the transitions from state to state are governed by a discrete-time Markov chain and characterized by state-transition probability matrix \( A = [a_{ij}] \), where

\[ a_{ij} = P(x_t = j|x_{t-1} = i) \]

The influence of the state upon the observation is also characterized by emission probability matrix \( B = [b_{jk}] \), where

\[ b_{jk} = P(y_t = k|x_t = j) \]

is the conditional probability of the observation being \( k \) when the state is \( j \). This represents a second simplifying assumption that the observation is effectively independent of prior states or observations. To predict the evolution of the system, it is necessary to characterize the initial state of the system \( x_0 \).

\[ \pi_j = P(x_t = j) \]

The dynamics of an HMM model is thus represented by two matrices \( A \), \( B \) and an initial state probability vector \( \pi \).

### 7.12.1 Example Problem

Let the system under study be a lane of road with a sensor to count traffic flow (number of vehicles passing the sensor in a five minute period). As a simple example, let the state of the road be whether or not there is an accident ahead. In other words, the state of road is either 0 (No-accident) or 1 (Accident). The only information available is the traffic counts and of course historical information for training an HMM model. Suppose the chance of an accident ahead is 10%.
As B

This may be done by computing the joint probability \( P \) states gives the highest probability of occurrence to the observed sequences. The number of observed values \( m = 4 \). Again from historical data the emission probability matrix \( B \) is estimated to be

\[
\pi = [0.9, 0.1]
\]

From historical information, two transition probabilities are estimated: the first is for the transition from no accident to accident which is 20%; the second from accident to no-accident state (i.e., the accident has been cleared) which is 50% (i.e., probability of one half that the accident will be cleared by the next time increment). The number of states \( n = 2 \). Therefore, the state-transition probability matrix \( A \) is

\[
\begin{bmatrix}
  0.8 & 0.2 \\
  0.5 & 0.5
\end{bmatrix}
\]

As \( A \) maps states to state, \( A \) is an \( n \)-by-\( n \) matrix.

Clearly, the state will influence the traffic flow (tens of cars per 5 minutes) with possible values of 0, 1, 2, 3. The number of observed values \( m = 4 \). Therefore, the state-transition probability matrix \( A \) is

\[
\begin{bmatrix}
  0.1 & 0.2 & 0.3 & 0.4 \\
  0.5 & 0.2 & 0.2 & 0.1
\end{bmatrix}
\]

As \( B \) maps states to observed values, \( B \) is an \( n \)-by-\( m \) matrix.

One question to address is, given a time series (observations sequence), what corresponding sequence of states gives the highest probability of occurrence to the observed sequences.

\[
y = [3, 3, 0]
\]

This may be done by computing the joint probability \( P(x, y) \)

\[
P(NNA, y) = \pi_0 \cdot b_{03} \cdot a_{00} \cdot b_{03} \cdot a_{00} \cdot b_{00} = 0.9 \cdot 0.4 \cdot 0.8 \cdot 0.4 \cdot 0.8 \cdot 0.1 = 0.009216
\]

\[
P(NAN, y) = \pi_0 \cdot b_{03} \cdot a_{00} \cdot b_{03} \cdot a_{01} \cdot b_{10} = 0.9 \cdot 0.4 \cdot 0.8 \cdot 0.4 \cdot 0.2 \cdot 0.5 = 0.011520
\]

\[
P(NAA, y) = \pi_0 \cdot b_{03} \cdot a_{01} \cdot b_{13} \cdot a_{10} \cdot b_{00} = 0.9 \cdot 0.4 \cdot 0.2 \cdot 0.1 \cdot 0.5 \cdot 0.1 = 0.000360
\]

\[
P(NNN, y) = \pi_0 \cdot b_{03} \cdot a_{00} \cdot b_{03} \cdot a_{00} \cdot b_{00} = 0.9 \cdot 0.4 \cdot 0.8 \cdot 0.4 \cdot 0.8 \cdot 0.1 = 0.009216
\]

\[
P(ANN, y) = \pi_1 \cdot b_{03} \cdot a_{10} \cdot b_{03} \cdot a_{00} \cdot b_{00} = 0.1 \cdot 0.1 \cdot 0.5 \cdot 0.4 \cdot 0.8 \cdot 0.1 = 0.000160
\]

\[
P(ANA, y) = \pi_1 \cdot b_{03} \cdot a_{10} \cdot b_{03} \cdot a_{01} \cdot b_{10} = 0.1 \cdot 0.1 \cdot 0.5 \cdot 0.4 \cdot 0.2 \cdot 0.5 = 0.000200
\]

\[
P(AAN, y) = \pi_1 \cdot b_{03} \cdot a_{11} \cdot b_{13} \cdot a_{10} \cdot b_{00} = 0.1 \cdot 0.1 \cdot 0.5 \cdot 0.1 \cdot 0.5 \cdot 0.1 = 0.000025
\]

\[
P(AAA, y) = \pi_1 \cdot b_{03} \cdot a_{11} \cdot b_{13} \cdot a_{11} \cdot b_{10} = 0.1 \cdot 0.1 \cdot 0.5 \cdot 0.1 \cdot 0.5 \cdot 0.5 = 0.000125
\]

The state giving the highest probability is \( x = NNA \). The marginal probability of the observed sequence \( P(y) \) can be computed by summing over all eight states.

\[
P(y) = \sum_x P(x, y) = 0.023406
\]

The algorithms given in the subsections below are adapted from [39]. For these algorithms, we divide the time series/sequence of observations into two parts (past and future).
\[ y^t = [y_0, y_1, y_t] \]

They allow one to calculate (1) the probability of arriving in a state at time \( t \) with observations \( y^t \), (2) the conditional probability of seeing future observations \( y^{t+} \) from a given state at time \( t \), and (3) the conditional probability of being in a state at time \( t \) given all the observations \( y \).

### 7.12.2 Forward Algorithm

For longer observation sequences/time series, the approach of summing over all possible state vectors (of which there are \( n^T \)) will become intractable. Much of the computation is repetitive anyway. New matrices \( A, B \) and \( \Gamma \) are defined to save such intermediate calculations.

The forward algorithm (\( \alpha \)-pass) computes the \( A \) matrix. The probability of being in state \( j \) at time \( t \) having observations up to time \( t \) is given by

\[ \alpha_{tj} = P(x_t = j, y = y^t) \]

Computation of \( \alpha_{tj} \) may be done efficiently using the following recurrence.

\[ \alpha_{tj} = b_{j, y_t} \sum_{i=0}^{n-1} \alpha_{t-1,i} a_{ij} = b_{j, y_t} [\alpha_{t-1} \cdot a_{ij}] \]

To get to state \( j \) at time \( t \), the system must transition from some state \( i \) at time \( t-1 \) and at time \( t \) emit the value \( y_t \). These values may be saved in a \( T \)-by-\( n \) matrix \( A = [\alpha_{tj}] \) and efficiently computed by moving forward in time.

```scala
// def forwardEval0 (): MatrixD =
{
    for (j <- rstate) alp(0, j) = pi(j) * b(j, y(0))
    for (t <- 1 until tt) {
        for (j <- rstate) {
            alp(t, j) = b(j, y(t)) * (alp(t-1) dot a.col(j))
        } // for
    } // for
    alp
} // forwardEval0
```

The marginal probability is now simply the sum of the elements in the last row of the \( \alpha \) matrix

\[ P(y) = \sum_{j=0}^{n} \alpha_{T-1,j} \]

SCALATION also provides a \texttt{forwardEval} method that uses scaling to avoid underflow.
7.12.3 Backward Algorithm

The backward algorithm (β-pass) computes the B matrix. The conditional probability of having future observations after time $t$ ($y = y^{t+}$) given the current state $x_t = i$ is

$$\beta_{ti} = P(y = y^{t+} | x_t = i)$$

Computation of $\beta_{ij}$ may be done efficiently using the following recurrence.

$$\beta_{ti} = \sum_{j=0}^{n-1} a_{ij} b_{j,y_{t+1}} \beta_{t+1,j}$$

From state $i$ at time $t$, the system must transition to some state $j$ at time $t+1$ and at time $t+1$ emit the value $y_{t+1}$. These values may be saved in a $T$-by-$n$ matrix $B = [\beta_{ti}]$ and efficiently computed by moving backward in time.

```scala
def backwardEval0 (): MatrixD = 
{ 
  for (i <- rstate) bet(tt-1, i) = 1.0 
  for (t <- tt-2 to 0 by -1) {
    for (i <- rstate) {
      bet(t, i) = 0.0
      for (j <- rstate) bet(t, i) += a(i, j) * b(j, y(t+1)) * bet(t+1, j)
    } // for
  } // for
  bet
} // backwardEval0
```

SCALATION also provides a `backwardEval` method that uses scaling to avoid underflow.

7.12.4 Viterbi Algorithm

The Viterbi algorithm (γ-pass) computes the Γ matrix. The conditional probability of the state at time $t$ being $i$, given all observations ($y = y$) is

$$\gamma_{ti} = P(x_t = i | y = y)$$

As $\alpha_{ti}$ captures the probability up to time $t$ and $\beta_{ti}$ captures the probability after time $t$, the conditional probability may be calculated as follows:

$$\gamma_{ti} = \frac{\alpha_{ti} \beta_{ti}}{P(y)}$$

In ScalaTion, the $\Gamma = [\gamma_{ti}]$ matrix is calculated using the Hadamard product.

```scala
def gamma (alp: MatrixD, bet: MatrixD): MatrixD = (alp ** bet) / probY (alp)
```

The conditional probability of being in state $i$ at time $t$ and transitioning to state $j$ at time $t+1$ given all observations ($y = y$) is
\[ \gamma_{tij} = P(x_t = i, x_{t+1} = j | y = y) \]

Note, this equation is not defined for the last time point \( T - 1 \), since there is no next state. Getting to state \( i \) at time \( t \) is characterized by \( \alpha_{ti} \), the probability of the state transitioning to from \( i \) to \( j \) is characterized \( a_{ij} \), the probability of emitting \( y_{t+1} \) from state \( j \) at time \( t + 1 \) is \( b_{j,y_{t+1}} \), and finally going from state \( j \) to the end is characterized by \( \beta_{t+1,j} \).

\[ \gamma_{tij} = \frac{\alpha_{ti} a_{ij} b_{j,y_{t+1}} \beta_{t+1,j}}{P(y)} \]

The Viterbi Algorithm \texttt{viterbiDecode} computes the \( \Gamma \) matrix (\texttt{gam} in code) from scaled versions of \texttt{alp} and \texttt{bet}. It also computes the \( \Gamma = [\gamma_{tij}] \) tensor (\texttt{gat} in code).

### 7.12.5 Training

The \texttt{train} method will call \texttt{forwardEval}, \texttt{backwardEval} and \texttt{viterbiDecode} to calculate updated values for the \( A \), \( B \) and \( \Gamma \) matrices as well as for the \( \Gamma \) tensor. These values are used to \texttt{re-estimate} the \( \pi \), \( A \) and \( B \) parameters.

```scala
def train (itest: Ints): HiddenMarkov =
{
  var oldLogPr = 0.0
  for (it <- 1 to MIT) {
    val logPr = logProbY (true)
    if (logPr > oldLogPr) {
      oldLogPr = logPr
      forwardEval ()
      backwardEval ()
      viterbiDecode ()
      reestimate ()
    } else {
      println (s"train: HMM model converged after $it iterations")
      return this
    } // if
  } // for
  println (s"train: HMM model did not converged after $MIT iterations")
  this
} // train
```

The training loop will terminate early when there is no improvement to \( P(y) \). To avoid underflow \(-\log(P(y))\) is used.

### 7.12.6 Reestimation of Parameters

The parameters for an HMM model are \( \pi, A \) and \( B \) and may be adjusted to maximize the probability of seeing the observation vector \( P(y) \). Since \( \alpha_{0i} = \pi_i b_{i,y_{0}} \), we can re-estimate \( \pi \) as follows:

\[ \pi_i = \frac{\alpha_{0i}}{b_{i,y_{0}}} = \gamma_{0i} \]
The $A$ matrix can re-estimated as follows:

$$a_{ij} = \frac{\sum_{t=0}^{T-2} \gamma_{tij}}{\sum_{t=0}^{T-2} \gamma_{tti}}$$

Similarly, the $B$ matrix can re-estimated as follows:

$$b_{ik} = \frac{\sum_{t=0}^{T-1} I_{y_t=k} \gamma_{tti}}{\sum_{t=0}^{T-1} \gamma_{tti}}$$

The detailed derivations are left to the exercises.

**HiddenMarkov Class**

**Class Methods:**

- @param y the observation vector/observed discrete-valued time series
- @param m the number of observation symbols/values \{0, 1, ... m-1\}
- @param n the number of (hidden) states in the model
- @param cn_ the class names for the states, e.g., ("Hot", "Cold")
- @param pi the probability vector for the initial state
- @param a the state transition probability matrix (n-by-n)
- @param b the observation probability matrix (n-by-m)

```scala
class HiddenMarkov (y: VectoI, m: Int, n: Int, cn_ : Strings = null,
private var pi: VectoD = null,
private var a: MatriD = null,
private var b: MatriD = null)
extends ClassifierInt (null, y, null, n, cn_)
```

override def size: Int = n

def parameter: VectoD = pi
def parameters: (MatriD, MatriD) = (a, b)
def jointProb (x: VectoI): Double =
def forwardEval0 () : MatriD =
def probY (scaled: Boolean = false): Double =
def logProbY (scaled: Boolean = false): Double =
def backwardEval0 () : MatriD =
def gamma () : MatriD = (alp ** bet) / probY ()
def forwardEval () : MatriD =
def getC: VectoD = c
def backwardEval () : MatriD =
def viterbiDecode () : MatriD =
def reestimate ()
def train (itest: Ints): HiddenMarkov =
override def report: String =
def classify (z: VectoI): (Int, String, Double) =
def reset () {} 
```
7.12.7 Exercises

1. Show that for \( t \in \{0, \ldots, T - 2\} \),

\[
\gamma_{ti} = \sum_{j=0}^{n-1} \gamma_{tij}
\]

2. Show that

\[
\pi_i = \frac{\alpha_0i}{b_{i,y_0}} = \gamma_0i
\]

3. Show that

\[
a_{ij} = \frac{\sum_{t=0}^{T-2} \gamma_{tij}}{\sum_{t=0}^{T-2} \gamma_{ti}}
\]

4. Show that

\[
b_{ik} = \frac{\sum_{t=0}^{T-1} I_{y_t = k} \gamma_{ti}}{\sum_{t=0}^{T-1} \gamma_{ti}}
\]

Further Reading

1. Data Classification Algorithms and Applications [2]
Chapter 8

Classification: Continuous Variables

For the problems in this chapter, the response/classification variable is still discrete, but some/all of the feature variables are now continuous. Technically, classification problems fit in this category, if it is infeasible or nonproductive to compute frequency counts for all values of a variable (e.g., for $x_j$, the value count $vc_j = \infty$). If a classification problem almost fits in the previous chapter, one may consider the use of binning to convert numerical variables into categorical variables (e.g., convert weight into weight classes). Care should be taken since binning represents hidden parameters in the model and arbitrary choices may influence results.
8.1 ClassifierReal

The ClassifierReal abstract class provides a common foundation for several classifiers that operate on continuous (or real-valued) data.

Class Methods:

@param x the real-valued training data vectors stored as rows of a matrix
@param y the training classification vector, where \( y_i \) = class for row \( i \) of the matrix \( x \)
@param fn the names of the features/variables
@param k the number of classes
@param cn the names for all classes
@param hparam the hyper-parameters

abstract class ClassifierReal (x: MatrixDouble, y: VectorInt, protected var fn: Strings,
                           k: Int, protected var cn: Strings, hparam: HyperParameter)
   extends ConfusionFit (y, k) with Classifier

    def vc_default: Array[Int] = Array.fill(n)(2)
def size: Int = m
def test (itest: Ints): Double =
def test (xx: MatrixDouble, yy: VectorDouble): Double =
def eval (xx: MatrixDouble, yy: VectorDouble = null): ClassifierReal =
def crossValidate (nx: Int = 10, show: Boolean = false): Array[Statistic] =
def crossValidateRand (nx: Int = 10, show: Boolean = false): Array[Statistic] =
def hparameter: HyperParameter = hparam
def report: String =
def classify (z: VectorDouble): (Int, String, Double) = classify (z.toDouble)
def classify (xx: MatrixDouble = x): VectorInt =
def calcCorrelation: MatrixDouble =
def calcCorrelation2 (zrg: Range, xrg: Range): MatrixDouble =
def featureSelection (TOL: Double = 0.01) =
8.2 Gaussian Naive Bayes

The `NaiveBayes` class implements a Gaussian Naïve Bayes Classifier, which is the most commonly used such classifier for continuous input data. The classifier is trained using a data matrix $X$ and a classification vector $y$. Each data vector in the matrix is classified into one of $k$ classes numbered $0, 1, \ldots, k - 1$.

Class probabilities are calculated based on the population of each class in the training-set. Relative probabilities are computed by multiplying these by values computed using conditional density functions based on the Normal (Gaussian) distribution. The classifier is naïve, because it assumes feature independence and therefore simply multiplies the conditional densities.

Starting with main results from the section on Naïve Bayes (equation 4.5),

$$y^* = \arg\max_{y \in \{0, \ldots, k-1\}} P(y) \prod_{j=0}^{n-1} P(x_j | y)$$

if all the variables $x_j$ are continuous, we may switch from conditional probabilities $P(x_j | y)$ to conditional densities $f(x_j | y)$. The best prediction for class $y$ is the value $y^*$ that maximizes the product of the conditional densities multiplied by the class probability.

$$y^* = \arg\max_{y \in \{0, \ldots, k-1\}} P(y) \prod_{j=0}^{n-1} f(x_j | y) \quad (8.1)$$

Although the formula assumes the conditional independence of $x_j$'s, the technique can be applied as long as correlations are not too high.

Using the Gaussian assumption, the conditional density of $x_j$ given $y$, is approximated by estimating the two parameters of the Normal distribution,

$$x_j | y \sim \text{Normal}(\mu_c, \sigma_c^2)$$

where class $c \in \{0, 1, \ldots, k - 1\}$, $\mu_c = \mathbb{E}[x | y = c]$ and $\sigma_c^2 = \mathbb{V}[x | y = c])$. Thus, the conditional density function is

$$f(x_j | y = c) = \frac{1}{\sqrt{2\pi} \sigma_c} e^{-\frac{(x - \mu_c)^2}{2\sigma_c^2}}$$

Class probabilities $P(y = c)$ may be estimated as $m_c / m$, where $m_c$ is the frequency count of the number of occurrences of $c$ in the class vector $y$. Conditional densities are needed for each of the $k$ class values, for each of the $n$ variables (each $x_j$) (i.e., $kn$ are needed). Corresponding means and variances may be estimated as follows:

$$\hat{\mu}_{cj} = \frac{1}{m_c} \sum_{i=0}^{m-1} (x_{ij} | y_i = c)$$

$$\hat{\sigma}_{cj}^2 = \frac{1}{m_c - 1} \sum_{i=0}^{m-1} ((x_{ij} - \hat{\mu}_{cj})^2 | y_i = c)$$

Using conditional density (cd) functions estimated in the `train` function (see code for details), an input vector $z$ can be classified using the `classify` function.
def classify (z: VectoD): Int =
{
    for (c <- 0 until k; j <- 0 until n) prob(c) *= cd(c)(j)(z(j))
    prob.argmax () // class c with highest probability
} // classify

Class Methods:

@param x the real-valued data vectors stored as rows of a matrix
@param y the class vector, where y_i = class for row i of the matrix x, x(i)
@param fn_ the names for all features/variables
@param k the number of classes
@param cn_ the names for all classes

class NaiveBayesR (x: MatriD, y: VectoI, fn_: Strings = null, k: Int = 2,
    cn_: Strings = null)
    extends ClassifierReal (x, y, fn_, k, cn_)

def calcStats ()
def calcHistogram (x_j: VectoD, intervals: Int): VectoD =
def train (itest: IndexedSeq [Int]): NaiveBayesR =
override def classify (z: VectoD): (Int, String, Double) =
def reset ()

8.2.1 Exercises

1. Use NaiveBayesR to classify manufactured parts according whether they should pass quality control based on curvature and diameter tolerances. See people.revoledu.com/kardi/tutorial/LDA/Numerical%20Example.html for details.

    // features/variable:
    // x1: curvature
    // x2: diameter
    // y: classification: pass (0), fail (1)
    //
    val xy = new MatrixD ((7, 3), 2.95, 6.63, 0, // joint data matrix
        2.53, 7.79, 0,
        3.57, 5.65, 0,
        3.16, 5.47, 0,
        2.58, 4.46, 1,
        2.16, 6.22, 1,
        3.27, 3.52, 1)
val fn = Array ("curvature", "diameter")  // feature names
val cn = Array ("pass", "fail")  // class names
val cl = NaiveBayesR (xy, fn, 2, cn)  // create NaiveBayesR classifier
8.3 Simple Logistic Regression

The SimpleLogisticRegression class supports simple logistic regression. In this case, the predictor vector \( \mathbf{x} \) is two-dimensional \([1, x_1]\). Again, the goal is to fit the parameter vector \( \mathbf{b} \) in the regression equation

\[
y = \mathbf{b} \cdot \mathbf{x} + \epsilon = b_0 + b_1 x_1 + \epsilon
\]

where \( \epsilon \) represents the residuals (the part not explained by the model). This looks like simple linear regression, with the difference being that the response variable \( y \) is binary \((y \in \{0, 1\})\). Since \( y \) is binary, minimizing the distance, as was done before, may not work well. First, instead of focusing on \( y \in \{0, 1\} \), we focus on the conditional probability of success \( p_y(\mathbf{x}) \in [0, 1] \), i.e.,

\[
p_y(\mathbf{x}) = P(y = 1|\mathbf{x})
\]

For example, the random variable \( y \) could be used to indicate whether a customer will pay back a loan (1) or not (0). The predictor variable \( x_1 \) could be the customer’s FICA score.

8.3.1 mtcars Example

Another example is from the Motor Trends Cars (mtcars) dataset (see https://stat.ethz.ch/R-manual/R-devel/library/datasets/html/mtcars.html, gist.github.com/seankross/a412dfbd88b3db70b74bd7). Try using mpg to predict/classify the car’s engine as either V-shaped(0) or Straight(1), as in V-6 or S-4. First, use SimpleRegression to predict \( p_y(\mathbf{x}) \) where \( y \) is V/S and \( x_1 \) is mpg, \(( \mathbf{x} = [1, x_1] \)). Plot \( y \) versus \( x_1 \) and then add a vector to the plot for the predicted values for \( p_y \). Utilizing simple linear regression to predict \( p_y(\mathbf{x}) \) would correspond to the following equation.

\[
p_y(\mathbf{x}) = b_0 + b_1 x_1
\]

8.3.2 Logistic Function

The linear relationship between \( y \) and \( x_1 \) may be problematic, in the sense that there is likely to be a range of rapid transition before which loan default is likely and after which loan repayment is likely. Similarly, there is rapid transition from S(1) to V(0) as mpg increases. This suggests that some “S-curve” function such as the logistic function may be more useful. The standard logistic function (sigmoid function) is

\[
\text{logistic}(z) = \frac{1}{1 + e^{-z}} = \frac{e^z}{1 + e^z} \tag{8.2}
\]

Letting \( z = b_0 + b_1 x_1 \), we obtain

\[
p_y(\mathbf{x}) = \text{logistic}(b_0 + b_1 x_1) = \frac{e^{b_0 + b_1 x_1}}{1 + e^{b_0 + b_1 x_1}} \tag{8.3}
\]

8.3.3 Logit Function

The goal now is to transform the right hand side into the usual linear form (i.e., \( \mathbf{b} \cdot \mathbf{x} \)).

\[
p_y(\mathbf{x}) = \frac{e^{\mathbf{b} \cdot \mathbf{x}}}{1 + e^{\mathbf{b} \cdot \mathbf{x}}}
\]
Multiplying through by $1 + e^{b \cdot x}$ gives

$$p_y(x) + e^{b \cdot x} p_y(x) = e^{b \cdot x}$$

Solving for $e^{b \cdot x}$ yields

$$e^{b \cdot x} = \frac{p_y(x)}{1 - p_y(x)}$$

Taking the natural logarithm of both sides gives

$$\ln \frac{p_y(x)}{1 - p_y(x)} = b \cdot x = b_0 + b_1 x_1$$

(8.4)

where the function on the left hand side is called the logit function.

$$\text{logit}(p_y(x)) = b \cdot x = b_0 + b_1 x_1$$

(8.5)

Putting the model in this form shows it is a special case of a Generalized Linear Model (see Chapter 8) and will be useful in the estimation procedure.

### 8.3.4 Maximum Likelihood Estimation

Imagine you wish to create a model that is able to generate data that looks like the observed data (i.e., the data in the dataset). The choice of values for the parameters $b$ (treated as a random vector) will impact the quality of the model. Define a function of $b$ that will be maximized when the parameters are ideally set to generate the observed data.

#### 8.3.5 Likelihood Function

We can think of this function as the likelihood of $b$ given the predictor vector $x$ and the response variable $y$.

$$L(b|x,y)$$

In this case, $y \in \{0, 1\}$, so if we estimate the likelihood for a single data instance (or row), we have

$$L(b|x,y) = p_y(x)^y (1 - p_y(x))^{1-y}$$

(8.6)

If $y = 1$, then $L = p_y(x)$ and otherwise $L = 1 - p_y(x)$. These are the probabilities for the two outcomes for a Bernoulli random variable (and equation 6.5 concisely captures both).

For each instance $i \in \{0, \ldots, m-1\}$, a similar factor is created. These are multiplied together for all the instances (in the dataset, or training or testing). The likelihood of $b$ given the predictor matrix $X$ and the response vector $y$ is then

$$L(b|x,y) = \prod_{i=0}^{m-1} p_y(x_i)^{y_i} (1 - p_y(x_i))^{1-y_i}$$

(8.7)
8.3.6 Log-likelihood Function

To reduce round-off errors, a log (e.g., natural log, ln) is taken

\[ l(\mathbf{b}|\mathbf{x}, y) = \sum_{i=0}^{m-1} y_i \ln(p_y(x_i)) + (1 - y_i)\ln(1 - p_y(x_i)) \]

This is referred as the log-likelihood function. Collecting \( y_i \) terms give

\[ l(\mathbf{b}|\mathbf{x}, y) = \sum_{i=0}^{m-1} y_i \ln \left( \frac{p_y(x_i)}{1 - p_y(x_i)} \right) + \ln(1 - p_y(x_i)) \]

Substituting \( \mathbf{b} \cdot \mathbf{x}_i \) for \( \text{logit}(p_y(x_i)) \) gives

\[ l(\mathbf{b}|\mathbf{x}, y) = \sum_{i=0}^{m-1} y_i (\mathbf{b} \cdot \mathbf{x}_i) + \ln(1 - p_y(x_i)) \] (8.8)

Multiplying the log-likelihood by -2 makes the distribution approximately Chi-square [?].

\[ -2l = -2 \sum_{i=0}^{m-1} y_i (\mathbf{b} \cdot \mathbf{x}_i) - \ln(1 + e^{\mathbf{b} \cdot \mathbf{x}_i}) \] (8.9)

Or since \( \mathbf{b} = [b_0, b_1] \),

\[ -2l = -2 \sum_{i=0}^{m-1} y_i (b_0 + b_1 x_{i1}) - \ln(1 + e^{b_0 + x_{i1}}) \]

Letting \( \beta_i = b_0 + b_1 x_{i1} \) gives

\[ -2l = -2 \sum_{i=0}^{m-1} y_i \beta_i - \ln(1 + e^{\beta_i}) \]

It is more numerically stable to perform a negative rather than positive \( e^z \) function.

\[ -2l = -2 \sum_{i=0}^{m-1} y_i \beta_i - \beta_i - \ln(e^{-\beta_i} + 1) \] (8.9)

8.3.7 Computation in ScalaTion

The computation of \(-2l\) is carried out in ScalaTion via the \texttt{ll} method. It loops through all instances computing \( \beta_i \) (\( \text{bx} \) in the code) and summing all the terms given in equation 6.9.
def ll (b: VectoD): Double =
{
    var sum = 0.0
    var bx = 0.0 // beta
    for (i <- y.range) {
        bx = b(0) + b(1) * x(i, 1)
        sum += y(i) * bx - bx - log (exp (-bx) + 1.0)
    } // for
    -2.0 * sum
} // ll

8.3.8 Making a Decision

So far, SimpleLogisticRegression is a model for predicting \( p_y(x) \). In order to use this for binary classification a decision needs to be made: deciding on either 0 (no) or 1 (yes). A natural way to do this is to choose 1 when \( p_y(x) \) exceeds 0.5.

override def classify (z: VectoD): (Int, String, Double) =
{
    val p_y = sigmoid (b dot z)
    val c = if (p_y > cThresh) 1 else 0
    (c, cn(c), p_y)
} // classify

In some cases, this may result in imbalance between false positives and false negatives. Quality of Fit (QoF) measures may improve by tuning the classification/decision threshold cThresh. Decreasing the threshold pushes false negatives to false positives. Increasing the threshold does the opposite. Ideally, the tuning of the threshold will also push more cases into the diagonal of the confusion matrix and minimize errors. Finally, in some cases it may be more important to reduce one more than the other, false negatives vs. false positives (see the exercises).

SimpleLogisticRegression Class

Class Methods:

@param x the input/input matrix augmented with a first column of ones
@param y the binary response vector, \( y_i \in \{0, 1\} \)
@param fn_ the names for all features/variable
@param cn_ the names for both classes

class SimpleLogisticRegression (x: MatriD, y: VectoI, fn_: Strings = Array ("one", "x1"),
    cn_: Strings = null)
    extends ClassifierReal (x, y, fn_, 2, cn_)

def ll (b: VectoD): Double =
def ll_null (b: VectoD): Double =
def train (itest: IndexedSeq [Int]): SimpleLogisticRegression =
def train_null ()
def parameter: VectoD = b
override def fit (y: VectoI, yp: VectoI, k: Int = 2): VectoD =
override def fitLabel: Seq [String] = super.fitLabel ++ Seq ("n_dev", "r_dev", "aic", "pseudo_rSq")
override def classify (z: VectoD): (Int, String, Double) =
def reset () { /* Not Applicable */ }

8.3.9 Exercises

1. Plot the standard logistic function (sigmoid).

   import scalation.analytics.ActivationFun.sigmoidV
   val z = VectorD.range (0, 160) / 10.0 - 8.0
   val fz = sigmoidV (z)
   new Plot (z, fz)

2. For the mtcars dataset, determine the model parameters $b_0$ and $b_1$ directly (i.e., do not call train). Rather perform a grid search for a minimal value of the $ll$ function. Use the x matrix (one, mpg) and y vector (V/S) from SimpleLogisticRegressionTest.

   // 32 data points: One Mpg
   val x = new MatrixD ((32, 2), 1.0, 21.0, // 1 - Mazda RX4
                        1.0, 21.0, // 2 - Mazda RX4 Wa
                        1.0, 22.8, // 3 - Datsun 710
                        1.0, 21.4, // 4 - Hornet 4 Drive
                        1.0, 18.7, // 5 - Hornet Sportabout
                        1.0, 18.1, // 6 - Valiant
                        1.0, 14.3, // 7 - Duster 360
                        1.0, 24.4, // 8 - Merc 240D
                        1.0, 22.8, // 9 - Merc 230
                        1.0, 19.2, // 10 - Merc 280
                        1.0, 17.8, // 11 - Merc 280C
                        1.0, 16.4, // 12 - Merc 450S
                        1.0, 17.3, // 13 - Merc 450SL
                        1.0, 15.2, // 14 - Merc 450SLC
                        1.0, 10.4, // 15 - Cadillac Fleetwood
                        1.0, 10.4, // 16 - Lincoln Continental
                        1.0, 14.7, // 17 - Chrysler Imperial
                        1.0, 32.4, // 18 - Fiat 128
                        1.0, 30.4, // 19 - Honda Civic
                        1.0, 33.9, // 20 - Toyota Corolla
1.0, 21.5, // 21 - Toyota Corona
1.0, 15.5, // 22 - Dodge Challenger
1.0, 15.2, // 23 - AMC Javelin
1.0, 13.3, // 24 - Camaro Z28
1.0, 19.2, // 25 - Pontiac Firebird
1.0, 27.3, // 26 - Fiat X1-9
1.0, 26.0, // 27 - Porsche 914-2
1.0, 30.4, // 28 - Lotus Europa
1.0, 15.8, // 29 - Ford Pantera L
1.0, 19.7, // 30 - Ferrari Dino
1.0, 15.0, // 31 - Maserati Bora
1.0, 21.4) // 32 - Volvo 142E

// V/S (e.g., V-6 vs. I-4)
val y = VectorI (0, 0, 1, 1, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0,
0, 1, 1, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1)


4. If the treatment for a disease is risky and consequences of having the disease are minimal, would you prefer to focus on reducing false positives or false negatives?

5. If the treatment for a disease is safe and consequences of having the disease may be severe, would you prefer to focus on reducing false positives or false negatives?
8.4 Logistic Regression

The LogisticRegression class supports logistic regression. In this case, \( x \) may be multi-dimensional \([1, x_1, \ldots, x_k]\). Again, the goal is to fit the parameter vector \( b \) in the regression equation

\[
y = b \cdot x + \epsilon = b_0 + b_1 x_1 + \ldots + b_k x_k + \epsilon
\]

where \( \epsilon \) represents the residuals (the part not explained by the model). This looks like multiple linear regression. The difference being that the response variable \( y \) is binary (\( y \in \{0, 1\} \)). Since \( y \) is binary, minimizing the distance, as was done before may not work well. First, instead of focusing on \( y \in \{0, 1\} \), we focus on the conditional probability of success \( p_y(x) \in [0, 1] \), i.e.,

\[
p_y(x) = P(y = 1|x)
\]

Still, \( p_y(x) \) is bounded, while \( b \cdot x \) is not. We therefore, need a transformation, such as the logit transformation, and fit \( b \cdot x \) to this function. Treating this as a GZLM problem,

\[
y = \mu(x) + \epsilon
\]

\[
g(\mu(x)) = b \cdot x
\]

we let the link function \( g = \text{logit} \).

\[
\text{logit}(\mu(x)) = \ln \frac{p_y(x)}{1 - p_y(x)} = b \cdot x
\]

This is the logit regression equation. Second, instead of minimizing the sum of squared errors, we wish to maximize the likelihood of predicting correct outcomes. For the \( i^{th} \) training case \( x_i \) with outcome \( y_i \), the likelihood function is based on the Bernoulli distribution.

\[
p_y(x_i)^{y_i} (1 - p_y(x_i))^{1-y_i}
\]

The overall likelihood function is the product over all \( m \) cases. The equation is the same as 6.6 from the last section.

\[
L(b|x, y) = \prod_{i=0}^{m-1} p_y(x_i)^{y_i} (1 - p_y(x_i))^{1-y_i}
\] (8.10)

Following the same derivation steps, will give the same log-likelihood that is in equation 6.7.

\[
l(b|x, y) = \sum_{i=0}^{m-1} y_i \cdot b \cdot x_i - \ln(1 + e^{b \cdot x_i})
\] (8.11)

Again, multiplying the log-likelihood function by -2 makes the distribution approximately Chi-square.

\[
-2l = -2 \sum_{i=0}^{m-1} y_i \cdot b \cdot x_i - \ln(1 + e^{b \cdot x_i})
\]

The likelihood can be maximized by minimizing \(-2l\), which is a nonlinear function of the parameter vector \( b \). Various optimization techniques may be used to search for optimal values for \( b \). Currently, SCALATion uses BFGS, a popular general-purpose QuasiNewton NLP solver. Other possible optimizers include LBFGS and IRWLS. For a more detailed derivation, see [http://www.stat.cmu.edu/~cshalizi/350/lectures/26/lecture-26.pdf](http://www.stat.cmu.edu/~cshalizi/350/lectures/26/lecture-26.pdf).
LogisticRegression Class

Class Methods:

@param x  the input/data matrix augmented with a first column of ones
@param y  the binary response vector, y_i in {0, 1}
@param fn_  the names for all features/variable
@param cn_  the names for all classes

class LogisticRegression (x: MatrixD, y: VectorI, fn_: Strings = null,
                       cn_: Strings = null)
    extends ClassifierReal (x, y, fn_, 2, cn_)

def ll (b: VectorD): Double =
def ll_null (b: VectorD): Double =
def train (itest: IndexedSeq[Int]): LogisticRegression =
def train_null ()
def parameter: VectorD = b
override def fit (y: VectorI, yp: VectorI, k: Int = 2): VectorD =
override def fitLabel: Seq[String] = super.fitLabel ++
    Seq("n_dev", "r_dev", "aic", "pseudo_rSq")
override def classify (z: VectorD): (Int, String, Double) =
def forwardSel (cols: Set[Int], adjusted: Boolean = true):
    (Int, VectorD, VectorD) =
def backwardElim (cols: Set[Int], adjusted: Boolean = true, first: Int = 1):
    (Int, VectorD, VectorD) =
def vif: VectorD =
def reset () { /* Not Applicable */ }

8.4.1 Exercises

1. Use Logistic Regression to classify whether stock market will be increasing or not. The Smarket dataset is in the ISLR library, see [46] section 4.6.2.

2. Use Logistic Regression to classify whether a customer will purchase caravan insurance. The Caravan dataset is in the ISLR library, see [46] section 4.6.6.
8.5 Simple Linear Discriminant Analysis

The SimpleLDA class supports Linear Discriminant Analysis which is useful for multiway classification of continuously valued data. The response/classification variable can take on \( k \) possible values, \( y \in \{0, 1, \ldots, k-1\} \). The feature variable \( x \) is one dimensional for SimpleLDA, but can be multi-dimensional for LDA discussed in the next section. Given the data about an instance stored in variable \( x \), pick the best (most probable) classification \( y = c \).

As was done for Naïve Bayes classifiers, we are interested in the probability of \( y \) given \( x \).\[
P(y|x) = \frac{P(x|y)P(y)}{P(x)}
\]
Since \( x \) is now continuous, we need to work with conditional densities as is done Gaussian Naïve Bayes classifiers,
\[
P(y|x) = \frac{f(x|y)P(y)}{f(x)} \tag{8.12}
\]
where
\[
f(x) = \sum_{c=0}^{k-1} f(x|y = c)P(y = c)
\]
Now let us assume the conditional probabilities are normally distributed with a common variance.
\[x|y \sim Normal(\mu_c, \sigma^2)\]
where class \( c \in \{0, 1, \ldots, k - 1\} \), \( \mu_c = \mathbb{E}[x|y = c] \) and \( \sigma^2 \) is the pooled variance (weighted average of \( \forall x|y = c \)). Thus, the conditional density function is
\[
f(x|y = c) = \frac{1}{\sqrt{2\pi\sigma}} e^{-\frac{(x-\mu_c)^2}{2\sigma^2}}
\]
Substituting into equation 6.10 gives
\[
P(y|x) = \frac{1}{\sqrt{2\pi\sigma}} e^{-\frac{(x-\mu_c)^2}{2\sigma^2}} P(y) f(x) \tag{8.13}
\]
where
\[
f(x) = \sum_{c=0}^{k-1} \frac{1}{\sqrt{2\pi\sigma}} e^{-\frac{(x-\mu_c)^2}{2\sigma^2}} P(y = c)
\]
Because of differing means, each conditional density will be shifted resulting in a mountain range appearance when plotted together. Given a data point \( x \), the question becomes, which mountain is it closest to in the sense of maximizing the conditional probability expressed in equation 6.11.
\[
P(y|x) \propto \frac{1}{\sqrt{2\pi\sigma}} e^{-\frac{(x-\mu_c)^2}{2\sigma^2}} P(y)
\]
Since the term \( \frac{1}{\sqrt{2\pi\sigma}} \) is same for all values of \( y \), it may be ignored. Taking the natural logarithm yields
\[
\ln(P(y|x)) \propto -\frac{(x - \mu_c)^2}{2\sigma^2} + \ln(P(y))
\]

Expanding \(-(x - \mu_c)^2\) gives \(-x^2 + 2x\mu_c - \mu_c^2\) and the first term may be ignored (same for all \(y\)).

\[
\ln(P(y|x)) \propto \frac{x\mu_c}{\sigma^2} - \frac{\mu_c^2}{2\sigma^2} + \ln(P(y))
\] (8.14)

The right hand side functions in 4.12 are linear in \(x\) and are called discriminant functions \(\delta_c(x)\).

Given training data vectors \(x\) and \(y\), define \(x_c\) (or \(xc\) in the code) to be the vector of all \(x_i\) values where \(y_i = c\) and let its length be denoted by \(m_c\). Now the \(k\) means may be estimated as follows:

\[
\hat{\mu_c} = \frac{1}{m_c} \cdot x_c
\]

The common variance may be estimated using a pooled variance estimator.

\[
\hat{\sigma^2} = \frac{1}{m-k} \sum_{c=0}^{k-1} ||x_c - \mu_c||^2
\]

Finally, \(m_c/m\) can be used to estimate \(P(y)\).

These can easily be translated into ScalaTion code. Most of the calculations are done in the \texttt{train} method. It estimates the class probability vector \(py\), the group means vector \(mu\) and the pooled variance. The vectors \texttt{term1} and \texttt{term2} capture the \(x\)-term \((\mu_c/\sigma^2)\) and the constant term \((\mu_c^2/2\sigma^2 - \ln(P(y)))\) in equation 6.12.

```scala
def train (itest: IndexedSeq[Int]): SimpleLDA =
{
  py = VectorD(xc.map (_.dim / md)) // probability y = c
  mu = VectorD(xc.map (_.mean)) // group means
  var sum = 0.0
  for (c <- 0 until k) sum += (xc(c) - mu(c)).normSq
  sig2 = sum / (m - k).toDouble // pooled variance
  term1 = mu / sig2
  term2 = mu~^2 / (2.0 * sig2) - py.map (log (_))
  this
} // train
```

Given the two precomputed terms, the \texttt{classify} method simply multiplies the first by \(z(0)\) and subtracts the second. Then it finds the \texttt{argmax} of the \texttt{delta} vector to return the class with the maximum \texttt{delta}, which corresponds the most probable classification.

\[
y^* = \arg\max_c \frac{x\mu_c}{\sigma^2} - \frac{\mu_c^2}{2\sigma^2} + \ln(P(y))
\] (8.15)

```scala
override def classify (z: VectoD): (Int, String, Double) =
{
  val delta = term1 * z(0) - term2
  val best = delta.argmax ()
  (best, cn(best), delta(best))
} // classify
```
Class Methods:

@param x the real-valued training/test data values stored in a vector
@param y the training/test classification vector, where y_i = class for x_i
@param fn_ the name of the feature/variable
@param k the number of possible values for y (0, 1, ... k-1)
@param cn_ the names for all classes

class SimpleLDA (x: VectoD, y: VectoI, fn_: Strings = Array("x1"), k: Int = 2,
                    cn_: Strings = null)
    extends ClassifierReal (MatrixD (x), y, fn_, k, cn_)

def train (itest: IndexedSeq[Int]): SimpleLDA =
override def classify (z: VectoD): (Int, String, Double) =
def reset () { /* Not Applicable */ }

8.5.1 Exercises

1. Generate two samples using Normal (98.6, 1.0) and Normal (101.0, 1.0) with 100 in each sample. Put the data instances into a single x vector. Let the y vector be 0 for the first sample and 1 for the second. Use SimpleLDA to classify all 200 data points and determine the values for tp, tn, fn and fp. See scalation.analytics.classifier.SimpleLDATest2.
8.6 Linear Discriminant Analysis

Like SimpleLDA, the LDA class supports Linear Discriminant Analysis that is used for multiway classification of continuously valued data. Similarly, the response/classification variable can take on \( k \) possible values, \( y \in \{0, 1, \ldots, k - 1\} \). Unlike SimpleLDA, this class is intended for cases where the feature vector \( x \) is multi-dimensional. The classification \( y = c \) is chosen to maximize the conditional probability of class \( y \) given the \( n \)-dimensional data/feature vector \( x \).

\[
P(y|x) = \frac{f(x|y)P(y)}{f(x)}
\]

where

\[
f(x) = \sum_{c=0}^{k-1} f(x|y=c)P(y=c)
\]

In the multi-dimensional case, \( x|y \) has a multivariate Gaussian distribution, \( \text{Normal}(\mu_c, \Sigma) \), where \( \mu_c \) are the mean vectors \( \mathbb{E}[x|y=c] \) and \( \Sigma \) is the common covariance matrix (weighted average of \( \mathbb{C}[x|y=c] \)). The conditional density function is given by

\[
f(x|y=c) = \frac{1}{(2\pi)^{\frac{n}{2}}|\Sigma|^{\frac{1}{2}}} e^{-\frac{1}{2}(x-\mu_c)^T \Sigma^{-1}(x-\mu_c)}
\]

Dropping factors independent of \( c \) and multiplying by \( P(y=c) \) gives

\[
f(x|y=c)P(y=c) \propto e^{-\frac{1}{2}(x-\mu_c)^T \Sigma^{-1}(x-\mu_c)}P(y=c)
\]

Taking the natural logarithm

\[
\ln(P(y|x)) \propto -\frac{1}{2}(x - \mu_c)^T \Sigma^{-1}(x - \mu_c) + \ln(P(y=c))
\]

The discriminant functions are obtained by multiplying out and again dropping terms independent of \( c \).

\[
\delta_c(x) = x^T \Sigma^{-1} \mu_c - \frac{1}{2} \mu_c^T \Sigma^{-1} \mu_c + \ln(P(y=c))
\]

As in the last section, the means for each class \( c \) (\( \mu_c \)), the common covariance matrix (\( \Sigma \)), and the class probabilities (\( P(y) \)) must be estimated.

Class Methods:

```
@param x the real-valued training/test data vectors stored as rows of a matrix
@param y the training/test classification vector, where y_i = class for row i of the matrix x
@param fn_ the names for all features/variables
@param k the number of classes (k in {0, 1, ...k-1})
@param cn_ the names for all classes

class LDA (x: MatrixD, y: VectorI, fn_: Strings = null, k: Int = 2, cn_: Strings = null)
    extends ClassifierReal (x, y, fn_, k, cn_)
    def corrected_cov (xc: MatrixD): MatrixD = (xc.t * xc) / xc.dim1
    def train (itest: IndexedSeq[Int]): LDA =
    def reset () { /* Not Applicable */ }
    override def classify (z: VectorD): (Int, String, Double) =
```

271
8.6.1 Exercises

1. Use LDA to classify manufactured parts according whether they should pass quality control based on curvature and diameter tolerances. See people.revoledu.com/kardi/tutorial/LDA/Numerical%20Example.html for details.
8.7 K-Nearest Neighbors Classifier

The `KNNClassifier` class is used to classify a new vector \( z \) into one of \( k \) classes \( y \in \{0, 1, \ldots, k-1\} \). It works by finding its \( \kappa \)-nearest neighbors to the point \( z \). These neighbors essentially vote according to their classification. The class with the most votes is selected as the classification of vector \( z \). Using a distance metric, the \( \kappa \) vectors nearest to \( z \) are found in the training data, which are stored row-wise in data matrix \( X \). The corresponding classifications are given in vector \( y \), such that the classification for vector \( x_i \) is given by \( y_i \).

In ScalaTion to avoid the overhead of calling `sqrt`, the square of the Euclidean distance is used (although other metrics can easily be swapped in). The squared distance from vector \( x \) to vector \( z \) is then

\[
d(x) = d(x, z) = \|x - z\|^2
\]

The distance metric is used to collect the \( \kappa \) nearest vectors into set \( \text{top}_\kappa(z) \), such that there does not exists any vector \( x_j \notin \text{top}_\kappa(z) \) that is closer to \( z \).

\[
\text{top}_\kappa(z) = \{x_i | i \in \{0, \ldots, \kappa - 1\} \text{ and } \nexists (x_j \notin \text{top}_\kappa(z) \text{ and } d(x_j) < d(x_i))\}
\]

In case of ties for the most distant point to include in \( \text{top}_\kappa(z) \) one could pick the first point encountered or the last point. A less biased approach would be to randomly break the tie.

Now \( y(\text{top}_\kappa(z)) \) can be defined to be the vector of votes from the members of the set, e.g., \( y(\text{top}_3(z)) = [1, 0, 1] \). The ultimate classification is then simply the mode (most frequent value) of this vector (e.g., 1 in this case).

\[
y^* = \text{mode } y(\text{top}_\kappa(z))
\]

8.7.1 Lazy Learning

Training in the `KNNClassifier` class is lazy, i.e., the work is done in the `classify` method, rather than the `train` method.

```scala
override def classify (z: VectorD): (Int, String, Double) =
{
    kNearest (z) // set topK to kappa nearest
    for (i <- 0 until kappa) count(y(topK(i)._1)) += 1 // tally votes per class
    val best = count.argmax () // class with maximal count
    reset () // reset topK and counters
    (best, cn(best), count(best)) // return best class, its name and votes
} // classify
```

The `kNearest` method finds the \( \kappa \times x \) vectors closest to the given vector \( z \). This method updates \( \text{topK} \) by replacing the most distant \( x \) vector in \( \text{topK} \) with a new one if it is closer. Each element in the \( \text{topK} \) array is a tuple \( (j, d(j)) \) indicating which vector and its distance from \( z \). Each of these selected vectors will have their vote taken, voting for the class for which it is labeled. These votes are tallied in the `count` vector. The class with the highest count will be selected as the best class.
KNN Classifier Class

Class Methods:

@param x the vectors/points of classified data stored as rows of a matrix
@param y the classification of each vector in x
@param fn_ the names of the features/variables
@param k the number of classes
@param cn_ the names for all classes
@param kappa the number of nearest neighbors to consider

class KNN_Classifier (x: MatrixD, y: VectorI, fn_: Strings = null, k: Int = 2, cn_: Strings = null, kappa: Int = 3)
  extends ClassifierReal (x, y, fn_, k, cn_)

def distance (x: VectorD, z: VectorD): Double = (x - z).normSq

def knNearest (z: VectorD)

def train (itest: IndexedSeq[Int]): KNN_Classifier =
  override def classify (z: VectorD): (Int, String, Double) =
  def reset ()

8.7.2 Exercises

1. Create a KNN Classifier for the joint data matrix given below and determine its tp, tn, fn, fp values upon re-classification of the data matrix. Let k = 3. Use Leave-One-Out validation for computing tp, tn, fn, fp.

   // x1  x2  y
   val xy = new MatrixD ((10, 3), 1, 5, 1, // joint data matrix 
                          2, 4, 1,
                          3, 4, 1,
                          4, 4, 1,
                          5, 3, 0,
                          6, 3, 1,
                          7, 2, 0,
                          8, 2, 0,
                          9, 1, 0,
                          10, 1, 0)

2. Under what circumstances would one expect a KNN Classifer to perform better than LogisticRegression?

3. How could KNN Classifer be adapted to work for prediction problems?
8.8 Decision Tree C45

The **DecisionTreeC45** class implements a Decision Tree classifier that uses the C4.5 algorithm. The classifier is trained using an $m$-by-$n$ data matrix $X$ and an $n$-dimensional classification vector $y$. Each data vector in the matrix is classified into one of $k$ classes numbered $0, \ldots, k-1$. Each column in the matrix represents a feature (e.g., Humidity). The value count $vc$ vector gives the number of distinct values per feature (e.g., 2 for Humidity).

Depending on the data type of a column, SCALATION’s implementation of C4.5 works like ID3 unless the column is continuous. A column is flagged `isCont` if it is continuous or relatively large ordinal. For a column that `isCont`, values for the feature are split into a left group and a right group based upon whether they are $\leq$ or $>$ an optimal threshold, respectively.

Candidate thresholds/split points are all the mid points between all column values that have been sorted. The threshold giving the maximum entropy drop (or gain) is the one that is chosen.

8.8.1 Example Problem

Consider the following continuous version of the play tennis example. The $x_1$ and $x_2$ columns (Temperature and Humidity) are now listed as continuous measurements rather than as categories as was the case for ID3.

```scala
// The 'ExampleTennis' object is used to test all integer based classifiers.
/** This is the well-known classification problem on whether to play tennis
 * based on given weather conditions.
 * The 'Cont' version uses continuous values for Temperature and Humidity,
 * @see sefiks.com/2018/05/13/a-step-by-step-c4-5-decision-tree-example
 */
object ExampleTennisCont {
  // combined data matrix [ x | y ]
  // dataset ---------------------------------------------------------------
  // x0: Outlook:     Rain (0), Overcast (1), Sunny (2)
  // x1: Temperature: Continuous
  // x2: Humidity:    Continuous
  // x3: Wind:        Weak (0), Strong (1)
  // y: the response/classification decision
  // variables/features:
  // x0  x1  x2  x3  y
  val xy = new MatrixD ((14, 5), 2, 85, 85, 0, 0, // day 1
                        2, 80, 90, 1, 0, // day 2
                        1, 83, 78, 0, 1, // day 3
                        0, 70, 96, 0, 1, // day 4
                        0, 68, 80, 0, 1, // day 5
                        0, 65, 70, 1, 0, // day 6
                        1, 64, 65, 1, 1, // day 7
                        2, 72, 95, 0, 0, // day 8
                        2, 69, 70, 0, 1, // day 9
                        0, 75, 80, 0, 1, // day 10
                        2, 75, 70, 1, 1, // day 11
  }"
As with the ID3 algorithm, the C4.5 algorithm picks \( x_0 \) as the root node. This feature is not continuous and has three branches. Branch \( b_0 \) will lead to a node where as before \( x_3 \) is chosen. Branch \( b_1 \) will lead to a leaf node. Finally, branch \( b_2 \) will lead to a node where continuous feature \( x_2 \) is chosen.

**Sub-problem \( x_0 = 2 \)**

Note that if \( x_0 = 0 \) or 1, the algorithm works like ID3. However, there is still some uncertainty left when \( x_0 = 2 \), so this node may be split and it turn out the split will involve continuous feature \( x_2 \). The sub-problem for Outlook: Rain (2) see Table 8.1 is defined as follows: Take all five cases/rows in the data matrix \( X \) for which \( x_{-0} = 2 \).

### Table 8.1: Sub-problem for node \( x_0 \) and branch 2

<table>
<thead>
<tr>
<th>Day</th>
<th>( x_{-1} )</th>
<th>( x_{-2} )</th>
<th>( x_{-3} )</th>
<th>( y )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>85</td>
<td>85</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>80</td>
<td>90</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>72</td>
<td>95</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>69</td>
<td>70</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>75</td>
<td>70</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

The distinct values for feature \( x_2 \) in sorted order are the following: \([70.0, 85.0, 90.0, 95.0] \). Therefore, the candidate threshold/split points for continuous feature \( x_2 \) are their midpoints: \([77.5, 87.5, 92.5] \). Threshold 77.5 yields (0-, 2+) on the left and (3-, 0+) on the right, 87.5 yields (1-, 2+) on the left and (2-, 0+) on the right, and 92.5 yields (2-, 2+) on the left and (1-, 0+) on the right. Clearly, the best threshold value is 77.5.

Since a continuous feature splits elements into low (left) and high (right) groups, rather than branching on all possible values, the same continuous feature may be chosen again by a descendant node.

### DecisionTreeC45 Class

**Class Methods:**

- @param \( x \) the data vectors stored as rows of a matrix
- @param \( y \) the class array, where \( y_i \) = class for row \( i \) of the matrix \( x \)
- @param \( \text{fn} \) the names for all features/variables
8.8.2 Exercises

1. Run `DecisionTreeC45` on the `ExampleTennis` dataset and verify that it produces the same answer as `DecisionTreeID3`.

2. Complete the C45 Decision Tree for the `ExampleTennisComp` problem.

3. Run `DecisionTreeC45` on the `winequality-white` dataset. Plot the accuracy versus the maximum tree depth (`td`).
8.9 Random Forest

The `RandomForest` class builds multiple decision trees for a given problem. Each decision tree is built using a sub-sample (rows) of the data matrix `x` and a subset of the columns/features. The fraction of rows used is given by `bR` the bagging ratio, while the number of columns used is given by `fS` the number of features used in building trees. Given a new instance vector `z`, each of the trees will classify it and the class with the most number of votes (one from each tree), will be the overall response of the random forest.

Class Methods:

```scala
@param x the data matrix (instances by features)
@param y the response class labels of the instances
@param nF the number of trees
@param bR bagging ratio (the portion of samples used in building trees)
@param fS the number of features used in building trees
@param k the number of classes
@param s seed for randomness
@param fn_ feature names (array of string)
@param cn_ class names (array of string)

class RandomForest (x: MatrixD, y: VectorI, nF: Int, bR: Double, fS: Int, k: Int, s: Int,
val fn_: Strings = null, val cn_: Strings = null)
extends ClassifierReal (x, y, fn_, k, cn_) with Error

def createSubsample (): MatrixD =
def selectSubFeatures (subSample: MatrixD): (MatrixD, VectorI) =
def train (testStart:Int, testEnd:Int) =
def classify (z: VectorD): (Int, String, Double) =
def reset() {}
```

278
8.10 Support Vector Machine

The `SupportVectorMachine` class implements linear support vector machines (SVM). A set of vectors stored in a matrix are divided into positive(1) and negative(-1) cases. The algorithm finds a hyperplane that best divides the positive from the negative cases. Each vector $x_i$ is stored as a row in the $x$ matrix.

---

**Example Problem:**

**Class Methods:**

```scala
@param x the matrix consisting of vectors
@param y the vector of outcomes (e.g., positive(1), negative(-1))
@param fn_ the names of the features/variables
@param cn_ the class names

class SupportVectorMachine (x: MatrixD, y: VectorI, fn_: String*, cn_: Array[String])
  extends ClassifierReal (x, y, fn_, 2, cn_)

  def l_D (a: VectorD): Double =
  def g (a: VectorD): Double = a dot y
  def find_w ()
  def find_b ()
  def train ()
  def fit: (VectorD, Double) = (w, b)
  def classify (z: VectorD): Int = (signum (w dot z + b)).toInt
```

---
Chapter 9

Generalized Linear and Additive Models

General Linear Models presented in Chapter 5 cover a wide range of simple models that are easy to use and explain and can be rapidly trained typically using Ordinary Least Squares (OLS) that boils down to matrix factorization.

Complex nonlinear models can provide improved Quality of Fit and handle cases where General Linear Models are not sufficiently predictive. However, they often result in the following disadvantage: harder to use and explain, substantially longer training times, more likely to overfit and require tuning of hyper-parameters.

This chapter examines two categories of modeling techniques that fall between the two extreme categories of models just discussed. The two categories of models of intermediate complexity are Generalized Linear Models and Generalized Additive Models.

9.1 Generalized Linear Model

A Linear Model may be viewed as fitting a distribution to a continuous random variable $y$.

$$ y \sim \text{Normal}(b \cdot x, \sigma^2) \quad (9.1) $$

By subtracting the mean $b \cdot x$, $y$ becomes the sum of two terms.

$$ y = b \cdot x + \epsilon \quad (9.2) $$

where $\epsilon \sim \text{Normal}(0, \sigma^2)$.

Now generalize the first term $b \cdot x$ into an optionally transformed linear function of $x$ called the mean function.

$$ y = \mu_y(x) + \epsilon \quad (9.3) $$

The mean function is restricted to be of the following form.

$$ g(\mu_y(x)) = b \cdot x \quad (9.4) $$
where \( g \) is an invertible function. One may think of \( g \) as a link function, or its inverse \( g^{-1} \) as an activation function. The idea of a link function is that it uncovers an underlying linear model \( \mathbf{b} \cdot \mathbf{x} \). In other words, \( g \) is a function that links \( y \)'s mean to a linear combination of the predictor variables. The idea of an activation function is that it allows a transformation function to be applied to the linear combination \( \mathbf{b} \cdot \mathbf{x} \), as is done in Perceptrons.

\[
\mu_y(x) = g^{-1}(\mathbf{b} \cdot \mathbf{x}) \tag{9.5}
\]

For example, a commonly used activation function in Perceptrons is the sigmoid function.

\[
\mu_y(x) = \text{sigmoid}(\mathbf{b} \cdot \mathbf{x}) \tag{9.6}
\]

where the sigmoid function (a simple form of the more general logistic function) is

\[
\text{sigmoid}(t) = \frac{1}{1 + e^{-t}} \tag{9.7}
\]

The inverse of the sigmoid function is the logit function.

\[
\logit(u) = \ln\frac{u}{1 - u} \tag{9.8}
\]

Applying the logit function to both sides of equation yields

\[
\logit(\mu_y(x)) = \mathbf{b} \cdot \mathbf{x} \tag{9.9}
\]

The form of the response random variable \( y \) is also generalized, e.g., it may be a discrete random variable.

So far, we have been focusing on the first term, the mean function, but what about the second term, the residual/error term? The treatment of residuals/errors in Generalized Linear Models, is generalized as well. For one, they need not follow the Normal distribution. Also, they are not required to be additive, e.g., they may be multiplicative.

To produce this level of generality/flexibility the methodology for Generalized Linear Models is to separate the model into a systematic component to determine the mean function, and a random component to handle the residuals/errors. Continuing with the mean function above, let \( y \) have domain \( D_y = \{0, 1\} \) and the errors \( \epsilon \) be additive and distributed as Bernoulli(\( p \)). Writing the two components together yields

\[
y = \mu_y(x) + \epsilon
\]

\[
\logit(\mu_y(x)) = \mathbf{b} \cdot \mathbf{x}
\]

These are the model equations of Logistic Regression. When the link function is the identity function, the response random variable is continuous and the errors \( \epsilon \) distributed as Normal(0, \( \sigma^2 \)), we are back to a Linear Model, written as a Generalized Linear Model.

\[
y = \mu_y(x) + \epsilon
\]

\[
\mu_y(x) = \mathbf{b} \cdot \mathbf{x}
\]

Several additional combinations of link functions and residual distributions are commonly used as shown in the table below.
<table>
<thead>
<tr>
<th>Model Type</th>
<th>Response Type (y)</th>
<th>Link Function</th>
<th>Residual Distribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logistic Regression</td>
<td>binary {0, 1}</td>
<td>logit</td>
<td>Bernoulli Distribution</td>
</tr>
<tr>
<td>Poisson Regression</td>
<td>integer {0, ..., ∞}</td>
<td>ln</td>
<td>Poisson Distribution</td>
</tr>
<tr>
<td>Exponential Regression</td>
<td>continuous (0, ∞)</td>
<td>ln or reciprocal</td>
<td>Exponential Distribution</td>
</tr>
<tr>
<td>General Linear Model (GLM)</td>
<td>continuous ((-∞, ∞))</td>
<td>identity</td>
<td>Normal Distribution</td>
</tr>
</tbody>
</table>

Table 9.1: Types of Generalized Linear Models


Since the response variable for Logistic Regression is defined on finite domains, it has been placed under Classification (see the last chapter).

Example Problem:

Class Methods:

```scala
object GZLM extends GLM

def apply (x: MatriD, y: VectoI, cn: Array[String]): LogisticRegression =

def apply (x: MatriD, y: VectoI, fn: Array[String], poisson: Boolean): PoissonRegression =

def apply (x: MatriD, nonneg: Boolean, y: VectoD): ExpRegression =
```

283
9.2 Maximum Likelihood Estimation

In this section, rather than estimating parameters using Least Squares Estimation (LSE), Maximum Likelihood Estimation (MLE) will be used. Given a dataset with $m$ instances, the model will produce an error for each instance. When the error is large, the model is in disagreement with the data. When errors are normally distributed, the probability density will be low for a large error, meaning this is an unlikely case. If this is true for many of the instances, the problem is not the data, it is the values given for the parameters. The parameter vector $b$ should be set to maximize the likelihood of seeing instances in the dataset. This notion is captured in the likelihood function $L(b)$. Note, for the Simpler Regression model there is only a single parameter, the slope $b$.

Given an $m$ instance dataset $(x,y)$ where both are $m$-dimensional vectors and a Simpler Regression model

$$y = bx + \epsilon$$

where $\epsilon \sim N(0, \sigma^2)$, let us consider how to estimate the parameter $b$. While LSE is based upon the distance of errors from zero, MLE transforms this distance based upon for example the pdf of the error distribution. Notice that for small errors, the Normal distribution is more tolerant than the Exponential distribution, while for larger errors it is less tolerant. See the exercises for how to visualize this.

For this model and dataset, the likelihood function $L(b)$ is the product of $m$ Normal density functions (making the assumption that the instances are independent).

$$L(b) = \prod_{i=0}^{m-1} \frac{1}{\sqrt{2\pi\sigma}} e^{-\epsilon_i^2/2\sigma^2}$$

Since $\epsilon_i = y_i - bx_i$, we may rewrite $L(b)$ as

$$L(b) = \prod_{i=0}^{m-1} \frac{1}{\sqrt{2\pi\sigma}} e^{-(y_i - bx_i)^2/2\sigma^2}$$

Taking the natural logarithm gives the log-likelihood function $l(b)$

$$l(b) = \sum_{i=0}^{m-1} -\ln(\sqrt{2\pi\sigma}) - (y_i - bx_i)^2/2\sigma^2$$

The derivative of $l(b)$ w.r.t. $b$ is

$$\frac{dl}{db} = \sum_{i=0}^{m-1} -2x_i(y_i - bx_i)/2\sigma^2$$

For optimization, the derivative may be set to zero

$$\sum_{i=0}^{m-1} x_i(y_i - bx_i) = 0$$

Solving for $b$ gives

$$b = \frac{\sum x_i y_i}{\sum x_i^2} = \frac{x \cdot y}{x \cdot x}$$
9.2.1 Akaike Information Criterion

The Akaike Information Criterion (AIC) serves as an alternative to measures like $R^2$ Adjusted, $\bar{R}^2$.

$$AIC = 2(n + 1) - 2l(b, \sigma^2)$$

It is twice the number of parameters $n + 1$ to be estimated minus twice the optimal log-likelihood. Note, the $n$ comes from estimating the coefficients $\dim(b)$ and 1 comes from estimating the error/residual variance $\sigma^2$. The smaller the AIC value, the better the model.

For linear regression, the above formula is equivalent to

$$AIC = 2(n + 1) - m \ln \left( \frac{sse}{m} \right) + constant$$

and the constant may be ignored as all models for the given dataset will have the same constant.

9.2.2 MLE for Generalized Linear Models

Maximum Likelihood Estimation (MLE) is general purpose and applies widely to statistical estimation problems.

Let us consider how it can be applied to Generalized Linear Models. A Generalized Linear Model for a response random variable $y$ is of the following form.

$$y = \mu_y(x) \odot \epsilon$$

$$g(\mu_y(x)) = b \cdot x$$

The $\odot$ is used to indicate either $+$ (for additive errors) or $\times$ (for multiplicative errors).

Now given an $m$-instance dataset $(X, y)$, we need to determine the link function $g$, the residual/error distribution $\epsilon \sim \text{Dist}(b)$, and finally the parameters $b$ using MLE.

When $y$ is continuous, the pdf conditioned on the parameters may be used to capture the error distribution.

$$f_\epsilon(\epsilon | b)$$

The error for each instance may be calculated from the dataset.

$$\epsilon_i = y_i - g^{-1}(b \cdot x_i) \quad \text{for additive errors}$$

$$\epsilon_i = y_i / g^{-1}(b \cdot x_i) \quad \text{for multiplicative errors}$$

When errors are highly correlated, Generalized Linear Models are not ideal. Thus, we assume the errors are independent and identically distributed (iid).

Because of the independence assumption, the joint error density is the product of the density for each instance $\epsilon_i$. Therefore, the likelihood function w.r.t. the parameter vector $b$ is

$$L(b) = \prod_{i=0}^{m-1} f_\epsilon(\epsilon_i | b)$$
Taking the natural logarithm yields

\[ l(\mathbf{b}) = \sum_{i=0}^{m-1} \ln f_{\epsilon_i}(\epsilon_i|\mathbf{b}) \]  \hspace{1cm} (9.10)

Analogously, for discrete random variables, the pdf is replaced with pmf

\[ l(\mathbf{b}) = \sum_{i=0}^{m-1} \ln p_{\epsilon_i}(\epsilon_i|\mathbf{b}) \]  \hspace{1cm} (9.11)

For Maximum Likelihood Estimation, the above two equations may serve as a starting point.
9.3 Poisson Regression

The PoissonRegression class can be used for developing Poisson Regression models. In this case, a response $y$ may be thought of as a count that may take on a non-negative integer value. The probability density function (pdf) for the Poisson distribution with mean $\lambda$ may be defined as follows:

$$f(y; \lambda) = \frac{\lambda^y}{y!} e^{-\lambda}$$

Again, treating this as a GZLM problem,

$$y = \mu(x) + \epsilon$$

$$g(\mu(x)) = b \cdot x$$

The link function $g$ for Poisson Regression is the ln (natural logarithm) function.

$$\ln(\mu(x)) = b \cdot x$$

The residuals $\epsilon_i$ are distributed according to the Poisson distribution.

$$\frac{\mu(x_i)^{y_i}}{y_i!} e^{-\mu(x_i)}$$

Therefore, the likelihood function for Poisson Regression is as follows:

$$L = \prod_{i=0}^{m-1} \frac{\mu(x_i)^{y_i}}{y_i!} e^{-\mu(x_i)}$$

Taking the natural logarithm gives the log-likelihood function.

$$LL = \sum_{i=0}^{m-1} y_i \ln(\mu(x_i)) - \mu(x_i) - \ln(y_i!)$$

Substituting $\mu(x_i) = e^{b \cdot x_i}$ yields the following:

$$LL = \sum_{i=0}^{m-1} y_i b \cdot x_i - e^{b \cdot x_i} - \ln(y_i!)$$

Since the last term is independent of the parameters, removing it will not affect the optimization.

$$LL_2 = \sum_{i=0}^{m-1} y_i b \cdot x_i - e^{b \cdot x_i}$$

See [http://www.stat.uni-muenchen.de/~helmut/Geo/stat_geo_11_Handout.pdf](http://www.stat.uni-muenchen.de/~helmut/Geo/stat_geo_11_Handout.pdf) for more details.

Example Problem:
PoissonRegression Class

Class Methods:

@param x the input/data matrix augmented with a first column of ones
@param y the integer response vector, y_i in {0, 1, ... }
@param fn the names of the features/variable

class PoissonRegression (x: Matrix, y: Vector, fn: Array[String] = null)
    extends Classifier with Error

    def ll (b: Vector): Double =
    def ll_null (b: Vector): Double =
    def train (yy: Vector) { throw new UnsupportedOperationException("train (yy) not implemented yet") }
    def train ()
    def train_null ()
    override def fit: Vector =
    override def fitLabels: Seq[String] = Seq("n_dev", "r_dev", "aic", "pseudo_rSq")
    def predict (z: Vector): Double = (round (exp (b dot z))).toDouble
9.4 Generalized Additive Models

A Generalized Additive Model (GAM) can be developed using the GZLM class.
9.5 Regression Trees

As with Decision (or Classification) Trees, Regression Trees make predictions based upon what range each variable/feature is in. If the tree is binary, there are two ranges for each feature split: low (below a threshold) and high (above a threshold). Building a Regression Tree essentially then requires finding thresholds for splitting variables/features. A threshold will split a dataset into two groups. Letting $\theta_k$ be a threshold for splitting variable $x_j$, we may split the rows in the $X$ matrix into left and right groups.

$$\text{left}_k(X) = \{x_i| x_{ij} \leq \theta_k\} \quad (9.12)$$
$$\text{right}_k(X) = \{x_i| x_{ij} > \theta_k\} \quad (9.13)$$

For splitting variable $x_j$, the threshold $\theta_k$ should be chosen to minimize the sum of the Mean Squared Error (MSE) of the left and right sides. Alternatively, one can minimize the Sum of Squared Errors (SSE). This variable becomes the root node of the regression tree. The dataset for the root node’s left branch consists of $\text{left}_k(X)$, while the right branch consists of $\text{right}_k(X)$. If the maximum tree depth is limited to one, the root’s left child and right child will be leaf nodes. For a leaf node, the prediction value that minimizes MSE is the mean $\mu(y)$.

9.5.1 Example Problem

Consider the following small dataset with just one predictor variable $x_0$.

```scala
val x = new MatrixD ((10, 1), 1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
val y = VectorD (5.23, 5.7, 5.91, 6.4, 6.8, 7.05, 8.9, 8.7, 9.0, 9.05)
```

In this case, $\theta_0 = 6.5$ divides the dataset into

$$\text{left}_0(X) = \{1, 2, 3, 4, 5, 6\}$$
$$\text{right}_0(X) = \{7, 8, 9, 10\}$$

with means $\mu_0(y) = 6.18$ (left) and $\mu_1(y) = 8.91$ (right). Further splitting may occur on $x_0$ (or $x_j$ for multidimensional examples). If we let the maximum tree depth be two, we obtain the following four regions, corresponding to the four leaf nodes,

Root (-Inf, Inf]
Node $x_0$ in (-Inf, 6.5]
Leaf $x_0$ in (-Inf, 3.5]
Leaf $x_0$ in (3.5, 6.5]
Node $x_0$ in (6.5, Inf]
Leaf $x_0$ in (6.5, 8.5]
Leaf $x_0$ in (8.5, Inf]

with means $\mu_0(y) = 5.61$, $\mu_1(y) = 6.75$, $\mu_2(y) = 8.80$ and $\mu_3(y) = 9.03$. Each internal (non-leaf) node will have a threshold. They are $\theta_0 = 6.5$, $\theta_1 = 3.5$ and $\theta_2 = 8.5$. 

290
9.5.2 Regions
The number of regions (or leaf nodes) is always one greater than the number of thresholds. The region for leaf node \( l \), \( R_l = (x_j, (a_l, b_l]) \), defines the feature/variable being split and the interval of inclusion. Corresponding to each region \( R_l \) is an indicator function,

\[
1_l(x) = x_j \in (a_l, b_l]
\] (9.14)

which simply indicates (false/true) whether variable \( x_j \) is in the interval \( (a_l, b_l] \). Now define \( 1^*_l(x) \) as the product of the indicator functions from leaf \( l \) until (not including) the root of the tree,

\[
1^*_l(x) = \prod_{h \in \text{anc}(l)} 1_h(x)
\] (9.15)

where \( \text{anc}(l) \) is the set of ancestors of leaf node \( l \) (inclusive of \( l \), exclusive of root). Since only one of these \( 1^*_l \) indicator functions can be true for any given \( x \) vector, we may concisely express the regression tree model as follows:

\[
y = \sum_{l \in \text{leaves}} 1^*_l(x) \mu_l(y) + \epsilon
\] (9.16)

Thus, given a predictor vector \( x \), predicting a value for the response variable \( y \) corresponds to taking the mean \( y \)-value of the vectors in \( x \)'s composite region (the intersection of regions from the leaf until the root). As locality determines the prediction for Regression Trees, they are similar to K-NN Predictors.

9.5.3 Determining Thresholds
For the \( k^{th} \) split, a simple way to determine the best threshold is to take each feature/variable \( x_j \) and find a value \( \theta_k \) that minimizes the sum of the MSEs.

\[
\min_{\theta_k} \text{mse}(\text{left}_k(X)) + \text{mse}(\text{right}_k(X))
\] (9.17)

Possible values for \( \theta_k \) are the values between any two consecutive values in vector \( x_j \) sorted. This will allow any possible split of \( x_j \) to be considered. For example, \( \{1, 10, 11, 12\} \) should not be split in the middle, e.g., into \( \{1, 10\} \) and \( \{11, 12\} \), but rather into \( \{1\} \) and \( \{10, 11, 12\} \). Possible thresholds (split points) are the averages of any two consecutive values, i.e., 5.5, 10.5 and 11.5. A straightforward way to implement determining the next variable \( x_j \) and its threshold \( \theta_k \) would be to iterate over all features/variables and split points. Calculating the sum of left and right mse (or sse) from scratch for each candidate split point is inefficient. These values may be computed incrementally using the fast thresholding algorithm.

RegressionTree Class

Class Methods:

- @param x the data vectors stored as rows of a matrix
- @param y the dependent value
- @param fn the names for all features/variables
- @param maxDepth the depth limit for tree
9.6 Exercises

1. Let $x \sim \text{Exponential}(\lambda)$ and $y \sim \text{Normal}(\mu, \sigma^2)$ with $\lambda = 1$, $\mu = 0$ and $\sigma = 1$, the corresponding pdfs are as follows:

$$f_x(x) = e^{-x}$$
$$f_y(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}}$$

Plot and compare the pdfs $f_x$ and $f_y$ vs. $x$ over the interval $[0, 4]$.

2. Consider the following two-dimensional Regression Tree problem. FIX.

3. Contrast K-NN Predictors with Regression Trees in terms of the shape of and how regions are formed.

9.7 Further Reading


2. Generalized Linear Models (GLM) [104]
Chapter 10

Nonlinear Models and Neural Networks
10.1 Nonlinear Regression

The NonlinRegression class supports Nonlinear Regression (NLR). In this case, the vector of input/predictor variable \( \mathbf{x} \in \mathbb{R}^n \) can be multi-dimensional \([1, x_1, \ldots, x_k]\) and the function \( f \) is nonlinear in the parameters \( \mathbf{b} \in \mathbb{R}^p \).

### 10.1.1 Model Equation

As before, the goal is to fit the parameter vector \( \mathbf{b} \) in the model/regression equation,

\[
y = f(\mathbf{x}; \mathbf{b}) + \epsilon
\]

where \( \epsilon \) represents the residuals (the part not explained by the model). The function \( f : \mathbb{R}^n \times \mathbb{R}^p \rightarrow \mathbb{R} \) is called the mean response function.

Note that \( y = b_0 + b_1 x_1 + b_2 x_1^2 + \epsilon \) is still linear in the parameters. The example below is not, as there is no transformation that will make the formula linear in the parameters.

\[
y = \frac{(b_0 + b_1 x_1)}{(b_2 + x_1)} + \epsilon
\]

Nonlinear Regression can more precisely model phenomena compared to using the linear approximations of multiple linear regression. Although quadratic and cubic regressions allow the lines to be bent to better fit the data, they do not provide the flexibility of nonlinear regression, see Chapter 13 of [53]. The functional forms in nonlinear regressions can be selected by the physics driving the phenomena. At the other extreme, simple nonlinear functions of linear combinations of several input variables, allow many diverse phenomena to be modeled, for example, using Neural Networks. Neural Networks trade interpretability for a universal modeling framework.

### 10.1.2 Training

A training dataset consisting of \( m \) input-output pairs is used to minimize the error in the prediction by adjusting the parameter vector \( \mathbf{b} \). Given an input matrix \( \mathbf{X} \) consisting of \( m \) input vectors and an output vector \( \mathbf{y} \) consisting of \( m \) output values, minimize the distance between the target output vector \( \mathbf{y} \) and the predicted output vector \( f(\mathbf{X}; \mathbf{b}) \).

The model training process depicted in Figure [10.1] involves

- a Dataset \((\mathbf{X}, \mathbf{y})\) where input/data matrix \( \mathbf{X} \in \mathbb{R}^{m \times n} \) and output/response vector \( \mathbf{y} \in \mathbb{R}^m \),
- a Model with mean response function \( f \) or vectorized mean response function \( f : \mathbb{R}^{m \times n} \times \mathbb{R}^p \rightarrow \mathbb{R}^m \),
  and
- a Trainer with loss function \( h : \mathbb{R}^p \rightarrow \mathbb{R} \).

In Figure [10.1] the \( i^{th} \) row/instance of the Dataset is taken as the input vector \( \mathbf{x} \in \mathbb{R}^n \) to the model to produce a predicted response,

\[
\hat{y} = f(\mathbf{x}; \mathbf{b}) \quad \text{instance level}
\]
\[
\hat{y} = f(\mathbf{X}; \mathbf{b}) \quad \text{dataset level}
\]
where \( f(\mathbf{f}) \) is the (vectorized) mean response function and the parameter vector \( \mathbf{b} \in \mathbb{R}^p \).

Note that for linear models typically \( p = n \), but for nonlinear models, it is not uncommon for \( p > n \), i.e., the number of parameters to be greater than the number of input/predictor variables.

**Loss Function**

Optimization techniques can be used to determine a “good” value for the parameter vector \( \mathbf{b} \), by minimizing a loss function.

\[
h(\mathbf{b}; \mathbf{y}, \hat{\mathbf{y}})
\]

Common forms of loss functions are based on minimizing an \( \ell_1 \) or \( \ell_2 \) norm of the error vector \( \mathbf{e} = \mathbf{y} - \hat{\mathbf{y}} = \mathbf{y} - f(\mathbf{X}; \mathbf{b}) \),

\[
\min_{\mathbf{b}} \| \mathbf{y} - f(\mathbf{X}; \mathbf{b}) \| \quad (10.2)
\]

Again for an \( \ell_2 \) norm, it is convenient to use its square and minimize the dot product of the error with itself (or rather half of that \( \frac{1}{2} \mathbf{e} \cdot \mathbf{e} \)).

\[
h(\mathbf{b}) = \frac{1}{2} (\mathbf{y} - f(\mathbf{X}; \mathbf{b})) \cdot (\mathbf{y} - f(\mathbf{X}; \mathbf{b})) \quad (10.3)
\]

An alternative equation for the loss function uses summation notation to add up the square of each error \( \epsilon_i = y_i - f(\mathbf{x}_i; \mathbf{b}) \) over all \( m \) instances.

\[
h(\mathbf{b}) = \frac{1}{2} \sum_{i=0}^{m-1} [y_i - f(\mathbf{x}_i; \mathbf{b})]^2 \quad (10.4)
\]
Special Case of a Linear Model

For a linear model, \( f \) is simply a linear combination of the input variables, i.e.,

\[
h(b) = \frac{1}{2} (y - Xb) \cdot (y - Xb)
\]

Recall that taking the gradient with respect to \( b \) and setting it equal to 0, yields the Normal Equations,

\[
(X^T X)b = X^T y
\]

Then matrix factorization can be used to relatively quickly find a globally optimal value for the parameter vector \( b \).

10.1.3 Optimization

For nonlinear regression, a Least-Squares (minimizing the errors/residuals) method can be used to fit the parameter vector \( b \). Finding optimal values for the parameters \( b \) may be found by either solving the Normal Equations or using an iterative optimization algorithm. In either case, gradients of the loss function are used to find these values.

\[
h(b) = \frac{1}{2} \sum_{i=0}^{m-1} [y_i - f(x_i; b)]^2
\]

The \( j^{th} \) partial derivative of this loss function \( h(b) \) may be determined using the chain rule.

\[
\frac{\partial h(b)}{\partial b_j} = - \sum_{i=0}^{m-1} [y_i - f(x_i; b)] \frac{\partial f(x_i; b)}{\partial b_j} \quad (10.5)
\]

Setting each partial derivative \((j = 0 \ldots p - 1)\) to zero yields,

\[
\sum_{i=0}^{m-1} f(x_i; b) \frac{\partial f(x_i; b)}{\partial b_j} = \sum_{i=0}^{m-1} y_i \frac{\partial f(x_i; b)}{\partial b_j}
\]

Collecting these equations for all values of \( j \) leads to the Normal Equations for nonlinear models. Unfortunately, they form a system of nonlinear equations requiring numerical solution. Instead of solving the Normal Equations for nonlinear models, one may use an optimization algorithm to minimize the loss function \( h(b) \).

First-Order Optimization Algorithms

First-order optimization algorithms utilize first derivatives, typically moving in the opposite direction to the gradient.

\[
\nabla h(b) = \left[ \frac{\partial h(b)}{\partial b_0}, \ldots, \frac{\partial h(b)}{\partial b_{p-1}} \right] \quad (10.6)
\]

Noting that \( \epsilon_i = y_i - f(x_i; b) \), the \( j^{th} \) partial derivative can be written more concisely.

\[
\frac{\partial h(b)}{\partial b_j} = - \sum_{i=0}^{m-1} \epsilon_i \frac{\partial f(x_i; b)}{\partial b_j} \quad (10.7)
\]
Therefore, an optimization algorithm essentially operates off of the gradient of the mean response function \( f \). For greater accuracy and efficiency for a given function \( f \), utilization of formulas for its partial derivatives is preferred over their numerical calculation.

**Second-Order Optimization Algorithms**

*Second-order optimization algorithms* utilize both first derivatives (gradient) and second derivatives (Hessian). A user defined mean response function \( f \) that takes a vector of inputs \( x \) and a vector of parameters \( b \),

\[
f: (\text{VectoD, VectoD}) \Rightarrow \text{Double}
\]

is passed as a parameter to the `NonLinearRegression` class. This function is used to create a predicted output value \( y_p \) for each input vector \( x_i \). The `sseF` method (as a loss function) applies function \( f \) to all \( m \) input vectors to compute predicted output values. These are then subtracted from the actual output to create an error vector, whose squared Euclidean norm is returned.

```scala
def sseF (b: VectoD): Double =  
{ 
  val yp = new VectorD (m) // create vector yp to hold predicted outputs 
  for (i <- 0 until m) yp(i) = f (x(i), b) // compute values for yp (y-predicted) 
  (y - yp).normSq // sum of squared errors (sse) 
} // sseF
```

**ScalaTion**’s `minima` and `maxima` packages provide several solvers for linear, quadratic, integer and nonlinear programming/optimization. Currently, the `QuasiNewton` class is used for finding an optimal \( b \) by minimizing `sseF`. It uses the second-order Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm. The `QuasiNewton` optimizer requires an initial guess for the parameter vector \( b \).

```scala
val bfgs = new QuasiNewton (sseF) // minimize sse using NLP 
val b = bfgs.solve (b_init) // estimate for b from optimizer
```


**NonLinRegression Class**

**Class Methods:**

- `@param x` the data/input matrix
- `@param y` the response/output vector
- `@param f` the nonlinear function \( f(x, b) \) to fit
- `@param b_init` the initial guess for the parameter vector \( b \)
- `@param fname_` the feature/variable names
- `@param hparam` the hyper-parameters (currently has none)

```scala
class NonLinRegression (x: MatriD, y: VectoD, f: FunctionP2S,
```
10.1.4 Exercises

1. Given the Normal Equations for nonlinear models,

\[
\sum_{i=0}^{m-1} f(x_i; b) \frac{\partial f(x_i; b)}{\partial b_j} = \sum_{i=0}^{m-1} y_i \frac{\partial f(x_i; b)}{\partial b_j}
\]

simply them assuming \( f \) is linear in the parameters \( b \), i.e.,

\[
f(x_i; b) = x_i \cdot b
\]

Hint:

\[
\frac{\partial f(x_i; b)}{\partial b_j} = x_{ij}
\]

2. Enzyme catalyzed biochemical reactions may be described by a Michaelis-Menten nonlinear regression model [64]. The reaction rate/velocity \( y = v \) is a function of the substrate concentration \( x \),

\[
y = \frac{b_0 x}{x + b_1} + \epsilon
\]

where parameter \( b_0 = V_{max} \) is the maximum reaction velocity (occurs at high substrate concentration) and parameter \( b_1 = K_m \) is the Michaelis constant that measures the strength of the enzyme-substrate interaction. Use the dataset given in Table 2 of [64] to train the model.
10.2 Simple Exponential Regression

The `SimpleExpRegression` class can be used for developing Simple Exponential Regression models. These are useful when data exhibit exponential growth or decay.

10.2.1 Model Equation

Simple Exponential Regression models have a single predictor variable \( x \). They are a type of Nonlinear Model for a response random variable \( y \) having a model equation of the following form,

\[
y = b_0 e^{b_1 x} + \epsilon
\]  

(10.8)

The mean response function has two parameters, a multiplier \( b_0 \) and a growth rate \( b_1 \).

\[
f(x, b) = b_0 e^{b_1 x}
\]  

(10.9)

Note, for `SimpleExpRegression` the number of parameters \( p = 2 \), while the number of predictor variables \( n = 1 \).

10.2.2 Training

Given a dataset \((x, y)\) where \( x \in \mathbb{R}^m \) and \( y \in \mathbb{R}^m \), the parameters \( b = [b_0, b_1] \) may be determined using Least Squares Estimation (LSE). The loss function may be deduced from the general NLR loss function in summation form.

\[
h(b) = \frac{1}{2} \sum_{i=0}^{m-1} [y_i - f(x_i; b)]^2
\]

Replacing the general mean response function \( f(x_i; b) \) with \( b_0 e^{b_1 x_i} \) yields,

\[
h(b) = \frac{1}{2} \sum_{i=0}^{m-1} [y_i - b_0 e^{b_1 x_i}]^2
\]  

(10.10)

When \( \epsilon \sim N(0, \sigma^2 I) \), i.e., each error has mean 0 and variance \( \sigma^2 \), the same parameter estimates can be obtained using the Maximum Likelihood Estimation (MLE), see Chapters 9 and 10. See the Exercises, to show the equivalence of the two estimation techniques.

10.2.3 Optimization

Optimization for Simple Exponential Regression involves determining the gradient of the loss function with respect to the parameters \( b \). Starting with the general formula for the \( j^{th} \) partial derivative of the loss function \( h(b) \)

\[
\frac{\partial h(b)}{\partial b_j} = -\sum_{i=0}^{m-1} [e_i] \frac{\partial f(x_i; b)}{\partial b_j}
\]

the partial derivatives of the mean response function are needed. In this case, there are two parameters \( b = [b_0, b_1] \) and two formulas:
\[
\frac{\partial h(b)}{\partial b_0} = -\sum_{i=0}^{m-1} [\epsilon_i] e^{b_1 x_i}
\]
(10.11)
\[
\frac{\partial h(b)}{\partial b_1} = -\sum_{i=0}^{m-1} [\epsilon_i] b_0 x_i e^{b_1 x_i}
\]
(10.12)

Since the \(i^{th}\) predicted response \(\hat{y}_i = b_0 e^{b_1 x_i}\), these two equations may be simplified as follows:

\[
\frac{\partial h(b)}{\partial b_0} = -\sum_{i=0}^{m-1} [\epsilon_i] \hat{y}_i / b_0 = -\epsilon \cdot (\hat{y} / b_0)
\]
(10.13)
\[
\frac{\partial h(b)}{\partial b_1} = -\sum_{i=0}^{m-1} [\epsilon_i] x_i \hat{y}_i = -\epsilon \cdot (x \ast \hat{y})
\]
(10.14)

where \(*\) is the element-wise vector product. The gradient of the loss function is the vector formed from the two partial derivatives given above.

\[
\nabla h(b) = \left[ \frac{\partial h(b)}{\partial b_0}, \frac{\partial h(b)}{\partial b_1} \right]
\]
(10.15)
\[
\nabla h(b) = -[\epsilon \cdot (\hat{y} / b_0), \epsilon \cdot (x \ast \hat{y})]
\]
(10.16)

A Gradient Descent Optimizer is a simple type of first-order optimization algorithm. It starts at a random (or guessed) point in parameter space and iteratively moves in the direction opposite to the gradient. To control how rapidly the algorithm moves in that direction, a learning rate \(\eta\) is introduced as a hyper-parameter to tune. With each iteration the parameter \(b\) is updated as follows:

\[
b = b - \eta \nabla h(b)
\]

The pseudocode for the Gradient Descent Algorithm for Simple Exponential Regression is as follows:

\[
\begin{align*}
b & = b_{\text{init}} \quad // \text{initial random/guessed parameter vector} \\
do & \quad // \text{initial random/guessed parameter vector} \\
\text{yp} & = f(x; b) \quad // \text{y-predicted vector} \\
\text{e} & = y - \text{yp} \quad // \text{error vector} \\
d0 & = \text{e dot yp} / b0 \quad // \text{delta 0 - partial of h w.r.t. b0} \\
d1 & = \text{e dot yp} \ast x \quad // \text{delta 1 - partial of h w.r.t. b1} \\
b0 & += \text{eta} \ast d0 \quad // \text{update to first parameter} \\
b1 & += \text{eta} \ast d1 \quad // \text{update to second parameter} \\
\text{while} \quad \text{loss still decreasing} & \quad // \text{stopping rule}
\end{align*}
\]

### 10.2.4 Linearization

In this subsection, we examine two versions of simple exponential regression to see if the model can be linearized, by transforming the model so that it becomes linear in the parameters. The mean response function for the simple exponential regression model is

\[
f(x, b) = b_0 e^{b_1 x}
\]
Multiplicative Errors

Consider this mean response function with multiplicative errors,

\[ y = b_0 e^{b_1 x} \epsilon \]

Taking the natural logarithm yields

\[ \ln(y) = \ln(b_0) + b_1 x + \ln(\epsilon) \]

This linearized form can be used to fit the transformed response as was done with TranRegression.

\[ \ln(y) = \beta_0 + b_1 x + e \]

Multiplicative Errors

Now consider this mean response function with additive errors,

\[ y = b_0 e^{b_1 x} + \epsilon \]

Taking the natural logarithm yields

\[ \ln(y) = \ln(b_0 e^{b_1 x} + \epsilon) \]

Notice that the error model is different (multiplicative vs. additive) so the transformed linear regression model is different from the original nonlinear exponential regression model. They may be viewed as competitors, see the exercises.

Some nonlinear models are intrinsically linear in that the data can be transformed into a linear form, although the transformation may transform the errors inappropriately. The following nonlinear model, however, cannot be transformed to a linear form, regardless of the error model. It adds an additional parameter to the simple exponential model.

\[ y = b_0 + b_1 e^{b_2 x} + \epsilon \] (10.17)

10.2.5 Exercises

1. Consider the following hospitalization dataset from Chapter 13 of [59].

```scala
val xy = new MatrixD((15, 2), 2, 54,
5, 50,
7, 45,
10, 37,
14, 35,
19, 25,
26, 20,
31, 16,
34, 18,
38, 13)
```
Plot \( y = xy\. \text{col}(1) \) patient prognostic index versus \( x = xy\. \text{col}(0) \) days hospitalized.

2. Model the hospitalization dataset using Simple Linear Regression, plot the regression line and show the Quality of Fit (QoF) measures.

3. Model the hospitalization dataset using Simple Exponential Regression, plot the regression curve and show the Quality of Fit (QoF) measures. How do the QoF measures compare with those produced by Simple Linear Regression?

4. Model the hospitalization dataset using Log Transformed Linear Regression, plot the regression curve and show the Quality of Fit (QoF) measures. How do the QoF measures compare with those produced by Simple Linear Regression and Simple Exponential Regression?

5. For the following exponential regression model

\[
  y = b_0 e^{b_1 x} + \epsilon
\]

show that when error vector \( \epsilon \sim N(0, \sigma^2 I) \), the LSE and MLE estimation techniques will produce the same parameter estimates.

6. Use the pseudo-code given in this section to write a Gradient Descent Optimizer for Simple Exponential Regression using ScalaTion.

7. Pass the loss function into a second-order Quasi-Newton Optimizer and compare to the first-order algorithm in term of the parameter solution, the value of the loss function, and the number of steps needed for convergence.
10.3 Exponential Regression

The simple exponential regression model can be extended to have multiple predictor variables, e.g., \( x = [x_1, x_2] \) and \( \mathbf{b} = [b_0, b_1, b_2] \).

\[
y = b_0 e^{b_1 - 2 \cdot x} + \epsilon
\]

(10.18)

Note, \( b_{1-2} = [b_1, b_2] \). The mean response function then is

\[
f(x, \mathbf{b}) = b_0 e^{b_1 - 2 \cdot x}
\]

By introducing a new variable, \( x_0 = 1 \), the equation may be rewritten.

\[
f(x, \mathbf{b}) = e^{\mathbf{b} \cdot x}
\]

(10.19)

The \( \ell_2 \) loss function for Exponential Regression is

\[
h(\mathbf{b}) = \frac{1}{2} \sum_{i=0}^{m-1} [y_i - e^{\mathbf{b} \cdot x_i}]^2
\]

Again, starting with the general formula for the \( j^{th} \) partial derivative of the loss function \( h(\mathbf{b}) \)

\[
\frac{\partial h(\mathbf{b})}{\partial b_j} = -\sum_{i=0}^{m-1} \epsilon_i \frac{\partial f(x_i; \mathbf{b})}{\partial b_j}
\]

the problem reduces to finding the \( j^{th} \) partial derivative of the mean response function.

\[
\frac{\partial f(x_i, \mathbf{b})}{\partial b_j} = x_{ij} e^{\mathbf{b} \cdot x_i}
\]

Substituting this result into the general formula gives

\[
\frac{\partial h(\mathbf{b})}{\partial b_j} = \sum_{i=0}^{m-1} \epsilon_i x_{ij} e^{\mathbf{b} \cdot x_i}
\]

(10.20)

This mean response function can be linearized using a log transformation. Alternatively, one can also look into using a Generalized Linear Model (GLM) for such datasets to provide more options for dealing with the error distribution. See http://www.stat.uni-muenchen.de/~leiten/Lehre/Material/GLM_0708/chapterGLM.pdf for more details. Also, see the exercises.

ExpRegression Class

Class Methods:

@param x the data/input matrix
@param y the response vector
@param nonneg whether to check that responses are nonnegative

class ExpRegression (x: MatriD, y: VectoD, nonneg: Boolean) extends PredictorMat (x, y)
def ll (b: VectoD): Double =
def ll_null (b: VectoD): Double =
def train (yy: VectoD = y): ExpRegression =
def train_null ()
def crossVal (k: Int = 10, rando: Boolean = true): Array [Statistic]

10.3.1 Exercises

1. A Generalized Linear Model (GLM) that utilized Maximum Likelihood Estimation (MLE) may be used for such data. The response variable \( y \) is modeled as the product of a mean response function \( f(x; b) = \mu_y(x) \) and exponentially distributed residuals/errors \( \epsilon \).

\[
y = \mu_y(x) \times \epsilon \\
\ln(\mu_y(x)) = b \cdot x
\]

where the link function \( g = \ln \) with inverse \( g^{-1} = \exp \).

\[
g(\mu_y(x)) = \ln(\mu_y(x)) = b \cdot x
\]

Explain why the errors/residuals \( \epsilon_i = y_i / \mu_y(x_i) \) follow and Exponential distribution.

\[
f(y_i / \mu_y(x_i)) = \frac{1}{\mu_y(x_i)} e^{-y_i / \mu_y(x_i)}
\]

2. Show that the likelihood function for Exponential Regression is the following:

\[
L = \prod_{i=0}^{m-1} \frac{1}{\mu_y(x_i)} e^{-y_i / \mu_y(x_i)}
\]

Substituting for \( \mu_y(x_i) = e^{b \cdot x_i} \) gives

\[
L = \prod_{i=0}^{m-1} e^{-b \cdot x_i} e^{-y_i / e^{b \cdot x_i}}
\]

Taking the natural logarithm gives the log-likelihood function.

\[
LL = \sum_{i=0}^{m-1} -b \cdot x_i - \frac{y_i}{e^{b \cdot x_i}}
\]

3. Write a function to compute the negative log-likelihood \((-LL)\) and pass it into a nonlinear optimization algorithm for minimization.
10.4 Perceptron

The Perceptron class supports single-valued 2-layer (input and output) Neural Networks. The inputs into a Neural Net are given by the input vector $x$, while the outputs are given by the output value $y$. As depicted in Figure 10.2 each component of the input $x_j$ is associated with an input node in the network, while the output $y$ is associated with the single output node. The input layer consists of $n$ input nodes, while the output layer consists of 1 output node.

An edge connects each input node with the output node, i.e., there are $n$ edges in the network. To include an intercept in the model (also referred to as bias) one of the inputs (say $x_0$) must always be set to 1. Alternatively, a bias offset $\beta$ can be associated with the output node and added to the weighted sum (see below).

10.4.1 Model Equation

The weights on the edges are analogous to the parameter vector $b$ in regression. The output $y$ has an associated parameter vector $b$, where parameter value $b_j$ is the edge weight connecting input node $x_j$ with output node $y$.

Recall the model equation for multiple linear regression.

$$ y = b \cdot x + \epsilon = b_0 + b_1 x_1 + ... + b_{n-1} x_{n-1} + \epsilon $$

We now take the linear combination of the inputs, $b \cdot x$, and apply an activation function $f_a$ (or simply $f$).

$$ y = f(b \cdot x) + \epsilon = f\left(\sum_{j=0}^{n-1} b_j x_j\right) + \epsilon $$

(10.21)

The general mean response function $f_r(x; b)$ of nonlinear regression, which takes two vectors as input, is now replaced by first taking a linear combination of $x$ and then applying a simpler activation function that takes a single scalar as input, i.e., $f(b \cdot x)$.
10.4.2 Training

Given several input vectors and output values (e.g., in a training set), optimize/fit the weights $b$ connecting the layers. After training, given an input vector $x$, the net can be used to predict the corresponding output value $\hat{y}$.

![Training Diagram for Perceptron](image)

Figure 10.3: Training Diagram for Perceptron

A training dataset consisting of $m$ input-output pairs is used to minimize the error in the prediction by adjusting the parameter/weight vector $b \in \mathbb{R}^n$. Given an input/data matrix $X \in \mathbb{R}^{m \times n}$ consisting of $m$ input vectors and an output vector $y \in \mathbb{R}^m$ consisting of $m$ output values, minimize the distance between the actual/target output vector $y$ and the predicted output vector $\hat{y}$,

$$\hat{y} = f(Xb)$$ (10.22)

where $f : \mathbb{R}^m \to \mathbb{R}^m$ is the vectorized version of the activation function $f$. The vectorization may occur over the entire training set or more likely, an iterative algorithm may work with a group/batch of instances at a time. In other words, the goal is to minimize some norm of the error vector.

$$\epsilon = y - \hat{y} = y - f(Xb)$$ (10.23)

Using the Euclidean ($\ell_2$) norm, we have

$$\min_b \|y - f(Xb)\|$$

As was the case with regression, it is convenient to minimize the dot product of the error with itself ($\|\epsilon\|^2 = \epsilon \cdot \epsilon$). In particular, we aim to minimize half of this value, half sse (loss function $h$).

$$h(b) = \frac{1}{2} [y - f(Xb)] \cdot [y - f(Xb)]$$ (10.24)

Using summation notation gives

306
\[ h(b) = \frac{1}{2} \sum_{i=0}^{m-1} [y_i - f(x_i \cdot b)]^2 \]  

(10.25)

### 10.4.3 Optimization

Optimization for Perceptrons and Neural Networks is typically done using an iterative optimization algorithm that utilizes gradients. Popular optimizers include Gradient Descent (GD), Stochastic Gradient Descent (SGD), Stochastic Gradient Descent with Momentum (SGDM), Root Mean Square Propagation (RMSProp) and Adaptive Moment Estimation (Adam) (see Chapter on Optimization for details).

The gradient of the loss function \( h \) is calculated by computing all of the partial derivatives with respect to the parameters/weights.

\[ \nabla h(b) = \left[ \frac{\partial h}{\partial b_0}, \ldots, \frac{\partial h}{\partial b_{n-1}} \right] \]

**Partial Derivative for \( b_j \)**

Starting with the nonlinear regression general formula for the \( j^{th} \) partial derivative of the loss function \( h(b) \)

\[ \frac{\partial h(b)}{\partial b_j} = -\sum_{i=0}^{m-1} [\epsilon_i] \frac{\partial f_r(x_i; b)}{\partial b_j} \]

and specializing the mean response function \( f_r(x_i; b) \) to \( f(x_i \cdot b) \), i.e., the function of two vectors is replaced with an activation function applied to the dot product of the two vectors, yields,

\[ \frac{\partial h(b)}{\partial b_j} = -\sum_{i=0}^{m-1} [\epsilon_i] \frac{\partial f(x_i \cdot b)}{\partial b_j} \]

Now, the partial derivative of \( f(x_i \cdot b) \) with respect to \( b_j \) can be determined via the *chain rule* using the scalar pre-activation response \( u_i = x_i \cdot b \).

\[ \frac{\partial f(x_i \cdot b)}{\partial b_j} = \frac{\partial f(u_i)}{\partial u_i} \frac{\partial u_i}{\partial b_j} \]  

(10.26)

**Derivative of the Activation Function**

Since \( \frac{\partial f(u_i)}{\partial u_i} \) is a regular derivative as it is one dimensional, it can be replaced with \( \frac{df(u_i)}{du_i} \) which may be denoted as \( f'(u_i) \).

\[ \frac{\partial f(u_i)}{\partial u_i} = \frac{df(u_i)}{du_i} = f'(u_i) \]  

(10.27)

**Derivative of the Pre-Activation Response**

\[ \frac{\partial u_i}{\partial b_j} = \frac{\partial}{\partial b_j} (b_0x_{i0} + \cdots + b_jx_{ij} + \cdots + b_kx_{ik}) = x_{ij} \]  

(10.28)
Combining Results

\[
\frac{\partial f(x_i \cdot b)}{\partial b_j} = f'(u_i) x_{ij}
\]

Therefore, the \(j^{th}\) partial derivative of the loss function \(\mathcal{h}\) becomes

\[
\frac{\partial \mathcal{h}(b)}{\partial b_j} = - \sum_{i=0}^{m-1} [\epsilon_i] x_{ij} f'(u_i)
\] (10.29)

The sum can be replaced with a dot product (sum of products),

\[
\frac{\partial \mathcal{h}(b)}{\partial b_j} = -x_{.j} \cdot (\epsilon \ast f'(u))
\] (10.30)

where \(x_{.j}\) is the \(j^{th}\) column of data matrix \(X\), \(\ast\) is the element-wise vector product, \(u = Xb\), and \(f'(u)\) is the vector extension of the derivative of the activation function \(\frac{df(u_i)}{du_i} = f'(u_i)\) over all \(m\) instances.

Alternative Derivation at the Vector Level

The same basic derivation can be carried out at the vector level as well. Taking the partial derivative with respect to the \(j^{th}\) parameter/weight, \(b_j\), is a bit complicated since we need to use the chain rule and the product rule. First, letting \(u = Xb\) (the pre-activation response) allows the loss function to be simplified to

\[
\mathcal{h}(b) = \frac{1}{2} [y - f(u)] \cdot [y - f(u)]
\] (10.31)

The chain rule from vector calculus to be applied is

\[
\frac{\partial \mathcal{h}(b)}{\partial b_j} = \frac{\partial \mathcal{h}(b)}{\partial u} \cdot \frac{\partial u}{\partial b_j}
\] (10.32)

The first partial derivative is

\[
\frac{\partial \mathcal{h}(b)}{\partial u} = - [y - f(u)] \ast f'(u)
\] (10.33)

where the first part of the r.h.s. is \(f'(u)\) which is the derivative of \(f\) with respect to vector \(u\) and the second part is the difference between the actual and predicted output/response vectors. The two vectors are multiplied together, elementwise.

The second partial derivative is

\[
\frac{\partial u}{\partial b_j} = x_{.j}
\] (10.34)

where \(x_{.j}\) is the \(j^{th}\) column of matrix \(X\) (see Exercises 4, 5 and 6 for details).

The dot product of the two partial derivatives gives

\[
\frac{\partial \mathcal{h}(b)}{\partial b_j} = -x_{.j} \cdot [y - f(u)] f'(u)
\]

Since the error vector \(\epsilon = y - f(u)\), we may simplify the expression.
The $j^{th}$ partial derivative (or $j^{th}$ element of the gradient) indicates the relative amount to move (change $b_j$) in the $j^{th}$ dimension to reduce $h$. Notice that equations 11.30 and 11.35 are the same.

\[
\frac{\partial h(b)}{\partial b_j} = -x_j \cdot (\epsilon * f'(u)) \tag{10.35}
\]

The $\delta$ Vector

The goal of training is to minimize the errors. The errors are calculated using forward propagation through the network. The parameters (weights and bias) are updated by back-propagation of the errors, or more precisely, the slope-adjusted errors as illustrated in Figure 10.4.

Therefore, it is helpful especially for multi-layer neural networks to define the delta vector $\delta$ as follows:

\[
\delta = \frac{\partial h(b)}{\partial u} = -\epsilon * f'(u) \tag{10.36}
\]

It multiplies the derivative of the vectorized activation function $f$ by the negative error vector, elementwise. If the error is small or the derivative is small, the adjustment to the parameter should be small. The partial derivative of $h$ with respect to $b_j$ now simplifies to

\[
\frac{\partial h(b)}{\partial b_j} = x_j \cdot \delta \tag{10.37}
\]

### 10.4.4 Example Calculation for $\epsilon$ and $\delta$

Assume that the activation function is the sigmoid function. Starting with the parameter/weight vector

\[
b = [.1, .2, .1],
\]

compute the $m$-dimensional vectors, $\epsilon$ and $\delta$, for Exercise 7. With these parameters, the predicted output/response vector $\hat{y}$ may be computed in two steps: The first step computes the response, pre-activation
The second step takes this vector and applies the activation function to each of its elements. This requires looking ahead to the subsection on \textit{activation functions}. The sigmoid function (abbreviated \texttt{sigm}) is defined as follows:

\[
\text{sigm}(u) = \frac{1}{1 + e^{-u}}
\]  

Figure 10.5 show the response surface for the example problem where the response \( y \) is color-coded.

![Figure 10.5: Example Perceptron Problem: response value black (.2), blue (.3), green (.5), purple/crimson (.3), red (1)](image)

**Pre-activation Vector \( \mathbf{u} \)**

From Figure 10.5 create the input/data matrix \( \mathbf{X} \in \mathbb{R}^{9 \times 3} \) and multiply it by the current parameter vector \( \mathbf{b} \in \mathbb{R}^{3} \). The pre-activation vector is the aggregated signal before activation.

\[
\mathbf{u} = \mathbf{Xb} = [1.15, 2.2, 2.25, 3, 3, 35, 4]
\]

**Predicted Response Vector \( \hat{\mathbf{y}} \)**

The predicted response vector is determined by applying the activation function to each element of the pre-activation vector.

\[
\hat{\mathbf{y}} = \text{sigm} (\mathbf{u}) = \text{sigm} ([1.15, 2.2, 2.25, 3, 3, 35, 4])
\]

\[
= [0.5249, 0.5374, 0.5498, 0.5498, 0.5621, 0.5744, 0.5744, 0.5866, 0.5986]
\]
**Error Vector $\epsilon$**

The error vector $\epsilon$ is simply the difference between the actual and predicted output/response vectors.

$$\epsilon = y - \hat{y}$$

$$[.5000, .3000, .2000, .8000, .5000, .3000, 1.0000, .8000, .5000] -$$

$$[.5249, .5374, .5498, .5498, .5621, .5744, .5744, .5866, .5986] =$$

$$[-.0249, -.2374, -.3498, .2501, -.0621, -.2744, .4255, .2133, -.0986]$$

**Delta Vector $\delta$**

To compute the delta vector $\delta$, we must look ahead to get the derivative of the activation function (see exercise 8).

$$\text{sigm}'(u) = \text{sigm}(u) [1 - \text{sigm}(u)]$$

Therefore, since $\text{sigm}(u) = \hat{y}$

$$\delta = -\epsilon * [\hat{y} * (1 - \hat{y})]$$

$$[.0062, .0590, .0865, -.0619, .0153, .0670, -.1040, -.0517, .0237]$$

**Forming the Gradient**

Combining the partial derivatives $\frac{\partial h}{\partial b_j}$ for the loss function into an $n$-dimensional vector (i.e., the gradient) yields

$$\nabla h(b) = \frac{\partial h(b)}{\partial b} = -X^T [\epsilon * f'(Xb)] = X^T \delta$$  \hspace{1cm} (10.39)

**Parameter Update**

Since many optimizers such as gradient-descent, move in the direction opposite to the gradient by a distance governed by the learning rate $\eta$ (alternatively step size), the following term should be subtracted from the weight/parameter vector $b$.

$$\nabla h(b) \eta = X^T \delta \eta$$  \hspace{1cm} (10.40)

The right hand side is an $n$-by-$m$ matrix, $m$ vector product yielding an $n$ vector result. Since gradient-based optimizers move in the negative gradient direction by an amount determined by the magnitude of the gradient times a learning rate $\eta$, the parameter/weight vector $b$ is updated as follows:

$$b = b - X^T \delta \eta$$  \hspace{1cm} (10.41)
10.4.5 Initializing Weights/Parameters

The weight/parameter vector $b$ should be randomly set to start the optimization.

Set the initial weight/parameter vector $b$ with values in $(0, limit)$ before training.

```scala
@param stream the random number stream to use
@param limit the maximum value for any weight
def setWeights (stream: Int = 0, limit: Double = 1.0 / sqrt (x.dim2))
{
  val rvg = new RandomVecD (n, limit, 0.0, stream = stream) // may change stream
  b = rvg.gen
} // setWeights
```

For testing or learning purposes, the weights may also be set manually.

```scala
def setWeights (w0: VectoD) { b = w0 }
```

10.4.6 Activation Functions

An activation function $f_a$ (or simple $f$) takes an aggregated signal and transforms it. The simplest activation function is the id or identity function where the aggregated signal is passed through unmodified. In this case, Perceptron is in alignment with Regression (see Exercise 9). This activation function is usually not intended for neural nets with more layers, since theoretically they can be reduced to a two-layer network (although it may be applied in the last layer). More generally useful activation functions include reLU, lreLU, eLU, seLU, sigm, tanh and gaussian. Several activation functions are compared in [63, 71]. For these activation functions the outputs in the $y$ vector need to be transformed into the range specified for the activation function, see Table 10.1. It may be also useful to transform/standardize the inputs.

<table>
<thead>
<tr>
<th>Name</th>
<th>Function $y = f(u)$</th>
<th>Dom</th>
<th>Range</th>
<th>Derivative $f'(u)$</th>
<th>Inverse $u = f^{-1}(y)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>$u$</td>
<td>$\mathbb{R}$</td>
<td>$\mathbb{R}$</td>
<td>1</td>
<td>$y$</td>
</tr>
<tr>
<td>reLU</td>
<td>$\max(0, u)$</td>
<td>$\mathbb{R}$</td>
<td>$\mathbb{R}^+$</td>
<td>$1_{u&gt;0}$</td>
<td>$y$ for $y &gt; 0$</td>
</tr>
<tr>
<td>lreLU</td>
<td>$\max(\alpha u, u)$, $\alpha &lt; 1$</td>
<td>$\mathbb{R}$</td>
<td>$\mathbb{R}$</td>
<td>$1_{u&lt;0}(\alpha, 1)$</td>
<td>$\min(\frac{y}{\alpha}, y)$</td>
</tr>
<tr>
<td>eLU</td>
<td>$\max(0, \alpha(e^u - 1), u)$</td>
<td>$\mathbb{R}$</td>
<td>$\mathbb{R}$</td>
<td>$1_{u&lt;0}(f(u) + \alpha, 1)$</td>
<td>$\text{if } y &lt; 0 (\ln(\frac{y}{\alpha} + 1), y)$</td>
</tr>
<tr>
<td>seLU</td>
<td>$\lambda[1_{u&lt;0}(\alpha(e^u - 1), u)]$</td>
<td>$\mathbb{R}$</td>
<td>$\mathbb{R}$</td>
<td>$\lambda[1_{u&lt;0}(f(u) + \alpha, 1)]$</td>
<td>$\frac{1}{\lambda}[1_{y&lt;0}(\ln(\frac{y}{\alpha} + 1), y)]$</td>
</tr>
<tr>
<td>sigm</td>
<td>$[1 + e^{-u}]^{-1}$</td>
<td>$\mathbb{R}$</td>
<td>$(0,1)$</td>
<td>$f(u)[1 - f(u)]$</td>
<td>$-\ln(\frac{1}{y} - 1)$</td>
</tr>
<tr>
<td>tanh</td>
<td>$\tanh(u)$</td>
<td>$\mathbb{R}$</td>
<td>$(-1,1)$</td>
<td>$1 - f(u)^2$</td>
<td>$.5 \ln(\frac{1 + y}{1 - y})$</td>
</tr>
<tr>
<td>gaussian</td>
<td>$e^{-u^2}$</td>
<td>$\mathbb{R}$</td>
<td>$(0,1)$</td>
<td>$-2ue^{-u^2}$</td>
<td>$\sqrt{-\ln(y)}$</td>
</tr>
</tbody>
</table>

The curves of three of the activation functions are shown in Figure 10.6.
The *sigmoid* function has an ‘S’ shape, which facilitates its use as a smooth and differentiable version of a step function, with larger negative values tending to zero and larger positive values tending to one. In the case of using sigmoid for the activation function, $f'(u) = f(u)[1 - f(u)]$, so

$$
\delta = -\epsilon \ast [f'(u)] = -\epsilon \ast [f(u) \ast [1 - f(u)]
$$

Gradient-descent algorithms iteratively move in the negative gradient direction by an amount determined by the magnitude of the gradient times a learning rate $\eta$, so the parameter/weight vector $\mathbf{b}$ is adjusted as follows:

$$
\mathbf{b} = \mathbf{b} - \mathbf{X}^\top \delta \eta
$$

**Continuation of Example Calculation**

Assuming the learning rate $\eta = 1$ and taking the $\delta$ vector from the example, the update to parameter/weight vector $\mathbf{b}$ is

$$
\mathbf{X}^\top \delta \eta = [.0402, -.1218, .1886]
$$

Consequently, the updated value for the parameter/weight vector $\mathbf{b}$ is

$$
\mathbf{b} = [.1, 2, .1] - [.0402, -.1218, .1886] = [.0597, .3218, -.0886]
$$

Check to see if the new values for $\mathbf{b}$ have improved/decreased the loss function $h$. 

313
10.4.7 Basic Gradient Descent Algorithm

A basic gradient descent algorithm is an iterative process that typically terminates when the drop in the loss function $h$ is small or a maximum number of iterations is exceeded. The parameters $\eta$ and $\text{max\_epochs}$ need careful adjustments to obtain nearly (locally) optimal values for $h$. Gradient-descent works by iteratively moving in the opposite direction as the gradient until a stopping rule evaluates to true (e.g., stop after $q^{th}$ increase in $h$ and return best solution so far). The rate of convergence can be adjusted using the learning rate $\eta$ which can modify the trajectory of the gradient. Setting it too low slows convergence, while setting it too high can cause oscillation or divergence. In SCALATION, the learning rate $\eta$ (eta in the code) is a hyper-parameter that defaults to 0.1, but is easily adjusted, e.g.,

```
Optimizer.hp ("eta") = 0.05
```

The `train0` method contains the main training loop that is shown below. Inside the loop, new values $y_p$ are predicted, from which an error vector $e$ is determined. This is used to calculate the delta vector $d$, which along $x.t$ and $\eta$ are used to update the parameter/weight vector $b$. Note, $x_-$, $y_-$ which default to $x, y$ constitute the training set.

```
def train0 (x_ : MatrixD = x, y_ : VectorD = y): Perceptron =
{
    if (b == null) b = weightVec (n) // initialize parameters/weights
    var sse0 = Double.MaxValue // hold prior value of sse
    for (epoch <- 1 until maxEpochs) {
        val yp = f.fV (x_ * b) // predicted output vector yp = f(Xb)
        e = y_ - yp // error vector for y
        val d = -e * f.dV (yp) // delta vector for y
        b -= x_.t * d * eta // update the parameters/weights
        val sse = sseF (y, f.fV (x_ * b)) // recompute sum of squared errors
        if (DEBUG) println (s"train0: parameters for $epoch th phase: b = $b, sse = $sse")
        if (sse >= sse0) return this // return when sse increases
        sse0 = sse // save prior sse
    } // for
    this
} // train0
```

The vector function $f.fV$ is the vectorization of the activation function $f.f$, and is created in SCALATION using the `vectorize` high-order function, e.g., given a scalar function $f$, it can produce the corresponding vector function $f$.

```
def vectorize (f: FunctionS2S): FunctionV_2V = (x: VectorD) => x.map (f(_))
val fV = vectorize (f)
```

The function $f.dV$ is the derivative of the vector activation function. The core of the algorithm is the first four lines in the loop. Table 10.2 shows the correspondence between these lines of code and the main/boxed equations derived in this section. Note, all the equations in the table are vector assignments.

The third line of code appears to be different from the mathematical equation, in terms of passing the pre-activation versus the post-activation response. It turns out that all derivatives for the activation functions
Table 10.2: Correspondence between Code and Boxed Equations

<table>
<thead>
<tr>
<th>Code</th>
<th>Equation</th>
<th>Equation Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>yp = f.fV (x_ * b)</td>
<td>ŷ = f(Xb)</td>
<td>11.22</td>
</tr>
<tr>
<td>e = y_ - yp</td>
<td>ϵ = y - ŷ</td>
<td>11.23</td>
</tr>
<tr>
<td>d = -e * f.dV (yp)</td>
<td>δ = − ϵ * f’(Xb)</td>
<td>11.36</td>
</tr>
<tr>
<td>b -= x_.t * d * eta</td>
<td>b = b − X’δη</td>
<td>11.41</td>
</tr>
</tbody>
</table>

(except Gaussian) are either formulas involving constants or simple functions of the activation function itself, so for efficiency, the yp vector is passed in.

A perceptron can be considered to be a special type of nonlinear or transformed regression, see Exercise 10.

The ActivationFun Object

The Perceptron class defaults to the f.sigmoid Activation Function Family (AFF), which is defined in the ActivationFun object.

val f_sigmoid = AFF (sigmoid, sigmoidV, sigmoidM, sigmoidDV, sigmoidDM, (0, 1))

In general, the AFF for family f contains the following:

val f = AFF (f, fV, fM, dV, dM, (lb, ub))

The first three are the activation function in scalar, vector and matrix forms, respectively, the next two are the derivative of the activation function in vector and matrix forms, respectively, and the last is a tuple giving the lower and upper bounds for the range of the activation function.

Rescaling: Depending on the activation function, rescaling of outputs and/or inputs may be necessary: (1) If the actual response vector y is outside the bounds/range of the activation function, it will be impossible for the predicted response vector ŷ to approximate it, so rescaling will be necessary. (2) If the linear combination of inputs takes the pre-activation value beyond the effective domain of the activation function, training will become slow, e.g., when u is large, f(u) will be nearly 1 for sigm, even when there are substantial changes to u. induced by updates to the parameters.

Other activation functions should be experimented with, as one may produce better results. All the activation functions shown in Table 10.1 are available in the ActivationFun object.

Essentially, parameter optimization in perceptrons involves using/calculating several vectors as summarized in Table 10.3 where n is the number of parameters and m is the number of instances used at a particular point in the iterative optimization algorithm, for example, corresponding to the total number of instances in a training set for Gradient Descent (GD).

Stochastic Gradient Descent (SGD) utilizes a fraction of the m instances to make updates to the parameters, corresponding to just 1 for pure SGD or to the size of a mini-batch (e.g., 30) for the common form of SGD. This has the obvious benefit of reducing the amount of computation required to make parameter updates. Somewhat surprisingly, it also makes it easier for the algorithm to escape local mimima and to generally be a more robust algorithm. The mini-batch size is another hyper-parameter that can be tuned.
Table 10.3: Vectors Used in Perceptrons

<table>
<thead>
<tr>
<th>Vector</th>
<th>Space</th>
<th>Formula</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_i$</td>
<td>$\mathbb{R}^n$</td>
<td>given</td>
<td>the $i^{th}$ row of the input/data matrix</td>
</tr>
<tr>
<td>$x_j$</td>
<td>$\mathbb{R}^m$</td>
<td>given</td>
<td>the $j^{th}$ column of the input/data matrix</td>
</tr>
<tr>
<td>$b$</td>
<td>$\mathbb{R}^n$</td>
<td>given</td>
<td>the parameter vector (updated per iteration)</td>
</tr>
<tr>
<td>$u$</td>
<td>$\mathbb{R}^m$</td>
<td>$Xb$</td>
<td>the pre-activation vector</td>
</tr>
<tr>
<td>$y$</td>
<td>$\mathbb{R}^m$</td>
<td>given</td>
<td>the actual output/response vector</td>
</tr>
<tr>
<td>$\hat{y}$</td>
<td>$\mathbb{R}^m$</td>
<td>$f(u)$</td>
<td>the predicted output/response vector</td>
</tr>
<tr>
<td>$\epsilon$</td>
<td>$\mathbb{R}^m$</td>
<td>$y - \hat{y}$</td>
<td>the error/residual vector</td>
</tr>
<tr>
<td>$\delta$</td>
<td>$\mathbb{R}^m$</td>
<td>$-\epsilon \ast f'(u)$</td>
<td>the negative-slope-weighted error vector</td>
</tr>
<tr>
<td>$\beta \oplus w$</td>
<td>$\mathbb{R}^n$</td>
<td>$b$</td>
<td>concatenation of bias scalar and weight vector</td>
</tr>
</tbody>
</table>

Perceptron Class

Class Methods:

```scala
class Perceptron (x: MatriD, y: VectoD, 
  fname_ : Strings = null, hparam: HyperParameter = Optimizer.hp, 
  f: AFF = f_sigmoid, val itran: FunctionV_2V = null) 
  extends PredictorMat (x, y, fname_, hparam) 

def setWeights (w0: VectoD) 
def reset (eta_ : Double) { eta = eta_ } 
def train0 (x_: MatriD = x, y_: VectoD = y): Perceptron = 
def train (x_: MatriD = x, y_: VectoD = y): Perceptron = 
override def train2 (x_: MatriD = x, y_: VectoD = y): Perceptron = 
def trainSwitch (which: Int, x_: MatriD = x, y_: VectoD = y): Perceptron = 
override def predict (z: VectoD): Double = f.f (b dot z) 
override def predict (z: MatriD = x): VectoD = f.fV (z * b) 
def buildModel (x_cols: MatriD): Perceptron =
```

The `train0` method uses Gradient Descent with a simple stopping rule and a non-adaptive learning rate. A better optimizer is used by the `train` method which uses Stochastic Gradient Descent, a better stopping rule (see `StoppingRule` class), and an adaptive learning rate. The work is delegated to the
Optimizer object and can easily be changed to use Stochastic Gradient Descent with Momentum using the Optimizer_SGDM object. The train2 method simply calls the train method with various learning rates over a given interval to find the best one. The trainSwitch method makes it easy to switch between these three methods by setting the which parameter to 0, 1 or 2.

10.4.8 Exercises

1. Plot the rest of activation functions in Table 10.1 in the same plot and compare them.

2. The Texas Temperature regression problem can also be analyzed using a perceptron.

```scala
val x = new MatrixD ((16, 4), 1.0, 29.767, 41.0, 95.367, // Harris
                      1.0, 32.850, 440.0, 96.850, // Dallas
                      1.0, 26.933, 25.0, 97.800, // Kennedy
                      1.0, 31.950, 2851.0, 102.183, // Midland
                      1.0, 34.800, 3840.0, 102.467, // Deaf Smith
                      1.0, 33.450, 1461.0, 99.633, // Knox
                      1.0, 28.700, 815.0, 100.483, // Maverick
                      1.0, 32.450, 2380.0, 100.533, // Nolan
                      1.0, 31.800, 3918.0, 106.400, // El Paso
                      1.0, 34.850, 2040.0, 100.217, // Collington
                      1.0, 30.867, 3000.0, 102.900, // Pecos
                      1.0, 36.350, 3693.0, 102.083, // Sherman
                      1.0, 30.300, 597.0, 97.700, // Travis
                      1.0, 26.900, 315.0, 99.283, // Zapata
                      1.0, 28.450, 459.0, 99.217, // Lasalle
                      1.0, 25.900, 19.0, 97.433) // Cameron

val y = VectorD (56.0, 48.0, 60.0, 46.0, 38.0, 46.0, 53.0, 46.0,
                      44.0, 41.0, 47.0, 36.0, 52.0, 60.0, 56.0, 62.0)

val nn = new Perceptron (x, y)
nn.train ().eval ()
println (nn.report)

val z = VectorD (1.0, 30.0, 1000.0, 100.0)
println (s"predict ($z) = ${nn.predict (z)}")
```

3. Analyze the ExampleConcrete dataset, which has three output variables $y_0$, $y_1$ and $y_2$. Create a perceptron for each output variable.

4. Use the following formula for matrix-vector multiplication
\[ u = Xb = \sum_j b_j x_j \]
to derive the formula for the following partial derivative
\[ \frac{\partial u}{\partial b_j} = x_j \]

5. Given the formula for the loss function \( h : \mathbb{R}^m \rightarrow \mathbb{R} \) expressed in terms of the pre-activation vector \( u = Xb \) and the vectorized activation function \( f : \mathbb{R}^m \rightarrow \mathbb{R}^m \),

\[ h(u) = \frac{1}{2} (y - f(u)) \cdot (y - f(u)) \]
derive the formula for the gradient of \( h \) with respect to \( u \).

\[ \nabla h = \frac{\partial h}{\partial u} = -f'(u)(y - f(u)) \]

**Hint:** Take the gradient, \( \frac{\partial h}{\partial u} \), using the product rule \((d_1 \cdot f_2 + f_1 \cdot d_2)\).

\[ \frac{\partial h}{\partial u} = -\frac{\partial f(u)}{\partial u} \cdot (y - f(u)) \]
where \( f_1 = f_2 = y - f(u) \) and \( d_1 = d_2 = -\frac{\partial f(u)}{\partial u} \). Next, assuming \( \frac{\partial f(u)}{\partial u} \) is a diagonal matrix, show that the above equation can be rewritten as

\[ \frac{\partial h}{\partial u} = -f'(u)(y - f(u)) \]
where \( f'(u) = [f'(u_0), \ldots, f'(u_{m-1})] \) and the two vectors, \( f'(u) \) and \( y - f(u) \), are multiplied, element-wise.

6. Show that the \( m \)-by-\( m \) Jacobian matrix, \( J f(u) = \frac{\partial f(u)}{\partial u} \), is a diagonal matrix, i.e.,

\[ J f(u) = \begin{bmatrix} \frac{\partial f_i(u)}{\partial u_j} \end{bmatrix} = 0 \text{ if } i \neq j \]
where \( f_i = f \) the scalar activation function. Each diagonal element is the derivative of the activation function applied to the \( i^{th} \) input, \( f'(u_i) \). See the section on Vector Calculus in Chapter 2 that discusses Gradient Vectors, Jacobian Matrices and Hessian Matrices.

7. Show the first 10 iterations that update the parameter/weight matrix \( b \) that is initialized to \([1, 2, 1]\).
Use the following combined input-output matrix. Let the perceptron use the default sigmoid function.

```java
// 9 data points: Constant x1 x2 y
val xy = new MatrixD ((9, 4), 1.0, 0.0, 0.0, 0.5,
                          1.0, 0.0, 0.5, 0.3,
                          1.0, 0.0, 1.0, 0.2,
                          1.0, 0.0, 0.0, 0.5,
...```
val hp = Optimizer.hp.updateReturn ("eta", 2.0)  // try several values for eta
val nn = Perceptron (xy, null, hp)  // create a perceptron

For each iteration, do the following: Print the weight/parameter update vector $X^T \delta \eta$ and the new value for weight/parameter vector $b$. Make a table with $m$ rows showing values for $x_1, x_2, y, \hat{y}, \epsilon, \epsilon^2, \hat{y}(1 - \hat{y})$ and $\hat{y}(1 - \hat{y}) \epsilon \eta$

Try letting $\eta = 1$ then 2. Also, compute $sse$ and $R^2$.

8. Show that for sigmoid function (abbreviated sigm)

$$\text{sigm}(u) = [1 + e^{-u}]^{-1} = \frac{1}{1 + e^{-u}}$$

its derivative is

$$\text{sigm}'(u) = \frac{d\text{sigm}(u)}{du} = \text{sigm}(u)[1 - \text{sigm}(u)]$$

9. Show that when the activation function $f$ is the id function, that $f'(u)$ is the one vector, $1$. Plug this into the equation for the gradient of the loss function to obtain the following result.

$$\frac{\partial h}{\partial b} = -X^T[1 \epsilon] = -X^T(y - Xb)$$

Setting the gradient equal to zero, now yields $(X^T X)b = X^T y$, the Normal Equations.

10. Show that a Perceptron with an invertible activation function $f$ is similar to TranRegression with transform $f^{-1}$. Explain any differences in the parameter/weight vector $b$ and the sum of squared errors $sse$. Use the sigmoid activation function and the AutoMPG dataset and make the following two plots (using PlotM): $y$, $ypr$, $ypt$ vs. $t$ and $y$, $ypr$, $ypp$ vs. $t$, where $y$ is the actual response/output, $ypr$ is the prediction from Regression, $ypt$ is the prediction from TranRegression and $ypp$ is the prediction from Perceptron.
10.5 Multi-Output Prediction

The PredictorMat2 abstract class provides the basic structure and API for a variety of modeling techniques that produce multiple responses/outputs, e.g., Neural Networks, Extreme Learning Machines and Multi-Variate Regression. It serves the same role that PredictorMat has for the regression modeling techniques.

10.5.1 Model Equation

For modeling techniques extending this abstract class, the model equation takes an input vector \(x\), pre-multiplies it by the transpose of the parameter matrix \(B\), applies a function \(f\) to the resulting vector and adds an error vector \(\epsilon\),

\[
y = f(B \cdot x) + \epsilon = f(B^T x) + \epsilon \tag{10.42}
\]

where

- \(y\) is an \(n_y\)-dimensional output/response random vector,
- \(x\) is an \(n_x\)-dimensional input/data vector,
- \(B\) is an \(n_x\)-by-\(n_y\) parameter matrix,
- \(f: \mathbb{R}^{n_y} \to \mathbb{R}^{n_y}\) is a function mapping vectors to vectors, and
- \(\epsilon\) is an \(n_y\)-dimensional residual/error random vector.

For Multi-Variate Regression, \(f\) is the identity function.

10.5.2 Training

The training equation takes the model equation and several instances in a dataset to provide estimates for the values in parameter matrix \(B\). Compared to the single response/output variable case, the main difference is that the response/output vector, the parameter vector, and the error vector now all become matrices.

\[
Y = f(XB) + E \tag{10.43}
\]

where \(X\) is an \(m\)-by-\(n_x\) data/input matrix, \(Y\) is an \(m\)-by-\(n_y\) response/output matrix, \(B\) is an \(n_x\)-by-\(n_y\) parameter matrix, \(f\) is a function mapping one \(m\)-by-\(n_y\) matrix to another, and \(E\) is an \(m\)-by-\(n_y\) residual/error matrix. Note, a bold function symbol \(f\) is used is used to denote a function mapping either vectors to vectors (as was the case in the Model Equation subsection) or matrices to matrices (as is the case here).

\[
f: \mathbb{R}^{m \times n_y} \to \mathbb{R}^{m \times n_y}
\]

If one is interested in referring to the \(k\)th component (or column) of the output, the model equation decomposes into

\[
y_k = f(Xb_k) + \epsilon_k \tag{10.44}
\]

Recall that \(y_k\) indicates the \(k\)th row of matrix \(Y\), while \(y_k\) indicates the \(k\)th column.
Analogous to other predictive modeling techniques, PredictorMat2 constructor takes four arguments: the data/input matrix x, the response/output matrix y, the feature/variable names fname, and the hyper-parameters for the model/network hparam.

PredictorMat2 Class

Class Methods:

@param x the m-by-nx data/input matrix (data consisting of m input vectors)
@param y the m-by-ny response/output matrix (data consisting of m output vectors)
@param fname the feature/variable names (if null, use x_j's)
@param hparam the hyper-parameters for the model/network

abstract class PredictorMat2 (x: MatriD, y: MatriD, 
protected var fname: Strings, hparam: HyperParameter)

extends Predictor with Error

def getX: MatriD = x
def getY: VectoD = y.col(0)
def getYY: MatriD = y
def reset (eta_ : Double) { eta = eta_ }
def train0 (x_ : MatriD = x, y_ : MatriD = y): PredictorMat2
def train (x_ : MatriD = x, y_ : MatriD = y): PredictorMat2
def train (x_ : MatriD, y_ : VectoD): PredictorMat2 = train (x_ , MatrixD (Seq (y_)))
def train2 (x_ : MatriD = x, y_ : MatriD = y): PredictorMat2 = train (x_, y_)
def trainSwitch (which: Int, x_ : MatriD = x, y_ : MatriD = y): PredictorMat2 = 
def resetDF (df_update: PairD)
def eval (x_e: MatriD = x, y_e: VectoD = y.col(0)): PredictorMat2 =
def eval (x_e: MatriD, y_e: VectoD): PredictorMat2 = ???
def analyze (x_ : MatriD = x, y_ : VectoD = y(0),
def residual: VectoD = ee.col(0)
def residuals: MatriD = ee
def fitLabel: Seq [String] = fitA(0).fitLabel
def fitMap: IndexedSeq [Map [String, String]] =
def hparameter: HyperParameter = hparam
def parameter: VectoD = parameters (0).w(0)
def parameters: NetParams
def report: String =
def buildModel (x_cols: MatriD): PredictorMat2
def forwardSel (cols: Set [Int], index_q: Int = index_rSqBar): (Int, PredictorMat2) =
def forwardSelAll (index_q: Int = index_rSqBar, cross: Boolean = true): (Set [Int], MatriD) =
def backwardElim (cols: Set [Int], index_q: Int = index_rSqBar, first: Int = 1): (Int, PredictorMat2) =
def backwardElimAll (index_q: Int = index_rSqBar, first: Int = 1, cross: Boolean = true): 
(Set [Int], MatriD) =
def vif (skip: Int = 1): VectoD =
def predict (z: VectoD): Double = predictV (z)(0)
The default hyper-parameters are defined in the `Optimizer` object.

```scala
val hp = new HyperParameter
hp += ("eta", 0.1, 0.1) // learning/convergence rate
hp += ("bSize", 10, 10) // mini-batch size
hp += ("maxEpochs", 10000, 10000) // maximum number of epochs/iterations
hp += ("lambda", 0.0, 0.0) // regularization hyper-parameter
```

**NetParam Class**

A model producing multiple output variables will have parameters as weight matrices. They may also have bias vectors. To unify these cases, SCALaTION utilizes the `NetParam` case class for holding a weight matrix along with an optional bias vector. Linear algebra like operators are provided for convenience, e.g., the `*:` allows one to write `x *:` `p`, corresponding to the mathematical expression `XP` where `X` is the input/data matrix and `P` holds the parameters. If the bias `b` is null, this is just matrix multiplication.

```scala
def *: (x: MatriD): MatriD = x * w + b
```

Inside, the `x` is multiplied by the weight matrix `w` and the bias vector `b` is added. Note, the `*:` is right associative since the `NetParam` object is on right (see `NeuralNet_2L` for an example of its usage).

**Class Methods:**

- `@param w` the weight matrix
- `@param b` the optional bias/intercept vector (null => not used)

```scala
case class NetParam (w: MatriD, var b: VectoD = null)
```

```scala
def copy: NetParam = NetParam (w.copy, b.copy)
def set (c: NetParam) { w.set (c.w); b.set (c.b()) }
def set (cw: MatriD, cb: VectoD = null)
def += (c: NetParam) { w += c.w; b += c.b }
def += (cw: MatriD, cb: VectoD = null)
def -= (c: NetParam) { w -= c.w; b -= c.b }
def -= (cw: MatriD, cb: VectoD = null)
def * (x: MatriD): MatriD = x * w + b
def *: (x: MatriD): MatriD = x * w + b
def dot (x: VectoD): VectoD = (w dot x) + b
override def toString: String = s"b.w = $w \n b.b = $b"
```
10.6 Two-Layer Neural Networks

The `NeuralNet_2L` class supports multi-valued 2-layer (input and output) Neural Networks. The inputs into a Neural Network are given by the input vector \( \mathbf{x} \), while the outputs are given by the output vector \( \mathbf{y} \). Each input \( x_j \) is associated with an input node in the network, while each output \( y_k \) is associated with an output node in the network, as shown in Figure 10.7. The input layer consists of \( n_x \) input nodes, while the output layer consists of \( n_y \) output nodes.

![Figure 10.7: Two-Layer (input, output) Neural Network](image)

An edge connects each input node with each output node, i.e., there are \( n_x n_y \) edges in the network. The edge connecting input node \( j \) to output node \( k \) has weight \( b_{jk} \). In addition, below each output node is a bias \( \beta_k \). An alternative to having explicit bias offsets, is to include an intercept in the model (as an implicit bias) by adding a special input node (say \( x_0 \)) having its input always set to 1.

10.6.1 Model Equation

The weights on the edges are analogous to the parameter vector \( \mathbf{b} \) in regression. Each output variable \( y_k \), has its own parameter vector \( \mathbf{b}_{k} \). These are collected as column vectors into a parameter/weight matrix \( \mathbf{B} \), where parameter value \( b_{jk} \) is the edge weight connecting input node \( x_j \) with output node \( y_k \).

After training, given an input vector \( \mathbf{x} \), the network can be used to predict the corresponding output vector \( \mathbf{y} \). The network predicts an output/response value for \( y_k \) by taking the weighted sum of its inputs and passing this sum through activation function \( f \).

\[
\begin{align*}
y_k &= f(\mathbf{b}_{k} \cdot \mathbf{x}) + \epsilon_k \\
&= f\left(\sum_{j=0}^{n_x-1} b_{jk} x_j\right) + \epsilon_k
\end{align*}
\]

The model equation for `NeuralNet_2L` can written in vector form as follows:

\[
\mathbf{y} = f(\mathbf{B} \cdot \mathbf{x}) + \mathbf{\epsilon} = f(\mathbf{B}^\top \mathbf{x}) + \mathbf{\epsilon}
\]  

(10.45)
10.6.2 Training

Given several input vectors and output vectors in a training dataset \((i = 0, \ldots, m - 1)\), the goal is to optimize/fit the parameters/weights \(B\). The training dataset consisting of \(m\) input-output pairs is used to minimize the error in the prediction by adjusting the parameter/weight matrix \(B\). Given an input matrix \(X\) consisting of \(m\) input vectors and an output matrix \(Y\) consisting of \(m\) output vectors, minimize the distance between the actual/target output matrix \(Y\) and the predicted output matrix \(\hat{Y}\),

\[
\hat{Y} = f(XB) \tag{10.46}
\]

This will minimize the error matrix \(E = Y - \hat{Y}\)

\[
\min_B \|Y - f(XB)\|_F \tag{10.47}
\]

where \(\|\cdot\|_F\) is the Frobenius norm, \(X\) is a \(m\)-by-\(n_x\) matrix, \(Y\) is a \(m\)-by-\(n_y\) matrix, and \(B\) is a \(n_x\)-by-\(n_y\) matrix. Other norms may be used as well, but the square of the Frobenius norm will give the overall sum of squared errors \(sse\).

10.6.3 Optimization

As was the case with regression, it is convenient to minimize the dot product of the error with itself. We do this for each of the columns of the \(Y\) matrix to get the \(sse\) for each \(y_k\) and sum them up. The goal then is to simply minimize the loss function \(h(B) = \frac{1}{2} sse(B)\). As in the Perceptron section, we work with half of the sum of squared errors \(sse\). Summing the error over each column vector \(y_{ik}\) in matrix \(Y\) gives

\[
h(B) = \frac{1}{2} \sum_{k=0}^{n_y-1} (y_{ik} - f(Xb_{ik})) \cdot (y_{ik} - f(Xb_{ik})) \tag{10.48}
\]

This nonlinear optimization problem may be solved by a variety of optimization techniques, including Gradient-Descent, Stochastic Gradient Descent or Stochastic Gradient Descent with Momentum.

Most optimizers require a derivative and ideally these should be provided in functional form (otherwise the optimizer will need to numerically approximate them). Again, for the sigmoid (abbreviated sigm) activation function,

\[
sigm(u) = \frac{1}{1 + e^{-u}}
\]

the derivative is

\[
sigm(u)[1 - sigm(u)]
\]

To minimize the loss function, we decompose it into \(n_y\) functions.

\[
h(b_{ik}) = \frac{1}{2} (y_{ik} - f(Xb_{ik})) \cdot (y_{ik} - f(Xb_{ik}))
\]

Notice that this is the same as the Perceptron loss function, just with subscripts on \(y\) and \(b\).

In Regression, we took the gradient and set it equal to zero. Here, gradients will need to be computed by the optimizer. The equations will be the same as given in the Perceptron section, again just with subscripts added. The boxed equations from the Perceptron section become the following: The prediction vector for the \(k^{th}\) response/output is

324
The error vector for the $k^{th}$ response/output is
\[ \epsilon_k = y_k - \hat{y}_k = y_k - f(Xb_k) \] (10.50)

The delta vector for the $k^{th}$ response/output is
\[ \delta_k = \frac{\partial h}{\partial u_k} = -\epsilon_k * f'(Xb_k) \] (10.51)

where $u_k = Xb_k$. The gradient with respect to the $k^{th}$ parameter vector is
\[ \frac{\partial h}{\partial b_k} = -X^T[\epsilon_k * f'(Xb_k)] = X^T\delta_k \]

Finally, the update for the $k^{th}$ parameter vector is
\[ b_k = b_k - X^T\delta_k \eta \] (10.52)

### Sigmoid Case

For the sigmoid function, $f'(Xb_k) = f(Xb_k) *[1 - f(Xb_k)]$, so
\[ \delta = -\epsilon_k * f(Xb_k) * [1 - f(Xb_k)] \]

#### 10.6.4 Matrix Version

Of course the boxed equations may be rewritten in matrix form. The $m$-by-$n_y$ prediction matrix $\hat{Y}$ has a column for each output variable.
\[ \hat{Y} = f(XB) \] (10.53)

The $m$-by-$n_y$ negative of the error matrix $E$ is the difference between the predicted and actual/target output/response.
\[ E = \hat{Y} - Y \] (10.54)

The $m$-by-$n_y$ delta matrix $\Delta$ adjusts the error according to the slopes within $f'(XB)$ and is the elementwise matrix (Hadamard) product of $f'(XB)$ and $E$.
\[ \Delta = f'(XB) \circ E \] (10.55)

In math, the Hadamard product is denoted by the $\circ$ operator, while in SCALATion it is denoted by the $*$ operator. Finally, the $n_x$-by-$n_y$ parameter matrix $B$ is updated by $-X^T \Delta \eta$.
\[ B = B - X^T\Delta \eta \] (10.56)

The corresponding code in the `train0` method is shown below:
def train0 (x_: MatriD = x, y_: MatriD = y): NeuralNet_2L =
{
    var sse0 = Double.MaxValue // hold prior value of sse
    for (epoch <- 1 to maxEpochs) { // iterate over each epoch
        val yp = f.fM (x_ :*: b) // Yp = f(XB)
        ee = yp - y_ // negative of error matrix
        val d = f.dM (yp) ** ee // delta matrix for yp
        b -= x_.t * d * eta // update 'b' parameters
        val sse = sseF (y_, f.fM (x_ :*: b))
        if (DEBUG) println (s"train0: parameters for $epoch th epoch: b = $b, sse = $sse")
        if (sse > sse0) return this // return when sse increases
        sse0 = sse // save prior sse
    } // for
} // train0

Note: f.fM is the matrix version of the activation function and it is created using the matrixize high-order function that takes a vector function as input.

def matrixize (f: FunctionV_2V): FunctionM_2M = (x: MatriD) => x.map (f(_))
val fM = matrixize (fV)

Similarly, f.dM is the matrix version of the derivative of the activation function. The NeuralNet_2L class also provides train (typically better than train0) and train2 (with built in η search) methods.

NeuralNet_2L Class

Class Methods:

@param x the m-by-nx input matrix (training data consisting of m input vectors)
@param y the m-by-ny output matrix (training data consisting of m output vectors)
@param fname_ the feature/variable names (if null, use x_j’s)
@param hparam the hyper-parameters for the model/network
@param f the activation function family for layers 1->2 (input to output)
@param itran the inverse transformation function returns responses to original scale

class NeuralNet_2L (x: MatriD, y: MatriD,
    fname_ : Strings = null, hparam: HyperParameter = Optimizer.hp,
    f: AFF = f_sigmoid, val itran: FunctionV_2V = null)
    extends PredictorMat2 (x, y, fname_, hparam)

def parameters: NetParams = Array (b)
def train0 (x_: MatriD = x, y_: MatriD = y): NeuralNet_2L =
def train (x_: MatriD = x, y_: MatriD = y): NeuralNet_2L =
override def train2 (x_: MatriD = x, y_: MatriD = y): NeuralNet_2L =
def buildModel (x_cols: MatriD): NeuralNet_2L =

326
def predictV (z: VectoD): VectoD = f.fV (b dot z)
def predictV (z: MatriD = x): MatriD = f.fM (b * z)

10.6.5 Exercises

1. The dataset in ExampleConcrete consists of 7 input variables and 3 output variables.

    // Input Variables (7) (component kg in one M^3 concrete):
    // 1. Cement
    // 2. Blast Furnace Slag
    // 3. Fly Ash
    // 4. Water
    // 5. Super Plasticizer (SP)
    // 6. Coarse Aggregate
    // 7. Fine Aggregate
    // Output Variables (3):
    // 1. SLUMP (cm)
    // 2. FLOW (cm)
    // 3. 28-day Compressive STRENGTH (Mpa)

Create a NeuralNet_2L model to predict values for the three outputs \( y_0 \), \( y_1 \) and \( y_2 \). Compare with the results of using three Perceptrons.

2. Create a NeuralNet_2L model to predict values for the one output for the AutoMPG dataset. Compare with the results of using the following models: (a) Regression, (b) Perceptron.

3. Were the results in for AutoMPG dataset the same for Perceptron and NeuralNet_2L? Please explain. In general, is a NeuralNet_2L equivalent to \( n_y \) Perceptrons?

4. Draw a NeuralNet_2L with \( n_x = 4 \) input nodes and \( n_y = 2 \) output nodes. Label the eight edges with weights from the 4-by-2 weight matrix \( B = [b_{jk}] \). Write the two model equations, one for \( y_0 \) and one for \( y_1 \). Combine these two equations into one vector equation for \( \mathbf{y} = [y_0, y_1] \). Given column vector \( \mathbf{x} = [1, x_1, x_2, x_3] \), express \( \hat{\mathbf{y}} = f(B^T \mathbf{x}) \) at the scalar level.
10.7 Three-Layer Neural Networks

The `NeuralNet_3L` class supports 3-layer (input, hidden and output) Neural Networks. The inputs into a Neural Net are given by the input vector \( x \), while the outputs are given by the output vector \( y \). Between these two layers is a single hidden layer, whose intermediate values will be denoted by the vector \( z \). Each input \( x_j \) is associated with an input node in the network, while each output \( y_k \) is associated with an output node in the network, as shown in Figure 10.8. The input layer consists of \( n_x \) input nodes, the hidden layer consists of \( n_z \) hidden nodes, and the output layer consists of \( n_y \) output nodes.

![Figure 10.8: Three-Layer (input, hidden, output) Neural Network](image)

There are two sets of edges. Edges in the first set connect each input node with each hidden node, i.e., there are \( n_x n_z \) such edges in the network. The parameters (or edge weights) for the first set of edges are maintained in matrix \( A = [a_{jk}]_{n_x \times n_z} \). Edges in the second set connect each hidden node with each output node, i.e., there are \( n_z n_y \) such edges in the network. The parameters (or edge weights) for the second set of edges are maintained in matrix \( B = [b_{hk}]_{n_z \times n_y} \).

There are now two activation functions, \( f_0 \) and \( f_1 \). \( f_0 \) is applied at each node in the hidden layer, while \( f_1 \) plays this role for the output layer. Having two activation functions allows greater capability to approximate a variety of functional forms (for more information see [21, 43] on Universal Approximation Theorems).

10.7.1 Model Equation

The model equation for `NeuralNet_3L` can be written in vector form as follows:

\[
y = f_1(B \cdot f_0(A \cdot x)) + \epsilon = f_1(B^T f_0(A^T x)) + \epsilon
\]  

(10.57)

The innermost matrix-vector product multiplies the transpose of the \( n_z \)-by-\( n_z \) matrix \( A \) by the \( n_z \)-by-1 vector \( x \), producing an \( n_z \)-by-1 vector, which is passed into the \( f_0 \) vectorized activation function. The outermost matrix-vector product multiplies the transpose of the \( n_z \)-by-\( n_y \) matrix \( B \) by the \( n_z \)-by-1 vector results, producing an \( n_y \)-by-1 vector, which is passed into the \( f_1 \) vectorized activation function.
Intercept/Bias

As before, one may include an intercept in the model (also referred to as bias) by having a special input node (say $x_0$) that always provides the value 1. A column of all one in an input matrix (see below) can achieve this. This approach could be carried forward to the hidden layer by including a special node (say $z_0$) that always produces the value 1. In such case, the computation performed at node $z_0$ would be thrown away and replaced with 1. The alternative is to replace the uniform notion of parameters with two types of parameters, weights and biases. SCALATIION supports this with the NetParam case class.

```scala
@param w the weight matrix
@param b the optional bias/intercept vector (null => not used)

case class NetParam (w: Matrix, var b: Vector = null)

def dot (x: Vector): Vector = (w dot x) + b
```

Following this approach, there is no need for the special nodes and the dot product is re-defined to add the bias $b$ to the regular matrix-vector dot product ($w \cdot x$). Note, the NetParam class defines several other methods as well. The vector version of the predict method in NeuralNet.3L uses this dot product to make predictions.

```scala
def predictV (v: Vector): Vector = f1.fV (b dot f0.fV (a dot v))
```

10.7.2 Training

Given a training dataset made up of an $m$-by-$n_x$ input matrix $X$ and an $m$-by-$n_y$ output matrix $Y$, training consists of making a prediction $\hat{Y}$,

$$\hat{Y} = f_1(f_0(XA)B)$$ (10.58)

and determining the error in prediction $E = Y - \hat{Y}$ with the goal of minimizing the error.

$$\min_B \|Y - f_1(f_0(XA)B)\|_F$$ (10.59)

Training involves an iterative procedure (e.g., stochastic gradient descent) that adjusts parameter values (for weights and biases) to minimize a loss function such as $sse$ or rather half $sse$ (or $h$). Before the main loop, random parameter values (for weights and biases) need to be assigned to NetParam $A$ and NetParam $B$. Roughly as outlined in section 3 of [83], the training can be broken into four steps:

1. Compute predicted values for output $\hat{y}$ and compare with actual values $y$ to determine the error $y - \hat{y}$.
2. Back propagate the adjusted error to determine the amount of correction needed at the output layer. Record this as vector $\delta^1$.
3. Back propagate the correction to the hidden layer and determine the amount of correction needed at the hidden layer. Record this as vector $\delta^0$.
4. Use the delta vectors, $\delta^1$ and $\delta^0$, to makes updates to NetParam $A$ and NetParam $B$, i.e., the weights and biases.
10.7.3 Optimization

In this subsection, the basic elements of the back-propagation algorithm are presented. In particular, we now go over the four steps outlined above in more detail. Biases are ignored for simplicity, so the \( A \) and \( B \) NetParams are treated as weight matrices. In the code, the same logic includes the biases (so nothing is lost, see exercises). Note that \( h \) denotes a loss function, while \( h \) is an index into the hidden layer.

1. Compute predicted values: Based on the randomly assigned weights to the \( A \) and \( B \) matrices, predicted outputs \( \hat{y} \) are calculated. First values for the hidden layer \( z \) are calculated, where the value for hidden node \( h, z_h \), is given by

\[
    z_h = f_0(a_h \cdot x) \quad \text{for } h = 0, \ldots, n_z - 1
\]

where \( f_0 \) is the first activation function (e.g., sigmoid), \( a_h \) is column-\( h \) of the \( A \) weight matrix, and \( x \) is an input vector for a training sample/instance (row in the data matrix). Typically, several samples (referred to as a mini-batch) are used in each step. Next, the values computed at the hidden layer are used to produce predicted outputs \( \hat{y} \), where the value for output node \( k, \hat{y}_k \), is given by

\[
    \hat{y}_k = f_1(b_k \cdot z) \quad \text{for } k = 0, \ldots, n_y - 1
\]

where the second activation function \( f_1 \) may be the same as (or different from) the one used in the hidden layer and \( b_k \) is column-\( k \) of the \( B \) weight matrix. Now the difference between the actual and predicted output can be calculated by simply subtracting the two vectors, or elementwise, the error for the \( k \)th output, \( \epsilon_k \), is given by

\[
    \epsilon_k = y_k - \hat{y}_k \quad \text{for } k = 0, \ldots, n_y - 1
\]

Obviously, for subsequent iterations, the updated/corrected weights rather than the initial random weights are used.

2. Back propagate from output layer: Given the computed error vector \( \epsilon \), the delta/correction vector \( \delta^1 \) for the output layer may be calculated, where for output node \( k, \delta^1_k \) is given by

\[
    \delta^1_k = [-\epsilon_k] \cdot f'(b_k \cdot z) \quad \text{for } k = 0, \ldots, n_y - 1 \quad (10.60)
\]

where \( f' \) is the derivative of the activation function (e.g., for sigmoid, \( f'(u) = f(u)[1 - f(u)] \)). The partial derivative of the loss function \( h \) with respect to the weight connecting hidden node \( h \) with output node \( k, b_{hk} \), is given by

\[
    \frac{\partial h}{\partial b_{hk}} = z_h \delta^1_k \quad (10.61)
\]

3. Back propagate from hidden layer: Given the delta/correction vector \( \delta^1 \) from the output layer, the delta vector for the hidden layer \( \delta^0 \) may be calculated, where for hidden node \( h, \delta^0_h \) is given by

\[
    \delta^0_h = [b_h \cdot \delta^1] \cdot f_0(a_h \cdot x) \quad \text{for } h = 0, \ldots, n_z - 1 \quad (10.62)
\]
This equation is parallel to the one given for $\delta^1_k$ in that an error-like factor multiplies the derivative of the activation function. In this case, the error-like factor is the weighted average of the $\delta^1_k$ for output nodes connected to hidden node $h$ times row-$h$ of weight matrix $B$. The weighted average is computed using the dot product.

$$b_h \cdot \delta^1 = \sum_{k=0}^{n_y-1} b_{hk} \delta^1_k$$

The partial derivative of $h$ with respect to the weight connecting input node $j$ with hidden node $h$, $a_{jh}$, is given by

$$\frac{\partial h}{\partial a_{jh}} = x_j \delta^0_h$$ (10.63)

4. Update weights: The weight matrices $A$ and $B$, connecting input to hidden and hidden to output layers, respectively, may now be updated based on the partial derivatives. For gradient descent, movement is in the opposite direction, so the sign flips from positive to negative. These partial derivatives are multiplied by the learning rate $\eta$ which moderates the adjustments to the weights.

$$b_{hk} = z_h \delta^1_k \eta$$ (10.64)

$$a_{jh} = x_j \delta^0_h \eta$$ (10.65)

Figure 10.9 shows the forward ($\to$) and backward ($\leftarrow$) propagation through the Neural Network and depicts the role of $\delta$ in updating the parameters from the perspective of hidden node $z_h$.

Figure 10.9: View for Hidden Node $z_h$
mini-batch including, at the end, updates to the $A$ and $B$ estimates, the current epoch is said to be complete. Correspondingly, the above equations may be vectorized/matrixized so that calculations are performed over many instances in a mini-batch using matrix operations. Each outer iteration (epoch) typically should improve the $A$ and $B$ estimates. Simple stopping rules include specifying a fixed number of iterations or breaking out of the outer loop when the loss function $\mathcal{h}$ fails to decrease for $q$ iterations.

### 10.7.4 Matrix Version

Given a training dataset consisting of an input/data matrix $X \in \mathbb{R}^{m \times n_x}$ and an output/response matrix $Y \in \mathbb{R}^{m \times n_y}$, the optimization equations may be re-written in matrix form as shown below.

The gradient descent optimizer used by the `train0` method has one main loop, while the stochastic gradient descent optimizer used by the `train` has two main loops. The outer loop iterates over *epochs* which serve to improve the parameters/weights with each iteration. If the fit does not improve in several epochs, the algorithm likely should break out of this loop.

The four boxed equations from the previous section become seven due to the extra layer. The optimizers compute predicted outputs and take differences between the actual/target values and these predicted values to compute an error matrix. These results are then used to compute delta matrices that form the basis for updating the parameter/weight matrices $A \in \mathbb{R}^{n_z \times n_x}$ and $B \in \mathbb{R}^{n_x \times n_y}$.

1. The hidden (latent) values for all $m$ instances and all $n_z$ hidden nodes are computed by applying the first matrixized activation function $f_0$ to the matrix product $XA$ to produce the latent feature matrix $Z \in \mathbb{R}^{m \times n_z}$.

   \[
   Z = f_0(XA) \tag{10.66}
   \]

   The predicted output matrix $\hat{Y} \in \mathbb{R}^{m \times n_y}$ is similarly computed by applying the second matrixized activation function $f_1$ to the matrix product $ZB$.

   \[
   \hat{Y} = f_1(ZB) \tag{10.67}
   \]

2. The negative of the error matrix $E \in \mathbb{R}^{m \times n_y}$ is just the difference between the predicted and actual/target values.

   \[
   E = \hat{Y} - Y \tag{10.68}
   \]

3. This information is sufficient to calculate delta matrices: $\Delta^1$ for adjusting $B$ and $\Delta^0$ for adjusting $A$. The output-layer delta matrix $\Delta^1 \in \mathbb{R}^{m \times n_y}$ is the elementwise matrix (Hadamard) product of $E$ and $f'_1(ZB)$.

   \[
   \Delta^1 = E \circ f'_1(ZB) \tag{10.69}
   \]

   The hidden-layer delta matrix $\Delta^0 \in \mathbb{R}^{m \times n_z}$ is the elementwise matrix (Hadamard) product of $\Delta^1 B'$ and $f'_0(XA)$.

   \[
   \Delta^0 = [\Delta^1 B'] \circ f'_0(XA) \tag{10.70}
   \]
4. As mentioned, the delta matrices form the basis (a matrix transpose × delta × the learning rate \( \eta \)) for updating the parameter/weight matrices, \( A \) and \( B \).

\[
B \; - \; Z^T \Delta^1 \eta
\]

\[
A \; - \; X^T \Delta^0 \eta
\]

10.7.5 Gradient Descent Algorithm

The basic gradient descent algorithm places the above matrix equations into a loop to iteratively update the weight matrices and bias vectors (unified via the \texttt{NetParam} class). The corresponding \texttt{ScalaTion} code for the \texttt{train0} method is show below. Note: \(*\): defined in \texttt{NetParam} computes value-matrix \(*\) weight-matrix \(+\) bias-vector.

```scala
def train0 (x_: MatriD = x, y_: MatriD = y): NeuralNet_3L =
{
  var sse0 = Double.MaxValue // hold prior value of sse
  for (epoch <- 1 to maxEpochs) {
    var z = f0.fM (x_*: a) // Z = f0(XA)
    var yp = f1.fM (z *: b)
    ee = yp - y_ // negative of the error matrix
    val d1 = ee ** f1.dM (yp) // delta matrix for yp
    val d0 = (d1 * b.w.t) ** f0.dM (z) // delta matrix for z
    a -= (x_.t * d0 * eta, d0.mean * eta) // update 'a' weights & biases
    b -= (z.t * d1 * eta, d1.mean * eta) // update 'b' weights & biases
    val sse = sseF (y_, f1.fM (f0.fM (x_*: a)) *: b)
    if (DEBUG) println (s"train0: parameters for $epoch th epoch: b = $b, sse = $sse")
    if (sse > sse0) return this // return early if moving up
    sse0 = sse // save prior sse
    this
  } // for
  this
} // train0
```

For stochastic gradient descent in the \texttt{Optimizer} object, the inner loop divides the training dataset into \( nB \) mini-batches. A batch is a randomly selected group/batch of rows. Each batch (\( ib \)) is passed to the \texttt{updateWeight (x(ib), y(ib))} method that updates the \( A \) and \( B \) parameter/weight matrices.

Neural networks may be used for prediction/regression as well as classification problems. For prediction/regression, the number of output nodes would corresponding to the number of responses. For example, in the \texttt{ExampleConcrete} example there are three response columns, requiring three instances of \texttt{Regression} or one instance of \texttt{NeuralNet_3L}. Three separate \texttt{NeuralNet_3L} instances each with one output node could be used as well. Since some activation functions have limited ranges, it is common practice for these types of problems to let the activation function in the last layer be identity \( \text{id} \). If this is not done, response columns need to be re-scaled based on the training dataset. Since the testing dataset may have values outside this range, this approach may not be ideal.
**Softmax Activation Function**

For classification problems, it is common to have an output node for each response value for the categorical variable, e.g., “no”, “yes” would have $y_0$ and $y_1$, while “red”, “green”, “blue” would have $y_0$, $y_1$ and $y_2$. The softmax activation function is a common choice for the last layer for classification problems.

$$f_i(u) = \frac{e^{u_i}}{1 + e^u} \quad \text{for } i = 0, \ldots, n - 1$$

The values produced by the softmax activation function can be thought of as giving a probability score to the particular class label, e.g., the image shows a “cat” vs. “dog”.

**10.7.6 Example Error Calculation Problem**

Draw a 3-layer (input, hidden and output) Neural Network (with sigmoid activation), where the number of nodes per layer are $n_x = 2$, $n_z = 2$ and $n_y = 1$.

**Input to Hidden Layer Parameters**

Initialize bias vector $\alpha$ to $[.1, .1]$ and weight matrix $A$ ($n_x$-by-$n_z$) to

$$
\begin{bmatrix}
.1 & .2 \\
.3 & .4
\end{bmatrix}
$$

**Hidden to Output Layer Parameters**

Initialize bias vector $\beta$ to $[.1]$ and weight matrix $B$ ($n_z$-by-$n_y$) to

$$
\begin{bmatrix}
.5 \\
.6
\end{bmatrix}
$$

**Compute the Error for the First Iteration**

Let $x = [x_0, x_1] = [2, 1]$ and $y_0 = .8$, and then compute the error $\epsilon_0 = y_0 - \hat{y}_0$, by feeding the values from vector $x$ forward. First compute values at the hidden layer for $z$.

$$
\begin{align*}
z_h &= f_0(a_h \cdot x + a_h) \\
z_0 &= f_0(a_0 \cdot x + a_0) \\
z_0 &= f_0([0.1, 0.3] \cdot [2.0, 1.0] + 0.1) \\
z_0 &= f_0(0.6) = 0.645656 \\
z_1 &= f_0(a_1 \cdot x + a_1) \\
z_1 &= f_0([0.2, 0.4] \cdot [2.0, 1.0] + 0.1) \\
z_1 &= f_0(0.9) = 0.710950
\end{align*}
$$

One may compute the values for sigmoid activation function as follows:

```java
println (ActivationFun.sigmoidV (VectorD (0.6, 0.9)))
```
Then compute predicted values at the output layer for $\hat{y}$.

$$\hat{y}_k = f_1(b_k \cdot z + \beta_k)$$
$$\hat{y}_0 = f_1(b_0 \cdot z + \beta_0)$$
$$\hat{y}_0 = f_1([0.5, 0.6] \cdot [0.645656, 0.71095] + 0.1)$$
$$\hat{y}_0 = f_1(0.849398) = 0.7004408$$

Therefore, the error is

$$e_0 = y_0 - \hat{y}_0$$
$$e_0 = 0.8 - 0.7004408 = 0.0995592$$

**NeuralNet_3L Class**

**Class Methods:**

@param x the m-by-nx input matrix (training data consisting of m input vectors)
@param y the m-by-ny output matrix (training data consisting of m output vectors)
@param nz the number of nodes in hidden layer
@param fname_ the feature/variable names (if null, use x_j’s)
@param hparam the hyper-parameters for the model/network
@param f0 the activation function family for layers 1->2 (input to hidden)
@param f1 the activation function family for layers 2->3 (hidden to output)
@param itran the inverse transformation function returns responses to original scale

class NeuralNet_3L (x: MatriD, y: MatriD, 
private var nz: Int = -1, 
fname_: Strings = null, hparam: HyperParameter = hp, 
f0: AFF = f_tanh, f1: AFF = f_id, 
val itran: FunctionV_2V = null) 
extends PredictorMat2 (x, y, fname_, hparam) // sets eta in parent class 

def compute_df_m (n: Int): Int = (nx + 2) * n 
def parameters: NetParams = Array (a, b) 
def train0 (x_: MatriD = x, y_: MatriD = y): NeuralNet_3L = 
def train (x_: MatriD = x, y_: MatriD = y): NeuralNet_3L = 
override def train2 (x_: MatriD = x, y_: MatriD = y): NeuralNet_3L = 
def buildModel (x_cols: MatriD): NeuralNet_3L = 
def predictV (v: VectoD): VectoD = f1.fV (b dot f0.fV (a dot v)) 
def predictV (v: MatriD = x): MatriD = f1.fM (b * f0.fM (a * v)) 
def showEstat ()

335
10.7.7 Exercises

1. **Delta Vectors**: For the example error calculation problem given in this section, calculate the $\delta^1 = [\delta^1_0]$ vector using the following formula.

$$\delta^1_k = - \epsilon_k f'_1 (b_k \cdot z)$$

Rework the problem using separate weights ($b_k$) and biases ($\beta_k$) (not lumped together as parameters) to compute $\delta^1_k$.

$$\delta^1_k = - \epsilon_k f'_1 (b_k \cdot z + \beta_k)$$

Calculate the $\delta^0 = [\delta^0_0, \delta^0_1]$ vector using the analogous reworked formula having separate weights ($a_h$) and biases ($\alpha_h$) to compute $\delta^0_h$.

$$\delta^0_h = [b_h \cdot \delta^1] f'_0 (a_h \cdot x + \alpha_h)$$

2. **Parameter Update Equations**: Use the $\delta_1$ vector to update weight matrix $B$, i.e., for each row $h$,

$$b_h = z_h \delta_1 \eta$$

and update the bias vector $\beta$ as follows:

$$\beta = \delta_1 \eta$$

Use the $\delta_0$ vector to update weight matrix $A$, i.e., for each row $j$,

$$a_j = x_j \delta_0 \eta$$

and update the bias vector $\alpha$ as follows:

$$\alpha = \delta_0 \eta$$

3. Derive the equation for the partial derivative of the loss function $h$ w.r.t. $b_{hk}$,

$$\frac{\partial h}{\partial b_{hk}} = z_h \delta^1_k$$

by defining pre-activation value $v_k = b_k \cdot z$ and applying the following chain rule:

$$\frac{\partial h}{\partial b_{hk}} = \frac{\partial h}{\partial v_k} \frac{\partial v_k}{\partial b_{hk}}$$

4. Explain the formulations for the two delta matrices.

$$\Delta^1 = E \circ f'_1 (ZB)$$

$$\Delta^0 = [\Delta^1 B'] \circ f'_0 (XA)$$
5. The dataset in ExampleConcrete consists of 7 input variables and 3 output variables. See the NeuralNet_2L section for details. Create a NeuralNet_3L model to predict values for the three outputs $y_0$, $y_1$ and $y_2$. Compare with the results of using a NeuralNet_2L model.

6. Create a NeuralNet_3L model to predict values for the one output for the AutoMPG dataset. Compare with the results of using the following models: (a) Regression, (b) Perceptron, (c) NeuralNet_2L.

7. Conduct a literature study on the effectiveness and efficiency of various training algorithms for neural networks, for example see [18].
10.8 Multi-Hidden Layer Neural Networks

The `NeuralNet_XL` class supports basic x-layer (input, {hidden} and output) Neural Networks. For example a four layer neural network (see Figure 10.10) with four layers of nodes with (one input layer numbered 0, two hidden layers numbered 1 and 2, and one output layer numbered 3). Note, since the input layer’s purpose is just to funnel the input into the model, it is also common to refer to such a neural network as a three layer network. This has the advantage that number layers now corresponds to the number parameter/weight matrices.

![Figure 10.10: Four-Layer (input, hidden (l = 0), hidden (l = 1), output (l = 2)) Neural Network](image)

In SCALATION, the number of active layers is denoted by n1 (which in this case equals 3 with l = 0, 1, 2). Since arrays of matrices are used in the SCALATION code, multiple layers of hidden nodes are supported. The matrix $B_l = [b_{jk}^l]$ maintains the weights for layer l, while $\beta_l = [\beta_k^l]$ maintains the biases for layer l.

In particular, parameter $b$ which holds the weights and biases for all layers is of type `NetParams` where

```
type NetParams = IndexedSeq[NetParam]
```

For simplicity, in the model equation below rolls the biases into the weights using `NetParam`.

### 10.8.1 Model Equation

The equations for `NeuralNet_XL` are the same as those used for `NeuralNet_3L`, except that the calculations are repeated layer by layer in a forward direction for prediction. The model equation for a four layer `NeuralNet_XL` can written in vector form as follows:

$$y = f_2(B_2 \cdot f_1(B_1 \cdot f_0(B_0 \cdot x))) + \epsilon$$  \hspace{1cm} (10.73)

where $B_l$ is the `NetParam` (weight matrix and bias vector) connecting layer l to layer l + 1 and $f_1$ is the vectorized activation function at layer l + 1.
10.8.2 Training

As before, the training dataset consists of an \( m \)-by-\( n_x \) input matrix \( X \) and an \( m \)-by-\( n_y \) output matrix \( Y \). During training, the predicted values \( \hat{Y} \) are compared to actual/target values \( Y \),

\[
\hat{Y} = f_2(f_1(f_0(XB_0)B_1)B_2)
\]  

(10.74)

to compute an error matrix \( E = Y - \hat{Y} \), to be minimized.

\[
\min_B \|Y - f_2(f_1(f_0(XB_0)B_1)B_2)\| F
\]  

(10.75)

Corrections based on these errors are propagated backward through the network to improve the parameter estimates (weights and biases) layer by layer.

10.8.3 Optimization

The seven boxed equations from the previous section become six due to unification of the last two. As before, the optimizers compute a predicted output matrix and then take differences between the actual/target values and these predicted values to compute an error matrix. These computed matrices are then used to compute delta matrices that form the basis for updating the weight matrices. Again for simplicity, biases are ignored in the equations below, but are taken care of in the code through the NetParam abstraction. See the exercises for details.

1. The values are feed forward through the network, layer by layer. For layer \( l \), these values are stored in matrix \( Z_l \). The first layer is the input, so \( Z_0 = X \). For the rest of the layers, \( Z_{l+1} \) equals the result of activation function \( f_l \) being applied to the product of the previous layer’s \( Z_l \) matrix times its parameter matrix \( B_l \).

\[
Z_0 = X
\]  

(10.76)

For each layer \( l \) in the forward direction:

\[
Z_{l+1} = f_l(Z_lB_l)
\]  

(10.77)

2. The negative of the error matrix \( E \) is just the difference between the predicted and actual/target values, where \( \hat{Y} = Z_{nl} \).

\[
E = \hat{Y} - Y
\]  

(10.78)

3. This information is sufficient to calculate delta matrices \( \Delta^l \). For the last layer:

\[
\Delta^{nl-1} = f'_{nl-1}(Z_{nl-1}B_{nl-1}) \circ E
\]  

(10.79)

For the rest of layers in the backward direction with \( l \) being decremented:

\[
\Delta^l = f'_l(Z_lB_l) \circ (\Delta^{l+1}B^T_{l+1})
\]  

(10.80)
4. As mentioned, the delta matrices form the basis (a matrix transpose × delta × the learning rate \( \eta \)) for updating the parameter/weight matrices, \( B_l \) for each layer \( l \).

\[
B_l \munuseq Z_l^\top \Delta_l \eta 
\]

(10.81)

The implementation of the `train0` encodes these equation and uses gradient descent to improve the parameters \( B_l \) over several epochs, terminating early when the objective/cost function fails to improve.

```scala
def train0 (x_ : MatriD = x, y_ : MatriD = y): NeuralNet_XL =
{
  var sse0 = Double.MaxValue // hold prior value of sse
  val z = Array.ofDim [MatriD] (nl+1); z(0) = x_ // storage: activations f(Z_l B_l)
  val d = Array.ofDim [MatriD] (nl) // storage: deltas

  for (epoch <- 1 to maxEpochs) { // iterate over each epoch
    for (l <- layers) z(l+1) = f(l).fM (z(l) :: b(l)) // feedforward and store activations
    ee = z.last - y_ // negative of error matrix
    d(nl-1) = f.last.dM (z.last) ** ee // delta for last layer before output
    for (l <- nl-2 to 0 by -1)
      d(l) = f(l).dM (z(l+1)) ** (d(l+1) * b(l+1).w.t) // deltas for previous hidden layers
    for (l <- layers) b(l) -= (z(l).t * d(l) * eta, d(l).mean * eta) // update (weights, biases)
    val sse = ee.normFSq
    if (DEBUG) println (s"train0: parameters for $epoch th epoch: b = $b, sse = $sse")
    if (sse > sse0) return this // return early if moving up
    sse0 = sse // save prior sse
  }
} // train0
```

The `train0` method is kept simple to facilitate understanding. In practice, the `train` and `train2` methods should be used.

### 10.8.4 Number of Nodes in Hidden Layers

The array `nz` gives the number of nodes for each of the hidden layers. If the user does not specify the number of nodes for each hidden layer, then based on the number of input nodes `nx` and number of output nodes `ny`, defaults are utilized according to one of two rules. The comments should be switched to change rules.

```scala
// if (nz == null) nz = compute_nz (nx) // Rule [1] default # nodes for hidden layers
if (nz == null) nz = compute_nz (nx, ny) // Rule [2] default # nodes for hidden layers
val df_m = compute_df_m (nz) // degrees of freedom for model
resetDF (df_m, x.dim1 - df_m) // degrees of freedom for (model, error)
```

If the array is `null`, then default numbers for the hidden layers are utilized. Rule [1] (drop one), simply decreases the number of nodes in each hidden layer by one, starting with `nx - 1, nx - 2,` etc. Rule [2] (average), takes the average between `nx` and `ny` for the first hidden layer, averages that with `ny` for the second layer, etc. The number of nodes in each layer is used make a rough estimate of the degrees of freedom for the model [?]. Currently, the degrees of freedom is only considered for the first output variable.
10.8.5 Avoidance of Overfitting

If efforts are not made to avoid overfitting, NeuralNet_XL models are likely to suffer from this problem. When $R^2$, $\bar{R}^2$ are much higher than $R^2_{cv}$ there may be two causes. One is that the tuning of hyper-parameters on the full dataset, is different from the tuning on training set slices. Two is that the optimization algorithm finished with the signal and continued on to fit the noise. The simplest way to reduce overfitting is to make the optimizer quit before focusing its efforts on the noise. If only we knew. A crude way to this is to reduce the maximum number of epochs. A better way to do this is to split a training set into two parts, one for training (iteratively adjusting the parameters) and other for the stopping rule. The stopping rule would compute the objective/loss function only using the validation data. Drop-out may be used in which a node temporarily taken out of the model. Regularization of the parameters, as was done for Ridge and Lasso Regression, may help as well.

10.8.6 Deep Learning

When the number of hidden layers are increased beyond the base levels of one or two, the learning may be described as deep. In addition, special types of layers or units may be included such as convolutional or recurrent.

NeuralNet_XL Class

Class Methods:

```scala
@linenotag
@linenotag
class NeuralNet_XL (x: MatriD, y: MatriD,
  private var nz: Array[Int] = null,
  fname_: Strings = null, hparam: HyperParameter = Optimizer.hp,
  f : Array[AFF] = Array (f_tanh, f_tanh, f_id),
  val itran: FunctionV_2V = null)
  extends PredictorMat2 (x, y, fname_, hparam) // sets eta in parent class

  def compute_nz (nx: Int): Array[Int] =

  def compute_nz (nx: Int, ny: Int): Array[Int] =

  def compute_df_m (n: Array[Int]): Int =

  def parameters: NetParams = b

  def train0 (x_: MatriD = x, y_: MatriD = y): NeuralNet_XL =

  def train (x_: MatriD = x, y_: MatriD = y): NeuralNet_XL =

  override def train2 (x_: MatriD = x, y_: MatriD = y): NeuralNet_XL =

  def buildModel (x_cols: MatriD): NeuralNet_XL =
```

341
When a neural network is being used for prediction/regression problems, it is often the case that there will be a single output node. SCALATION provides the `NeuralNet_XL1` class for this.

### NeuralNet_XL1 Class

**Class Methods:**

- @param x the m-by-nx data/input matrix (training data having m input vectors)
- @param y the m response/output vector (training data having m output values)
- @param nz_ the number of nodes in each hidden layer, e.g., Array (9, 8)
- @param fname_ the feature/variable names (if null, use x_j's)
- @param hparam the hyper-parameters for the model/network
- @param f the array of activation function families between every pair of layers
- @param itran the inverse transformation function returns responses to original scale

```scala
class NeuralNet_XL1 (x: MatriD, y: VectoD, 
    nz_ : Array[Int] = null, 
    fname_ : Strings = null, hparam: HyperParameter = hp, 
    f: Array[AFF] = Array (f_tanh, f_tanh, f_id), 
    itran: FunctionV_2V = null) 
    extends NeuralNet_XL (x, MatrixD (y), nz_, fname_, hparam, f, itran)
```

### 10.8.7 Exercises

1. Examine the implementation of the `train0` method and the `NetParam` case class, where the net parameter b has two parts: the weight matrix b.w and the bias vector b.b. Show how the biases affect the calculation of prediction matrix \( \hat{Y} = Z_{nl} \) in the feed forward process.

2. Examine the implementation of the `train0` method and the `NetParam` case class and show how the biases affect the update of the weights b.w in the back-propagation process.

3. Examine the implementation of the `train0` method and the `NetParam` case class and show how the biases b.b are updated in the back-propagation process.

4. The dataset in `ExampleConcrete` consists of 7 input variables and 3 output variables. See the `NeuralNet_2L` section for details. Create a `NeuralNet_XL` model with four layers to predict values for the three outputs \( y_0, y_1 \) and \( y_2 \). Compare with the results of using a `NeuralNet_3L` model.
5. Create a NeuralNet_XL model with four layers to predict values for the one output for the AutoMPG dataset. Compare with the results of using the following models: (a) Regression, (b) Perceptron, (c) NeuralNet_2L, (d) NeuralNet_3L.

6. **Tuning the Hyper-Parameters:** The learning rate $\eta$ (eta in the code) needs frequent tuning. ScalaTion as with most packages has limited auto-tuning of the learning rate. Tune the other hyper-parameters for the AutoMPG dataset.

   ```scala
   /** hyper-parameters for tuning the optimization algorithms - user tuning */
   val hp = new HyperParameter
   hp += ("eta", 0.1, 0.1) // learning/convergence rate
   hp += ("bSize", 20, 20) // mini-batch size, common range 10 to 30
   hp += ("maxEpochs", 500, 500) // maximum number of epochs/iterations
   hp += ("patience", 4.0, 4.0) // number of subsequent upward steps before stopping
   hp += ("lambda", 0.01, 0.00) // regularization hyper-parameter

   // example adjustments - to be done before creating the neural network model
   hp("eta") = 0.05
   hp("bSize") = 25
   hp("maxEpochs") = 200
   hp("patience") = 5
   hp("lambda") = 0.02
   ```

   Note, in other packages patience is number of upward steps, not the number of subsequent upward steps. Below are additional neural network hyper-parameters for a future release of ScalaTion.

   ```scala
   hp += ("momentum", 0.9, 0.9) // momentum from prior gradients
   hp += ("dropout", 0.05, 0.05) // probability of neuron dropping out
   hp += ("valSplit", 0.1, 0.1) // training-validation set split fraction
   ```

7. **Tuning the Network Architecture:** The architecture of the neural network can be tuned by

   (a) changing the number of layers,
   (b) changing the number of nodes in each hidden layer, and
   (c) changing the activation functions.

   Tune the architecture for the AutoMPG dataset. The number of layers and number of nodes in each layer should only be increased when there is non-trivial improvement.

8. **Feature Selection for Neural Networks.** Although Neural Networks may be used without Feature Selection, it can still be useful to consider. An improvement over Forward Selection and Backward Elimination is Stepwise Regression. Start with no variables in the model and add one variable that improves the selection criterion the most. Add the second best variable for step two. After the second step determine whether it is better to add or remove a variable. Continue in this fashion until no improvement in the selection criterion is found. For Forward Selection and Backward Elimination it may instructive to continue all the way to the end (all variables for forward/no variables for backward).
Stepwise regression may lead to coincidental relationships being included in the model, particularly if a penalty-free QoF measure such as $R^2$ is used. Typically, this approach is used when there a penalty for having extra variables/parameters, e.g., $R^2$ adjusted $\bar{R}^2$, $R^2$ cross-validation $R^2_{cv}$ or Akaike Information Criterion (AIC). See the section on Maximum Likelihood Estimation for a definition of AIC. Alternatives to Stepwise Regression include Lasso Regression ($\ell_1$ regularization) and to a lesser extent Ridge Regression ($\ell_2$ regularization).

Perform Forward Selection, Backward Elimination, and Stepwise Regression with all four criteria: $R^2$, $\bar{R}^2$, $R^2_{cv}$, and AIC. Plot the curve for each criterion, determine the best number of variables and what these variables are. Compare the four criteria.

As part of a larger project compare this form of feature selection with that provided by Ridge Regression and Lasso Regression.

Use the following types of models: TranRegression, Perceptron, NeuralNet_3L, and NeuralNet_XL (4 layers).

In addition to the AutoMPG dataset, use the Concrete dataset and three more datasets from UCI Machine Learning Repository. The UCI datasets should have more instances ($m$) and variables ($n$) than the first two datasets. The testing is to be done in ScalaTion and Keras.

9. **Double Descent in Deep Learning.** Conventional wisdom indicates that as more features, nodes, or parameters are added the network/model, the training loss (e.g., based on $mse$ or $sse$) will continue to decrease, but the testing loss (out-of-sample) will start to increase. Recent research suggests that as even more parameters are added to the network/model, the testing loss may begin to decrease for a second time. Try this with some of the datasets from the last question. Note, that a decrease in the loss function corresponds to an increase in $R^2$ for training or $R^2_{cv}$ for testing.
10.9 Convolutional Networks

One way to think about a Convolution Network (or Conv-Net) is to take an ordinary Neural Network such as a 2-layer Neural Network and add a special non-neural layer in front of it, as depicted in Figure 10.11. This special layer is called the convolutional layer (there may be more than one). Its purpose is to reduce the volume of input in such a way that the salient features are brought forward. When the input is huge, such as image data, it is important to reduce the amount of data sent to the last layers, now referred to as the Fully-Connected (FC) layers.

![Figure 10.11: High-Level View of a Convolutional Network](image)

The convolution layers will take the input and apply convolutional filtering along with pooling operations. Inputs to the convolutional layers may be 1D (vector), 2D (matrix) or 3D or higher (tensor). After the convolutional layers, a flattening layer is used to turn matrices/tensors into vectors. Whatever the dimensionality of the input, the output of the flattening layer will be one dimensional. The flattening layer will fully connect to the first FC layer using a parameter matrix for edge weights. In general, the box labelled Conv Layers may consist of multiple convolutional layers with multiple pooling layer interspersed, followed by a flattening layer. Some of the advantages of convolutional networks are listed below:

- Input Data Reduced for Subsequent Layers.
- Learning Involves Fewer Parameters.
- Faster Network Training Time.
- Bringing Salient Features to the Forefront.

For additional background, there are multiple basic introductions to convolutional networks including [51, 56]. In the next two sections, 1D and 2D convolutional networks will be presented.
10.10 1D Conv-Net

A 1D Conv-Net takes the simplest form of input, that of a one-dimensional vector. Such input is found in sequence data such as time-series, sound/audio and text. More well known are 2D Conv-Nets where the input is two-dimensional as in grayscale images. Color images would be 3D as there are typically three color channels.

Convolutional Networks can be deep and complex, but in their simplest form they may be viewed as a modified Fully-Connected Neural Network. The example shown in Figure 10.12 of a one-dimensional convolutional network replaces the first dense layer with a convolutional layer. There are two differences: First the connections are sparse (at least not full) and the weights/parameters are shared by all second layer nodes. Notice that each node $z_h$ is only connected to three input nodes forming a window of width $n_c = 3$ starting at index $h$,

$$x_{[h,n_c]} = x_{h:h+n_c-1} = [x_h, x_{h+1}, x_{h+2}] \quad \text{for } h = 0, 1, 2 \quad (10.82)$$

and that the parameters are always $c = [c_0, c_1, c_2]$. The second set of edges fully connect the $z$ nodes with the $y$ nodes.

![Figure 10.12: Simple 1D Conv-Net](image)

**Convolutional Filter**

As the edge weights are shared, rather than duplicating them for multiple edges, we may think of the vector $c = [c_0, c_1, c_2]$ as a convolutional filter (or cofilter) that is used to calculate second layer node values,

$$z_h = x_{[h,3]} \cdot c = x_{h:h+2} \cdot c \quad (10.83)$$
for each index value $h$ by shifting the window/slice $x_{h:h+2}$ and taking the dot product with the cofilter $c$. For example, if the input $x = [1, 2, 3, 4, 5]$ and the cofilter $c = [0.5, 1.0, 0.5]$, then

$$z = [4, 6, 8]$$

To keep things simple, we assume the bias $\alpha$ is zero and that the convolutional layer’s activation function $f_0$ is the identity function. The $z$ vector is then propagated through the fully connected layer to obtain predicted output $\hat{y}$.

$$\hat{y}_k = f_1(b_k \cdot z)$$

### 10.10.1 Model Equation

The calculation of second layer node values can be vectorized by introducing a convolutional operator $\ast_c$ that shifts the cofilter $c$ over the vector $x$, slicing and computing dot products, which are collected into vector $z$.

$$z = x \ast_c c$$

(10.84)

In cases where activation function $f_0$ is not the identity function, the equation becomes

$$z = f_0(x \ast_c c)$$

(10.85)

SCALATion’s conv function in object CoFilter.1D implements this convolution operator $\ast_c$. Note, the subscript $c$ is used to avoid confusion with element-wise vector multiplication.

```scala
def conv (x: VectoD, c: VectoD): VectoD =
{
  val nc = c.dim
  val phi = new VectorD (x.dim + 1 - nc)
  for (j <- phi.range) phi(j) = x.slice (j, j+nc) dot c
  phi
} // conv
```

The second equation takes the $z$ vector and propagates it through the Fully-Connected (FC) component of the network

$$y = f_1(B \cdot z) + \epsilon$$

(10.86)

where the $B$ matrix is a 3-by-2 matrix in the example.

Recall the model equation for NeuralNet.3L written in vector form.

$$y = f_1(B \cdot f_0(A \cdot x)) + \epsilon$$

For 1D Conv-Nets, the model equation is very similar.

$$y = f_1(B \cdot f_0(x \ast_c c)) + \epsilon$$

(10.87)
Rather than taking the dot product with a matrix of unshared parameters/weights $A$, the input vector $x$ undergoes convolution with cofilter vector $c$. Adding in the shared scalar bias $\alpha$ for the convolutional layer and the unshared bias vector $\beta$ for the fully connected layer gives

$$y = f_1(B \cdot f_0(x \ast_c c + \alpha) + \beta) + \epsilon$$

(10.88)

When input vector $x$ is large it may be useful to further reduce the vector sizes by pooling, e.g., max-pooling with stride $s = 2$, would take the max of two adjacent elements and then move on to the next two. SCALATion’s `pool` function in object CoFilter.1D implements the pooling operator.

```scala
def pool (x: VectorD, s: Int = 2): VectorD =
{
  val p = new VectorD (x.dim / s)
  for (j <- p.range) {
    val jj = s * j
    p(j) = x.slice (jj, jj+s).max ()
  } // for
  p
} // pool
```

## 10.10.2 Training

Training 1D Convolutional Networks is more complicated, although more efficient, than training Fully-Connected Neural Networks [51]. Training involves finding values for the parameters $B \in \mathbb{R}^{n_z \times n_y}$, $\beta \in \mathbb{R}^{n_y}$, $c \in \mathbb{R}^{n_c}$ and $\alpha \in \mathbb{R}$ that minimize a given loss function $h$. To simplify the development, the biases will be ignored.

For a single input vector $x$ yielding

$$\hat{y} = f_1(B \cdot f_0(x \ast_c c))$$

(10.89)

the $\ell_2$ loss function will be

$$h(B, c) = \frac{1}{2}[(y - \hat{y}) \cdot (y - \hat{y})]$$

(10.90)

## 10.10.3 Optimization

As with NeuralNet.3L, there are basically four steps in optimization.

1. **Compute predicted values:** Based on the randomly assigned weights to vector $c$ and matrix $B$, predicted outputs $\hat{y}$ are calculated. First values for the hidden layer $z$ are calculated, where the value for hidden node $h$, $z_h$, is given by

$$z_h = f_0(x_{[h:n_z]} \cdot c) \quad \text{for } h = 0, \ldots, n_z - 1$$

where $f_0$ is the first activation function (e.g., reLU), $c$ is the convolutional filter which contains the shared weights/parameters, and $x$ is an input vector for a training sample/instance (row in the data
matrix). Next, the values computed at the hidden layer are used to produce predicted outputs \( \hat{y} \), where the value for output node \( k \), \( \hat{y}_k \), is given by

\[
\hat{y}_k = f_1(b_k \cdot z) \quad \text{for } k = 0, \ldots, n_y - 1
\]

where the second activation function \( f_1 \) may be the same as (or different from) the one used in the hidden layer and \( b_k \) is column-\( k \) of the \( B \) weight matrix. Now the difference between the actual and predicted output, the error, for the \( k^{th} \) output, \( \epsilon_k \), is given by

\[
\epsilon_k = y_k - \hat{y}_k \quad \text{for } k = 0, \ldots, n_y - 1
\]

2. **Back propagate from output layer**: Given the computed error vector \( \epsilon \), the delta/correction vector \( \delta^1 \) for the output layer may be calculated, where for output node \( k \), \( \delta^1_k \) is given by

\[
\delta^1_k = [-\epsilon_k] f'_1(b_k \cdot z) \quad \text{for } k = 0, \ldots, n_y - 1 \tag{10.91}
\]

where \( f'_1 \) is the derivative of the activation function. The partial derivative of the loss function \( h \) with respect to the weight connecting hidden node \( h \) with output node \( k \), \( b_{hk} \), is given by

\[
\frac{\partial h}{\partial b_{hk}} = z_h \delta^1_k \tag{10.92}
\]

3. **Back propagate from hidden layer**: Given the delta/correction vector \( \delta^1 \) from the output layer, the delta vector for the hidden layer \( \delta^0 \) may be calculated, where for hidden node \( h \), \( \delta^0_h \) is given by

\[
\delta^0_h = [b_h \cdot \delta^1] f'_0(x_{[h, n_c]} \cdot c) \quad \text{for } h = 0, \ldots, n_z - 1 \tag{10.93}
\]

The partial derivative of \( h \) with respect to the \( j^{th} \) weight in the convolutional filter, \( c_j \), is given by

\[
\frac{\partial h}{\partial c_j} = \sum_{h=0}^{n_k-1} x_{j+h} \delta^0_h = x_{[j, n_z]} \cdot \delta^0 \tag{10.94}
\]

4. **Update weights**: The weights \( c \) and \( B \), in the convolutional and full-connected layers, respectively, may now be updated based on the partial derivatives. For gradient descent, movement is in the opposite direction, so the sign flips from positive to negative. These partial derivatives are multiplied by the learning rate \( \eta \) which moderates the adjustments to the weights.

\[
b_{hk} -= z_h \delta^1_k \eta \tag{10.95}
\]

\[
c_j -= x_{[j, n_z]} \cdot \delta^0 \eta \tag{10.96}
\]

349
10.10.4 Matrix Version

1. The hidden (latent) values for all \( m \) instances and all \( n_z \) hidden nodes are computed by applying the first matrixized activation function \( f_0 \) to the convolution \( X \ast_c c \) to produce the latent feature matrix \( Z \in \mathbb{R}^{m \times n_z} \).

\[
Z = f_0(X \ast_c c)
\]  

(10.97)

The predicted output matrix \( \hat{Y} \in \mathbb{R}^{m \times n_y} \) is similarly computed by applying the second matrixized activation function \( f_1 \) to the matrix product \( ZB \).

\[
\hat{Y} = f_1(ZB)
\]  

(10.98)

2. The negative of the error matrix \( E \in \mathbb{R}^{m \times n_y} \) is just the difference between the predicted and actual/target values.

\[
E = \hat{Y} - Y
\]  

(10.99)

3. This information is sufficient to calculate delta matrices: \( \Delta^1 \) for adjusting \( B \) and \( \Delta^0 \) for adjusting \( A \). The output-layer delta matrix \( \Delta^1 \in \mathbb{R}^{m \times n_y} \) is the elementwise matrix (Hadamard) product of \( E \) and \( f'_1(ZB) \).

\[
\Delta^1 = E \circ f'_1(ZB)
\]  

(10.100)

The hidden-layer delta matrix \( \Delta^0 \in \mathbb{R}^{m \times n_z} \) is the elementwise matrix (Hadamard) product of \( \Delta^1B^\top \) and \( f'_0(XA) \).

\[
\Delta^0 = [\Delta^1B^\top] \circ f'_0(X \ast_c c)
\]  

(10.101)

4. As mentioned, the delta matrices form the basis (a matrix transpose \( \times \) delta \( \times \) the learning rate \( \eta \)) for updating the parameter/weight matrix \( B \) and cofilter vector \( c \).

\[
B \leftarrow Z^\top \Delta^1 \eta
\]  

(10.102)

\[
c_j \leftarrow [X^\top_{\ast,j,n_z}\Delta^0]_{\text{mean}} \eta
\]  

(10.103)

10.10.5 Gradient Descent Algorithm

The basic gradient descent algorithm adapts the above equations into a loop to iteratively update the weight matrices and bias vectors (unified via the \texttt{NetParam} class). The corresponding \texttt{ScalaTion} code for the \texttt{train0} method is shown below. Note: \( * \): defined in \texttt{NetParam} computes value-matrix \( \ast \) weight-matrix + bias-vector.
def train0 (x_ : MatriD = x, y_ : MatriD = y): ConvNet_1D =
{
    println (s"train0: eta = $eta")
    var sse0 = Double.MaxValue // hold prior value of sse

    for (epoch <- 1 to maxEpochs) { // iterate over each epoch
        var z = f0.fM (conv (x_, c)) // Z = f(conv (X, c))
        var yp = f1.fM (z *: b) // Yp = f(ZB)
        ee = yp - y_ // negative of the error matrix
        val d1 = ee ** f1.dM (yp) // delta matrix for y
        val d0 = (d1 * b.w.t) ** f0.dM (z) // delta matrix for z
        ConvNet_1D.updateParam (x_, z, d0, d1, eta, c, b)
        val yp_ = f1.fM (f0.fM (conv (x, c)) *: b) // updated predictions
        val sse = sseF (y_, yp_)
        if (sse > sse0) return this // return early if moving up
        sse0 = sse // save prior sse
    } // for
    this
} // train0

The updateParam method produces new values for c and b.

def updateParam (x_ : MatriD, z: MatriD, d0: MatriD, d1: MatriD, eta: Double,
       c: VectoD, b: NetParam) =
{
    for (j <- c.range) {
        var sum = 0.0
        for (i <- x_.range1; h <- z.range2) sum += x_(i, h+j) * d0(i, h)
        c(j) -= (sum / x_.dim1) * eta // update 'c' weights in conv filter
    } // for
    b -= (z.t * d1 * eta, d1.mean * eta) // update 'b' weights & biases
} // updateParam

10.10.6 Example Error Calculation Problem

Given the following problem, compute the negative error matrix.

val x = new MatrixD ((2, 5), 1, 2, 3, 4, 5,
6, 7, 8, 9, 10)
val y = new MatrixD ((2, 2), 6, 9,
16, 24)
val c = VectorD (0.5, 1, 0.5)
val b = NetParam (new MatrixD ((3, 2), 0.1, 0.2,
0.3, 0.4,
0.5, 0.6))

Use the following code to compute the negative error matrix E.
val z = f0.fM (conv (x, c))                  // Z = f(conv (X, c))
val yp = f1.fM (z *: b)                    // Yp = f(2Z)
val ee = yp - y                            // negative error E = Yp - Y
println (s"ee = $ee")

10.10.7 Two Convolutional Filters

Typically, a convolutional layer may have multiple cofilters. Figure [10.13] has two cofilters: The first cofilter incident upon the yellow nodes in the middle has weights $c^0 = [c_{00}^0, c_{10}^0, c_{20}^0]$, while the second cofilter incident upon the orange nodes in the middle has weights $c^1 = [c_{01}^1, c_{11}^1, c_{21}^1]$. The superscripts are left off in the figure due to crowding of edge labels.

![Figure 10.13: A 1D Conv-Net with Two Convolution Filters $c^0$ and $c^1$](image)

The cofilter $c^0$ will move over the input nodes producing feature map $\phi^0$ (the yellow nodes in the middle), while cofilter $c^1$ will also move over the input nodes producing feature map $\phi^1$ (the orange nodes in the middle). As there are no other layers before the fully-connected layer, these two feature maps are combined
(or flattened) and activated to form a hidden \((z = [z_0, z_1, z_2, z_3, z_4, z_5])\) layer. The stride indicates how far the cofilter moves (e.g., down one node, down two nodes) on each step. Here the stride is equal to one.

Figure 10.14 shows the same Convolutional Network from the point of view of tensors flowing through the network (in this case the tensors are just vectors). Cofilter \(c^0\) moves through the input vector \(x\) taking the dot product of itself with the blue, purple/crimson and black windows/slices, respectively. These three dot products produce the values in feature map \(\phi^0\). (Note that blue line from cofilter \(c^0\) to feature map \(\phi^0\) has utilized all the values in the cofilter and all values in the blue window and is shown as a single line to reduce clutter.) The same logic is used for the second cofilter \(c^1\) to create feature map \(\phi^1\). The difference lies only in the values in each of these two convolution vectors.

Now the two feature maps \(\phi^0\) and \(\phi^1\) need to be combined. In general this is done by a flattening operation. For the 1D case, the feature maps are already vectors so flattening reduces to vector concatenation. This flattened vector is activated using \(f_0\) to produce \(z\) that serves as an entry point into the fully-connected part of the network. Further computations produce the output vector \(\hat{y}\) based on vector \(z\) and parameter matrix \(B\), with their product activated using \(f_1\).

\[
\begin{align*}
z &= f_0(\cdot) \\
\hat{y} &= f_1(\cdot)
\end{align*}
\]

Figure 10.14: Tensor Flow Diagram Showing Two Convolution Filters \(c^0, c^1\) and Two Feature Maps \(\phi^0, \phi^1\) followed by a Fully-Connected Layer

A Convolutional Network may have multiple convolutional layers as well as pooling layers (discussed in more detail in the next section). Activation (e.g., using ReLU) occurs at the end of paired convolutional-pooling layers. In addition, the fully-connected part may consist of multiple layers.
ConvNet_1D Class

Class Methods:

@param x the input/data matrix with instances stored in rows
@param y the output/response matrix, where y_i = response for row i of matrix x
@param nf the number of filters for this convolutional layer
@param nc the width of the filters (size of cofilters)
@param fname_ the feature/variable names (if null, use x_j's)
@param hparam the hyper-parameters for the model/network
@param f0 the activation function family for layers 1->2 (input to hidden)
@param f1 the activation function family for layers 2->3 (hidden to output)
@param itran the inverse transformation function returns responses to original scale

class ConvNet_1D (x: MatrixD, y: MatrixD, nf: Int = 1, nc: Int = 3,
fname_ : Strings = null, hparam: HyperParameter = hp,
f0: AFF = f_reLU, f1: AFF = f_reLU,
val itran: FunctionV_2V = null)
extends PredictorMat2 (x, y, fname_, hparam)

def filter (i: Int, f: Int): VectorD =
def updateFilterParams (f: Int, vec2: VectorD)
def parameters: NetParams = Array (NetParam (MatrixD (Seq (c))), b)
def train0 (x_ : MatrixD = x, y_ : MatrixD = y): ConvNet_1D =
def train (x_ : MatrixD = x, y_ : MatrixD = y): ConvNet_1D =
def predictV (v: VectorD): VectorD = f1.fV (b dot f0.fV (conv (v, c)))
def predictV (v: MatrixD = x): MatrixD = f1.fM (f0.fM (conv (v, c)) *: b)
def buildModel (x_cols: MatrixD): ConvNet_1D =

10.10.8 Exercises

1. Delta Vectors: For the example error calculation problem given in this section, calculate the \( \delta^1 \) and \( \delta^0 \) vectors.

2. Parameter Update Equations: For the same example, use the \( \delta^1 \) vector to update weight matrix \( B \) for the fully-connected layer. Use the \( \delta^0 \) vector to update weight vector \( c \) for the convolutional layer.

3. For the same example, make four training steps and see what happens to \( sse \).

4. Create a ConvNet_1D model for the AutoMPG dataset.

   val cn = new ConvNet_1D (x, MatrixD (y))

5. Create and demo a 1D Conv-Net for the MIT-BIH Arrhythmia Database using SCALATION’s ConvNet_1D class and Keras. See [https://physionet.org/content/mitdb/1.0.0](https://physionet.org/content/mitdb/1.0.0).
10.11 2D Conv-Net

Two-dimensional Convolution Networks (2D Conv-Nets) are needed when the data instances are two-dimensional and based on a regular grid pattern where the values in the grid can be mapped to a matrix. For 2D Conv-Nets, flattening operators are added to the convolutional and pooling operators. The following references focus on 2D Convolutional Networks [48, 117]. To clarify what can happen in a 2D Conv-Net convolutional layer, consider the well-known MNIST problem to classify hand written digits 0 to 9. Given 10,000 grayscale images, each having 28-by-28 pixels, the goal is to train a convolutional network to classify an image as one of 10 digits. The following simple network architecture may be used as a starting point for the MNIST problem:

- **Convolutional Layer.** A convolutional layer consists of one or more convolutional filters, for which a convolutional operation is applied to the input for each filter, conceptually in parallel. The result of the convolutions may then be passed into an activation function. For this problem, assume there is just one 5-by-5 convolutional filter.

- **Pooling Layer.** A pooling layer takes the output of a convolutional layer and reduces/aggregates it. A common pooling operation is generally applied to the results of the convolution. For this problem, assume there is a 2-by-2 max-pooling operation applied to the one filter.

- **Flattening Layer.** Between the convolutional layers and fully-connected layers a flattening layer turns the matrices (or tensors for 3D Conv-Nets) flowing through the convolutional layers and pooling layers into vectors as required by the fully-connected layers.

- **Fully Connected Output Layer.** A 10 node layer where the $i^{th}$ node represents the $i^{th}$ digit. Whichever output node gets the highest score is the classification result. In general, there may be multiple fully-connected layers.

Advanced network architectures may have many convolutional layers and pooling layers. Since one layer comes after the other, the signals are processed in series (not parallel). For this problem, assume there is just one convolutional layer.

10.11.1 Filtering Operation

A filter is a small matrix, e.g., a 5-by-5 matrix that scans across the image taking the dot product of its values with the values in a 5-by-5 window of the image, as depicted in Figure 10.15. Commonly a (convolutional) stride of 1 is used that moves the window one position/pixel at a time. This process produces a 24-by-24 result matrix referred to as a feature map. Padding can be used to maintain the size of the image. The commonly used zero-padding would put zeros around the image. Each of the filters will produce its own feature map.

The MNIST dataset consisting of many images can be collected into a 3-dimensional tensor $X$. The $i^{th}$ row (or renamed as a sheet) of the tensor $X$, $X_i$, will be a 28-by-28 input matrix.

Instead of looking at a large input matrix $X$, consider one of the matrices given on the following Webpage [https://cs231n.github.io/convolutional-networks](https://cs231n.github.io/convolutional-networks). It is a 5-by-5 matrix, extracted below.
The effect of applying the following 2-by-2 convolutional filter matrix $C$ to $X$, \[ C = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \] (10.105)
is produced by using the convolution operator ($*_c$), which will give the following feature map matrix $\Phi$, \[ \Phi = X *_c C \] (10.106)where each element in the resulting 4-by-4 feature map matrix $\Phi$ is the dot product of a subimage and the convolutional filter. The subimage is a shifting 2-by-2 window/slice of the original image with top-left position $(j,k)$. \[ \phi_{jk} = X_{j:j+1,k:k+1} \cdot C \] (10.107)The resulting feature map matrix $\Phi$ is shown below. \[ \Phi = \begin{bmatrix} 0 & 2 & 4 & 3 \\ 2 & 2 & 1 & 5 \\ 3 & 4 & 2 & 3 \\ 4 & 2 & 0 & 2 \end{bmatrix} \] (10.108)Notice for this example, if a reLU activation function is applied, the final feature map is unaltered. SCALATION’s `conv` function in the `Filter2D` object implements the convolution operator.
```scala
def conv (x: Matrix, c: Matrix): Matrix = {  
  val (m, n) = (c.dim1, c.dim2)  
}
```
val phi = new MatrixD(x.dim1 + 1 - m, x.dim2 + 1 - n)
for (j <- phi.range1; k <- phi.range2) {
    phi(j, k) = (x.slice(j, j+m, k, k+n) ** c).sum
} // for
phi
} // conv

10.11.2 Pooling Operation

Each feature map can be further reduced to a pooled feature map using a pooling operation. Max pooling takes the maximum value in each non-overlapping 2-by-2 region as the value for the pooled feature map. Each of the pooled feature maps will be 12-by-12 matrices. All these values will be placed in the flattening layer consisting of $12 \times 12 = 144$ nodes.

Using 2-by-2 max pooling, the maximum of each non-overlapping 2-by-2 submatrix is taken.

$$p_{jk} = \max \Phi_{2j:2j+1,2k:2k+1}$$

(10.109)

to produce the 2-by-2 matrix $P$. Note, having a pooling stride of 2 will make the regions non-overlapping.

$$P = \begin{bmatrix} 2 & 5 \\ 4 & 3 \end{bmatrix}$$

(10.110)

SCALATION's pool function in the Filter2D object implements the pooling operator.

```scala
def pool (x: MatrixD, s: Int = 2): MatrixD = 
{
    val p = new MatrixD(x.dim1 / s, x.dim2 / s)
    for (j <- p.range1; k <- p.range2) {
        val (jj, kk) = (s*j, s*k)
        p(j, k) = x.max (jj until jj+s, kk until kk+s)
    } // for
    p
} // pool
```

10.11.3 Flattening Operation

Flattening takes either a matrix or tensor and forms a single vector that includes all the elements. A matrix may be flattened in either row-major or column-major style. The SCALATION method in MatrixD shown below flattens a matrix in row-major fashion.

```scala
def flatten: VectorD = 
{
    val u = new VectorD (dim1 * dim2)
    var k = 0
    for (i <- range1; j <- range2) { u(k) += v(i)(j); k += 1 }
    u
} // flatten
```
Therefore, the flattened pooled matrix $P$ is

$$\text{P.flatten} = [2, 5, 4, 3]$$

### 10.11.4 Model Equation

The first equation for 2D Conv-Nets computes the intermediate $z$ vector for a particular image, $X$, by taking the following steps: (i) convolve $X$ with convolutional filter $C$, (ii) apply the first activation function $f_0$ (e.g., reLU), (iii) pool to reduce size, and (iv) flatten matrix into a vector.

$$z = \text{[pool}(f_0(X * _c C))\text{].flatten} \quad (10.111)$$

The second equation takes the $z$ vector and propagates it through the Fully-Connected (FC) layer of the network. The model equation includes these two steps with an error term added.

$$y = f_1(B \cdot z) + \epsilon \quad (10.112)$$

where the $B$ matrix connects each of the 144 nodes in the flattened layer to the 10 output nodes of the FC layer. Combining these two equations yields

$$y = f_1(B \cdot \text{[pool}(f_0(X * _c C))\text{].flatten}) + \epsilon \quad (10.113)$$

More generally, multiple convolutional filters (along with their pooling) can be used simultaneously. See the exercises for an example with multiple convolutional filters. In addition, multiple combination convolution-pooling layers are often used, as in Deep Learning. The fully-connected part of the network may also consist of multiple layers as well.

### 10.11.5 Training

Training involves finding values for the parameters $B \in \mathbb{R}^{n_z \times n_y}$, $\beta \in \mathbb{R}^{n_y}$, $C \in \mathbb{R}^{n_c \times n_c}$ and $\alpha \in \mathbb{R}$ that minimize a given loss function $h$. To simplify the development, the biases will be ignored.

For a single input matrix $X$ yielding

$$\hat{y} = f_1(B \cdot \text{[pool}(f_0(X * _c C))\text{].flatten}) \quad (10.114)$$

the $\ell_2$ loss function will be

$$h(B, C) = \frac{1}{2}[(y - \hat{y}) \cdot (y - \hat{y})] \quad (10.115)$$

### 10.11.6 Optimization

### 10.11.7 Exercises

1. Develop the modeling equations for a 2D Conv-Net that consists of the following layers:

   (a) Convolutional Layer with two feature maps. $5 \times 5$ convolutional filters, no padding, stride 1 and reLU activation.

   (b) Pooling Layer using max-pooling. $2 \times 2$ pooling with a stride of 2.
(c) Flattening Layer to convert matrices into a vector.
(d) Fully-Connected Layer with 10 output nodes.

2. If the above network is used for a $28 \times 28$ MNIST image/matrix, give the dimensions for all components in the network. How many parameters will need to be optimized?


4. It is common in data science to normalize input data. This is especially true when there are nonlinear transformations. For Conv-Nets and deep learning, normalizing the initial input may be insufficient, so other types of normalization have been introduced, such as batch normalization, and layer normalization. Discuss the advantages and disadvantages of the various normalization techniques.

5. The convolution operator slides a window the size of the convolution filter over the image. For a dilated Conv-Net, when the dilation rate is 2, the window size is enlarged in each dimension since every other pixel is skipped. Given a 3-by-3 convolution filter, the window size (size of the receptive field in the image) is 3-by-3. In this case the dilation rate is 1. In general, a dilation rate of $n$, will mean $n - 1$ pixels will skipped for every pixel used. What would the window size if the dilation rate is 2?

6. Create a 3D Conv-Net that processes color images using Keras. It will take a four-dimensional input tensor $X = [x_{ijkl}]$ where $i$ indicates which image, $j$ indicates which (horizontal) row, $k$ indicates which (vertical) column, and $l$ indicates which color channel. Use the CIFAR-10 dataset [https://www.cs.toronto.edu/~kriz/cifar.html](https://www.cs.toronto.edu/~kriz/cifar.html).
10.12 Graph Neural Networks

The convolutional filters used in Conv-Nets allow one to extract hidden features by looking at a collection of nearby points. The notion of nearby may be thought of in terms of Euclidean distance.

Graphs can be used to provide a more versatile notion of what points are nearby. This may be done by counting hops, the number of edges traversed to get from one vertex to another. If distance is an edge property (given or derived) then nearby vertices may be defined as those within a certain distance, as shown in Figure 10.16.

There are many types of Graph Neural Networks (GNNs) [119, 112]. A GNN may be defined for several types of graphs. Consider the following slightly modified definition of a vertex-labeled directed graph as a four-tuple \( G(V, E, d, x) \) where

- \( V \) = set of vertices
- \( E \subseteq V \times V \) set of directed edges
- \( d = \) dimensionality of the property vectors
- \( x : V \rightarrow \mathbb{R}^d \)

Hence, \( x_v \) denotes a property/attribute vector associated with vertex \( v \). In general, a GNN could also use values on edges. The following definitions are needed for understanding how GNNs work [114].

**Neighborhood**

The neighborhood of vertex \( v \in V \) is given as follows for a directed graph.

\[
\mathcal{N}(v) = \{u|(v, u) \in E\} \cup \{u|(u, v) \in E\}
\]  

(10.116)

The neighborhood of vertex \( v_4 \) is \( \{v_1, v_3\} \), where \( v_1 \) is an upstream vertex and \( v_3 \) is a downstream vertex.

10.12.1 AGGREGATE and COMBINE Operations

As with Conv-Nets, the initial processing of data occurs with non-neural layers. For each layer \( l \), data from a vertex is mixed with its neighbors’ data using the AGGREGATE and COMBINE operations [36, 69].
AGGREGATE

The values for the neighborhood of vertex $v$ are aggregated to form a vector of values representing the neighborhood.

$$x_{N(v)} = \text{AGGREGATE}\{x_u^{(l-1)}|u \in N(v)\} \quad (10.117)$$

The AGGREGATE operation may be as simple as element-wise mean or max-pooling.

COMBINE

A new value for vertex $v$, $x_v^{(l)}$ is calculated by combining the previous vector value for vertex $v$, $x_v^{(l-1)}$, and the vector value produced from $v$’s neighborhood.

$$x_v^{(l)} = \text{COMBINE}(x_v^{(l-1)}, x_{N(v)}) \quad (10.118)$$

The COMBINE operation may be as simple as element-wise mean or vector addition (+).

Example from Traffic Domain

As an example, consider applying a Graph Neural Network to vehicle traffic flow on a highway system. Let the vertices in the graph represent sensors. At a particular time $t$, the sensor records the flow (vehicles in last 5 minutes) and speed (in mph). The neighborhood may capture the notion of upstream flow past one sensor and downstream flow past another sensor. Suppose the initial property/attribute vectors are the following:

<table>
<thead>
<tr>
<th>$v$</th>
<th>flow</th>
<th>speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v_4$</td>
<td>99</td>
<td>66</td>
</tr>
<tr>
<td>$v_1$</td>
<td>90</td>
<td>68</td>
</tr>
<tr>
<td>$v_3$</td>
<td>80</td>
<td>72</td>
</tr>
</tbody>
</table>

Applying an element-wise mean AGGREGATE operation gives

$$x_{N(v_4)} = \text{AGGREGATE}\{x_u^{(0)}|u \in \{1, 3\}\} = [85, 70]$$

Applying an element-wise mean COMBINE operation gives

$$x_v^{(1)} = \text{COMBINE}(x_v^{(0)}, x_{N(v)}) = [93, 67]$$

Typically, the COMBINE operation includes a learnable weight/parameter matrix $B$ and an activation function $f$ (e.g., reLU).

$$x_v^{(l)} = f(B[x_v^{(0)}, x_{N(v)}]) \quad (10.119)$$

$f$ is the vectorization of activation function $f$. The weights/parameters in the GNN indicate the importance of data from upstream and downstream sensors in, for example, classifying, predicting or forecasting results at a particular sensor. Pooling can also be used to reduce the size of the vectors.
SCALATiON WILL supports Graph Neural Networks of the spatial variety, so those of the spectral variety are not considered (see [112] for a comparison).

- **Group Lectures 5: Graphs and Machine Learning, November 3, 5, 9**

  Include Graph Neural Networks.

- **Group Lectures 6: Knowledge Graphs in Machine Learning, November 10, 12, 16**

  Include the use of knowledge, especially in the form of knowledge graphs, in machine learning.
10.13 Transfer Learning

Transfer learning is applicable to many types of machine learning problems [74]. In this section, we focus on transfer learning for neural networks [103].

The first sets of transformations (from input to the first hidden layer and between the first and second hidden layers) in a Neural Network allow nonlinear effects to be created to better capture characteristics of the system under study. These are taken in combination in later latter layers of the Neural Network. The thought is that related problems share similar nonlinear effects. In other words, two datasets on related problems used to train two Neural Networks are likely to develop similar nonlinear effects at certain layers. If this is the case, the training of the first Neural Network could expedite the training of the second Neural Network. Some research has shown the Quality of Fit (QoF) may also be enhanced as well.

The easiest way to imagine this is to have two four-layer Neural Networks, say with 30 inputs for the first and 25 for the second. Let the first hidden layer have 50 nodes and second have 20 nodes, with an output layer for the single output/response value. The only difference in node count is in the input layer. For the first Neural Network, the sizes of the parameter/weight matrices are 30-by-50, 50-by-20 and 20-by-1. The only difference in the second Neural Network is that the first matrix is 25-by-50. After the first Neural Network is trained, its second matrix (50-by-20) along with its associated bias vector could be transferred to the second Neural Network. When training starts for the second Neural Network, random initialization is skipped for this matrix (and its associated bias vector). A training choice is whether to freeze this layer or allow its values to be adjusted during back-propagation.

10.13.1 Definition of Transfer Learning

In its simplest form transfer learning involves two related datasets and two related tasks.

- **Dataset.** A dataset $D$ may be defined as an ordered pair of tensors $D = (X, Y)$ where $X$ is the input tensor and $Y$ is the output tensor. It is assumed that there is an unknown function $f$ and a noise generator $\epsilon$ that characterizes the relationship between input tensor $X$ and output tensor $Y$.

  \[
  Y = f(X) + \epsilon \tag{10.120}
  \]

- **Task.** In a limited sense, a task $\tau$ may be viewed a procedure to produce a model function $f_m$ that approximates $f$. Secondary goals include interpretability and generalizability. The closeness of the approximation can be measured by some norm of the differences between the functions or more generally by a task inspired loss function (e.g., MSE for regression taks or cross entropy for classifications tasks),

  \[
  f_m = \min\{|f_k : \|f - f_k\| \text{ for } f_k \in \mathbb{F}\} \tag{10.121}
  \]

  where $\mathbb{F}$ is a function space. For example, if the input tensor is two-dimensional (a matrix) and the output tensor is one-dimensional (a vector), $\mathbb{F}$ could be the set of all linear transformations of the column space of the input matrix to $\mathbb{R}$. The goal of multiple linear regression is to find a point in the column space of matrix $X$ closest to the vector $y$.

- **Transfer Learning.** Now given two datasets $D_1$ and $D_2$, and two tasks $\tau_1$ and $\tau_2$, the question is whether the efforts in task $\tau_1$ can reduce the effort and/or improve the quality of outcome of task $\tau_2$. 

363
Transfer learning is ideally applied when both the datasets and the tasks are similar (or related in some sense) and when the first dataset is large enough to support accurate training. Applying transfer learning to the second task may be motivated by limited time/computational resources or lack of data. The lack of data often happens for classification tasks with missing labels ($y$ values).

### 10.13.2 Type of Transfer Learning

Transfer Learning can involve domain adaptation and instance selection. For neural networks, components of one network developed for task $\tau_1$ may be transferred to another network under development for task $\tau_2$.

#### Transfer Learning in Neural Networks

An area in which transfer learning has demonstrated considerable success is in Convolutional Networks. Therefore, consider a dataset where the input is a three-dimensional tensor $X \in \mathbb{R}^{m \times w \times h}$, where $m$ is the number of grayscale images, $w$ is their pixel width and $h$ is their pixel height. For image classification, the output is a one-dimensional vector $y \in \{0, \ldots, K - 1\}^m$, where $K$ is the number of classes in classification $C$.

For simplicity, assume that $m_2 < m_1$, $w_2 = w_1$, $h_2 = h_1$, and $K_2 \leq K_1$. We are left with two issues: how similar/related are the two sets of images and what is the logical/semantic connection between classification $C_2$ and classification $C_1$. The similarity of classification schemes could be based on the fraction of classes in $C_2$ that are also in $C_1$.

$$r = \frac{|C_2 \cap C_1|}{|C_2|}$$

(10.122)

This can be refined when there is an underlying ontology to provide meaning to the classifications and their class labels, as well as, a metric for semantic distance. Note, some pre-trained models may have been trained to discriminate between hundreds or thousands of image types, e.g., ImageNet [52].

Selection of a source dataset may now be decomposed into the following steps:

1. **Initial Screening.** Initial screening of potential candidate source datasets can be done by comparing their similarity to that of the target dataset $D_2$. Information divergence (e.g., KL divergence) or distances based on optimal transport may be used to explore the similarity of datasets (or their underlying probability measures). Source datasets sufficiently similar and with pre-trained models should be tested in step 2.

2. **Choosing Candidates.** Datasets identified from step 1 may be selected as candidates by applying them to the target problem in the following way. Replace the last fully-connected layer of the source convolutional network with a new one, where the output layer is reduced from $K_1$ to $K_2$ nodes. Call this the adapted model. Freeze all the previous layers and train the last layer from scratch. A layer being frozen means that its parameters (weights and biases) will not be changed during back-propagation. Choose the models with better Quality of Fit (QoF) measures or lower loss functions.

3. **Fine-Tuning.** Let $D_1$ be one of the candidate datasets. One option would be to train the adapted model/network from scratch. Unfortunately, this would be time consuming at best and likely to fail at worst due to lack of class labels. Retraining involves a choice for each layer, whether to freeze, fine-tune, or train from scratch. Making the right choice for each layer can lead to a more accurate model. Unfortunately, a convolution network with 16 trainable layers, such as VGG-16 [102], would
require $3^{16} = 43,047,721$ models to train. Therefore, establishing criteria for deciding how to handle each layer is important. Although it tends to be generally the case, that the later layers are specific to the given task and that earlier layers are more generic and hence transferable as is, recent research has shown benefits in fine-tuning some early layers [34].

Domain Adaptation

To summarize, given two datasets $D_1 = (X_1, Y_1)$ and $D_2 = (X_2, Y_2)$, and two tasks $\tau_1$ and $\tau_2$, utilize trained model/network $f_{m_1}$ to produce a new model $f_{m_2}$ by reusing much of $f_{m_1}$ to make training have similar or better outcomes in less time.

Typically in domain adaptation, we may assume that $\tau_1 = \tau_2$ and the task is classification (thus, the outputs are vectors). It may be further assumed that a label mapping function can be defined so that class labels in classification 1 can be mapped without loss of information to labels in classification 2. Thus, we may stipulate that the output vectors are points in the following label spaces.

\[
y_1 \in [0, \ldots, K-1]^{m_1} \quad (10.123)
\]
\[
y_2 \in [0, \ldots, K-1]^{m_2} \quad (10.124)
\]

The focus in domain adaptation is on applying transformations on the input datasets, $X_1$ and $X_2$. Let the form of inputs now be reduced to matrices $X_1 \in \mathbb{R}^{m_1 \times n_1}$ and $X_2 \in \mathbb{R}^{m_2 \times n_2}$. To facilitate visualization, let the number of columns in both matrices be 2 ($n_1 = n_2 = 2$). Now, two-dimensional histograms may be produced for each input data matrix.

Instance Selection

NeuralNet_XLT Class

Class Methods:

- `@param x` the m-by-nx input matrix (training data consisting of m input vectors)
- `@param y` the m output vector (training data consisting of m output integer values)
- `@param nz` the number of nodes in each hidden layer, e.g., Array (9, 8) => sizes 9 and 8
- `@param fname_` the feature/variable names (if null, use x_j’s)
- `@param hparam` the hyper-parameters for the model/network
- `@param f` the array of activation function families between every pair of layers
- `@param l_trans` the first layer to be transferred in
- `@param transfer` the saved network parameters from layers of a related neural network
- `@param itran` the inverse transformation function returns responses to original scale

```text
class NeuralNet_XLT (x: MatriD, y: MatriD,
    nz: Array [Int], fname_ : Strings = null,
    hparam: HyperParameter = hp,
    f: Array [AFF] = Array (f_tanh, f_tanh, f_id),
    l_trans: Int = 1, transfer: NetParams = null,
    itran: FunctionV_2V = null)
```

365
extends NeuralNet_XL (x, y, nz, fname_, hparam, f, itran)

def trim (bl: NetParam, tl: NetParam): NetParam =

10.13.3 Exercises

1.
10.14 Extreme Learning Machines

An Extreme Learning Machine (ELM) may be viewed as three-layer Neural Network with the first set of fixed-parameters (weights and biases) frozen. The values for these parameters may be randomly generated or transferred in from a Neural Network. With the first set of fixed-parameters frozen, only the second set of parameters needs to be optimized. When the second activation function is the identity function (id), the optimization problem is the same as for the Regression problem, so that matrix factorization may be used to train the model. This greatly reduces the training time over Neural Networks. Although the first set of fixed-parameters is not optimized, nonlinear effects are still created at the hidden layer and there may be enough flexibility left in the second set of parameters to retain some of the advantages of Neural Networks.

10.14.1 Model Equation

A relatively simple form of Extreme Learning Machine in SCALATION is ELM_3L1. It allows for a single output/response variable \( y \) and multiple input/predictors variables \( x = [x_0, x_1, \ldots, x_{n-1}] \). The number of nodes per layer are \( n_x \) for input, \( n_z \) for hidden and \( n_y = 1 \) for output. The second activation function \( f_1 \) is implicitly id and be left out.

The model equation for ELM_3L1 can be written in vector form as follows,

\[
y = b \cdot f_0(A^T x) + \epsilon
\]  

(10.125)

where \( A \) is the first layer NetParam consisting of an \( n_x \)-by-\( n_z \) weight matrix and an \( n_z \) bias vector. In SCALATION, these parameters are initialized as follows (see exercises for details):

```scala
private var a = new NetParam (weightMat3 (n, nz, s),
weightVec3 (nz, s))
```

The second layer is simply an \( n_z \) weight vector \( b \). The first layer’s weights and biases \( A \) are frozen, while the second layer parameters \( b \) are optimized.

10.14.2 Training

Given a dataset \((X, y)\), training will be used to adjust values of the parameter vector \( b \). The objective is to minimize the distance between the actual and predicted response vectors.

\[
f_{obj} = ||y - b \cdot f_0(XA)||
\]  

(10.126)

10.14.3 Optimization

An optimal value for parameter vector \( b \) may be found using Regression.

```scala
def train (x_ : MatriD = x, y_ : VectoD = y): ELM_3L1 =
{
  val z = f0.fM (x_ *: a) // Z = f(XA)
  val rg = new Regression (z, y_)
  rg.train ()
  b = rg.parameter
```
Extreme Learning Machines typically are competitive with lower order polynomial regression and may have fewer parameters than CubicXRegression.

**ELM_3L1 Class**

**Class Methods:**

- @param x the m-by-n input matrix (training data consisting of m input vectors)
- @param y the m output vector (training data consisting of m output scalars)
- @param nz the number of nodes in hidden layer (-1 => use default formula)
- @param fname_ the feature/variable names (if null, use x_j’s)
- @param hparam the hyper-parameters for the model/network
- @param f0 the activation function family for layers 1->2 (input to hidden)
- @param itran the inverse transformation function returns responses to original scale

```scala
class ELM_3L1 (x: MatriD, y: VectoD, private var nz: Int = -1,
    fname_: Strings = null, hparam: HyperParameter = null,
    f0: AFF = f_tanh, val itran: FunctionV_2V = null)
    extends PredictorMat (x, y, fname_, hparam)

    def compute_df_m (nz_: Int): Int = nz_
    def parameters: VectoD = b
    def train (x_: MatriD = x, y_: VectoD = y): ELM_3L1 =
    def buildModel (x_cols: MatriD): ELM_3L1 =
    override def predict (v: VectoD): Double = b dot f0.fV (a dot v)
    override def predict (v: MatriD = x): VectoD = f0.fM (a * v) * b
```

When the second activation function is not \( id \), then the optimization of the second set of parameters works like it does for Transformed Regression. Also, when multiple outputs are needed, the ELM_3L may be used.

### 10.14.4 Exercises

1. Create an ELM_3L1 model to predict values for the AutoMPG dataset. Compare with the results of using the following models: (a) Regression, (b) Perceptron, (c) NeuralNet_2L, (d) NeuralNet_3L, (e) NeuralNet_XL

2. Time each of the six model given above using SCALATION’s time method (`import scalation.util.time`).

   ```scala```
def time [R] (block: => R): R =
```

This method can time the execution of any block of code `{ time { block } }`.

3. Compare the following strategies for initializing NetParam A (weights and biases for the first layer). ...
Chapter 11

Time-Series/Temporal Models

For time-series forecasting, time-series/temporal models are used in order to make forecasts into the future. Applications are numerous and include weather, sales projections, energy consumption, economic indicators, financial instruments and traffic conditions. For time-series classification, related models are applied to sequential data that includes Speech Recognition and Natural Language Processing (NLP).

Until this chapter, predictive models have been of the form

\[ y = f(x; b) + \epsilon \]

where \( x \) is the vector of predictive variables/features, \( y \) is the response variable and \( \epsilon \) is the residual/error.

In order to fit the parameters \( b \), \( m \) samples are collected into a data/input matrix \( X \) and a response/output vector \( y \). The samples are treated as though they are independent. In many cases, such as data collected over time, they are often not independent. For example, the current Gross Domestic Product (GDP) will likely show high dependency upon the previous quarter’s GDP. If the model is to forecast the next quarter’s GDP, surely it should take into account the current (or recent) GDP values.

**General Time-Series Model**

To begin with, one could focus on forecasting the value of response \( y \) at time \( t \) as a function of past (or lagged) values of \( y \), e.g.,

\[
y_t = f(y_{t-p:t-1}; \phi) + \epsilon_t
\]

(11.1)

where \( f \) is a function of the slice of the past \( p \) values \( y_{t-p:t-1} \) and the parameter vector \( \phi \) is analogous to the \( b \) vector. The lags may also be dilated, e.g., with a dilation rate of 2,

\[
[y_{t-2p}, \ldots, y_{t-2}]
\]

Some forecasting techniques such as Time-Series Regression with Lagged Variables can select the features using for example forward selection, backward elimination or stepwise refinement.

Initially in this chapter, time will be treated as discrete time.
11.1 ForecasterVec

The ForecasterVec abstract class within the scalation.analytics.forecaster package provides a common framework for several forecasters. Note, the train method must be called first followed by eval. The forecast method is used to make forecasts h steps (time units) into the future.

ForecasterVec Abstract Class

Class Methods:

@param y the response vector (time series data)
@param n the number of parameters
@param hparam the hyper-parameters

abstract class ForecasterVec (y: VectoD, n: Int, hparam: HyperParameter = null)
    extends Fit (y, n, n, y.dim - n)) with Predictor

    def getX: MatriD = null
    def getY: VectoD = y
    def acF: VectoD = stats.acr
    def pacF: VectoD = pacf
    def train (x_null: MatriD, y_: VectoD): ForecasterVec =
    def eval (x_e: MatriD, y_e: VectoD): ForecasterVec = eval (y_e)
    def evalf (y_e: VectoD, yf: VectoD): ForecasterVec =
    def analyze (x_: MatriD = null, y_: VectoD = y, 
        def hparameter: HyperParameter = hparam
    def report: String =
    def residual: VectoD = e
    def predict (y_null: VectoD = null): Double = forecast (y.dim-1)(0)
    def predict (z: MatriD): VectoD =
    def predictAllz (): VectoD
    def predictAll (): VectoD = predictAllz () + stats.mu
    def forecastAll (h: Int, p: Int): MatriD =
    def forecast (t: Int, h: Int = 1): VectoD
    def forecastX (y: VectoD, t: Int, h: Int = 1, o: Int = 0): Double = forecast (t, h).last
    def plotFunc (fVec: VectoD, name: String, show: Boolean = true)
    def plotFunc2 (fVec: VectoD, name: String, show: Boolean = true)

Stats Case Class

The Stats case class defines the Auto-Correlation Function (ACF). It holds the mean, variance, the auto-covariance vector and the auto-correlation vector.
private val y = if (y_.isInstanceOf [VectorD]) y_.asInstanceOf [VectorD]
else y_.toDense

val mu = y.mean       // mean
val sig2 = y.variance  // variance
val acv = new VectorD (lags + 1)  // auto-covariance vector
for (k <- acv.range) acv(k) = y acov k  // k-th lag auto-covariance
val acr = acv / sig2  // auto-correlation function

11.1.1 Auto-Correlation Function

To better understand the dependencies in the data, it is useful to look at the auto-correlation. Consider the following time series data used in forecasting lake levels recorded in the Lake Level Times-series Dataset. (see cran.r-project.org/web/packages/fpp/fpp.pdf):

val m = 98
val t = VectorD.range (0, m)
val y = VectorD (580.38, 581.86, 580.97, 580.80, 579.79, 580.39, 580.42, 580.82, 581.40, 581.32,
  581.44, 581.24, 579.10, 579.09, 579.35, 578.82, 579.32, 579.01, 579.00, 579.80,
  579.83, 579.72, 579.89, 580.01, 579.37, 578.69, 578.19, 578.67, 579.55, 579.92,
  578.09, 579.37, 580.13, 580.14, 579.51, 579.24, 578.66, 578.86, 578.05, 577.79,
  576.75, 576.75, 577.82, 578.64, 580.58, 579.48, 577.38, 576.90, 576.94, 576.24,
  576.84, 576.85, 576.90, 577.79, 578.18, 577.51, 577.23, 578.42, 579.61, 579.05,
  579.26, 579.22, 579.38, 579.10, 577.95, 578.12, 579.75, 580.85, 580.41, 579.96,
  579.61, 578.76, 578.18, 577.21, 577.13, 579.10, 578.25, 577.91, 576.89, 575.96,
  576.80, 577.68, 578.38, 578.52, 579.74, 579.31, 579.89, 579.96)

The data also available in the scalation.analytics.forecaster package.

import ForecasterVec.{t, y}

First plot this dataset and then look at its Auto-Correlation Function (ACF).

new Plot (t, y, null, "Plot of y vs. t", true)

The Auto-Correlation Function (ACF) measures how much the past can influence a forecast. If the forecast is for time \( t \), then the past time points are \( t - 1, \ldots, t - p \). The \( k \)-th lag auto-covariance (auto-correlation), \( \gamma_k (\rho_k) \) is the covariance (correlation) of \( y_t \) and \( y_{t-k} \).

\[
\gamma_k = \text{C}[y_t, y_{t-k}] \\
\rho_k = \text{corr}(y_t, y_{t-k})
\]

(11.2) (11.3)

Note that \( \gamma_0 = \text{V}[y_t] \) and \( \rho_k = \gamma_k / \gamma_0 \). These equations assume the stochastic process \( \{y_t | t \in [0, m-1]\} \) is covariance stationary (see exercises).

Although vectors need not be created, to compute \( \text{corr}(y_t, y_{t-2}) \) one could imagine computing the correlation between \( y_.slice (2, m) \) and \( y_.slice (0, m-2) \). In SCALATION, the ACF is provided by the acF function from the ForecasterVec companion object.

371
import ForecasterVec._

val stats = Stats(y, MAX_LAGS) // e.g., 30
println(stats)
val zero = new VectorD(stats.acr.dim)
new Plot(null, stats.acr, zero, "ACF vs. k", true)

The first point in plot is the auto-correlation of $y_t$ with itself, while the rest of the points are ACF($k$), the $k^{th}$ lag auto-correlation.

### 11.1.2 Quality of Fit (QoF) for Time-Series Data

Notice that the ForecasterVec abstract class extends the Fit class and so several QoF measures are available, including $R^2$, $\bar{R}^2$, MAE, MSE, RMSE, and AIC.

Two that are widely used in time-series analysis are the following:

- **Mean Absolute Percentage Error (MAPE)**

$$ MAPE = \frac{100}{m} \sum_{t=0}^{m-1} \frac{|y_t - \hat{y}_t|}{|y_t|} \quad (11.4) $$

- **symmetric Mean Absolute Percentage Error (sMAPE)**

$$ sMAPE = \frac{200}{m} \sum_{t=0}^{m-1} \frac{|y_t - \hat{y}_t|}{|y_t| + |\hat{y}_t|} \quad (11.5) $$
11.2 Random Walk

The physical property of inertia would suggest that many processes may be accurately modeled as a random walk, at least in the short run. This model states the future value $y_t$ is the previous value $y_{t-1}$ disturbed by white noise $\epsilon_t$.

$$y_t = y_{t-1} + \epsilon_t$$  \hspace{1cm} (11.6)

Notice that this model has no parameters to estimate and as such is considered to be a baseline model for more sophisticated to improve upon. If they cannot, their value is questionable.

11.2.1 White Noise

A stochastic process $\{\epsilon_t|t \in \{0, \ldots, m - 1\}\}$ is said to be white noise if

$$E[\epsilon_t] = 0 \hspace{1cm} \text{zero mean} \hspace{1cm} (11.7)$$

$$E[\epsilon_t^2] = \sigma^2 \hspace{1cm} \text{constant variance} \hspace{1cm} (11.8)$$

$$E[\epsilon_t\epsilon_{t-k}] = 0 \hspace{0.5cm} \text{for } k \geq 1 \hspace{1cm} \text{uncorrelated} \hspace{1cm} (11.9)$$

Several distributions may be used to generate white noise with the most common being the Gaussian (Normal) distribution, $\mathcal{N}(0, \sigma^2 I)$.

11.2.2 Detecting Random Walks

To determine whether a stochastic process is a random walk, take its first difference.

$$y'_t = y_t - y_{t-1} = \epsilon_t$$  \hspace{1cm} (11.10)

Then apply the Ljung-Box Test to see if it qualifies as white noise.

$$q = m(m + 2) \sum_{k=1}^{h} \frac{\rho_k^2}{m - k} \sim \chi^2_h$$  \hspace{1cm} (11.11)

where $\rho_k$ is the $k$-lag autocorrelation and $m$ is the length of the time-series. Large values for $q$ indicate significant autocorrelation. Hyndman [45] recommends that the number of lags $h$ be 10 for non-seasonal processes. See the section on SARIMA for a discussion of seasonal models.

11.2.3 Forecasting Lake Levels

As a simple example, run and compare the two baseline modeling techniques, RandomWalk and NullModel of the Lake Level Time Series Dataset.

```scala
import ForecasterVec.{t, y}

banner("Build RandomWalk Model")
val rw = new RandomWalk (y) // time series model
rw.train (null, y).eval () // train for RandomWalk model
```

373
println (rw.report)
new Plot (t, y, rw.predictAll (), s"Plot of y, RandomWalk vs. t", true)

banner ("Build NullModel")
val nm = new NullModel (y) // time series model
nm.train (null, y).eval () // train for RandomWalk model
println (nm.report)
new Plot (t, y, nm.predictAll (), s"Plot of y, NullModel vs. t", true)

In the plot, the forecasts (in red) simply follow the actual values (in black) with a one-step delay. The report for RandomWalk shows the quality of fit.

REPORT
hparameter hp = null
parameter phi = null
fitMap qof = Map(rSq -> 0.680473, rBarSq -> 0.677179, sst -> 168.577367, sse -> 53.865000,
mse0 -> 0.549643, rmse -> 0.741379, mae -> 0.579592, dfm -> 1.000000, df -> 97.000000, fStat -> 206.573835,
aic -> -105.730135, bic -> -100.560200, mape -> 0.100114, smape -> 0.100122)

In particular, the $R^2$ is 0.680473, the Mean Absolute Percentage Error (MAPE) is 0.100114 and is symmetric version is 0.100122.

The report for NullModel shows a reduced quality of fit.

REPORT
hparameter hp = null
parameter phi = VectorD(579.004)
fitMap qof = Map(rSq -> 0.000000, rBarSq -> -0.010309, sst -> 168.577367, sse -> 168.577367,
mse0 -> 1.720177, rmse -> 1.311555, mae -> 1.058684, dfm -> 1.000000, df -> 97.000000, fStat -> 0.000000,
aic -> -161.634915, bic -> -156.464980, mape -> 0.182880, smape -> 0.182862)

RandomWalk Class

Class Methods:

@tag param y the response vector (time series data)
@tag param hparam the hyper-parameters
class RandomWalk (y: VectoD, hparam: HyperParameter = null)
    extends ForecasterVec (y, 1) with NoFeatureSelection

override def train (x_null: MatriD, y_: VectoD): RandomWalk = { super.train (null, y_); this }
def parameter: VectoD = null
def predictAllz (): VectoD =
def forecast (t: Int, h: Int = 1): VectoD = VectorD.fill (h)(y(max (0, t-1)))
override def forecastX (y: VectoD, t: Int, h: Int = 1, o: Int = 0): Double = y(max (0, t-1))
11.2.4 Random Walk with Drift

NullModel Class

A simpler baseline modeling technique that is not as accurate as RandomWalk is the NullModel

Class Methods:

\[
\text{Class Methods:}
\]

- @param y the response vector (time series data)
- @param hparam the hyper-parameters

```scala
class NullModel (y: VectorD, hparam: HyperParameter = null)
    extends ForecasterVec (y, 1, hparam) with NoFeatureSelection
```

```scala
override def train (x_null: MatrixD, y_: VectorD): NullModel = {
    super.train (null, y_);
    this
}
```

```scala
def parameter: VectorD = VectorD (stats.mu)
```

```scala
def predictAllz (): VectorD = new VectorD (m)
```

```scala
def forecast (t: Int, h: Int = 1): VectorD = VectorD.fill (h)(stats.mu)
```

```scala
override def forecastX (y: VectorD, t: Int, h: Int = 1, o: Int = 0): Double = stats.mu
```

11.2.5 Exercises

1. Generate Gaussian white noise with a variance of one and plot it.

   ```scala
   val noise = Normal (0, 1)
   val y = VectorD (for (i <- 0 until 100) yield noise.gen)
   new Plot (null, y, null, "white noise")
   ```

2. Create a Random Walk model for the above white noise process, plot the process value vs. the forecasted value and assess the Quality of Fit (QoF). Also, examine the ACF plot. What would it mean if a peak in the ACF plot was outside the error bands. Ignore the first value corresponding to the zeroth lag where the correlation \( \rho_0 \) should one and the partial correlation \( \psi_{00} \) should be zero.

   ```scala
   val rw = new RandomWalk (y) // time series model
   rw.train (null, y).eval () // train for RandomWalk model
   println (rw.report)
   new Plot (null, y, rw.predictAll (), s"Plot of y, RandomWalk vs. t", true)
   rw.plotFunc2 (rw.acF, "ACF")
   ```

   Note: plotFunc2 includes error bands/confidence intervals, while plotFunc does not.

3. For the Lake-Level Dataset, apply the Ljung-Box Test to see if \( y_t \) or \( \epsilon_t \) are white noise. Determine the values for \( q \) for each case and the 95% critical value of \( \chi^2_{10} \).

   ```scala
   import scalation.variate.Quantile.chiSquareInv
   println (chiSquareInv (0.95, 10))
   ```

375
11.3 Simple Exponential Smoothing

The basic idea of Simple Exponential (SES) is to start with Random Walk,

\[ \hat{y}_t = y_{t-1} \]  

but stabilize/smooth out the forecast by maintaining a state variable that summarizes the past.

The state variable \( s_t \) is a weighted combination of the most recent past value \( y_{t-1} \) and the previous value of the state variable \( s_{t-1} \). The forecasted value \( \hat{y}_t \) is taken as the value of the state variable \( s_t \).

\[ s_t = \alpha y_{t-1} + (1 - \alpha) s_{t-1} \]  
\[ \hat{y}_t = s_t \]  

The state variable may be initialized as the initial value of the time-series (other options include taking an average or letting it be a second parameter passed to an optimizer).

\[ s_0 = y_0 \]  

The parameter \( \alpha \in [0, 1] \) is referred to as the smoothness parameter. When \( \alpha = 1 \), SES becomes Random Walk as older values in the time-series are ignored. When \( \alpha = 0 \), SES the state remains locked at its initial value. Thus, \( \alpha \in (0, 1) \) is a more practical range.

The reason it is called exponential smoothing is that the impact of older values in the time series decreases exponentially.

11.3.1 Model Equation

The model equation indicates that next forecasted value (one step ahead) is the value of the state variable plus a random element/shock.

\[ y_t = s_t + \epsilon_t \]  

11.3.2 Training

For SES, one may minimize the sum of squared errors (sse). The error at time \( t \) is given by

\[ \epsilon_t = y_t - \hat{y}_t \]  

In vector form \( \epsilon = [\epsilon_0, \ldots, \epsilon_{m-1}] \), the equation becomes,

\[ \epsilon = y - \hat{y} \]  

A loss function \( h(\alpha) = sse = \epsilon \cdot \epsilon \) may be minimized to determine the parameter \( \alpha \) (alternatively, both \( s_0 \) and \( \alpha \)).

\[ h(\alpha) = (y - \hat{y}) \cdot (y - \hat{y}) \]  

In ScalaTion, the train method is used to find an optimal value for \( \alpha \) (a).
override def train (x_null: MatriD, y_: VectoD): SimpleExpSmoothing =
{
    def f_obj (x: VectoD): Double = (y_ - smooth (x(0), y_)).normSq // only one param
    super.train (null, y_)
    if (opt) {
        val solver = new L_BFGS_B (f_obj, l = lo, u = up) // Quasi-Newton optimizer
        val x = solver.solve (VectorD (a), toler = TOL) // optimize value for a
        a = x(0) // pull a from vector result
    } // if
    s = smooth (a) // vector of predicted values
    this
} // train

y_ holds the vector of $y_t$ values, while smooth calculates $\hat{y}_t$

### 11.3.3 Effect of the Smoothing Parameter

As the smoothing parameter $\alpha$ get smaller, the forecast curve becomes more smooth, as shown in the plot produced by the following code applied to the Lake Level database.

```scala
import ForecasterVec.{t, y}
val ses = new SimpleExpSmoothing (y)
for (i <- 0 to 5) {
    val a = i.toDouble / 5.0
    banner (s"Build SimpleExpSmoothing model with a = $a")
    ses.reset (a)
    ses.train (null, y)
    ses.eval ()
    val yf = ses.predictAll ()
    println (ses.report)
    new Plot (t, y, yf, s"Plot of y, SimpleExpSmoothing (a = $a) vs. t", true)
} // for
```

### 11.3.4 Simple Exponential Smoothing with Growth

SimpleExpSmoothing Class

---

Class Methods:

- @param y the response vector (original time series data)
- @param hparam the hyper-parameters

```scala
class SimpleExpSmoothing (y: VectoD, hparam: HyperParameter = SimpleExpSmoothing.hp)
    extends ForecasterVec (y, n = 1, hparam) with NoFeatureSelection
```

```scala
def reset (a: Double) { aa = a }
```
def toggleOpt () { opt = ! opt }
def smooth (a: Double = aa, y_: VectoD = y): VectoD =
override def train (x_null: MatriD, y_: VectoD): SimpleExpSmoothing =
def parameter: VectoD = VectorD (aa)
override def predictAll (): VectoD = s
def predictAllz (): VectoD = s - stats.mu
def forecast (t: Int, h: Int = 1): VectoD =

11.3.5 Exercises

1. Test customized ($\alpha = 0.5$) vs. optimized ($\alpha$ optimizer determined) smoothing on the following synthetic data.

```scala
val m = 50
val r = Random ()
val y = VectorD (for (i <- 0 until m) yield i + 10.0 * r.gen)

val ses = new SimpleExpSmoothing (y)
ses.smooth (0.5) // use customized parameters, don't train
ses.eval ()
println (ses.report)
val yf = ses.predictAll () // y forecasted

ses.train (null, y) // train to use optimal a
ses.eval ()
println (ses.report)
val yf2 = ses.predictAll () // y forecasted
```

2. For some optimization software it is necessary to pass in an objective function and derivative function(s). For this type of optimization, $\hat{y}$ needs to be replaced.

$$\hat{y}_t = \alpha y_{t-1} + \alpha (1-\alpha)y_{t-2} + \alpha(1-\alpha)^2y_{t-3} + \ldots$$ (11.20)

As a sum this becomes,

$$\hat{y}_t = \alpha \sum_{k=0}^{t-1} (1-\alpha)^k y_{t-1-k}$$ (11.21)

Therefore, the loss function is expressed as a double sum.

$$h(\alpha) = \sum_{t=0}^{m-1} \left[ y_t - \alpha \sum_{k=0}^{t-1} (1-\alpha)^k y_{t-1-k} \right]^2$$ (11.22)

Create a formula for the derivative of $h(\alpha)$ and pass the function and its derivative into the NewtonRaphson class found in the scalation.minima package to find an optimal value for $\alpha$. 

378
11.4 Auto-Regressive (AR) Models

In order to predict future values for a response variable \( y \), the obvious thing to do is to find variables that may influence the value of \( y \). The most obvious is prior (or lagged) values of itself. Often, other variables may be helpful as well, some of which may be time-series themselves. There are numerous modeling techniques used for time-series forecasting \[10, 45\].

One of the simplest types of forecasting models of the form given in the first section is to make the future value \( y_t \) be linearly dependent on the last \( p \) of \( y \) values. In particular, a \( p^{th} \)-order Auto-Regressive AR\((p)\) model predicts the next value \( y_t \) from the sum of the last \( p \) values each weighted by its own coefficient/parameter \( \phi_j \),

\[
y_t = \delta + \phi \cdot \left< y_{t-p:t-1} \right> + \epsilon_t \tag{11.23}
\]

where

- \( \cdot \left< \cdot \right> \) is the dot product with the second argument reversed,
- the parameter vector \( \phi = [\phi_0, \ldots, \phi_{p-1}] \),
- the slice of the past \( p \) values \( y_{t-p:t-1} = [y_{t-p}, \ldots, y_{t-1}] \),
- the constant/intercept/drift \( \delta = \mu(1 - 1 \cdot \phi) \), and
- the error/noise is represented by \( \epsilon_t \).

This equation may be expanded out into

\[
y_t = \delta + \phi_0 y_{t-1} + \ldots + \phi_{p-1} y_{t-p} + \epsilon_t \tag{11.24}
\]

\( \epsilon_t \) represents noise that shocks the system at each time step. We require that \( E[\epsilon_t] = 0, \ V[\epsilon_t] = \sigma^2 \), and that all the noise shocks be independent.

**Zero-Centered**

To better capture the dependency, the data can be zero-centered, which can be accomplished by subtracting the mean \( \mu \), \( z_t = y_t - \mu \).

\[
z_t = \phi_0 z_{t-1} + \ldots + \phi_{p-1} z_{t-p} + \epsilon_t \tag{11.25}
\]

Notice that since \( z_t \) is zero-centered, \( \delta \) drops out (see exercises) and the formulas for the mean, variance and covariance are simplified.

\[
E[z_t] = 0 \tag{11.26}
\]

\[
V[z_t] = E[z_t^2] = \gamma_0 \tag{11.27}
\]

\[
C[z_t, z_{t-k}] = E[z_t z_{t-k}] = \gamma_k \tag{11.28}
\]

Recall that covariance \( C[x, y] = E[(x - \mu_x)(y - \mu_y)] \).
11.4.1 AR(1) Model

When the future is mainly dependent only on the most recent value, e.g., $\rho_1$ is high and rest $\rho_2, \rho_3, \text{ etc.}$ are decaying, then an AR(1) model may be sufficient.

$$z_t = \phi_0 z_{t-1} + \epsilon_t$$ (11.29)

An estimate for the parameter $\phi_0$ may be determined from the Auo-Correlation Function (ACF).

Solving for the $\phi_0$ Parameter

Take this equation and multiply it by $z_{t-k}$.

$$z_t z_{t-k} = \phi_0 z_{t-1} z_{t-k} + \epsilon_t z_{t-k}$$

Taking the expected value of the above equation gives,

$$E[z_t z_{t-k}] = \phi_0 E[z_{t-1} z_{t-k}] + E[\epsilon_t z_{t-k}]$$ (11.30)

Using the definition for $\gamma_k$ given above for $k \geq 1$, this can be rewritten as

$$\gamma_k = \phi_0 \gamma_{k-1} + 0$$

where past value $z_{t-k}$ is independent of future noise shock $\epsilon_t$. Now dividing by $\gamma_0$ yields

$$\rho_k = \phi_0 \rho_{k-1}$$ (11.31)

An estimate for parameter $\phi_0$ may be easily determined by simply setting $k = 1$ in the above equation.

$$\phi_0 = \rho_1$$ (11.32)

Furthermore, $\rho_0$ is 1 and

$$\rho_k = \phi_0^k \quad \text{for } |\phi_0| < 1$$ (11.33)

Solving for the Noise Variance

When $k = 0$, multiplying by $z_t$ gives,

$$E[z_t z_t] = \phi_0 E[z_{t-1} z_t] + E[\epsilon_t z_t]$$ (11.34)

Using gamma notation and replacing $z_t$ with $\hat{z}_t + \epsilon_t$ yields

$$\gamma_0 = \phi_0 \gamma_1 + E[\epsilon_t (\hat{z}_t + \epsilon_t)]$$

Since $\hat{z}_t$ is independent of $\epsilon_t$, the last term becomes $E[\epsilon_t \epsilon_t]$ and as $\gamma_1 = \gamma_0 \rho_1$, the variance of the noise may be written as follows:

$$\sigma^2_\epsilon = V[\epsilon_t] = \gamma_0 (1 - \phi_0 \rho_1) = \gamma_0 (1 - \phi_0^2)$$ (11.35)
### 11.4.2 AR($p$) Model

The zero-centered model equation for an AR($p$) model includes the past $p$ values.

$$z_t = \phi_0 z_{t-1} + \phi_1 z_{t-2} + \ldots + \phi_{p-1} z_{t-p} + \epsilon_t$$

The forecasted value for one step ahead ($h = 1$) is calculated as follows:

$$\hat{z}_t = \phi_0 z_{t-1} + \phi_1 z_{t-2} + \ldots + \phi_{p-1} z_{t-p}$$

Using the reverse dot product the equation becomes,

$$\hat{z}_t = \phi \cdot z_{t-p:t-1}$$  \hspace{1cm} (11.36)

This is depicted in Figure 11.1 that shows each subsequent parameter is paired with a value from the time-series which is farther back in time. The paired values are multiplied and their products are summed to provide the next forecasted value.

![Figure 11.1: AR($p$) Model with $p = 3$](image)

As with convolutional filters, the parameter vector $\phi$, can slide along the zero-centered time-series $z_t$ to compute forecasted values $\hat{z}_t$. To obtain forecasted values for the original time-series, simply add back the mean,

$$y_t = z_t + \mu$$  \hspace{1cm} (11.37)  

**Determining Parameter Values from the ACF**

Working with the zero-centered equations, one can derive a system of linear equations to solve for the parameters. First, multiplying by $z_{t-k}$ gives

$$z_t z_{t-k} = \phi_0 z_{t-1} z_{t-k} + \phi_1 z_{t-2} z_{t-k} + \ldots + \phi_{p-1} z_{t-p} z_{t-k} + \epsilon_t z_{t-2}$$

and then taking the expectation produces

$$\gamma_k = \phi_0 \gamma_{k-1} + \phi_1 \gamma_{k-2} + \ldots + \phi_{p-1} \gamma_{k-p}$$
Dividing by $\gamma_0$ yields equations relating the parameters and correlations,

$$\rho_k = \phi_0 \rho_{k-1} + \phi_1 \rho_{k-2} + \cdots + \phi_{p-1} \rho_{k-p}$$ (11.38)

These equations contains $p$ unknowns and by letting $k = 1, 2, \ldots, p$, it can be used to generate $p$ equations, or one matrix equation.

**Yule-Walker Equations**

The equations below are known as the Yule-Walker equations. Note that $\rho_{-j} = \rho_j$, $\rho_0$ equals 1, and $k$ advances row by row.

\[
\begin{align*}
\rho_1 &= \phi_0 \rho_0 + \phi_1 \rho_1 + \cdots + \phi_{p-1} \rho_{p-1} \\
\rho_2 &= \phi_0 \rho_1 + \phi_1 \rho_0 + \cdots + \phi_{p-1} \rho_{p-2} \\
\vdots \\
\rho_p &= \phi_0 \rho_{p-1} + \phi_1 \rho_{p-2} + \cdots + \phi_{p-1} \rho_0
\end{align*}
\]

Letting $\rho$ be the $p$-dimensional vector of lag auto-correlations and $\phi$ be the $p$-dimensional vector of parameters/coefficients, we may concisely write

$$\rho = R \phi$$ (11.39)

where $R$ is a $p$-by-$p$ symmetric Toeplitz (one value per diagonal) matrix of correlations with ones on the main diagonal.

\[
R = \begin{bmatrix}
1 & \rho_1 & \cdots & \rho_{p-2} & \rho_{p-1} \\
\rho_1 & 1 & \cdots & \rho_{p-3} & \rho_{p-2} \\
\vdots & \vdots & \ddots & \vdots & \vdots \\
\rho_{p-2} & \rho_{p-3} & \cdots & 1 & \rho_1 \\
\rho_{p-1} & \rho_{p-2} & \cdots & \rho_1 & 1
\end{bmatrix}
\] (11.40)

One way to solve for the parameter vector $\phi$ is to take the inverse (or use related matrix factorization techniques).

$$\phi = R^{-1} \rho$$ (11.41)

The noise variance $\sigma^2$ is given by (see exercises)

$$\sigma^2 = \gamma_0 (1 - \rho \cdot \phi) = \gamma_0 (1 - \rho^T R^{-1} \rho)$$ (11.42)

Due to the special structure of the $R$ matrix, more efficient techniques may be used, see the next subsection.
11.4.3 Training

The steps in training include computing the Auto-Correlation Function, executing the Durbin-Levinson algorithm, and zero-centering the response. The `train` method of the `AR` class is in its superclass `ForecasterVec`.

```scala
def train (x_null: MatriD, y_: VectoD): ForecasterVec =
{
  stats = Stats (y_, ml) // compute stats including ACF (acv)
  psi = durbinLevinson (stats.acv, ml) // pass in ACF and max lags to Durbin-Levinson
  pacf = psi.getDiag () // Partial Auto-Correlation Function (PACF)
  z = y_ - stats.mu // zero-center the training response vector
  e = new VectorD (y_.dim) // make space for error results
  this
} // train
```

The parameter/coefficient vector $\phi$ can be estimated using multiple training algorithms/estimation procedures [106]:

1. **Method of Moments (MoM).** The first two moments, mean and covariance, are used in the Yule-Walker Method. In particular the auto-correlations in ACF are used.

2. **Maximum Likelihood Estimation (MLE).** TBD.

3. **Least Squares Estimation (LSE).** TBD.

In Scalation, the coefficients $\phi$ are estimated using the Durbin-Levinson algorithm and extracted from the $p^{th}$ row of the $\psi$ (psi) matrix. Define $\psi_{kj}$ to be $\phi_j$ for an AR($k$) model. Letting $k$ range up to $p$ allows the $\phi_j$ parameters to be calculated. Letting $k$ range up to the maximum number of lags ($ml$) allows the Partial Auto-Correlation Function (PACF) to be computed.

Invoke the `durbinLevinson` method [79] passing in the auto-covariance vector $\gamma$ ($g$) and the maximum number of lags ($ml$). From 1 up to the maximum number of lags, iteratively compute the following:

\[
\psi_{kk} = \gamma_k - \sum_{j=1}^{k-1} \psi_{k-1,j} \gamma_{k-j} / r_{k-1}
\]
\[
\psi_{kj} = \psi_{k-1,j} - \psi_{kk} \psi_{k-1,k-j}
\]
\[
r_k = r_{k-1}(1 - \psi_{kk}^2)
\]

```scala
def durbinLevinson (g: VectoD, ml: Int): MatriD =
{
  val psi_ = new MatrixD (ml+1, ml+1) // psi matrix (ml = max lags)
  val r   = new VectorD (ml+1); r(0) = g(0)
  for (k <- 1 to ml) {
    val sum = 0.0
    for (j <- 1 until k) sum += psi_(k-1, j) * g(k-j)
    val a = (g(k) - sum) / r(k-1)
    psi_(k, k) = a
    for (j <- 1 until k) psi_(k, j) = psi_(k-1, j) - a * psi_(k-1, k-j)
    r(k) = r(k-1)(1 - psi_(k, k))
  }
  this
} // durbinLevinson
```
\[ r(k) = r(k-1) \times (1.0 - a \times a) \]

} // for

psi_

} // durbinLevinson

In particular, \( \psi_{kk} \) is the \( k^{th} \) partial auto-correlation and the parameter vector is

\[ \phi = \psi_p \]

the \( p^{th} \) row of the \( \Psi \) matrix (11.43)

**Partial Auto-Correlation Function**

The Partial Auto-Correlation Function (PACF) that is extracted from the main diagonal of the \( \Psi \) matrix and can be used along with ACF to select the appropriate type of model. As \( k \) increases, the \( \rho_k \) will decrease and adding more parameters/coefficients will become of little help. Deciding where to cut off \( \rho_k \) from the ACF is somewhat arbitrary, but the \( k^{th} \) partial auto-correlation \( \psi_{kk} \) drops toward zero more abruptly giving a stronger signal as to what model to select. See the exercises for details.

### 11.4.4 Forecasting

After the parameters/coefficients have been estimated as part of the train method, the AR(\( p \)) model can be used for forecasting.

```scala
AR.hp("p") = p // reassign hyper-parameter p
val ar = new AR (y) // time series data
ar.train ().eval () // train for AR(p) model
val y_f = ar.forecast (2) // make forecasts for 1 and 2 steps
```

The forecast method takes two parameters: \( t \) the time index for the first forecast in the forecasting horizon and \( h \) the forecasting horizon indicating how many steps into the future to forecast. The method uses the last \( p \) actual values to make the first forecast, and then uses the last \( p-1 \) actual values and the first forecast to make the next forecast. As expected, the quality of the forecast will degrade as \( h \) gets larger.

```scala
def forecast (t: Int, h: Int = 1): VectoD =
{
    if (t > m) flaw ("forecast", "no forecasts with starting t > m are provided")
    val zf = new VectorD (p+h) // copy first p values into zf
    for (l <- 0 until p) zf(l) = z(max (0, t-p+l)) // copy first p values into zf
    for (k <- 1 to h) zf(p-1+k) = phi.rot (zf)(p-2+k) // predict the rest
    zf.slice (p) += stats.mu // return only forecasted values
} // forecast
```

The rot method implements a reversed dot product (\( \cdot \)). It pairs each coefficient \( \phi_j \) with increasing older values in the time-series (moving backward in the time-series) and sums the paired products. The first argument is the time-series/vector and second is the most recent index/time position to take (moving backward from there).
AR Class

Class Methods:

@param y the response vector (time-series data)
@param hparam the hyper-parameters

class AR (y: VectoD, hparam: HyperParameter = ARMA.hp)
    extends ForecasterVec (y, hparam("p").toInt, hparam)

override def train (x_null: MatriD, y_ : VectoD): AR =
def retrain (p_ : Int): AR =
def parameter: VectoD = phi
override def report: String =
def predictAllz (): VectoD =
def forecast (t: Int, h: Int = 1): VectoD =
override def forecastX (y: VectoD = y, t: Int = y.dim, h: Int = 1, o: Int = 0): Double =
def forwardSel (cols: Set[Int], index_q: Int = index_rSqBar): (Int, AR) =

11.4.5 Exercises

1. Compute ACF for the Lake Level Time-series Dataset. For AR(1) set parameter $\phi_0$ to $\rho_1$, the first lag auto-correlation. Compute $\hat{y}_t$ by letting $\hat{y}_0 = y_0$ and for $k <- 1$ until $y.dim$

$$z_t = y_t - \mu$$
$$\hat{y}_t = \rho_1 z_{t-1} + \mu$$

Plot $\hat{y}_t$ and $y_t$ versus $t$.

2. Consider the following AR(2) Model.

$$z_t = \phi_0 z_{t-1} + \phi_1 z_{t-2} + \epsilon_t$$

Derive the following equation:

$$\rho_k = \phi_0 \rho_{k-1} + \phi_1 \rho_{k-2}$$

Setting $k = 1$ and then $k = 2$ produces two equations which have two unknowns $\phi_0$ and $\phi_1$. Solve for $\phi_0$ and $\phi_1$ in terms of the first and second lag auto-correlations, $\rho_1$ and $\rho_2$. Compute $\hat{y}_t$ by letting $\hat{y}_0 = y_0$, $\hat{y}_1 = y_1$ and for $k <- 2$ until $y.dim$
\[ z_t = y_t - \mu \]
\[ \hat{y}_t = \phi_0 z_{t-1} + \phi_1 z_{t-2} + \mu \]

Plot \( \hat{y}_t \) and \( y_t \) versus \( t \).

3. Use the ScalaTion class AR to develop Auto-Regressive Models for \( p = 1, 2, 3 \), for the Lake Level Time-series Dataset. Plot \( \hat{y}_t \) and \( y_t \) versus \( t \) for each model. Also, compare the first two models with those developed in the previous exercises.

4. Generate a dataset for an AR(\( p \)) model as follows:

```scala
val sig2 = 10000.0
val noise = Normal (0.0, sig2)
val n = 50
val t = VectorD.range (0, n)
val y = VectorD (for (i <- 0 until n) yield 40 * (i-1) - (i-2) * (i-2) + noise.gen)
```

Now create an AR(1) model, train it and show its report.

```scala
val ar = new AR (y) // time series data
ar.train ().eval () // train for AR(1) model
println (ar.report)
```

Also, plot \( \hat{y}_t \) and \( y_t \) versus \( t \). Look at the ACF and PACF to see if some other AR(\( p \)) might be better.

```scala
ar.plotFunc (ar.acF, "ACF")
ar.plotFunc (ar.pacF, "PACF")
```

Choose a value for \( p \) and retrain the model. Explain your choice of \( p \).

```scala
ar.retrain (p).eval () // retrain for AR(p) model
println (ar.report)
```

Also, plot \( \hat{y}_t \) and \( y_t \) versus \( t \).

5. The \( k^{th} \) lag auto-covariance is useful when the stochastic process \( \{y_t \mid t \in \{0, m-1\}\} \) is covariance stationary.

\[ \gamma_k = \mathbb{C} [y_t, y_{t-k}] \]

When \( y_t \) is covariance stationary, the covariance is only determined by the lag between the variables and not where they occur in the time series, e.g., \( \mathbb{C} [y_4, y_3] = \mathbb{C} [y_2, y_1] = \gamma_1 \). Thus, the covariance matrix \( \Gamma \) is Toeplitz. Covariance stationarity also requires that \( \mathbb{E} [y_t] = \mu \), i.e., the mean is time invariant.

Repeat the previous exercise, but generate a time series from a process that is not covariance stationary. What can be done to transform such a process into a covariance stationary process? Hint: read ahead to the section on ARIMA models or additional reading such as [92], Chapter 23.
6. For a covariance stationary process \( y_t \), derive the relationship between the drift parameter \( \delta \) and the mean \( \mu \). Further, given an AR\( (p) \) model,

\[
y_t = \delta + \phi_0 y_{t-1} + \phi_1 y_{t-2} + \ldots + \phi_{p-1} y_{t-p} + \epsilon_t
\]

Show that \( \delta \) drops out of the equation when \( z_t + \mu \) is substituted for \( y_t \) in the above equation.

\[
z_t = \phi_0 z_{t-1} + \phi_1 z_{t-2} + \ldots + \phi_{p-1} z_{t-p} + \epsilon_t
\]

7. Assume the time indexed, \( m \) dimensional vector \( y \) is a zero-centered, covariance stationary Gaussian process \( \mathbf{N}(0, \Gamma) \) having the following probability density function (pdf),

\[
f(y; \phi, \sigma^2) = \left[ (2\pi)^m \det \Gamma \right]^{-\frac{1}{2}} e^{-\frac{1}{2} y' \Gamma^{-1} y}
\]

Recall, that the covariance matrix for a covariance stationary process is Toeplitz, meaning

\[
\Gamma = [\gamma_{i-j}]_{i,j=0,m-1}
\]

Write the likelihood and log-likelihood functions. Discuss how to create an algorithm for Maximum Likelihood Estimation. Be sure to exploit the special structure of the covariance matrix.

8. Derive the formula for the noise variance \( \sigma^2 = \text{Var}[z_t] = \text{E}[z_t z_t] \),

\[
\sigma^2 = \gamma_0(1 - \rho \cdot \phi)
\]

from the following equation:

\[
\gamma_0 = \phi_0 \gamma_1 + \phi_1 \gamma_2 + \ldots + \phi_{p-1} \gamma_p + \text{E}[\epsilon_t \epsilon_t]
\]

9. Create a process with \( \phi_0 \) and \( \phi_1 \) nonzero, such that \( \rho_1 \) is approximately zero, while \( \rho_2 \) is large. Compare the Quality of Fit (QoF) of an AR\( (2) \) model versus an AR\( (1) \) model for this process.

```scala
val m = 100
val y = new VectorD (m)
val noise = Normal (0, 1)
val phi = VectorD (? , ?)
y(0) = noise.gen; y(1) = noise.gen
for (t <- 2 until m) y(t) = phi(0) * y(t-1) + phi(1) * y(t-2) + noise.gen
```

Plot the ACF to view the auto-correlations.
11.5 Moving-Average (MA) Models

The Auto-Regressive (AR) models predict future values based on past values. Let us suppose the daily values for a stock are 100, 110, 120, 110, 90. To forecast the next value, one could look at these values or focus on each day’s errors or shocks. Whether previous values of the time-series (AR) or unexpected changes (MA) (e.g., while predictions indicated that the Dow-Jones Industrial Average would continue its slow rise, news of the its large drop has investor nervous) are more important depends on the situation. In the simplest case an MA model is of the following form,

\[ y_t = \mu + \theta_0 \varepsilon_{t-1} + \varepsilon_t \]

To make the data zero-centered, again let \( z_t = y_t - \mu \)

\[ z_t = \theta_0 \varepsilon_{t-1} + \varepsilon_t \] \hspace{1cm} (11.44)

Errors are computed in the usual way.

\[ \varepsilon_t = y_t - \hat{y}_t \]

The question of AR(1) versus MA(1) may be viewed as which column \( z_{t-1} \) or \( \varepsilon_{t-1} \) leads to better forecasts. It depends on the data, but considering the GDP example again, \( z_{t-1} \) indicates how far above or below the mean the current GDP is. One could imagine shocks \( \varepsilon_{t-1} \) to GDP, such as new tariffs, tax cuts or bad weather being influential shocks to the economy. In such cases, an MA model may provide better forecasts than an AR model.

11.5.1 MA(\( q \)) Model

The model equation for an MA(\( q \)) model includes the past \( q \) values of \( \varepsilon_t \).

\[ y_t = \delta + \theta \cdot \varepsilon_{t-q:t-1} + \varepsilon_t \] \hspace{1cm} (11.45)

where the parameter vector \( \theta = [\theta_0, \ldots, \theta_{q-1}] \), the drift \( \delta = \mu \) and the error/noise is represented by \( \varepsilon_t \). This may be expanded out into

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|c|c|}
\hline
\( t \) & \( y_t \) & \( z_t \) & \( \varepsilon_{t-1} \) & \( \hat{y}_t \) & \( \hat{y}_t \) & \( \varepsilon_t \) & \( \varepsilon_t^2 \) \\
\hline
0 & 100.0 & 0.0 & - & - & - & 0.0 & 0.0 \\
1 & 110.0 & 10.0 & 0.0 & 0.0 & 100.0 & 10.0 & 100.0 \\
2 & 120.0 & 20.0 & 10.0 & 8.0 & 108.0 & 12.0 & 144.0 \\
3 & 110.0 & 10.0 & 12.0 & 9.6 & 109.6 & 0.4 & 0.2 \\
4 & 90.0 & -10.0 & 0.4 & 0.3 & 100.3 & -10.3 & 106.1 \\
\hline
\end{tabular}
\caption{MA(1) Forecast Chart (to one decimal place)}
\end{table}

The question of AR(1) versus MA(1) may be viewed as which column \( z_{t-1} \) or \( \varepsilon_{t-1} \) leads to better forecasts. It depends on the data, but considering the GDP example again, \( z_{t-1} \) indicates how far above or below the mean the current GDP is. One could imagine shocks \( \varepsilon_{t-1} \) to GDP, such as new tariffs, tax cuts or bad weather being influential shocks to the economy. In such cases, an MA model may provide better forecasts than an AR model.
\[ y_t = \mu + \theta_0 \epsilon_{t-1} + \ldots + \theta_{q-1} \epsilon_{t-q} + \epsilon_t \]

**Zero-Centered**

Zero-centering the data \( z_t = y_t - \mu \) produces

\[ z_t = \theta_0 \epsilon_{t-1} + \ldots + \theta_{q-1} \epsilon_{t-q} + \epsilon_t \quad (11.46) \]

In order to estimate the parameter vector \( \theta = [\theta_0, \ldots, \theta_{q-1}] \), we would like to develop a system of equations like the Yule-Walker equation. Proceeding likewise, the auto-covariance function for \( \text{MA}(q) \) is

\[ \gamma_k = \mathbb{C}[z_t, z_{t-k}] = \mathbb{E}[z_t, z_{t-k}] \]

\[ \gamma_k = \mathbb{E}[(\theta_0 \epsilon_{t-1} + \ldots + \theta_{q-1} \epsilon_{t-q} + \epsilon_t)(\theta_0 \epsilon_{t-k-1} + \ldots + \theta_{q-1} \epsilon_{t-k-q} + \epsilon_{t-k})] \quad (11.47) \]

**When \( k = 0 \)**

Letting \( k = 0 \) will give the variance \( \gamma_0 = \mathbb{V}[z_t] \)

\[ \gamma_0 = \mathbb{E}[(\theta_0 \epsilon_{t-1} + \ldots + \theta_{q-1} \epsilon_{t-q} + \epsilon_t)(\theta_0 \epsilon_{t-1} + \ldots + \theta_{q-1} \epsilon_{t-q} + \epsilon_{t-k})] \]

Since the noise shocks are independent, i.e., \( \mathbb{C}[\epsilon_t, \epsilon_u] = 0 \) unless \( t = u \), many of the terms in the product drop out.

\[ \gamma_0 = \theta_0^2 \mathbb{E} [\epsilon_{t-1}^2] + \ldots + \theta_{q-1}^2 \mathbb{E} [\epsilon_{t-q}^2] + \mathbb{E} [\epsilon_t^2] \]

The variance of the noise shocks is defined, for any \( t \), to be \( \mathbb{V}[\epsilon_t] = \mathbb{E}[\epsilon_t^2] = \sigma_\epsilon^2 \), so

\[ \gamma_0 = (\theta_0^2 + \theta_1^2 + \ldots + \theta_{q-1}^2 + 1) \sigma_\epsilon^2 \quad (11.48) \]

**When \( k \geq 1 \)**

For \( k \in [1, \ldots, q] \), similar equations can be created, only with parameter index shifted so the noise shocks match up.

\[ \gamma_1 = \mathbb{E}[(\theta_0 \epsilon_{t-1} + \theta_1 \epsilon_{t-2} + \ldots + \theta_{q-1} \epsilon_{t-q} + \epsilon_t)(\theta_0 \epsilon_{t-1-1} + \theta_1 \epsilon_{t-1-2} + \ldots + \theta_{q-1} \epsilon_{t-1-q} + \epsilon_{t-1})] \]

\[ \gamma_1 = (\theta_0 + \theta_1 \theta_0 + \theta_2 \theta_1 + \ldots + \theta_{q-1} \theta_{q-2}) \sigma_\epsilon^2 \]

\[ \gamma_k = \mathbb{E}[(\theta_0 \epsilon_{t-1} + \theta_1 \epsilon_{t-2} + \ldots + \theta_{q-1} \epsilon_{t-q} + \epsilon_t)(\theta_0 \epsilon_{t-k-1} + \theta_1 \epsilon_{t-k-2} + \ldots + \theta_{q-1} \epsilon_{t-k-q} + \epsilon_{t-k})] \]

\[ \gamma_k = (\theta_{k-1} + \theta_k \theta_0 + \theta_{k+1} \theta_1 + \ldots + \theta_{q-1} \theta_{q-k-1}) \sigma_\epsilon^2 \quad (11.49) \]
As before the equation for the $k^{th}$-lag auto-correlation (part of the Auto-Correlation Function (ACF)) is simply $\gamma_k / \gamma_0$.

$$\rho_k = (\theta_{k-1} + \theta_k \theta_0 + \theta_{k+1} \theta_1 + \ldots + \theta_{q-1} \theta_{q-k-1}) \frac{\sigma^2}{\gamma_0}$$ (11.50)

Notice that when $k > q$, the ACF will be zero. This is because only the last $q$ noise shocks are included in the model, so any earlier noise shocks before that are forgotten. MA processes will tend to exhibit a more rapid drop off of the ACF compared to the slow decay for AR processes.

Unfortunately, the system of equations that can be generated from these equations are nonlinear. Consequently, training is more difficult and less efficient.

### 11.5.2 Training

**Training for MA(1)**

Training for Moving Average models is easiest to understand for the case of a single parameter $\theta_0$. In general, training is about errors and so rearranging the MA equation gives the following:

$$e_t = z_t - \theta_0 e_{t-1}$$ (11.51)

Given a value for parameter $\theta_0$, this is a recursive equation that can be used to compute subsequent errors from previous ones. Unfortunately, the equation cannot be used to compute the first error $e_0$. One approach to deal with indeterminacy of $e_0$ is to assume (or condition on) it being 0.

$$e_0 = 0$$ (11.52)

Note, this affects more than the first error, since the first affects the second and so on. Next, we may compute a sum of squared errors, in this case called Conditional Sum of Squared Errors (denoted in ScalaTion as \text{csse}).

$$\text{csse} = \sum_{t=0}^{m-1} e_t^2 = \sum_{t=1}^{m-1} (z_t - \theta_0 e_{t-1})^2$$ (11.53)

One way to find a near optimal value for $\theta_0$ is to minimize the Conditional Sum of Squared Error.

$$\arg\min_{\theta_0} \sum_{t=1}^{m-1} (z_t - \theta_0 e_{t-1})^2$$ (11.54)

As the parameter $\theta_0 \in (-1, 1)$, an optimal value minimizing $\text{csse}$ may be found using Grid Search. A more efficient approach is to use the Newton method for optimization.

$$\theta_0' = \theta_0^{-1} - \frac{d \text{csse}}{d \theta_0} \frac{d^2 \text{csse}}{d \theta_0^2}$$ (11.55)

where

$$\frac{d \text{csse}}{d \theta_0} = -2 \sum_{t=1}^{m-1} e_{t-1} (z_t - \theta_0 e_{t-1})$$
\[ \frac{d^2 \text{csse}}{d\theta_0^2} = 2 \sum_{t=1}^{m-1} e_{t-1}^2 \]

Substituting gives

\[ \theta_0^i = \theta_0^{i-1} + \sum_{t=1}^{m-1} e_{t-1} (z_t - \theta_0 e_{t-1}) / \sum_{t=1}^{m-1} e_{t-1}^2 \]  

(11.56)

Maximum Likelihood Estimation (MLE) is also used for MA(q) parameter estimation.

### 11.5.3 Exercises

1. For an MA(1) model, solve for \( \theta_0 \) using the equation for the noise variance and the equation for the correlation.

\[ \gamma_0 = (1 + \theta_0^2) \sigma_t^2 \]
\[ \rho_1 = (\theta_0) \sigma_t^2 = \frac{\theta_0}{1 + \theta_0^2} \]

Solve for \( \theta_0 \) in terms of \( \rho_1 \).

2. Develop an MA(1) model for the Lake Level Time-series Dataset using the solution you derived in the previous question. Plot \( y_t \) and \( \hat{y}_t \) vs. \( t \). Estimate the mean Lake level and then plot \( z_t \) and \( \epsilon_t \).

3. Use the MA(1) solution you derived above for \( \theta_0 \) and the fact that \( \phi_0 = \rho_1 \) for AR(1) to produce two Forecast Charts for the first 10 values in the Lake-Level Dataset, one for MA(1) and one for AR(1).

\[ \hat{z}_t = \theta_0 \epsilon_{t-1} + \epsilon_t \]  
MA(1)

\[ \hat{z}_t = \phi_0 y_{t-1} + \epsilon_t \]  
AR(1)

Compute \( R^2 \) and SMAPE from information in the charts. Which model has a better Quality of Fit (QoF)?

4. Use ScalaTion to assess the quality of an MA(1) model versus an MA(2) model for the Lake Level Time-series Dataset.

5. The position in the ACF plot where \( \rho_k \) drops off can be used for \( q \), the order of the model. What does this plot suggest for the Lake Level Time-series Dataset?

6. Deduce the fact the drift \( \delta = \mu \) for all MA(q) models.
The ARMA class provides basic time series analysis capabilities for Auto-Regressive (AR) and Moving Average (MA) models. In an ARMA($p$, $q$) model, $p$ and $q$ refer to the order of the Auto-Regressive and Moving Average components of the model. ARMA models are often used for forecasting.

Recall that a $p$th-order Auto-Regressive AR($p$) model predicts the next value $y_t$ from a weighted combination of prior values and that a $q$th-order Moving Average MA($q$) model predicts the next value $y_t$ from the combined effects of prior noise/disturbances.

A combined $p$th-order Auto-Regressive, $q$th-order Moving Average ARMA($p$, $q$) model predicts the next value $y_t$ from both a weighted combination of prior values and the combined effects of prior noise/disturbances.

$$y_t = \delta + \phi \cdot y_{t-p} + \theta \cdot \epsilon_{t-q} + \epsilon_t$$ (11.57)

This can be expanded into

$$y_t = \delta + \phi_0 y_{t-1} + ... + \phi_{p-1} y_{t-p} + \theta_0 \epsilon_{t-1} + ... + \theta_{q-1} \epsilon_{t-q} + \epsilon_t$$ (11.58)

### 11.6.1 Selection Based on ACF and PACF

The Auto-Correlation Function (ACF) and Partial Auto-Correlation Function (PACF) may be used for choosing values for the hyper-parameters $p$ and $q$, for AR($p$), MA($q$) and ARMA($p$, $q$) models. The ACF is computed to a maximum number of lags and when plotted shows the auto-correlation $\rho_k$ versus the number of lags. When the ACF as a function of $k$ drops in value toward zero, one may select a value for $q$ around this $k$.

$$\rho_k = \frac{C[z_t, z_{t-k}]}{\sqrt{C[z_t]}} = \frac{\mathbb{E}[z_t z_{t-k}]}{\mathbb{E}[z_t z_t]} = \frac{\gamma_k}{\gamma_0}$$ (11.59)

Similarly, the PACF is computed to a maximum number of lags and when plotted shows the partial auto-correlation $\psi_{kk}$ versus the number of lags. When the PACF as a function of $k$ drops in value toward zero, one may select a value for $p$ around this $k$.

$$\psi_{kk} = \left[R^{-1} \rho_{1:k}\right]_k$$ (11.60)

Figure 11.2 shows the ACF $\rho_k$ and PACF $\psi_{kk}$ values versus the number of lags $k$. The PACF indicates the correlation between $z_t$ and $z_{t-k}$ beyond the chaining effect. For example, $z_t$ and $z_{t-2}$ would naturally be correlated at the level $\rho_2^2$, since the correlation between $z_t$ and $z_{t-1}$ is $\rho_1$, and $z_{t-1}$ and $z_{t-2}$ is $\rho_1$. Thus, partial correlation is a better indication of whether a term $\phi_k z_{t-k}$ contributes to the model.

From Figure 11.2 three partial correlations are worthy of consideration: $\rho_1 = 0.831911$, $\rho_2 = -0.285357$, and maybe $\rho_{10} = -0.237926$. From this perspective, candidate models are AR(1), AR(2) and AR(10). Of course, AR(10) will include several terms that contribute very little, so if a software package supports feature selection, they can be removed so that only three terms will be included in the model. Other options include maximizing $\bar{R}^2$ or minimizing AIC. See the exercise.
ACF(-) and PACF(+) vs lags $k$

Figure 11.2: Auto-Correlation Function (ACF) and Partial Auto-Correlation Function (PACF)

Table 11.2: AR Optimization Algorithms

<table>
<thead>
<tr>
<th>Technique</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MoM</td>
<td>Durbin-Levinson Solution to Yule-Walker Equations</td>
</tr>
<tr>
<td>OLS</td>
<td>Ordinary Least Squares</td>
</tr>
<tr>
<td>WLS</td>
<td>Weighted Least Squares</td>
</tr>
<tr>
<td>GLS</td>
<td>Generalized Least Squares</td>
</tr>
<tr>
<td>MLE</td>
<td>Maximum Likelihood Estimation</td>
</tr>
</tbody>
</table>

11.6.2 Training

Training for ARMA models involves simultaneously finding $p$ values for the AR part and $q$ values for the MA part. We have seen that are multiple optimization algorithms for finding the $\phi$ parameters for AR models.

Similarly, there are also multiple, more complex, optimization algorithms for finding the $\theta$ parameters for MA models.

Table 11.3: MA Optimization Algorithms

<table>
<thead>
<tr>
<th>Technique</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MoI</td>
<td>Method of Innovations</td>
</tr>
<tr>
<td>cSSE</td>
<td>Conditional Sum of Squares</td>
</tr>
<tr>
<td>MLE</td>
<td>Maximum Likelihood Estimation</td>
</tr>
</tbody>
</table>

393
**ARMA Class**

**Class Methods:**

```java
@param y_ the original input vector (time series data)
@param n the total number of parameters to estimate

class ARMA (y_ : VectoD, n: Int)
    extends ForecasterVec (y_, n, null)

def setTS (ts: VectoD)
def setPQ (p_: Int = 0, q_: Int = 0)
def train (): ARMA =
def nll (b: VectoD): Double =
def updateFittedValues (): Double =
def fittedValues (): VectoD = zp + mu_
override def predictAllz (): VectoD = fittedValues () - mu_
override def predictAll (): VectoD =
def forecast (t: Int = y.dim, h: Int = 1): VectoD =
def forecastAll (h: Int = 1): MatriD =
override def parameter: VectoD = if (q == 0) phi else if (p == 0) theta else phi ++ theta
def eval_ (e: VectoD = e, y: VectoD = y, yp: VectoD = fittedValues): ARMA =
def eval_ (y: VectoD, yp: VectoD): ARMA = eval_ (y - yp, y, yp)
```

**11.6.3 Exercises**

1. Plot the PACF for the Lake Level Time-series Dataset and use it to pick a value for $p$ based on rule 1: “stop at the drop” and rule 2: “pull all peaks”. Run AR for $p$ ranging from 1 to 10. Assess the Quality of Fit (QoF) for each AR($p$) model over this range. What would $\bar{R}^2$, AIC, and MAPE/sMAPE pick, respectively?

2. Rule 1 and especially rule 2 should be done within the context of significance, based on the following critical values,

$$
\rho_k^* = \frac{z_{\alpha/2}}{\sqrt{m-k}}
$$

where $m$ is the length of the time-series, $k$ is the lag, and $\alpha$ is the significance level (e.g., for $\alpha = .95$, $z_{\alpha/2} = 1.96$). Determine all peaks (both positive and negative) that are outside the interval $[-\rho_k^*, \rho_k^*]$.

3. Plot the ACF for the Lake Level Time-series Dataset and use it to pick a value for $q$. Assess the quality of an MA($q$) with this value for $q$. Try it for $q$ being one lower and one higher.

4. Using the selected values for $p$ and $q$ from the two previous exercises, assess the quality of an ARMA($p$, $q$). Try the four possibilities around the point ($p$, $q$).
5. For an ARMA \((p, q)\) model, explain why the Partial Auto-correlation Function (PACF) is useful in choosing a value for the \(p\) AR hyper-parameter.

6. For an ARMA \((p, q)\) model, explain why the Auto-correlation Function (ACF) is useful in choosing a value for the \(q\) MA hyper-parameter.
11.7 Rolling-Validation

Although cross-validation is very useful for prediction and classification problems, there is no direct analog for time-series data. Instances cannot be randomly selected for training and testing, because the instances are now time ordered and dependency is high between instances that are close in time (e.g., as measured by auto-correlation).

In time-series, a rolling-validation can be used as a replacement for cross-validation. Rolling-validation works by defining two adjacent, non-overlapping time windows: the first containing training data and the second containing testing data. These two windows are rolled forward in time together.

11.7.1 1-Fold Rolling Validation

The simplest form of rolling-validation is one-fold. The first time window is for training and has length $l$, while the second time window is for testing and has length $\lambda = m - l$.

\[
W^{(r)} = \{t_0, \ldots, t_{l-1}\} \quad \text{training} \tag{11.62}
\]

\[
W^{(e)} = \{t_l, \ldots, t_{m-1}\} \quad \text{testing} \tag{11.63}
\]

Based on these time windows, the response vector $y$ is chopped into two subvectors:

\[
y^{(r)} = y[W^{(r)}] \quad \text{training} \tag{11.64}
\]

\[
y^{(e)} = y[W^{(e)}] \quad \text{testing} \tag{11.65}
\]

Start by training the model on $y^{(r)}$. Then iterate through $y^{(e)}$ making a forecast for each element in this vector, producing a forecast vector $\hat{y}$. The forecast error is then

\[
\epsilon = y^{(e)} - \hat{y} \tag{11.66}
\]

Retraining

If the testing window is long, training may not be up to date, so one would expect the forecasts to degrade as the algorithm iterates through the testing data. This would argue for periodic retraining within the testing loop. Depending on the runtime efficiency of the training algorithm, the retaining may be more or less frequent. See the SimpleRollingValidation object in the scalation.analytics.forecaster package for details.

Multi-Horizon Forecasting

Although one step ahead forecasting is the most accurate, today’s forecasting models are capable of producing useful forecasts for several steps ahead. As an example, Numerical Weather Prediction (NWP) produces 14-day forecasts. With a time-step on 6 hours, this would imply 56 step ahead forecasts. In this case the forecasting horizon $h = 56$. Note, among other techniques, NWP uses Ensemble Kalman Filters, see the chapter on State-Space Models.

There are three main techniques available for multi-horizon forecasting [101];
1. The **Direct Method** applies a trained forecasting function \( f \) to recent data to obtain an \( h \)-step ahead forecast.

\[
\hat{y}_{t+h-1} = f(y_{t-p:t-1}; \phi) \quad (11.67)
\]

\[
\hat{y}_{t+h-1} = f([y_{t-p}, y_{t-p+1}, \ldots, y_{t-2}, y_{t-1}]; \phi) \quad (11.68)
\]

Clearly, the training must be based upon minimizing the \( h \)-step ahead forecast error (or maximizing the likelihood).

2. The **Recursive Method** builds up forecasts one step at a time. For efficiency it is implemented in an iterative rather than recursive manner. For example, for \( h = 3 \) and \( p \geq h \), forecasts would be produced as follows:

\[
\hat{y}_{t} = f([y_{t-p}, y_{t-p+1}, \ldots, y_{t-2}, y_{t-1}]; \phi) \quad (11.69)
\]

\[
\hat{y}_{t+1} = f([y_{t-p+1}, y_{t-p+2}, \ldots, y_{t-1}, \hat{y}_{t}]; \phi) \quad (11.70)
\]

\[
\hat{y}_{t+2} = f([y_{t-p+2}, y_{t-p+3}, \ldots, \hat{y}_{t}, \hat{y}_{t+1}]; \phi) \quad (11.71)
\]

Notice that the farther into the future the forecast is made, the more it depends on prior forecasts (not actual data). As errors can compound, longer term forecast may degrade.

**Warning Concerning Multi-Horizon Forecasting with MA(\( q \)) Models**

Predicting future shocks/errors is precarious, so order \( q \) needs to be at least the maximum forecasting horizon \( h \) for MA(\( q \)) models. When \( h > q \), all the errors will need to be forecasted and the best estimate for future noise is zero, so MA(\( q \)) forecasts will quickly degenerate to the mean of the time-series (like the NullModel).

Table 11.4 show how Multi-Horizon Forecasting is performed in SCALATION. Given a time-series split into a training set \( W^{(r)} \) and testing set \( W^{(e)} \) with sizes,

\[
I = 0.6m
\]

\[
\lambda = m - I
\]

the table lists the actual/true values in the first column. The next three columns show the 1-step, 2-step and 3-step ahead forecasts for the time corresponding to the \( y_t \) given in the first column, i.e., \( t+1, t+2 \) and \( t+3 \).

The values for the same time forecast (e.g., \( \hat{y}_1 \) for \( h = 1 \) and for \( h = 2 \)) are different for different horizons, so computation of forecasts should be column-by-column. At the beginning of the table, forecasts are blank that cannot be computed unless imputation or backcasting is performed. For example, \( \hat{y}_1 \) would require actual vales for \( y_{-2}, y_{-1} \) and \( y_0 \). Similarly, \( \hat{y}_2 \) would require actual vales for \( y_{-1}, y_{-0} \) and \( y_1 \).
Values at negative times are not available. The question marks at the end of the table indicate that a forecast can be made, but QoF measures such as MAPE cannot use them since actual/true values are unknown. The double line in the table divides the forecasts in those used for training and those used for testing.

The retraining period $\tau$ indicates the number of rows to be forecasted before retraining occurs to re-estimate the parameters $\phi$. Using the direct method, a parameter vector is created for each horizon: $\phi^{(1)}$, $\phi^{(2)}$ and $\phi^{(3)}$, while for the recursive method only $\phi^{(1)}$ is needed. In the table, the retraining period $\tau = 4$. The retraining period hyper-parameter $\tau$ should be adjusted to optimize between Quality of Fit (QoF) and model execution times (which can be substantial for Neural Networks).

Table 11.4: Multi-Horizon ($h = 3$) Forecasting for AR(3) with $\tau = 4$

<table>
<thead>
<tr>
<th>value $y_t$</th>
<th>forecast $h = 1$</th>
<th>forecast $h = 2$</th>
<th>forecast $h = 3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y_0$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$y_1$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$y_2$</td>
<td>$\hat{y}_3$</td>
<td>$\hat{y}_4$</td>
<td>$\hat{y}_5$</td>
</tr>
<tr>
<td>$y_3$</td>
<td>$\hat{y}_4$</td>
<td>$\hat{y}_5$</td>
<td>$\hat{y}_6$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$y_{l-3}$</td>
<td>$\hat{y}_{l-2}$</td>
<td>$\hat{y}_{l-1}$</td>
<td>$\hat{y}_l$</td>
</tr>
<tr>
<td>$y_{l-2}$</td>
<td>$\hat{y}_{l-1}$</td>
<td>$\hat{y}_l$</td>
<td>$\hat{y}_{l+1}$</td>
</tr>
<tr>
<td>$y_{l-1}$</td>
<td>$\hat{y}_l$</td>
<td>$\hat{y}_{l+1}$</td>
<td>$\hat{y}_{l+2}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>train</th>
<th>$\phi^{(1)}$</th>
<th>$\phi^{(2)}$</th>
<th>$\phi^{(3)}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y_l$</td>
<td>$\hat{y}_{l+1}$</td>
<td>$\hat{y}_{l+2}$</td>
<td>$\hat{y}_{l+3}$</td>
</tr>
<tr>
<td>$y_{l+1}$</td>
<td>$\hat{y}_{l+2}$</td>
<td>$\hat{y}_{l+3}$</td>
<td>$\hat{y}_{l+4}$</td>
</tr>
<tr>
<td>$y_{l+2}$</td>
<td>$\hat{y}_{l+3}$</td>
<td>$\hat{y}_{l+4}$</td>
<td>$\hat{y}_{l+5}$</td>
</tr>
<tr>
<td>$y_{l+3}$</td>
<td>$\hat{y}_{l+4}$</td>
<td>$\hat{y}_{l+5}$</td>
<td>$\hat{y}_{l+6}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>retrain</th>
<th>$\phi^{(1)}$</th>
<th>$\phi^{(2)}$</th>
<th>$\phi^{(3)}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$y_{m-4}$</td>
<td>$\hat{y}_{m-3}$</td>
<td>$\hat{y}_{m-2}$</td>
<td>$\hat{y}_{m-1}$</td>
</tr>
<tr>
<td>$y_{m-3}$</td>
<td>$\hat{y}_{m-2}$</td>
<td>$\hat{y}_{m-1}$</td>
<td>$\hat{y}_m$</td>
</tr>
<tr>
<td>$y_{m-2}$</td>
<td>$\hat{y}_{m-1}$</td>
<td>$\hat{y}_m$</td>
<td>$\hat{y}_{m+1}$</td>
</tr>
<tr>
<td>$y_{m-1}$</td>
<td>$\hat{y}_m$</td>
<td>$\hat{y}_{m+1}$</td>
<td>$\hat{y}_{m+2}$</td>
</tr>
</tbody>
</table>

Another potential problem may occur if the training set $l$ is not much greater than the order $p$ ($l \gg p$), as training cannot utilize the first $p-1$ rows, so for the Lake-Level Dataset where $m = 98$ and $l = 59$, the first training will involve only 57 instances, while the rest will have 59 instances.

Finally, the last $h = 3$ rows cannot be used for Quality of Fit (QoF) calculations.

3. **Hybrid Methods** try to combine the best the direct and recursive methods.
11.7.2 k-Fold Rolling-Validation

1-fold Rolling-Validation with retraining and multi-horizon forecasting is already complex. Still the approach lacks the ability to create confidence intervals as was done with $k$-fold cross-validation.

One can define $k$-fold rolling-validation by using $j \in \{0, \ldots, k - 1\}$ to keep track of the current fold. Now each fold has two adjacent, non-overlapping time windows. The first time window is for training and has length $l$, while the second time window is for testing and has length $\lambda = \lfloor \frac{m-l}{k} \rfloor$.

$$\mathcal{W}_j^{(r)} = \{t_{\lambda j}, \ldots, t_{l+\lambda j-1}\}$$  training  \hspace{1cm} (11.72)

$$\mathcal{W}_j^{(e)} = \{t_{l+\lambda j}, \ldots, t_{l+\lambda (j+1)-1}\}$$  testing  \hspace{1cm} (11.73)

For example, let $m = 200$, $k = 10$, $l = 100$, then $\lambda = 10$ and the ten pairs of time windows are shown below.

$$\mathcal{W}_i^{(r)} = \{t_0, \ldots, t_{99}\}, \{t_{10}, \ldots, t_{109}\}, \ldots, \{t_{90}, \ldots, t_{189}\}$$  \hspace{1cm} (11.74)

$$\mathcal{W}_i^{(e)} = \{t_{100}, \ldots, t_{109}\}, \{t_{110}, \ldots, t_{119}\}, \ldots, \{t_{190}, \ldots, t_{199}\}$$  \hspace{1cm} (11.75)

$$\mathcal{W}_i^{(s)} = \{t_{200}, \ldots, t_{209}\}, \{t_{210}, \ldots, t_{219}\}, \ldots, \{t_{290}, \ldots, t_{299}\}$$  \hspace{1cm} (11.76)

See the RollingValidation object in the scalation.analytics.forecaster package for details.
11.8 ARIMA

In cases where a time-series is not stationary, an ARIMA \((p, d, q)\) may be used. The new hyper-parameter \(d\) is the number of times the time-series is differenced. When there is a trend in the data (e.g., the values or levels are increasing over time), an ARIMA \((p, 1, q)\) may be effective. This model will take a first difference of the values in the time-series, i.e.,

\[
y_t' = \Delta y_t = y_t - y_{t-1}
\]  

(11.77)

This new 'differenced' time-series can then be put into an ARMA\((p, q)\) model to predict the next difference \(y_t'\) from both a weighted combination of prior values and the combined effects of prior noise/disturbances.

\[
y_t' = \delta + \phi \cdot y_{t-p} + \theta \cdot \epsilon_{t-q} + \epsilon_t
\]  

(11.78)

This can be expanded into

\[
y_t' = \delta + \phi_0 y_{t-1} + \ldots + \phi_{p-1} y_{t-p} + \theta_0 \epsilon_{t-1} + \ldots + \theta_{q-1} \epsilon_{t-q} + \epsilon_t
\]  

(11.79)

A second difference \((d = 2)\) is computed as the difference of the first difference.

\[
y_t'' = y_t' - y_{t-1} = y_t - 2y_{t-1} + y_{t-2}
\]  

(11.80)

Higher differencing may, but is not commonly used.

Training for an ARIMA applies the optimization algorithms used for ARMA models (see the last section). Note, the “I” in ARIMA could be interpreted to mean, once you have taken a difference (like a derivative), it should be “Integrated” back to get the final forecast (see the subsection below).

11.8.1 Differencing

The `diff` method in the `package` object for `Forecaster` takes a time-series \(y\) as a vector and takes the first difference in-place (i.e., it is overwritten). Note that the 'differenced' series has one less element than the original series. This allows the first element in the original time-series to be stored in the 'differenced' series. Maintaining this value allows the original time-series to be exactly restored from the 'differenced' series.

```scala
def diff (y: VectoD)
{
    for (i <- y.dim - 1 to 1 by -1) y(i) -= y(i-1)
} // diff
```

Restoration of the original times-series is achieved using the `diffinv` (inverse difference) method.

```scala
def diffinv (y: VectoD)
{
    for (i <- 1 until y.dim) y(i) += y(i-1)
} // diffinv
```
11.8.2 Forecasting

The difference forecast is produced from the ARMA model.

\[ \hat{y}_t' = \delta + \phi \cdot \hat{y}_{t-p:t-1} + \theta \cdot \epsilon_{t-q:t-1} \]  \hspace{1cm} (11.81)

The level forecast then simply undoes the differencing.

\[ \hat{y}_t = \hat{y}_t' + y_{t-1} \]  \hspace{1cm} (11.82)

11.8.3 Backshift Operator

For more complex models, it become convenient to define the backshift operator \( B^k \),

\[ B^k y_t = y_{t-k} \]  \hspace{1cm} (11.83)

that moves back \( k \) positions in the time series. Consequently, an AR(\( p \)) model may be written as

\[ y_t = \delta + [\phi_0 B^p + \cdots + \phi_{p-1} B^1] y_t + \epsilon_t \]  \hspace{1cm} (11.84)

In vector notation, this becomes

\[ y_t = \delta + \phi \cdot [B^p, \ldots, B^1] y_t + \epsilon_t \]  \hspace{1cm} (11.85)

Similarly, an MA(\( q \)) may be written as

\[ y_t = \delta + \theta \cdot [B^q, \ldots, B^1, 1] \epsilon_t \]  \hspace{1cm} (11.86)

Combining these two gives an ARMA(\( p, q \)) model.

\[ y_t = \delta + \phi \cdot [B^p, \ldots, B^1] y_t + \theta \cdot [B^q, \ldots, B^1, 1] \epsilon_t \]  \hspace{1cm} (11.87)

Notice that \((1 - B^1)y_t = y_t - y_{t-1}\), i.e., the first difference and in general, \((1 - B^1)^d\) is the \( d^{th} \) difference. This can be used to define an ARIMA(\( p, 0, q \)) model.

\[ \phi \cdot [-B^p, \ldots, -B^1, 1] y_t = \delta + \theta \cdot [B^q, \ldots, B^1, 1] \epsilon_t \]  \hspace{1cm} (11.88)

This is just the ARMA model with the AR part collected on the right and the MA collected on the left. In particular, \( y_t \) is not repeated. So why the more obscure formulation. Well, it makes it easy to apply the differencing in an ARIMA(\( p, d, q \)) model.

\[ \phi \cdot [-B^p, \ldots, -B^1, 1](1 - B)^d y_t = \delta + \theta \cdot [B^q, \ldots, B^1, 1] \epsilon_t \]  \hspace{1cm} (11.89)
11.8.4 Stationarity

ARIMA Class

Class Methods:

@param y_ the original input vector (time series data)
@param d the order of Integration/simple differencing
@param n the total number of parameters to estimate

class ARIMA (y_ : VectoD, d: Int, n: Int)
    extends ForecasterVec (y_, n, null)

    def differenceTrend (xx: VectoD, d: Int): VectoD =
    def setTS (ts: VectoD)
    def setPQ (p_: Int = 0, q_ : Int = 0)
    def train (): ARIMA =
    def nll (input: VectoD): Double =
    def updateFittedValues (): Double =
    def fittedValues (): VectoD = if (differenced) transformBackTrend (zp, y, d) else zp + mu_
    override def predictAllz (): VectoD =
    override def predictAll (): VectoD =
    def forecast (t: Int = y_.dim, h: Int = 1): VectoD =
    def forecastAll (h: Int = 1): MatriD =
    override def parameter: VectoD = if (q == 0) phi else if (p == 0) theta else phi ++ theta
    def residuals () = if (differenced) y - fittedValues() else e
    def eval_ (e: VectoD = residuals, y: VectoD = y, yp: VectoD = fittedValues): ARIMA =
    def eval_ (y: VectoD, yp: VectoD): ARIMA = eval_ (y - yp, y, yp)

11.8.5 Exercises

1. Use the diff method to create a version that takes the number of differences d as a second parameter. Do this (a) iteratively, (b) recursively.

2. Take a first difference of the Lake Level time-series dataset and plot the values versus time.

3. Look at the ACF and PACF plots for the Lake Level 'differenced series' to determine values for p and q. Create an ARIMA (p, 1, q) model. Plot its in-sample forecast \( \hat{y} \) versus the actual value \( y \) and determine its in-sample Quality of Fit (QoF) measures.

4. Use Grid Search on 0 to 19 for p and q to find the hyper-parameter values that (a) minimize AIC, (b) minimize sMAPE, (c) minimize MAE, (d) minimize RMSE, (e) maximize \( R^2 \), and (f) maximize \( \bar{R}^2 \). How do these compare?

5. Take the best model according to the consensus in the last question and determine its out-of-sample Quality of Fit (QoF) measures for forecasting horizon \( h = 1 \). Plot its out-of-sample forecast \( \hat{y} \) versus the actual value \( y \). Try this for a training to testing ratio of (a) 60-40 and (b) 70-30.
6. Determine the out-of-sample Quality of Fit (QoF) measures for forecasting horizons $h = 2$ to $14$ (two weeks ahead). How rapidly do the QoF measures decline.

7. Repeat the questions on the Lake-Level dataset for the Covid-19 dataset.

8. Compare an ARIMA (0, 1, 0) model and a Random Walk with Drift model. Remove the constant/drift from this ARIMA model and determine what other model type it matches.

9. Compare an ARIMA (0, 1, 1) model and Simple Exponential Smoothing with Growth model. Remove the constant/drift from this ARIMA model and determine what other model type it matches.
11.9 SARIMA

In this context, the word “seasonal” is used to mean any time period longer than one. For example, vehicle traffic may exhibit periodic behavior every 24 hours. As the time resolution is 5 minutes, the period or seasonal length \( s = 288 \) time units. Seasonal ARIMA allows for seasonal differences, so a SARIMA(\( p,d,q \),\( P,D,Q \)) is

\[
\phi \cdot [-B^p, \ldots, -B^1, 1](1 - B)^d(1 - B^s)^D y_t = \delta + \theta \cdot [B^q, \ldots, B^1, 1] \epsilon_t \tag{11.90}
\]

The seasonal differencing \((1 - B^s)^D\) happens first, followed by regular differencing \((1 - B)^d\) and then essentially an ARMA model. In addition, seasonal autoregressive parameter vector \(\phi^s \in \mathbb{R}^P\) and seasonal moving-average parameter vector \(\theta^s \in \mathbb{R}^Q\) may be added to a SARIMA(\( p,d,q \),\( P,D,Q \)) model.

\[
[\phi \cdot [-B^p, \ldots, -B^1, 1]] [\phi^s \cdot [-B^{sp}, \ldots, -B^s, 1]] (1 - B)^d(1 - B^s)^D y_t = \delta + [\theta \cdot [B^q, \ldots, B^1, 1]] [\theta^s \cdot [B^{sq}, \ldots, B^s, 1]] \epsilon_t \tag{11.91}
\]

The expression \([\phi \cdot [-B^p, \ldots, -B^1, 1]]\) may be viewed as a polynomial and the polynomials are multiplied. Because of this, it is referred to as a multiplicative SARIMA model. Other types of SARIMA models include additive and subset \([?]?\).

11.9.1 Exercises

1. Write out a SARIMA(5,1,3)\( \times (4,1,2)_s \) model, with and without using the backshift operator.
11.10 VAR

*Multivariate Time-Series Analysis* extends univariate time-series by analyzing multiple variables. It works on multiple interrelated time-series,

\[ y_t = [y_{t0}, y_{t1}, \ldots, y_{tn-1}] \tag{11.93} \]

with one time-series for each of the \( n \) variables [109]. By convention in this text, the time-series are oriented in a matrix so that the \( j^{th} \) variable is held in the \( j^{th} \) column and the \( t^{th} \) time point is held in the \( t^{th} \) row. This way the matrix holding the data will correspond to how data is typically organized in a csv data file.

The forecasted value for the \( j^{th} \) variable at time \( t \), \( y_{tj} \), can depend on the previous (or lagged) values of all the variables. This notion is captured in Vector Auto-Regressive Models [97, 120]. A Vector Auto-Regressive Model of order \( p \) with \( n \) variables, VAR\((p, n)\), will utilize the most recent \( p \) values for each variable to produce a forecast.

11.10.1 VAR\((p, 2)\)

When \( n = 1 \), VAR\((p, n)\) becomes AR\((p)\), so the simplest actual vector case is VAR\((p, 2)\) also known as a bivariate VAR\((p)\) model. Such models can be useful in traffic forecasting as flow and speed are related variables for which time-series data is maintained for each, i.e., \( y_{t0} \) is the traffic flow at a particular sensor at time \( t \) and \( y_{t1} \) is the traffic speed at that sensor at time \( t \).

For each lag, each of the lag variables is used to predict each response variable, so the complete set of parameters forms a tensor consisting of \( p \) matrices.

\[ \Phi = [\Phi^{(0)}, \Phi^{(1)}, \ldots, \Phi^{(p-1)}] \quad \text{where} \quad \Phi^{(l)} \in \mathbb{R}^{n \times n} \tag{11.94} \]

Written in matrix-vector form [42], a bivariate \((n = 2)\) VAR\((3)\) model is

\[ y_t = \delta + \Phi^{(0)}y_{t-3} + \Phi^{(1)}y_{t-2} + \Phi^{(2)}y_{t-1} + \epsilon_t \tag{11.95} \]

where \( \delta \in \mathbb{R}^n \) is a constant, \( \Phi^{(0)} \in \mathbb{R}^{n \times n} \) is the parameter matrix (first row of tensor) for first lags, \( \Phi^{(1)} \in \mathbb{R}^{n \times n} \) is the parameter matrix (second row of tensor) for second lags, \( \Phi^{(2)} \in \mathbb{R}^{n \times n} \) is the parameter matrix (third row of tensor) for third lags, and \( \epsilon_t \in \mathbb{R}^n \) is the residual/error/shock vector. Notice that \( \Phi^{(0)}y_{t-3} \) is a matrix-vector product yielding an \( n \)-dimensional vector. Also see the note about index (superscript/subscript) ordering in the section on AR models.

The equations for each component of the vector \( y_t = [y_{t0}, y_{t1}] \) are

\[
\begin{align*}
y_{t0} &= \delta_0 + \phi_{00}^{(0)} \cdot y_{t-3} + \phi_{01}^{(1)} \cdot y_{t-2} + \phi_{02}^{(2)} \cdot y_{t-1} + \epsilon_{t1} \quad \text{flow} \\
y_{t1} &= \delta_1 + \phi_{10}^{(0)} \cdot y_{t-3} + \phi_{11}^{(1)} \cdot y_{t-2} + \phi_{12}^{(2)} \cdot y_{t-1} + \epsilon_{t1} \quad \text{speed}
\end{align*}
\]

where for example, the vector \( \phi_0^{(1)} = [\phi_{00}^{(1)}, \phi_{01}^{(1)}] \) is the first row of matrix \( \Phi^{(1)} \). The two equations may be combined and rewritten in expanded matrix-vector notation.

\[
y_t = \begin{bmatrix} y_{t0} \\ y_{t1} \end{bmatrix} = \begin{bmatrix} \delta_0 \\ \delta_1 \end{bmatrix} + \begin{bmatrix} \phi_{00}^{(0)} & \phi_{01}^{(0)} \\ \phi_{10}^{(0)} & \phi_{11}^{(0)} \end{bmatrix} y_{t-3} + \begin{bmatrix} \phi_{00}^{(1)} & \phi_{01}^{(1)} \\ \phi_{10}^{(1)} & \phi_{11}^{(1)} \end{bmatrix} y_{t-2} + \begin{bmatrix} \phi_{00}^{(2)} & \phi_{01}^{(2)} \\ \phi_{10}^{(2)} & \phi_{11}^{(2)} \end{bmatrix} y_{t-1} + \epsilon_t
\]
11.10.2 VAR\((p, n)\)

A general Vector Auto-Regressive VAR\((p, n)\) model with \(n\) variables each with \(p\) lags will have \(p\) parameter matrices. The equation for the response vector \(y_t \in \mathbb{R}^n\) may be written in matrix-vector form as follows:

\[
y_t = \delta + \Phi^{(0)} y_{t-p} + \Phi^{(1)} y_{t-p+1} + \ldots + \Phi^{(p-1)} y_{t-1} + \epsilon_t \tag{11.96}
\]

where constant vector \(\delta \in \mathbb{R}^n\), parameter matrices \(\Phi^{(l)} \in \mathbb{R}^{n \times n}\) \((l = 0, \ldots, p-1)\), and error vector \(\epsilon_t \in \mathbb{R}^n\).

11.10.3 Training

One way to train a VAR model is to treat each variable as a separate regression problem and use least squares to estimate the parameters, e.g., the parameters for \(y_{tj}\) are matrix \(\Phi_{.j}\).

11.10.4 AR\(^*\)(\(p, n\))

An AR\(^*\)(\(p, n\)) model is a VAR\((p, n)\) model where all of the parameter matrices are diagonal. Thus, each variable/time-series \(y_{tj}\) is modeled independently, so for an AR\(^*\)(3, 2) model, the \(j^{th}\) equation is

\[
y_{tj} = \delta_j + \phi^{(0)}_{jj} y_{t-3,j} + \phi^{(1)}_{jj} y_{t-2,j} + \phi^{(2)}_{jj} y_{t-1,j} + \epsilon_{tj}
\]

11.10.5 Exercises

1. Plot the flow \(y_{t0}\) and speed \(y_{t1}\) given in the Traffic Sensor Dataset (traffic.csv). Estimate values for the four parameters contained in the one parameter matrix \(\Phi^{(0)}\) of a VAR (1, 2) model \((p = 1, n = 2)\). Compute the Quality of Fit (QoF).

2. Use the Traffic Sensor Dataset (traffic.csv) to estimate values for the three parameter matrices \(\Phi^{(0)}\), \(\Phi^{(1)}\) and \(\Phi^{(2)}\) of a VAR (3, 2) model \((p = 3, n = 2)\). Compute the Quality of Fit (QoF) and compare with the VAR(1, 2) model.

3. Consider what happens if two times series \(\{y_{t0}\}\) and \(\{y_{t1}\}\) are following similar trends. What difficulties could this cause in a VAR model.

4. How could a Vector Error Correction Model (VECM) handle the above problem?
11.11 Time-Series Regression with Lagged Variables

Given a predictor time-series \( \{x_t\} \) and a response time-series \( \{y_t\} \), a simple times-series regression model uses the current value of \( x_t \) to predict \( y_t \).

\[
y_t = b_0 + b_1 x_t + \epsilon_t
\] (11.97)

Knowing previous values of \( x_t \) often helps with the predictions, so it is common to introduce a lagged predictor variable \( x_{t-1} \).

\[
y_t = b_0 + b_1 x_t + b_2 x_{t-1} + \epsilon_t
\] (11.98)

One may also introduce a lagged response variable \( y_{t-1} \).

\[
y_t = b_0 + \phi_1 y_{t-1} + b_1 x_t + b_2 x_{t-1} + \epsilon_t
\] (11.99)

Doing so may introduce estimation problems for Ordinary Least Squares (OLS), so in some cases Generalized Least Square (GLS) may work better, or more likely, VAR may be a better alternative [50].

When contemporaneous values are not known, \( x_t \) may be dropped from the model equation. If the all \( x \)-values are dropped, then the model equation becomes the same as AR(\( p \)), and its estimation procedures will often be preferred (see [?] for conditions where the opposite is the case).

Of course these equations can be extended to multiple predictor variables and multiple lags.

11.11.1 Exercises

1. Compare Time-Series Regression with Lagged Variables with VAR models.
11.12 Nonlinear Time-Series Models

11.12.1 Nonlinear Autoregressive (NAR)

As was the case for Nonlinear Regression, Nonlinear Time-Series models have the potential for better fitting models.

A \( p^{th} \) order Nonlinear Autoregressive NAR(\( p \)) model is a generalization of an AR(\( p \)) model. The forecasted value at time \( t \), \( y_t \), is a function of previous values of \( y \), e.g.,

\[
\begin{align*}
x_t &= [y_{t-p}, \ldots, y_{t-1}] \\
y_t &= f(x_t; \phi) + \epsilon_t
\end{align*}
\]

where \( \phi \) is a vector of \( p \) parameters and \( p \) is also taken the number of lagged values to use. In general, for nonlinear models the number of lags may or may not equal the number of parameters \( p \).

11.12.2 Autoregressive Neural Network (ARNN)

A special case of a NAR(\( p \)) is an Autoregressive Neural Network ARNN(\( p \)) model. It is a three-layer (one hidden) neural network, where the input layer has a node for each of the \( p \) lags, the hidden layer also has \( p \) nodes, and the output layer has 1 node.

\[
\begin{align*}
x_t &= [y_{t-p}, \ldots, y_{t-1}] \\
y_t &= w \cdot f(\Phi x_t) + \epsilon_t
\end{align*}
\]

The \( p \times p \) matrix \( \Phi \) holds the parameters/weights connecting the input and hidden layers, while the \( p \) dimensional vector \( w \) holds the parameters/weights connecting the hidden and output layers. There is only an activation function (vectorized \( f \)) for the hidden layer. Note that \( \Phi x_t \) is matrix-vector multiplication. Of course, the basic ARNN model can be embellished.

11.12.3 Nonlinear Autoregressive, Moving-Average (NARMA)

A \( (p,q) \) order Nonlinear Autoregressive, Moving-Average NARMA(\( p, q \)) model, is a generalization of an ARMA(\( p, q \)) model.

\[
\begin{align*}
x_t &= [y_{t-p}, \ldots, y_{t-1}] & \text{input : past values} \\
e_t &= [\epsilon_{t-p}, \ldots, \epsilon_{t-1}] & \text{input : past shocks} \\
y_t &= f(x_t, e_t; \phi, \theta) + \epsilon_t & \text{output}
\end{align*}
\]
11.13 Recurrent Neural Networks (RNN)

11.13.1 RNN(1)

A basic Recurrent Neural Network cell/unit of order one makes forecasts based on the most recent information. In order to forecast a value for \( y_t \), denoted \( \hat{y}_t \), a weighted combination of the previous value \( x_t = y_{t-1} \) and the previous state \( h_{t-1} \) are passed to an activation function \( f \).

\[
\begin{align*}
x_t &= y_{t-1} \\
h_t &= f(\phi x_t + \theta h_{t-1} + \beta) \\
y_t &= h_t + \epsilon_t
\end{align*}
\]

(11.107) (11.108) (11.109)

where \( \phi \) and \( \theta \) are scalar parameters/weights and \( \beta \) is the bias. The above RNN model is a NARMA(1, 1) model [19].

The hidden (or state) variable \( h_t \) is defined recursively and as such provides memory to the model, since it values results from all previous values in the times [67].

Notice that if the activation function \( f \) is the identity function, the middle equation becomes,

\[
h_t = \beta + \phi x_t + \theta h_{t-1}
\]

(11.110)

so in this case it is an ARMA(1, 1) model, since \( h_{t-1} = y_{t-1} - \epsilon_{t-1} \).

11.13.2 RNN(\( p \))

A basic Recurrent Neural Network cell/unit of order \( p \) makes forecasts based on information going back \( p \) lags.

\[
\begin{align*}
x_t &= [y_{t-p}, \ldots, y_{t-1}] \\
h_t &= f(\Phi x_t + \Theta h_{t-1} + \beta) \\
y_t &= h_t(0) + \epsilon_t
\end{align*}
\]

where \( \Phi \) and \( \Theta \) are \( p \times p \) parameters/weights matrices and \( \beta \) is the \( p \)-dimensional bias vector. The activation function \( f \) defaults to the sigmoid function (vectorized). This RNN model is a NARMA(\( p \), \( p \)) model [19].

Unfortunately, such neural networks may have stability problems, so work moved unto units with gates that add stability (e.g., vanishing or exploding gradients). Gates are used to control how much of the previous state is preserved as signals propagate through the units. This allows gated units to have longer memories that simple RNNs.
11.14 Gated Recurrent Unit (GRU) Networks

Gated units have alleviated some of the problems with traditional RNNs. As the simplest gated unit, a Minimal Gated Unit (MGU) has a single gate and a minimal number of parameters compared to other gated units.

A Gated Recurrent Unit (GRU) is slightly more complex, but is more commonly used than an MGU. It adds a second gate, a third pair of parameter matrices and a third bias vector. The two gates in a GRU are the reset gate and the update gate. These are used to control the degree to which the previous state $h_{t-1}$ factors, along with the current input $x_t$, into the calculation of the new state $h_t$, which will be a mixture of the previous state $h_{t-1}$ and new candidate state $\tilde{h}_t$ coming out of the activation function $f$. Figure 11.3 shows how signals propagate through one unit. This unit would be connected to one on the left taking input $x_{t-1}$ and one on the right taking input $x_{t+1}$. The state provides memory for the next unit.

![Figure 11.3: Gated Recurrent Unit (GRU)](image)

The elements in the control vectors, reset $r_t$ and update $z_t$, come through sigmoid activation so they are always between 0 (open circuit) and 1 (closed circuit) and are shown in purple.

The following two equations indicate one way that a GRU could be set up to handle univariate time-series data, e.g., a NAR($p$) model.

$$x_t = [y_{t-1}, \ldots, y_{t-p}]$$  \hspace{1cm} \text{input}  \\
$$\hat{y}_t = g(h_t)$$  \hspace{1cm} \text{forecast}

Other forms of feature selection or engineering could be done as well; one could include time $t$, shocks, etc. or extend to multivariate time-series analysis.
GRU Equations

The equations below show how information flows through a gated recurrent unit [17, 118, 115].

**Reset Gate.** The degree to which the previous state $h_{t-1}$ influences the new candidate state $\hat{h}_t$ is controlled by the reset gate. When the reset gate is open ($r_t$ is close to zero) the previous state is all but ignored, whereas, when the gate is closed, its influence will be strong.

$$ r_t = f_r(U_r \cdot h_{t-1} + W_r \cdot x_t + \beta_r) $$ (11.111)

**Update Gate.** The update gate controls the relative mixture of the previous state $h_{t-1}$ and the new candidate state $\hat{h}_t$ used to form the new state $h_t$. When the update gate is open ($z_t$ is close to zero) the previous state is passed through almost intact, whereas, when the gate is closed, the new state is essentially the candidate state.

$$ z_t = f_z(U_z \cdot h_{t-1} + W_z \cdot x_t + \beta_z) $$ (11.112)

**Activate.** A candidate state $\hat{h}_t$ can be created by applying the activation function $f$ (e.g., tanh or reLU) to the weighted combination of the previous state $h_{t-1}$ and the current input $x_t$. However, the reset control $r_t$ can be used to cut off the influence of the previous state.

$$ \hat{h}_t = f(U_h \cdot [r_t \cdot h_{t-1}] + W_h \cdot x_t + \beta_h) $$ (11.113)

**Mix.** The new state $h_t$ is created as a mixture of the previous state $h_{t-1}$ and the newly created candidate state $\hat{h}_t$. The update control $z_t$ determines how much of each are put into the new state $h_t$.

$$ h_t = (1 - z_t) \cdot h_{t-1} + z_t \cdot \hat{h}_t $$ (11.114)

Recall that a matrix-vector dot product is equivalent to transposing the matrix and multiplying by the vector, e.g., $W_r \cdot x_t = W_r^T x_t$.

**GRU Variables and Parameters**

A Gated Recurrent Unit introduces a greater number of variables and parameters than a dense Feed Forward Neural Network. The variables for a GRU unit consist of one variable that serves as input $x_t$, one that represents state and is modified by the unit having before $h_{t-1}$ and after $h_t$ values, a candidate state $\hat{h}_t$, and two control variables, $r_t$ and $z_t$. These five vector-valued variables are listed in Table 11.5. All but one the variables has dimension $n$, the number of units in the GRU. That one, $x_t$ has dimension $p$ corresponding to the number of lags in univariate time-series (or course feature engineering can be used to add more).

Table 11.5: GRU Variables

<table>
<thead>
<tr>
<th>variable</th>
<th>dimensions</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_t$</td>
<td>$\mathbb{R}^p$</td>
<td>input</td>
</tr>
<tr>
<td>$r_t$</td>
<td>$\mathbb{R}^n$</td>
<td>reset control</td>
</tr>
<tr>
<td>$z_t$</td>
<td>$\mathbb{R}^n$</td>
<td>update control</td>
</tr>
<tr>
<td>$\hat{h}_t$</td>
<td>$\mathbb{R}^n$</td>
<td>candidate state</td>
</tr>
<tr>
<td>$h_t$</td>
<td>$\mathbb{R}^n$</td>
<td>state</td>
</tr>
</tbody>
</table>
The parameters for a GRU unit consist of six weight/parameter matrices (in three pairs) and three bias vectors as listed in Table 11.6.

<table>
<thead>
<tr>
<th>parameter</th>
<th>dimensions</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>$U_r$</td>
<td>$\mathbb{R}^{n \times n}$</td>
<td>state-to-reset weight matrix</td>
</tr>
<tr>
<td>$W_r$</td>
<td>$\mathbb{R}^{p \times n}$</td>
<td>input-to-reset weight matrix</td>
</tr>
<tr>
<td>$U_z$</td>
<td>$\mathbb{R}^{n \times n}$</td>
<td>state-to-update weight matrix</td>
</tr>
<tr>
<td>$W_z$</td>
<td>$\mathbb{R}^{p \times n}$</td>
<td>input-to-update weight matrix</td>
</tr>
<tr>
<td>$U_h$</td>
<td>$\mathbb{R}^{n \times n}$</td>
<td>state-to-activate matrix</td>
</tr>
<tr>
<td>$W_h$</td>
<td>$\mathbb{R}^{p \times n}$</td>
<td>input-to-activate weight matrix</td>
</tr>
<tr>
<td>$\beta_r$</td>
<td>$\mathbb{R}^n$</td>
<td>reset bias vector</td>
</tr>
<tr>
<td>$\beta_z$</td>
<td>$\mathbb{R}^n$</td>
<td>update bias vector</td>
</tr>
<tr>
<td>$\beta_h$</td>
<td>$\mathbb{R}^n$</td>
<td>activate bias vector</td>
</tr>
</tbody>
</table>

### 11.14.1 A GRU Layer

So far, a GRU has been discussed in isolation. To better visualize the role of GRUs in a neural network, first consider the three-layer Dense Feed-Forward Neural Network consisting of an input, hidden and output layer as shown in Figure 11.3.

![Figure 11.4: Three-Layer (input, hidden, output) Neural Network with Biases Removed](image)

Now, swap a GRU layer in for the dense hidden layer. Let the number of lags $p = 2$. Then at time $t$, the network maps input $x_t = [y_{t-2}, y_{t-1}]$ to output $y_t$. The weights shown in Figure 11.5 are meant to represent the weights in the three weight matrices, $W_r$, $W_u$ and $W_h$. Each of the two units takes the input vector $x_t$ and the previous state $h_{t-1}$ and computes the next state $h_t$. Note that the states are scalars here for an individualized unit. Computations, however, are performed over all $n$ units in the form of matrix-vector (or dot) products, e.g., $U_r \cdot h_{t-1}$ as depicted in Figure 11.3. The states coming out of the GRU layer are feed into a final dense layer to make one step-head forecasts $\hat{y}_t$. (This treatment can be extended for multi-horizon forecasts.)
The GRU iterates through time (using a loop in implementation). A useful way to visualize the execution is to duplicate the unit for each of the $m$ timestamps in the time-series. Figure 11.6 illustrates this, imagining $h_0$ and $h_1$ executing at time 0, $t - 1$, $t$, $t + 1$, and $m - 1$. Note, unless imputation or backcasting is used, there is no input for time 0. Consequently, for this example, one would set $x_0 = 0$ and $h_0 = 0$.

\[ h_0 @ t = 0 \quad \ldots \quad h_0 @ t - 1 \quad h_0 @ t \quad h_0 @ t + 1 \quad \ldots \quad h_0 @ m - 1 \]

\[ h_1 @ t = 0 \quad \ldots \quad h_1 @ t - 1 \quad h_1 @ t \quad h_1 @ t + 1 \quad \ldots \quad h_1 @ m - 1 \]

\[ x_{t - 1} \quad \quad x_t \quad \quad x_{t + 1} \quad \quad x_{m - 1} \]

\[ [y_{t - 3}, y_{t - 2}] \quad \quad [y_{t - 2}, y_{t - 1}] \quad \quad [y_{t - 1}, y_t] \quad \quad [y_{m - 3}, y_{m - 2}] \]

Figure 11.6: GRU in Execution over Time

See [26, 115] for additional details.

### 11.14.2 Training

### 11.14.3 Exercises

1. Given $U_r \in \mathbb{R}^{n \times n}$, $W_r \in \mathbb{R}^{p \times n}$, $h_{t - 1} \in \mathbb{R}^n$, and $x_t \in \mathbb{R}^p$, show that if $W = [U_r, W_r] \in \mathbb{R}^{n \times (n + p)}$, then

\[ W \cdot \begin{bmatrix} h_{t - 1} \\ x_t \end{bmatrix} = U_r \cdot h_{t - 1} + W_r \cdot x_t \tag{11.115} \]

Again, recall that $W_r \cdot x_t = W_r^T x_t$.

2. Use the above identity to rewrite the GRU equations using three parameter matrices and three bias vectors.
3. Write a formula for the total number of trainable parameters in a GRU where the state vector is \( n \) dimensional and the input vector is \( p \) dimensional.

4. Explain the difference between stateless and stateful training of a GRU (or MGU, LSTM) [26].
11.15 Other Recurrent Neural Networks

11.15.1 Minimal Gated Unit (MGU)

A Minimal Gated Unit (MGU) has a single gate and a minimal number of parameters compared to other gated units. Note, the GRU reset and update gates as well as their corresponding vectors are unified into the forget gate in an MGU. An MGU has a forget gate and its purpose is to weigh previous forecasts versus new information. The greater the forgetting the greater the reliance on recent data.

The input can be engineered like it was for GRUs, e.g.,

\[ x_t = [y_{t-1}, \ldots, y_{t-p}] \]

The equations below show how information flows through a minimal gated unit \[ [118, 115] \].

\[
\begin{align*}
  z_t &= f_s(U_z \cdot h_{t-1} + W_z \cdot x_t + \beta_z) \\
  \tilde{h}_t &= f(U_h \cdot [z_t \cdot h_{t-1}] + W_h \cdot x_t + \beta_h) \\
  h_t &= (1 - z_t) \cdot h_{t-1} + z_t \cdot \tilde{h}_t
\end{align*}
\]

The parameters consist of two pairs of weight/parameter matrices \( U_z \in \mathbb{R}^{n \times n}, W_z \in \mathbb{R}^{p \times n} \) and \( U_h \in \mathbb{R}^{n \times n}, W_h \in \mathbb{R}^{p \times n} \), as well as two bias vectors \( \beta_z \in \mathbb{R}^n \) and \( \beta_h \in \mathbb{R}^n \). The first activation function serves as a switch (typically sigmoid) and the second activation function defaults to tanh, but other activation functions (e.g., reLU) may be used.

In the above formulation, the dimensions of the vectors are follows: \( x_t \in \mathbb{R}^p \), \( z_t \), \( \tilde{h}_t \) and \( h_t \in \mathbb{R}^n \), where \( p \) is the number features engineered into the input and \( n \) is the number of units.

**Forget Gate.** The purpose of the forget gate is to determine how much of the previous state to forget. When the forget gate is open (\( z_t \) is close to zero) the previous state \( h_{t-1} \) is passed through almost intact, whereas, when the gate is closed, the new state is essentially the candidate state. The value of \( z_t \) also determines the influence of the previous state \( h_{t-1} \) has over the new candidate state \( \tilde{h}_t \).

11.15.2 Long Short Term Memory (LSTM) Networks

Long Short Term Memory (LSTM) Networks provide increased memory by introducing another state variable called the cell state \( c_t \) that works in parallel with the hidden state \( h_t \).

An LSTM has advantages over a GRU when its longer memory is beneficial and the time series is long enough for effective training. An LSTM has more parameters to train than a GRU, so it takes longer to train.

See \[ [115] \] for the equations specifying how information flows through an LSTM \[ [40, 31, 26] \]. The modern version has three gates: forget, input and output, as well as peephole connection.
Chapter 12

Dimensionality Reduction

When data matrices are very large with high dimensionality, analytics becomes difficult. In addition, there is likely to be co-linearity between vectors, making the computation of inverses or pseudo-inverses problematic. In such cases, it is useful to reduce the dimensionality of the data.
12.1 Reducer

The Reducer trait provides a common framework for several data reduction algorithms.

Trait Methods:

trait Reducer

def reduce (): MatriD
def recover (): MatriD
12.2 Principal Component Analysis (PCA)

The PrincipalComponents class computes the Principal Components (PCs) for data matrix \( X \) with the following dimensions

\[
X \in \mathbb{R}^{m \times n} \tag{12.1}
\]

where the number of rows \( m \) is the number of instances/samples and the number of columns \( n \) is the number of predictor variables.

Principal Component Analysis (PCA) can be used to reduce the dimensionality of the data matrix. Using the PrincipalComponents class, first find the PCs by calling ‘findPCs’ and then call ‘reduce’ to reduce the data (i.e., reduce matrix \( X \) to a lower dimensionality matrix).

### 12.2.1 Representation

PCA will replace the data matrix \( X \in \mathbb{R}^{m \times n} \) with a lower-dimensional reduced matrix \( Z \in \mathbb{R}^{m \times k} \) for \( k \leq n \).

\[
Z = X E_k \tag{12.2}
\]

The matrix \( E \) is the full eigen-decomposition of the covariance matrix of \( X \) \[1\].

\[
\Sigma [X] = \frac{X_c^t X_c}{m-1} \tag{12.3}
\]

where \( X_c = X - \mu_X \) is the centered data matrix.

\[
E = \text{eigendecomp} \left[ \frac{X_c^t X_c}{m-1} \right] \tag{12.4}
\]

The matrix \( E_k \) is the first \( k \) columns of \( E \) where the columns of \( E \) are ordered by the magnitude of the eigenvalues. The columns with the largest eigenvalues will explain most of the covariance. See the exercises to understand why the covariance matrix is used as the basis for the dimensionality reduction.

For a given vector \( x \) of dimension \( n \), its reduced representation \( z \) will have dimension \( k \).

\[
z = E_k x \tag{12.5}
\]

Note, that each element of vector \( z \) is a linear combination of elements in the original vector \( x \).

---

**Example Problem:**

**Class Methods:**

```python
@param x the data matrix to reduce, stored column-wise

class PrincipalComponents (x: MatriD)

def meanCenter (): VectoD =
def computeCov (): MatriD =
def computeEigenVectors (eVal: VectoD): MatriD =
def findPCs (k: Int): MatriD =
```
12.2.2 Exercises

1.
12.3 Autoencoder (AE)

An Autoencoder (AE) is a fully-connected neural network that contains a middle layer of lower dimensionality than the input layer. The output layer has the same dimensionality as the input layer, as shown in figure 12.1. The loss function then simply measures the difference between the input vectors and output vectors. If the loss is small, then the middle layer may be used as a lower-dimensional representation of the input.

Figure 12.1: Three-Layer Autoencoder Neural Network

A common choice for the loss function is the sum of squared reconstruction errors,

\[ sse = \| X - \hat{Y} \|_F \]  \hspace{1cm} (12.6) 

where \( X \) is the input data matrix and \( \hat{Y} \) is the reconstruction of \( X \).

12.3.1 Representation

For a three layer autoencoder, the lower dimensional representation vector \( z \) corresponds to the values at the middle hidden layer and is given by the following equation,

\[ z = f_0(A^t x + \beta) \]  \hspace{1cm} (12.7) 

where \( x \) is the input vector, \( A \) is the parameter/weight matrix, \( \beta \) is the bias vector and \( f_0 \) is the vectorized activation function.

Notice that if \( f_0 \) is the identity function, then \( z \) becomes a linear transformation of \( x \), as is the case for PCA.

12.3.2 Denoising Autoencoder (DEA)
Clustering

Clustering is related to classification, except that specific classes are not prescribed. Instead data points (vectors) are placed into clusters based on some similarity or distance metric (e.g., Euclidean or Manhattan distance). It is also related to prediction in the sense that a predictive model may be associated with each cluster. Points in a cluster are according to some metric, closer to each other than to points not in their cluster. Closeness or similarity may be defined in terms of $\ell_p$ distance $\|x - z\|_p$, correlation $\rho(x, z)$, or cosine $\cos(x, z)$. Abstractly, we may represent any of these by distance $d(x, z)$. In ScalaTion, the function `dist` in the package object computes the square of Euclidean distance between two vectors, but may easily be changed (e.g., `(u - v).norm1` for Manhattan distance).

```scala
def dist (u: VectoD, v: VectoD): Double =
{
    (u - v).normSq // squared Euclidean norm used for efficiency, may use other norms
} // dist
```

Consider a general modeling equation, where the parameters $b$ are estimated based on a dataset $(X, y)$.

$$y = f(x; b) + \epsilon$$

Rather than trying to approximate the function $f$ over the whole data domain, one might think that given point $z$, that points similar to (or close to) $z$, might be more useful in making a prediction $f(z)$.

A simple way to do this would be to find the $k$-nearest neighbors to point $z$, $\text{top}_k(z) = \{x_i \in X \mid x_i \text{ is among the } k \text{ closest points to } z\}$

and simply average the responses or $y$-values.

Instead of surveying the responses from the $k$-nearest neighbors, one could instead survey an entire group of similar points and take their averaged response (or utilize a linear model where each cluster $c$ has its own parameters $b_c$). The groups may be pre-computed and the averages/parameters can be maintained for each group. The groups are made by clustering the points in the $X$ matrix into say $k$ groups.

Clustering will partition the $m$-points $\{x_i \in X\}$ into $k$ groups/clusters. Group membership is based on closeness or similarity between the points. Commonly, algorithms form groups by establishing a centroid (or center) for each group/cluster. Centroids may defined as means (or medians) of the points in the group. In this way the data matrix $X$ is partitioned into $k$ submatrices.
\{X_c \mid c \in \{0, \ldots, k-1\}\}

each with centroid \(\xi_c = \mu(X_c)\). Typically, point \(x_i\) is in cluster \(c\) because it is closer to its centroid than any other centroid, i.e.,

\[x_i \in X_c \implies d(x_i, \xi_c) \leq d(x_i, \xi_h)\]

Define the cluster assignment function \(\xi\) to take a point \(x_i\) and assign it to the cluster with the closest centroid \(\xi_c\), i.e.,

\[\xi(x_i) = c\]

The goal becomes to find an optimal cluster assignment function by minimizing the following objective/cost function:

\[
\min_{\xi} \sum_{i=0}^{m} d(x_i, \xi(\xi(x_i)))
\]

If the distance \(d\) is \(\|x_i - \xi(\xi(x_i))\|^2\) (the default in SCALATION), then above sum may be viewed as a form of sum of squared errors (sse).

If one knew the optimal centroids ahead of time, finding an optimal cluster assignment function \(\xi\) would be trivial and would take \(O(kmn)\) time. Unfortunately, \(k\) centroids must be initially chosen, but then as assignments are made, the centroids will move, causing assignments to need re-evaluation. The details vary by clustering algorithm, but it is useful to know that finding an optimal cluster assignment function is \(NP\)-hard [?].

Other factors that can be considering in forming clusters include, balancing the size of clusters and maximizing the distance between clusters.
13.1 KNN_Predictor

Similar to the KNN_Classifier class, the KNN_Predictor class makes predictions based on individual predictions of its $\kappa$-nearest neighbors. For prediction, its function is analogous to using clustering for prediction and will be compared in the exercises in later sections of this chapter.

Training in KNN_Predictor is lazy and is done in the predict method, based on the following equation:

$$ \hat{y} = \frac{1}{\kappa} \sum y(top_{\kappa}(z)) $$  \hspace{1cm} (13.1)

Given point $z$, find $\kappa$ points that are the closest, sum their response values $y$, and return the average.

```
override def predict (z: VectoD): Double =
{
    kNearest (z) // set top-kappa to kappa nearest
    var sum = 0.0
    for (i <- 0 until kappa) sum = y(topK(i)._1) // sum the individual predictions
    val yp = sum / kappa // divide to get average
    reset () // reset topK
    yp // return the predicted value
}
```

The kNearest method is same as the one in KNN_Classifier.

KNN_Predictor Class

Class Methods:

- @param x the vectors/points of predictor data stored as rows of a matrix
- @param y the response value for each vector in x
- @param fname_ the names for all features/variables
- @param hparam the number of nearest neighbors to consider

class KNN_Predictor (x: MatriD, y: VectoD,
                      fname_: Strings = null, hparam: HyperParameter = KNN_Predictor.hp)
                 extends PredictorMat (x, y, fname_, hparam)

def distance (x: VectoD, z: VectoD): Double = (x - z).normSq

def train (yy: VectoD = y): KNN_Predictor = this
override def eval (xx: MatriD = x, yy: VectoD = y): KNN_Predictor =
override def predict (z: VectoD): Double =

def reset ()
def forwardSel (cols: Set [Int], adjusted: Boolean): (Int, VectoD, VectoD) =
def backwardElim (cols: Set [Int], adjusted: Boolean, first: Int): (Int, VectoD, VectoD) =
def crossVal (xx: MatriD = x, k: Int = 10, rando: Boolean = true): Array [Statistic] =

425
Note, the \texttt{forwardSel} and \texttt{backwardElim} methods are not relevant and just throw exceptions if called. Also, the \texttt{train} method has nothing to do, so it need not be called.

### 13.1.1 Exercises

1. Apply \texttt{KNN.Predictor} to the following combined data matrix.

```scala
// x1  x2  y
val xy = new MatrixD ((10, 3), 1, 5, 1,  // joint data matrix
                    2, 4, 1,
                    3, 4, 1,
                    4, 4, 1,
                    5, 3, 0,
                    6, 3, 1,
                    7, 2, 0,
                    8, 2, 0,
                    9, 1, 0,
                   10, 1, 0)

val knn = KNN_Predictor (xy)
val (x, y) = PredictorMat.pullResponse (xy)
val yp = knn.predict (x)
knn.eval (x, y) // due to lazy/late training
println (knn.report)
new Plot (xy.col(0), y, yp, lines = true)
```
13.2 Clusterer

The Clusterer trait provides a common framework for several clustering algorithms.

Clusterer Trait

Trait Methods:

trait Clusterer

  def name_ (nm: Strings) { _name = nm }
  def name (c: Int): String =
  def setStream (s: Int) { stream = s }
  def train (): Clusterer
  def cluster: Array[Int]
  def csize: VectorI
  def centroids: MatrixD
  def initCentroids (): Boolean = false
  def calcCentroids (x: MatrixD, to_c: Array[Int], sz: VectorI, cent: MatrixD)
  def classify (z: VectorD): Int
  def distance (u: VectorD, cn: MatrixD, kc_: Int = -1): VectorD =
  def sse (x: MatrixD, to_c: Array[Int]): Double =
  def sse (x: MatrixD, c: Int, to_c: Array[Int]): Double =
  def sst (x: MatrixD): Double =
  def checkOpt (x: MatrixD, to_c: Array[Int], opt: Double): Boolean = sse (x, to_c) <= opt

For readability, names may be given to clusters (see `name` and `name_`). To obtain a new (and likely different) cluster assignment, `setStream` method may be called to change the random number stream. The `train` methods in the implementing classes will take a set of points (vectors) and apply iterative algorithms to find a “good” cluster assignment function. The `cluster` method may be called after `train` to see the cluster assignments. The centroids are returned as rows in a matrix by calling `centroids`, whose cluster sizes are given by `csize`. The `initCentroids` method initializes the centroids, while the `calcCentroids` calculates the centroids based in the points contained in the each cluster.

def calcCentroids (x: MatrixD, to_c: Array[Int], sz: VectorI, cent: MatrixD)
{
  cent.set (0.0) // set cent matrix to all zeros
  for (i <- x.range1) {
    val c = to_c(i) // x_i currently assigned to cluster c
    cent(c) = cent(c) + x(i) // add the next vector in cluster
  } // for
  for (c <- cent.range1) cent(c) = cent(c) / sz(c) // divide to get averages/means
} // calcCentroids
Given a new point/vector $z$, the \texttt{classify} method will indicate which cluster it belongs to (in the range 0 to k-1). The distances between a point and the centroids is computed by the \texttt{distance} method. The objective/cost function is defined to be the sum of squared errors (\texttt{sse}). If the cost of an optimal solution is known, \texttt{checkOpt} will return true if the cluster assignment is optimal.
13.3 K-Means Clustering

The KMeansClustering class clusters several vectors/points using \( k \)-means clustering. The user selects the number of clusters desired (\( k \)). The algorithm will partition the points in \( X \) into \( k \) clusters. Each cluster has a centroid (mean) and each data point \( x_i \in X \) is placed in the cluster whose centroid it is nearest to.

13.3.1 Initial Assignment

There are two ways to initialize the algorithm: Either (1) randomly assign points to \( k \) clusters or (2) randomly pick \( k \) points as initial centroids. Technique (1) tends to work better and is the primary technique used in SCALA\text{ION}. Using the primary technique, the first step is to randomly assign each point \( x_i \) to a cluster.

\[
\xi(x_i) = \text{random integer from } \{0, \ldots, k-1\}
\]

In SCALA\text{ION}, this is carried out by the assign method, that uses the Randi random integer generator. It also uses multiple counters for determining the size \( sz \) of each cluster (i.e., the number of points in each cluster).

```scala
protected def assign ()
{
  val ran = new Randi (0, k-1, s) // for random integers: 0, ..., k-1
  for (i <- x.range1) {
    to_c(i) = ran.igen // randomly assign x(i) to a cluster
    sz(to_c(i)) += 1 // increment size of that cluster
  } // for
} // assign
```

See the exercises for more details on the second technique for initializing clusters/centroids.

Handling Empty Clusters

If any cluster turns out to be empty, move a point from another cluster. In SCALA\text{ION} this is done by removing a point from the largest cluster and adding it to the empty cluster. This is performed by the fixEmptyClusters method.

```scala
protected def fixEmptyClusters ()
```

After the assign and fixEmptyClusters methods have been called, the data matrix \( X \) will be logically partitioned into \( k \) non-empty submatrices \( X_c \) with cluster \( c \) having \( n_c \) \( (sz(c)) \) points/rows.

Calculating Centroids

The next step is to calculate the centroids using the calcCentroids method. For cluster \( c \), the centroid is the vector mean of the rows in submatrix \( X_c \).

\[
\xi_c = \frac{1}{n_c} \sum_{x_i \in X_c} x_i
\]

SCALA\text{ION} iterates over all points and based on their cluster assignment adds them to one of the \( k \) centroids (stored in the cent matrix). After the loop, these sums are divided by the cluster sizes \( sz \) to get means. The calcCentroids method is defined in the base trait Clusterer.
13.3.2 Reassignment of Points to Closest Clusters

After initialization, the algorithm iteratively reassigns each point to the cluster containing the closest centroid. The algorithm stops when there are no changes to the cluster assignments. For each iteration, each point $x_i$ needs to be re-evaluated and moved (if need be) to the cluster with the closest centroid. Reassignment is based on taking the argmin of all the distances to the centroids with ties going to the current cluster.

$$\xi(x_i) = \text{argmin}_c d(x_i, \xi_c)$$

In ScalaTion, this is done by the `reassign` method which iterates over each $x_i \in X$ computing the distance to each of $k$ centroids. The cluster ($c_2$) with the closest centroid is found using the `argmin` method. The distance to $c_2$'s centroid is then compared to the distance to its current cluster $c_1$'s centroid, and if the distance to $c_2$'s centroid is less, $x_i$ will be moved and a done flag will be set to false, indicating that during this reassignment phase at least one change was made.

```scala
protected def reassign (): Boolean =
{
  var done = true // done indicates no changes
  for (i <- x.range1) { // standard order for index i
    val c1 = to_c(i) // c1 = current cluster for point x_i
    if (sz(c1) > 1) { // if size of c1 > 1
      val d = distance (x(i), cent) // distances to all centroid
      val c2 = d.argmin () // c2 = cluster with closest centroid to x_i
      if (d(c2) < d(c1)) { // if closest closer than current
        sz(c1) -= 1 // decrement size of current cluster
        sz(c2) += 1 // increment size of new cluster
        to_c(i) = c2 // reassign point x_i to cluster c2
        done = false // changed clusters => not done
        if (immediate) return false // optionally return after first change
      } // if
    } // if
  } // for
  done
} // reassign
```

The exercises explore a change to this algorithm by having it return after the first change.

13.3.3 Training

The `train` method simply uses these methods until the `reassign` method returns true (internally the done flag is true). The method is set up to work for this and derived classes. It assigns points to clusters and then either initializes/picks centroids or calculates centroids from the first cluster assignment. Inside the loop, `reassign` and `calcCentroid` are called until there is no change to the cluster assignment. After the loop, an exception is thrown if there are any empty clusters (a useful safe-guard since this method is used by derived classes). Finally, if post-processing is to be performed (`post = true`), then the `swap` method is called. This method will swap two points in different clusters, if the swap results in a lower sum of squared error (sse).
def train (): KMeansClusterer =
{
    sz.set (0) // cluster sizes initialized to zero
    raniv = PermutedVecI (VectorI.range (0, x.dim1), stream) // for randomizing index order
    assign () // randomly assign points to clusters
    fixEmptyClusters () // move points into empty clusters
    if (! initCentroids ()) calcCentroids (x, to_c, sz, cent) // pick points for initial centroids
    breakable { for (l <- 1 to MAX_ITER) {
        if (reassign ()) break // reassign points (no change => break)
        calcCentroids (x, to_c, sz, cent) // re-calculate the centroids
    }} // for
    val ce = sz.indexOf (0) // check for empty clusters
    if (ce != -1) throw new Exception (s"Empty cluster c = $ce")
    if (post) swap () // swap points to improve sse
    this
} // train

KMeansClusterer Class

The KMeansClusterer class and its derived classes take a data matrix, the desired number clusters and an array of flags as input parameters. The array of flags are used to make adjustments to the algorithms. For this class, there are two: flags(0) or post indicates whether to use post-processing, and flags(1) or immediate indicates whether return upon the first change in the reassign method.

Class Methods:

@param x the vectors/points to be clustered stored as rows of a matrix
@param k the number of clusters to make
@param flags the array of flags used to adjust the algorithm

    default: no post processing, no immediate return upon change

class KMeansClusterer (x: MatriD, k: Int, val flags: Array[Boolean] = Array (false, false)) extends Clusterer with Error

def train (): KMeansClusterer =
def cluster: Array[Int] = to_c
def centroids: MatriD = cent
def csize: VectoI = sz
protected def assign ()
protected def fixEmptyClusters ()
protected def reassign (): Boolean =
protected def swap ()
def classify (z: VectoD): Int = distance (z, cent).argmin ()
def show (l: Int) { println (s"($l) to_c = ${to_c.deep} \n($l) cent = $cent") }

431
13.3.4 Exercises

1. Plot the following points.

```scala
// x0  x1
val x = MatrixD((6, 2), 1.0, 2.0,
                 2.0, 1.0,
                 4.0, 5.0,
                 5.0, 4.0,
                 8.0, 9.0,
                 9.0, 8.0)

new Plot(x.col(0), x.col(1), null, "x0 vs. x1")
```

For \(k = 3\), determine the optimal cluster assignment \(\xi\). What is the sum of squared errors \(sse\) for this assignment?

2. Using the data from the previous exercise, apply the K-Means Clustering Algorithm by hand to complete the following cluster assignment function table. Let the number of clusters \(k\) be 3 (clusters 0, 1 and 2). The \(\xi^0\) column is the initial random cluster assignment, while the next two columns represent the cluster assignments for the next two iterations.

Table 13.1: Cluster Assignment Function Table

<table>
<thead>
<tr>
<th>point</th>
<th>((x_0, x_1))</th>
<th>(\xi^0)</th>
<th>(\xi^1)</th>
<th>(\xi^2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(1, 2)</td>
<td>0</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>1</td>
<td>(2, 1)</td>
<td>2</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>2</td>
<td>(4, 5)</td>
<td>0</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>3</td>
<td>(5, 4)</td>
<td>1</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>4</td>
<td>(8, 9)</td>
<td>1</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>5</td>
<td>(9, 8)</td>
<td>2</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>

3. The `test` function in the Clusterer object is used test various configurations of classes extending Clusterer, such as the KMeansClusterer class.

```scala
@def test (x: MatrixD, fls: Array[Boolean], alg: Clusterer, opt: Double = -1.0)
```

Explain the meaning of each of the flags: `post` and `immediate`. Call the `test` function, passing in \(x\) and \(k\) from the last exercise. Also, let the value `opt` be the value determined in the last exercise. The `test` method will give the number of test cases out of \(NTESTS\) that are correct in terms of achieving the minimum \(sse\).
4. The primary versus secondary techniques for initializing the clusters/centroids are provided by the `KMeansClusterer` class and the `KMeansClusterer2` class, respectively. Test the quality of these two techniques.

5. Show that the time complexity of the `reassign` method is $O(kmn)$. The time complexity of K-Means Clustering using Lloyd’s Algorithm is simply the complexity of the `reassign` method times the number of iterations. In practice, the number of iterations tends to be small, but in the worst case only upper and lower bounds are known, see [5] for details.

6. Consider the objective/cost function given in ISL equation 10.11 in [46]. What does it measure and how does it compare to `sse` used in this book?
## 13.4 K-Means Clustering - Hartigan-Wong

An alternative to the Lloyd algorithm that often produces more tightly packed clusters is the Hartigan-Wong algorithm. Improvement is seen in the fraction of times that optimal clusters are formed as well as the reduction in sum of squared errors ($\text{sse}$). The change to the code is minimal in that only the `reassign` method needs to be overridden.

The basic difference is that rather than simply reassigning each point to the cluster with the closest centroid (the Lloyd algorithm), the Hartigan-Wong algorithm weights the distance by the relative changes in the number of points in a cluster. For example, if a point is to be moved into a cluster with 10 points currently, the weight would be $10/11$. If the point is to stay in its present cluster with 10 points currently, the loss in removing it would be weighted as $10/9$. The weighting scheme has two effects: First it makes it more likely to move a point out of its current cluster. Second it makes it more likely to join a small cluster.

Mathematically, the weighted distance $d'$ to cluster $c$ when the point $x_i \notin X_c$ is given by

$$d'(x_i, \xi_c) = \frac{n_c}{n_c + 1} d(x_i, \xi_c)$$

(13.2)

When the point $x_i \in X_c$, the weighted distance $d'$ to cluster $c$ is given by

$$d'(x_i, \xi_c) = \frac{n_c}{n_c - 1} d(x_i, \xi_c)$$

(13.3)

The code for the `reassign` method is similar to the one in `KMeansClusterer`, except that the private method `closestByR2` calculates weighted distances to return the closest centroid.

```scala
protected override def reassign (): Boolean =
{
  var done = true // done indicates no changes
  for (i <- raniv.igen) { // randomize order of index i
    val c1 = to_c(i) // c1 = current cluster for point x_i
    if (sz(c1) > 1) {
      val d = distance2 (x(i), cent, c1) // adjusted distances to all centroid
      val c2 = d.argmin () // c2 = cluster with closest centroid to x_i
      if (d(c2) < d(c1)) { // if closest closer than current
        sz(c1) -= 1 // decrement the size of cluster c1
        sz(c2) += 1 // increment size of cluster c2
        to_c(i) = c2 // reassign point x_i to cluster c2
        done = false // changed clusters => not done
        if (immediate) return false // optionally return after first change
      }
    }
  }
  done
}
```

Besides switching from distance $d$ to weighted distance $d'$, the code also randomizes the index order and has the option of returning immediately after a change is made.
13.4.1 Adjusted Distance

The distance2 method computes the adjusted distance of point \( u \) to all of the centroids \( \text{cent} \), where \( \text{cc} \) is the current centroid that \( u \) is assigned to. Notice the inflation of distance when \( c == \text{cc} \), and its deflation, otherwise.

```scala
def distance2 (u: VectoD, cent: MatriD, cc: Int): VectoD =
{
  val d = new VectorD (cent.dim1)
  for (c <- 0 until k) {
    d(c) = if (c == cc) (sz(c) * dist (u, cent(c))) / (sz(c) - 1)
               else (sz(c) * dist (u, cent(c))) / (sz(c) + 1)
  } // for
  d
} // distance2
```

KMeansClusteringHW Class

Class Methods:

- `@param x` the vectors/points to be clustered stored as rows of a matrix
- `@param k` the number of clusters to make
- `@param flags` the flags used to adjust the algorithm

```scala
class KMeansClustererHW (x: MatriD, k: Int, flags: Array[Boolean] = Array (false, false)) extends KMeansClusterer (x, k, flags)

protected override def reassign (): Boolean =

def distance2 (u: VectoD, cent: MatriD, cc: Int): VectoD =
```

13.4.2 Exercises

1. Compare `KMeansClustererHW` with `KMeansClusterer` for a variety of datasets, starting with the six points given in the last section (Exercise 1). Compare the quality of the solution in terms the fraction of optimal clusterings and the mean of the `sse` over the `NTESTS` test cases.
13.5 K-Means++ Clustering

The KMeansClustererPP class clusters several vectors/points using a $k$-means++ clustering algorithm. The class may be derived from a K-Means clustering algorithm and in SCALA they are derived from the Hartigan-Wong algorithm (KMeansClustererHW). The innovation for KMeansClustererPP is to pick the initial centroids wisely, yet randomly. The wise part is to make sure points are well separated. The random part involves making a probability mass function (pmf) where points farther away from the current centroids are more likely to be selected as the next centroid. Picking the initial centroids entirely randomly leads to KMeansClusterer2 which typically does not perform as well as KMeansClusterer. However, maintaining randomness while giving preference to more distant points becoming the next centroid has been shown to work well.

13.5.1 Picking Initial Centroids

In order to pick $k$ initial centroids, the first one, cent(0), is chosen entirely randomly, using the ranI random variate generator object. The method call ranI.igen will pick one of the $m = x$.dim1 points as the first centroid.

```scala
val ranI = new Randi (0, x.dim1-1, stream) // uniform random integer generator
cent(0) = x(ranI.igen) // pick first centroid uniformly at random
```

The rest of the centroids are chosen following a distance-derived discrete distribution, using the ranD random variate generator object. The probability mass function (pmf) for this discrete distribution is produced so that the probability of a point being selected as the next centroid is proportional to its distance to the closest existing centroid.

```scala
for (c <- 1 until k) { // pick remaining centroids
  val ranD = update_pmf (c) // update distance derived pmf
  cent(c) = x(ranD.igen) // pick next centroid according to pmf
} // for
```

Each time a new centroid is chosen, the pmf must be updated as it is likely to be the closest centroid for some of the remaining as yet unchosen points. Given that the next centroid to selected is the $c^{th}$ centroid, the update_pmf method will update the pmf and return a new distance-derived discrete distribution.

```scala
def update_pmf (c: Int): Discrete =
{
  for (i <- x.range1) pmf(i) = distance (x(i), cent, c).min () // shortest distances
  pmf /= pmf.sum // divide by sum
  Discrete (pmf, stream = (stream + c) % NSTREAMS) // distance-derived generator
} // update_pmf
```

The pmf vector initially records the shortest distance from each point $x_i$ to any of the existing already selected centroids $\{0, \ldots, c-1\}$. These distances are turned into probabilities by dividing by their sum. The pmf vector then defines a new distance-derived random generator that is returned.
KMeansClustererPP Class

Class Methods:

@param x the vectors/points to be clustered stored as rows of a matrix
@param k the number of clusters to make
@param flags the flags used to adjust the algorithm

class KMeansClustererPP (x: Matrix, k: Int, flags: Array[Boolean] = Array(false, false))
    extends KMeansClustererHW (x, k, flags)

    override def initCentroids (): Boolean =
    def update_pmf (c: Int): Discrete =

13.5.2 Exercises

1. Compare KMeansClustererPP with KMeansClustererHW and KMeansClusterer for a variety of datasets, starting with the six points given in the KMeansClusterer section (Exercise 1). Compare the quality of the solution in terms the fraction of optimal clusterings and the mean of the sse over the NTESTS test cases.
13.6 Clustering Predictor

The ClusteringPredictor class is used to predict a response value for new vector \( z \). It works by finding the cluster that the point \( z \) would belong to. The recorded response value for \( y \) is then given as the predicted response. The per cluster recorded response value is the consensus (e.g., average) of the response values \( y_i \) for each member of the cluster. Training involves clustering the points in data matrix \( X \) and then computing each cluster’s response. Assuming the closest centroid to \( z \) is \( \xi_c \), the predicted value \( \hat{y} \) is

\[
\hat{y} = \frac{1}{n_c} \sum_{\xi(x_i)=c} y_i \tag{13.4}
\]

where \( n_c \) is the number points in cluster \( c \) and \( \xi(x_i)=c \) means that the \( i^{th} \) point is assigned to cluster \( c \).

13.6.1 Training

The train method first clusters the points/rows in data matrix \( X \) by calling the train method of a clustering algorithm (e.g., \( clust = KMeansClusterer(...) \)). It then calls the assignResponse method to assign a consensus (average) response value for each cluster.

```scala
def train (yy: VectoD = y): ClusteringPredictor =
{
    clust.train ()
    val clustr = clust.cluster
    assignResponse (clustr)
    this
} // train
```

The computed consensus values are stored in yclus, so that the predict method may simply use the underlying clustering algorithm to classify a point \( z \) to indicate which cluster it belongs to. This is then used to index into the yclus vector.

```scala
override def predict (z: VectoD): Double = yclus (clust.classify (z))
```

ClusteringPredictor Class

Class Methods:

- `x`: the vectors/points of predictor data stored as rows of a matrix
- `y`: the response value for each vector in \( x \)
- `fname`: the names for all features/variables
- `hparam`: the number of nearest neighbors to consider

```scala
class ClusteringPredictor (x: MatriD, y: VectoD,
    fname_: Strings = null, hparam: HyperParameter = ClusteringPredictor.hp)
extends PredictorMat (x, y, fname_, hparam)
```

```scala
def train (yy: VectoD = y): ClusteringPredictor =
```

438
override def eval (xx: MatriD = x, yy: VectoD = y): ClusteringPredictor =
    def classify (z: VectoD): Int =
    override def predict (z: VectoD): Double = yclus (classify (z))
    def reset ()
    def forwardSel (cols: Set [Int], adjusted: Boolean): (Int, VectoD, VectoD) =
    def backwardElim (cols: Set [Int], adjusted: Boolean, first: Int): (Int, VectoD, VectoD) =
    def crossVal (xx: MatriD = x, k: Int = 10, rando: Boolean = true): Array [Statistic] =

13.6.2 Exercises

1. Apply ClusteringPredictor to the following combined data matrix.

    //
    val xy = new MatrixD ((10, 3), 1, 5, 1,  // joint data matrix
                        2, 4, 1,
                        3, 4, 1,
                        4, 4, 1,
                        5, 3, 0,
                        6, 3, 1,
                        7, 2, 0,
                        8, 2, 0,
                        9, 1, 0,
                        10, 1, 0)

    val cp = ClusteringPredictor (xy)
    cp.train ().eval ()
    val (x, y) = PredictorMat.pullResponse (xy)
    val yp = cp.predict (x)
    println (cp.report)
    new Plot (xy.col(0), y, yp, lines = true)

    Compare its results to that of KNN_Predictor.

2. Compare Regression, KNN_Predictor and ClusteringPredictor on the AutoMPG dataset.
13.7 Hierarchical Clustering

One critique of K-Means Clustering is that the user chooses the desired number of clusters ($k$) beforehand. With modern computing power, several values for $k$ may be tried, so this is less of an issue now. There is, however, a clustering technique called Hierarchical Clustering where this is a non-issue.

In SCALaTion the `HierClusterer` class starts with each point in the data matrix $X$ forming its own cluster ($m$ clusters). For each iteration, the algorithm will merge two clusters into a one larger cluster, thereby reducing the number of clusters by one. The two clusters that are closest to each other are chosen as the clusters to merge. The `train` method is shown below.

```scala
def train (): HierClusterer =
{
  sz.set (0) // initialize cluster sizes to zero
  initClusters () // make a cluster for each point

  for (kk <- x.dim1 until k by -1) {
    val (si, sj) = bestMerge (kk) // find the 2 closest clusters
    clust += si | sj // add the union of sets i and j
    clust -= si // remove set i
    clust -= sj // remove set j
    if (DEBUG) println (s"train: for cluster (${kk-1}), clust = $clust")
  } // for

  finalClusters () // make final cluster assignments
  calcCentroids (x, to_c, sz, cent) // calculate centroids for clusters
  this
} // train
```

After reducing the number of clusters to the desired number $k$ (which defaults to 2), final cluster assignments are made and centroids are calculated. Intermediate clustering results are available making it easier for the user to pick the desired number of clusters after the fact. The algorithm can be rerun with this value for $k$.

HierClusterer Class

**Class Methods:**

```scala
@param x the vectors/points to be clustered stored as rows of a matrix
@param k stop when the number of clusters equals k
class HierClusterer (x: MatrixD, k: Int = 2)
  extends Clusterer with Error

def train (): HierClusterer =
def cluster: Array [Int] = to_c
def centroids: MatrixD = cent
def csize: VectorI = sz
def classify (z: VectorD): Int = distance (z, cent).argmin ()
```
13.7.1 Exercises

1. Compare HierClusterer with KMeansClustererHW and KMeansClusterer for a variety of datasets, starting with the six points given in the KMeansClusterer section (Exercise 1). Compare the quality of the solution in terms the fraction of optimal clusterings and the mean of the sse over the NTESTS test cases.

2. K-Means Clustering techniques often tend to produce better clusters (e.g., lower sse) than Hierarchical Clustering techniques. For what types of datasets might Hierarchical Clustering be preferred?

3. What is the relationship between Hierarchical Clustering and Dendrograms?
13.8 Markov Clustering

The MarkovClusterer class implements a Markov Clustering Algorithm (MCL) and is used to cluster nodes in a graph. The graph is represented as an edge-weighted adjacency matrix (a non-zero cell indicates nodes i and j are connected).

The primary constructor takes either a graph (adjacency matrix) or a Markov transition matrix as input. If a graph is passed in, the normalize method must be called to convert it into a Markov transition matrix. Before normalizing, it may be helpful to add self loops to the graph. The matrix (graph or transition) may be either dense or sparse. See the MarkovClusteringTest object at the bottom of the file for examples.

MarkovClusterer Class

Class Methods:

- @param t either an adjacency matrix of a graph or a Markov transition matrix
- @param k the strength of expansion
- @param r the strength of inflation

```scala
class MarkovClusterer (t: MatriD, k: Int = 2, r: Double = 2.0)
extends Clusterer with Error
```

```scala
def train (): MarkovClusterer =
def cluster: Array [Int] = clustr
def centroids: MatriD = throw new UnsupportedOperationException ("not applicable")
def csize: VectoI = throw new UnsupportedOperationException ("not applicable")
def addSelfLoops (weight: Double = 1.0)
def normalize ()
def processMatrix (): MatriD =
def classify (y: VectoD): Int = throw new UnsupportedOperationException ()
```

13.8.1 Exercises

1. Draw the directed graph obtained from the following adjacency matrix, where g(i, j) == 1.0 means that a directed edge exists from node i to node j.

```scala
val g = new MatrixD ((12, 12),
  0.0, 1.0, 0.0, 0.0, 0.0, 1.0, 1.0, 0.0, 0.0, 1.0, 0.0, 0.0,
  1.0, 0.0, 1.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
  0.0, 1.0, 0.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
  0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 0.0, 1.0, 0.0,
  0.0, 1.0, 1.0, 0.0, 0.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0,
  1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0,
  1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0,
  1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0,
  1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0,
  1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0,
  0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0,
  1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0,
  0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0)
```

442
Apply the MCL Algorithm to this graph and explain the significance of the resulting clusters.

```scala
val mg = new MarkovClusterer(g)
mg.addSelfLoops()
mg.normalize()
println("result = " + mg.processMatrix())
mg.train()
println("cluster = " + mg.cluster.deep)
```
Part III

Simulation
Chapter 14

Simulation Foundations

ScalaTion supports multi-paradigm modeling that can be used for simulation, optimization and analytics. The focus of this chapter is simulation modeling. Viewed as a black-box, a simple model maps an input vector $x$ and a scalar time $t$ to an output/response vector $y$,

$$y = f(x, t) + \epsilon$$  \hspace{1cm} (14.1)

where $\epsilon$ is the error/residual vector.

A simulation model typically adds to these the notion of state, represented by a vector-valued function of time $x(t)$. External input (e.g., an external driving force) is now renamed to $u(t)$ and the error vector is replaced with a noise process $w(t)$ (e.g., a Gaussian white noise process). Commonly, simulations model systems that evolve over time.

$$y(t) = f(x(t), u(t), t) + w(t)$$ \hspace{1cm} (14.2)

Figure 14.1: Conceptual Framework for Simulation Models

Knowledge about a system or process is used to define state as well as how state can change over time. Theoretically, this should make such models more accurate, more robust, and have more explanatory power. Ultimately, we may still be interested in how inputs affect outputs, but to increase the realism of the model with the hope of improving its accuracy, much attention must be directed in the modeling effort to state and state transitions. This is true to a degree with most simulation modeling paradigms or world views.

Once a simulation model has been validated, one of its strengths is that it can be used to address what-if questions. What if we add another lane to an interstate highway. Will this lead to reduced traffic congestion
and reduced travel times? Such capabilities allow simulation models to play a larger role in perspective analytics. This can be taken farther with simulation optimization, which can seek improvements to systems.

This chapter focuses on foundations necessary for creating simulation models, as well as some simulation modeling techniques that can be performed without substantial software.

The following textbooks on Discrete-Event Simulation are recommended:

14.1 Basic Concepts

The following basic concepts are common to many types of simulation models.

- **Simulation Model**: A simulation model consists of collection of entities that interact with each other. The model may be view as simplified version of a system (existing or imagined). The model should be useful for description, prediction, and/or prescription. For improved explainability, the it is often desirable that the model mimics the behavior the real system.

- **Entity**: An entity is an identifiable object in a simulation model, e.g., a customer entering a bank, or a vehicle traveling on a road. One may think of an entity having a trajectory in time and space.

- **Attribute**: An attribute is a property of an entity that is relevant for the model, e.g., the speed and weight of the vehicle.

- **State**: The current values for all the variable in the model (or attribute values for all entities). These may be collected into a state vector $\mathbf{x}(t)$ that evolves over time. The *Restorable State* of the model may be thought of a sufficient recording of the execution of the model so far. This would allow a snapshot to be saved and later restored for continued execution. For some models, recent history needs to be maintained along with the current state. For a Markov Models, the state only depends on current values, i.e., the future conditioned on the present is independent of the past. For such models, the two notions of state are identical.

- **Event**: An event is an *instantaneous occurrence* that has the potential to change the state of the system being modeled, e.g., the arrival of customer at a bank. It may also trigger other events to occur in the future.

- **Simulation Clock**: To keep track of the advancement of time, a simulation clock is maintained. For continuous-time simulation, time smoothly advances in small increments. For discrete-time simulation, time advances by one time-unit (e.g., set to 1) for each tick of the clock. For discrete-event simulation, time jumps from the “event time of the current event” to the “event time of the next event” (any intermediate time is skipped).

- **Activity**: Entities in a model undergo activities that start on one event, have a duration described by a random variable, and end on another event, e.g., a customer being served by a bank teller.

- **Indefinite Delay**: An entity must wait for a server or other resource to become available. As this delay depends on other entities, the delay in not definite as it is for activities, e.g., waiting time in a queue.
14.2 Types of Models

While many modeling techniques, such as Regression, focus on predicting expected values,

\[ \hat{y} = E[y|x] \quad (14.3) \]

simulation models generate data. Then techniques discussed in this text, can be used to analyze the data instances produced as output of the simulation model.

14.2.1 Example: Modeling an M/M/1 Queue

A single server queue is characterized by an arrival process and a service distribution. For an M/M/1 Queue the inter-arrival time distribution and service distribution are both Exponential,

\[ i_a \sim \text{Exponential}(\lambda) \quad (14.4) \]
\[ s_j \sim \text{Exponential}(\mu) \quad (14.5) \]

where \( \lambda \) is the arrival rate and \( \mu \) is the service rate (not to be confused the mean).

The queue can be modeled as a Continuous-Time Markov Chain and the expected waiting time in the queue can be determined. Unfortunately, as queuing systems become more complex, analytic solution may not be available.

A more general, although less efficient approach is to simulate customers making their way through the queue. They arrive a certain time \( t_a \), begin service at time \( t_s \) and depart at time \( t_d \). Averages for \( m \) customers may then be used to estimate expected waiting times \( T_q \) and service times \( T_s \).

The construction of such simulation models is straightforward. One approach is based on the observation that the number of customers in a queueing system remained constant, except at special points in time, where an event occurs. Changes of the state of systems (e.g., number of customers) can only occur at events. The simulation therefore consists of programming logic that indicates what happens when an event occurs. This is the first major paradigm for simulation modeling and is referred to as event-scheduling.

For complex simulations, the logic may become fragmented, so an alternative is to track active entities in the systems and give the logic that they follow. For example, one may think of a customer entering a bank as an actor following a script. The programming logic for the simulation model then becomes writing scripts of each type of actor. This is the second major paradigm for simulation modeling and is referred to as process-interaction.

There are additional paradigms for simulation modeling that will be discussed later.
14.3 Random Number Generation

Let us imagine you are one of the actors following the script that you were given and you notice that all actors are doing exactly the same thing, following the same steps, take the same routes through the system and experiencing the same service times. Clearly, such simulations would not mimic reality. There should be uncertainty or variability in behavior. This is accomplished by introducing randomness.

The question becomes how to introduce randomness in a deterministic digital computer. In particular, one wants to generate a sequence of random numbers of the following form.

\[ r_i \sim \text{UID}(0, 1) \] (14.6)

The random numbers should be Uniformly and Independently Distributed. A Random Number Generator (RNG) may be used to produce such numbers.

14.3.1 Example RNG: Random0

Random number generators work by producing a long sequence of integers (Int or Long) that do not repeat. An obvious way to do this is to pick a large modulus \( M \), increment an integer \( x \) and take the mod. Such a sequence of numbers will not repeat until \( M \) are generated. The Random0 class implements this approach.

```scala
@param stream the random number stream index
case class Random0 (stream: Int = 0)
  extends RNG (stream):
  private val M = 2147483647 // modulus for a popular 32-bit generator (2^31 - 1)
  private val NORM = 1.0 / M.toDouble // normalization to (0, 1)
  private var x = stream // set stream value to its seed = stream index here

  // Return the modulus used by this random number generator.
  def getM: Double = M.toDouble

  // Return the next random number as a 'Double' in the interval (0, 1).
  def gen: Double = (x = (x + 1) % M; x * NORM)

  // Return the next stream value as a 'Int' in the set {1, 2, ..., m-1}.
  def igen: Int = (x = (x + 1) % M; x)

end Random0
```
The Random0 class will produce numbers in the interval [0, 1) when the \texttt{gen} method is called. The period of the generator (before it repeats itself) is equal to \( M \). So far, so good. Before using this generator, a battery of tests should be applied to check its suitability.

14.3.2 Testing Random Number Generators

Means Test

The Means Test simply computes several means by averaging sub-sequences of the random number stream. Suppose the test has a sample of \( n \) means from sub-sequences of length \( m \).

\[
\mu_i = \frac{1}{m} \sum_{j=0}^{m-1} r_{im+j} \quad \text{for } i = 0, \ldots, n - 1
\]  

(14.7)

Due to the Central Limit Theory, the means should be Normally distributed and the sample should have the following expected value and variance.

\[
\mathbb{E} [\mu_i] = \frac{1}{m} \mathbb{E} [r_i] = 0.5
\]

(14.8)

\[
\mathbb{V} [\mu_i] = \frac{1}{m^2} \mathbb{V} [r_i] = \frac{1}{12m}
\]

(14.9)

Distribution Test

The Distribution Test determines how well sub-streams of the generated random numbers are distributed over the unit interval. Are they uniformly spread out over the interval or concentrated in certain regions. This can be assessed by a Goodness-of-Fit Test, e.g., the Chi-square Goodness-of-Fit Test or the Kolmogorov-Smirnov Goodness-of-Fit Test (see the exercises).

The Chi-square Goodness-of-Fit Test checks how well a histogram from a subsequence of length \( m \) matches the density function of the Uniform(0, 1) distribution. Each interval, say \( I_j \), in the histogram will have an observed \( o_j \) and expected \( e_j \) number of generated random numbers within interval \( I_j \).

\[
o_j = \nu \{ r_i \in I_j : i \in \{0, \ldots, m - 1\} \}
\]

\[
e_j = m \cdot f(\text{middle}(I_j))
\]

\( f \) is the probability density function (pdf) and in this case it is the pdf for the Uniform (0, 1) distribution. Suppose there are \( n \) intervals, then \( e_j = m/n \), while \( o_j \) is determined by a counter that is incremented whenever a random number \( r_i \) is generated that is within interval \( I_j \). The Chi-square test statistic is then

\[
\chi^2 = \sum_{j=0}^{n-1} \frac{(o_j - e_j)^2}{e_j}
\]

(14.10)

When \( \chi^2 > \chi^2_{\alpha, n-1} \), the sample suggests the distribution is not Uniform, where \( \alpha \) is the significance level (e.g., .95).
Auto-Correlation Test

The are many tests related to checking correlation. As with time-series, the auto-correlation may be examined by looking at a Correlogram that indicates the Auto-Correlation Function (ACF) and Partial Auto-Correlation Function (PACF) for increasing lags. The $k$-lag auto-correlation $\rho_k$ is given by

$$\rho_k = \frac{C[r_j,r_{j+k}]}{\sqrt{V[r_j]}} \text{ for any } j \quad (14.11)$$

Ideally, $\rho_0 = 1$ and the rest are close to zero, indicating lack of correlation. Note, the above formula for $\rho_k$ assumes stationarity (see the chapter on time-series).

The Random0 class fails all three tests, see the exercises.

14.3.3 Example RNG: Random3

Rather than adding one each time before using the mod operator (%), it will work better to multiply the stream value by a constant.

```scala
def gen: Double = { x = A * x % M; x * NORM }
```

Extensive testing has been used to find good values for the constant $A$. One example is $A = 16807$. Random number generators of this form are know as Multiplicative Linear Congruential Generators (MLCG). This value for $A$ is an example of primitive-element modulo $M$, allowing the generator to exhibit a full period, $period = M-1$. In other words, the generator will produce all stream values from 1 to $M-1$, inclusive, before repeating itself. In ScalaTion, this generator is available in the Random3 class.

$$x_i = Ax_{i-1} \% M \quad \text{stream value} \quad (14.12)$$
$$r_i = x_i/M \quad \text{random number} \quad (14.13)$$

where $A = 7^5 = 16807$ and $M = 2^{31} - 1 = 2147483647$.

In general, a Linear Congruential Generators (LCG) take the following form,

$$x_i = (Ax_{i-1} + C) \% M \quad \text{stream value} \quad (14.14)$$
$$r_i = x_i/M \quad \text{random number} \quad (14.15)$$

The next step up (longer period and better properties) is a Multiple Recursive Generator (MRG). These can be combined for further improvement into a Combined Multiple Recursive Generator (CMRG). ScalaTion’s Random class is an example of an CMRG developed by L’Ecuyer and Touzin [?].

14.3.4 Exercises

1. Apply the above three tests to the Random0, Random and Random3 random number generators. See the RNGTester object in the scalation.random package for the test methods: meansTest, distributionTest, and correlationTest.

2. Discuss additional tests that are applied to assess the quality of a random number generator.
3. Explain how Multiple Recursive Generators (MRG) work and what advantages they may have over Linear Congruential Generators (LCG).

4. Explain how Kolmogorov-Smirnov Goodness-of-Fit tests work.

5. Finish the implementation of the `distributionTest_KS` function to implement the Kolmogorov-Smirnov Goodness-of-Fit test.
14.4 Random Variate Generation

With a good quality random number generator as a foundation, Random Variate Generators (RVG) for a variety of probability distributions can be created.

14.4.1 Inverse Transform Method

One of the simplest methods for converting random numbers into random variates following some distribution $F$ is the inverse transform method that uses the iCDF $F^{-1}$ to transform the random number.

**General Uniform Distribution**

Suppose $y \sim F$ and $r \sim \text{Uniform}(0, 1)$. Now let $F$ be a general uniform distribution, Uniform($a, b$).

Conceptually, the Inverse Transform Method generates a random number $r$ and uses it to select the height of a horizontal line in the diagram. Drop a vertical line from where it intersects the Cumulative Distribution Function (CDF) $F$. The position in the horizontal axis is the value for the random variate $y$. For example, if $r = 0.5$, then $y = 3.0$.

Mathematically, the relationship is

$$F(y) = r \quad (14.16)$$

$$y = F^{-1}(r) \quad (14.17)$$

The CDF for the general uniform distribution is $F(y) = \frac{y - a}{b - a}$, so

$$y = a + (b - a)r \quad (14.18)$$
**Exponential Distribution**

The Inverse Transform Method also works well for the *Exponential*(*λ*) distribution where the CDF \( F(y) = 1 - e^{-\lambda y} \). Setting \( F(y) = r \) and solving for the inverse yields,

\[
F(y) = 1 - e^{-\lambda y} = r \\
e^{-\lambda y} = 1 - r \\
e^{-\lambda y} = r \\
-\lambda y = \ln(r) \\
y = - \frac{\ln(r)}{\lambda}
\]

The third step relies on the fact that \( 1 - r \) and \( r \) have the same distribution.

To illustrate its use, for example with \( \lambda = 1 \), if the generated random number \( r = 0.5 \), then generated random variate \( y = 0.693 \).

There are numerous Random Variate Generators in SCALATION that extend the Variate abstract class, including the Exponential class.

```scala
case class Exponential (mu: Double = 1.0, stream: Int = 0)
  extends Variate (stream):
    if mu <= 0.0 then flaw ("constructor", "parameter mu must be positive")

  private val l = 1.0 / mu // lambda, the rate parameter

  val mean = mu
```

456
def pf (z: Double): Double = if z >= 0 then l * exp (-l*z) else 0.0

def gen: Double = -mu * log (r.gen)

def gen1 (z: Double): Double = -z * log (r.gen)

end Exponential

Note, since the mean is the reciprocal of the rate, \( \mu = \frac{1}{\lambda} \), the simple gen method \(-mu * log (r.gen)\) produces exponentially distributed random variates.

### 14.4.2 Convolution Method

Another simple method for generating random random variates is the convolution method that involves the summation of multiple simpler random variates.

For example, a Binomial random variate may be created by adding \( n \) Bernoulli random variates. The Bernoulli(\( p \)) distributions models the flip of coin where the probability of head (success) is \( p \) (and \( q = 1 - p \)). The probability mass function (pmf) is shown below.

\[
p_x (x) = p^x q^{1-x} \quad \text{where} \quad x \in \{0,1\}
\]

The Binomial(\( p, n \)) distribution models the number of heads (successes) in flipping \( n \) coins \( (y = x_1 + \cdots + x_n) \).

\[
p_y (y, n) = \binom{n}{y} p^y q^{n-y} \quad \text{where} \quad y \in \{0,1,\ldots n\}
\]

The gen method in the Binomial class therefore simply adds up the results of \( n \) coin flips where a head is 1 and a tail is 0.

@param p the probability of success
@param n the number of independent trials
@param stream the random number stream

case class Binomial (p: Double = .5, n: Int = 10, stream: Int = 0)
  extends Variate (stream):

  if p < 0.0 || p > 1.0 then flaw ("constructor", "parameter p must be in [0, 1]")
  if n <= 0 then flaw ("constructor", "parameter n must be positive")
  _discrete = true
  private val q = 1.0 - p // probability of failure
  private val p_q = p / q // the ratio p divided by q
  private val coin = Bernoulli (p, stream) // coin with prob of success of p

  val mean = p * n

  def pf (z: Double): Double = { val k = z.toInt; if z == k then pf (k) else 0.0 }

  def pf (k: Int): Double = if k in (0, n) then choose (n, k) * p^k * q^(n-k) else 0.0
override def pmf (k: Int): Array[Double] =
    val d = Array.ofDim[Double] (n+1) // array to hold pmf distribution
    d(0) = q^n
    for k <- 1 to n do d(k) = d(k-1) * p_q * (n-k+1) / k.toDouble
    d
end pmf

def gen: Double = summation (n)(coin.gen)
def gen1 (z: Double): Double = summation (z.toInt)(coin.gen)
end Binomial

Note, the `summation` top-level function is defined in Util.scala.

inline def summation (n: Int)(formula: => Double): Double =
    var sum = 0.0
    for i <- 0 until n do sum += formula
    sum
end summation

It evaluates the given `formula` a total of `n` times and returns the sum.

### 14.4.3 Exercises

1. Consider a distribution where the density linearly increases on the interval $[0, b]$. The pdf for this distribution is the following:

$$f(y) = \frac{2y}{b^2} \quad \text{on} \quad [0, b]$$

Use the Inverse Transform Method (ITM) to generate random variates following this distribution.
(a) Determine the Cumulative Distribution Function (CDF) $F(y)$.
(b) Determine the inverse Cumulative Distribution Function (iCDF) $F^{-1}(r)$.
(c) Write code for the `gen` method.
(d) Create a case class to contain the `gen` method and produce a Histogram that shows how the generated random variates are distributed.

2. Use the convolution method to generate random variates following the Erlang($\lambda, k$) distribution, where $\lambda$ is the rate parameter and $k$ is the number of events. The random variable can be used to measure the time for $k$ events to occur. When $k = 1$, it reduces to the Exponential distribution. Since an Erlang random variable is the sum of $k$ independent exponential random variables, the convolution method may be applied. The pdf for the Erlang distribution is shown below.

$$f(y) = \frac{\lambda^k y^{k-1} e^{-\lambda y}}{(k-1)!} \quad \text{on} \quad [0, \infty)$$
3. Test the Convolution Method for generating Binomial random variates for \( p = .5 \) and \( n = 4 \). Generate 10,000 random variates and show the histogram.


5. Illustrate the application the Acceptance-Rejection method for a particular distribution.

6. Consider a distribution with density on the interval \([0, 2]\). Let the probability density function (pdf) for this distribution be the following:

\[
f_y(y) = \frac{y}{2} \quad \text{on } [0, 2]
\]

Use the Inverse Transform Method (ITM) to generate random variates following this distribution.

(i) Determine the inverse Cumulative Distribution Function (iCDF) \( F^{-1}_y(r) \). Recall \( r \) denotes a random number.

(ii) Write code for the gen method for its Random Variate Generator (RVG).

(iii) Draw the CDF \( F_y(y) \) vs. \( y \) and illustrate how the ITM works in this case.
14.5 Poisson Process

In this section, the relationships between three random variables are examined. Consider a system in which events (e.g., arrivals) occur randomly, but at a constant rate $\lambda$. It is further assumed that the time to the next arrival is independent of the previous arrival.

- **Inter-arrival Time.** $T$ = the time interval between subsequent arrivals/events. As the time duration $\Delta t$ becomes arbitrarily small, the probability of an arrival equals the arrival rate multiplied by the time span.

\[
F_T(\Delta t) = P(T \leq \Delta t) = \lambda \Delta t
\]  

(14.19)

- **Time of $n^{th}$ Arrival.** $S_n$ = the time span for $n$ arrivals/events to occur.

\[
S_n = \sum_{i=1}^{n} T_i
\]

(14.20)

- **Counting Process.** $N(t)$ = the counter of the number events (e.g., arrivals) by time $t$.

\[
N(t) < n \iff S_n > t
\]

(14.21)

\[
N(t) = 0 \iff T > t
\]

(14.22)

Define the complementary CDF (cCDF) as follows:

\[
\bar{F}_T(t) = P(T > t) = 1 - F_T(t)
\]

(14.23)

Due to the independence assumption, the probability of no arrivals by time $t + \Delta t$, is the product of the following two probabilities.

\[
\bar{F}_T(t + \Delta t) = \bar{F}_T(t)\bar{F}_T(\Delta t)
\]

(14.24)

Since $\bar{F}_T(\Delta t) = 1 - \lambda \Delta t$, the following holds.

\[
\bar{F}_T(t + \Delta t) = \bar{F}_T(t)(1 - \lambda \Delta t)
\]

(14.25)

Multiplying out the rhs and rearranging gives,

\[
\bar{F}_T(t + \Delta t) - \bar{F}_T(t) = -\lambda \Delta t \bar{F}_T(t)
\]

(14.26)

Dividing both sides by $\Delta t$ results in

\[
\frac{\bar{F}_T(t + \Delta t) - \bar{F}_T(t)}{\Delta t} = -\lambda \bar{F}_T(t)
\]

(14.27)

A derivative is produced by taking the limit as $\Delta t \to 0$

\[
\frac{d}{dt} \bar{F}_T(t) = -\lambda \bar{F}_T(t)
\]

(14.28)
Both sides may now be integrated.

\[
\int \frac{d\bar{F}_T(t)}{\bar{F}_T(t)} = -\int \lambda dt \tag{14.29}
\]

The integral of a reciprocal introduces a natural logarithm.

\[
\ln \bar{F}_T(t) = -\lambda t \tag{14.30}
\]

Taking the exp function of both sides yields,

\[
\bar{F}_T(t) = e^{-\lambda t} \tag{14.31}
\]

See the exercises for more details. Switching back to the regular CDF, shows that inter-arrival time follows the Exponential distribution.

\[
F_T(t) = 1 - e^{-\lambda t} \tag{14.32}
\]

Consequently, \( S_n \) follows the Erlang distribution (see the exercises from the last section).

The counting process, \( N(t) \), is a Poisson Process with the following pmf:

\[
p_{N(t)}(n) = \frac{(\lambda t)^n}{n!}e^{-\lambda t} \tag{14.33}
\]

The probability of no arrivals by time \( t \), which is

\[
p_{N(t)}(0) = e^{-\lambda t} = \bar{F}_T(t) \tag{14.34}
\]

corresponds to the probability that the inter-arrival time is greater time \( t \).

### 14.5.1 Generating a Poisson Process

In ScalaTion, the `PoissonProcess` class is used to simulate a Poisson Process. For such processes, there is often interest in generating three things: arrival times, the counting process itself, and flow per time interval.

```scala
class PoissonProcess (t: Double, lambda: Double = 1.0, stream: Int = 0)
  extends VariateVec (stream):

  def mean: VectorD = VectorD.fill (1)(lambda * t)  // mean of N(t)
  def pf (z: VectorD): Double = ???
  def igen: VectorI = gen.toInt
  def gen: VectorD =
  def num (tt: Double): Int =
  def flow (t_span: Double): VectorI =
```

461
First is the event/arrival times, e.g., the times that vehicles pass a road sensor. The gen method will produce the arrival times from time zero up to the terminal/end time of the simulation \( t \). These arrival times are returned in a vector. As the inter-arrival time \( t_{ia} \) distribution is Exponential (\( \mu, \text{stream} \)) where \( \mu = 1.0 / \lambda \), it is used to generate each time increment.

```java
def gen: VectorD =
  val atime = ArrayBuffer[Double] ()
  var now = 0.0
  while now <= t do
    now += t_ia.gen
    atime += now
  end while
  t_a = VectorD (atime)
  t_a
end gen
```

Note that arrival time \( t_a(k) \) will be distributed as Erlang(\( \lambda, k \)).

Second is the counting process as a function of time \( N(t) \), e.g., the number of vehicles passing a sensor since the beginning of the simulation. Given that the arrival times have been generated, the num method returns the index value where the next arrival time exceeds the specified time \( tt \).

```java
def num (tt: Double): Int =
  if t_a == null then gen
  for i <- t_a.indices if t_a(i) > tt do return i
  t_a.dim
end num
```

Note that count \( \text{num}(tt) \) follows a PoissonProcess(\( t, \lambda \)).

Third is the incremental counts per an interval of time, e.g., the number of vehicles passing a sensor over a 5 minute interval of time. The flow method counts the number arrivals for each time interval (of length \( t_{span} \)) by taking the difference between the count at the end of interval \( n2 \) and the beginning of the interval \( n1 \).

```java
def flow (t_span: Double): VectorI =
  if t_a == null then gen
  val flw = ArrayBuffer[Int] ()
  var now = 0.0
  var n1  = 0
  while now <= t do
    val n2 = num (now)
    flw += n2 - n1
    now += t_span
    n1 = n2
  end while
  VectorI (flw)
end flow
```

The distribution of the flow is left as an exercise.
14.5.2 Generating a Non-Homogeneous Poisson Process

A simulation using a constant arrival rate to model traffic flow will be of low fidelity. Vehicle arrival rates vary dramatically over a day, with low vehicle counts at night and high spikes during morning and late afternoon rush hours.

To create more realistic, or higher fidelity, simulation models, a Non-Homogeneous Poisson Process (NHPP) may be used. The extension can be accomplished by converting the constant $\lambda$ to a function of time $\lambda(t)$. In SCALATION, the `NH_PoissonProcess` can be used to generate arrivals where the arrival rate is given by the `lambdaf` function.

```scala
class NH_PoissonProcess (t: Double, lambdaf: FunctionS2S, stream: Int = 0)
  extends PoissonProcess (t, 1.0, stream):

  override def mean: VectorD = VectorD.fill (1)(lambdaBar * t) // mean of N(t)
  override def pf (z: VectorD): Double = ???
  override def gen: VectorD =

  val atime = ArrayBuffer[Double] ()
  var now = 0.0
  while now <= t do
    val lamb = lambdaf (now) // current value of the lambda function
    now += t_ia.gen / lamb // adjust by dividing current lambda
    atime += now
  end while
  t_a = VectorD (atime)
  t_a

  override def gen: VectorD =
```

The `gen` must be overridden to adjust the time jump based on the current arrival rate. Fortunately, this can be done dividing an Exponential (1) random variate by the current arrival rate.

```scala
  val atime = ArrayBuffer[Double] ()
  var now = 0.0
  while now <= t do
    val lamb = lambdaf (now) // current value of the lambda function
    now += t_ia.gen / lamb // adjust by dividing current lambda
    atime += now
  end while
  t_a = VectorD (atime)
  t_a

override def gen: VectorD =
```

14.5.3 Exercises

1. Plot the pdf of the Erlang($1,k$) distribution, for $k = 2$ and $k = 3$.

2. Compare the above pdf with a histogram of the time for the second arrival ($k = 2$) and the third arrival ($k = 3$). See the `Histogram` class in the `mathstat` package.

3. Call the `num` method for several time points and plot it ($N(t)$) versus time $t$.

4. For the vehicle arrival process simulation, collect data on the number of arrivals every 5 minutes. Create a histogram to show the distribution.
5. One way to create a time-dependent lambda function \texttt{lambdaf} is to take a data file, say consisting of traffic counts, e.g., \texttt{travelTime.csv} in the \texttt{data} directory. Then create a Polynomial Regression model using \texttt{PolyRegression} from the \texttt{modeling} package. Finally, define a \texttt{lambdaf} that calls the Polynomial Regression model \texttt{predict} method.

6. The following equation states that

\[ \frac{d}{dt} \bar{F}_T(t) = -\lambda \bar{F}_T(t) \]

the rate of decrease in the cCDF is proportional to its value. This is analogous to the phenomena of radioactive decay, where the rate of decay is proportional to the amount of radioactive material. Both are described to the above Ordinary Differential Equation (ODE). The following steps may be followed to solve the ODE.

Step 1: Separation of Variables

\[ \frac{d\bar{F}_T}{\bar{F}_T} = -\lambda dt \]

Step 2: Integrate Both Sides

\[ \int \frac{d\bar{F}_T}{\bar{F}_T} = \int -\lambda dt \]

Step 3: Use the Fact that the Derivative of \( \ln x = \frac{1}{x} \)

\[ \ln \bar{F}_T + C = -\lambda t \]

Step 4: Determine Constant of Integration \( C \) using the Initial Condition (IC): \( \bar{F}_T(0) = 1 \)

\[ \ln 1 + C = 0 \]

Since \( C = 0 \), we have,

\[ \ln \bar{F}_T = -\lambda t \]

Step 5: Take the \texttt{exp} Function on Both Sides

\[ \bar{F}_T = e^{-\lambda t} \]

Verify that the solution is correct by showing that it satisfies both the ODE and the IC.
14.6 Monte Carlo Simulation

Many problems can be addressed by drawing a sample and determining whether it satisfies some criterion. For example, draw five cards and determine whether the hand is a full house (three-of-a-kind and pair/two-of-a-kind). If this process is repeated enough times, an estimate for the probability of a full house may be obtained.

14.6.1 Simulation of Card Games

Many card games such as Poker and Blackjack can be simulated to determine probabilities or expected earning/losses. A deck of 52 playing cards can be simulated using an array of cards. Initially, card $i$ will have the number $i$ associated with it. The card ordinal number (0 to 51) is mapped to the (face-value, suit) using the value method.

- The face-value is 1 (Ace), 2, 3, 4, 5, 6, 7, 8, 9, Jack, Queen, or King. The rank (low to high) usually moves Ace to high end of the list.
- The suit is ordered (low to high) in some games like bridge: Clubs (♣), Diamonds (♦), Hearts (♥), and Spades (♠).

The deck will be shuffled using the shuffle method to randomize the positions of cards in the deck. The draw method pulls the next card from the top of the deck. Calling it five times yields a poker hand. A counter can be incremented in case the hand is a full house. The ratio of this counter to the number of hands dealt becomes and estimate for the probability of a full-house.

class Cards:

```java
private val NUM_CARDS = 52 // number of cards in deck
private val card = Array.range (0, NUM_CARDS) // the cards themselves
private val rn = Randi (0, NUM_CARDS - 1) // random number generator
private var top = 0 // index of top card
private val suit = Array ('C', 'D', 'H', 'S') // Clubs, Diamonds, Hearts, Spades

//::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
/** Draw the top card from the deck and return it. Return -1 if no cards are left in the deck. */
def draw (): Int =
  if top == NUM_CARDS then -1
  else { val c = card(top); top += 1; c }

//::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
/** Shuffle the deck of cards. */
def shuffle (): Unit =
  for i <- card.indices do swap (card, i, rn.igen)
  top = 0
end shuffle
```

465
// Convert the card number c (0 to 51) to the face value (1 to 13) and
// suit (0(C), 1(D), 2(H), 3(S)).
// @param c the card ordinal number
// */
def value (c : Int): (Int, Char) = (c % 13 + 1, suit (c / 13))

/** Convert the card deck to a string. */
override def toString: String = "Cards ( " + stringOf (for c <- card yield value (c)) + " )"

end Cards

Probability of Drawing a Particular Hand

The estimate of the probability of a full house can be checked against probability theory. The total number
of hands consisting of five cards is

\[
\binom{52}{5} = \frac{52 \cdot 51 \cdot 50 \cdot 49 \cdot 48}{1 \cdot 2 \cdot 3 \cdot 4 \cdot 5} = 2,598,960
\]

The number of ways \( n_{\text{ways}} \) for get a full house may be determined as follows: (a) choose 1 of 13 face values
for the three-of-a-kind, (b) choose 1 of 12 remaining face values for the pair, (c) choose 3 of 4 suits for the
three-of-a-kind, and (d) choose 2 of 4 suits for the pair, i.e.,

\[
\binom{13}{1} \binom{12}{1} \binom{4}{3} \binom{4}{2} = [13][12]\left[\frac{4 \cdot 3 \cdot 2}{1 \cdot 2 \cdot 3}\right]\left[\frac{4 \cdot 3}{1 \cdot 2}\right] = 78 \cdot 4 \cdot 6 = 3,744
\]

Letting \( y \) be the random draw of five cards, the probability of a full house is simply the ratio of the two
numbers.

\[
P(y = \text{full house}) = \frac{3,744}{2,598,960} = 0.00144
\]

14.6.2 Integral of a Complex Function

As discussed earlier, there is no closed-form formula for the Cumulative Distribution Function (CDF) for the
Normal distribution. As the integral of the probability density function (pdf), the Numerical Integration can
be used to compute the CDF. Monte Carlo integration offers one approach for doing this, which is practically
effective for multiple integrals in higher dimensions.

To illustrate, consider the following one dimensional function defined on the domain \([0, 1]\).

\[
y = f(x) = \sqrt{1 - x^2}
\]
Integration: Area Under the Curve

The integral is the area under the curve. This is the same as the mean height of the curve times the length/size of the domain (1 in this case). The mean height of a function may be estimated by computing the height at many randomly selected points and taking the average.

\[
\bar{y} = \frac{1}{m} \sum_{i=0}^{m-1} f(x_i) = \frac{1}{m} \sum_{i=0}^{m-1} \sqrt{1-x_i^2}
\]

In ScalaTion, this capability is provided by the `MonteCarloIntegration.integrate` method.

```scala
def integrate (f: Function2[Double, Double, Double], a: Double, b: Double, m: Int, s: Int = 0): Double =
val length = b - a
val x = Uniform (a, b, s)
var sum = 0.0
for it <- 0 until m do sum += f(x.gen)
sum * length / m
end integrate
```

The particular function \( f \) is passed into `integrate` in the code below. As the function traces the unit circle in the first quadrant, multiplying by 4 allows \( \pi \) to be approximated.

```scala
object MonteCarloIntegrationTest extends App:

import MonteCarloIntegration.integrate

def f(x: Double): Double = sqrt (1 - x^2)

for k <- 1 to 9; s <- 0 to 1 do
  val pi = 4 * integrate (f, 0, 1, 10^k, s)
  println (s"for k = $k, s = $s: pi = $pi")
end for

end MonteCarloIntegrationTest
```

467
The case class **Uniform** is a random variate generator from the **random** package. **Uniform** \((a, b, s)\) generates (via the **gen** method) uniformly distributed random numbers in the interval \([a, b]\) using random number stream \(s\).

### 14.6.3 Grain Dropping Experiment

The **RandomVec** class may be used to produce random vectors. As a simpler cousin to the Buffon Needle experiment, the Grain Dropping experiment provides a very simple way to compute \(\pi\). The first parameter for the **RandomVec** class is the dimensionality of the space (so use 2 here). The idea is to generate many grains and drop them falling at \(x\)-\(y\) coordinates inside a square centered at the origin having sides of length 2. The grains at a distance of one or less from the origin will also be inside the unit circle (radius = 1) centered at the origin.

@param stream the random number stream to use

class GrainDrop (stream: Int):

    private val grain = RandomVecD (2, max = 1, min = -1, stream = stream)

    //:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
    /** Return the fraction of grains found inside the unit circle.
     * @param n the number of grains to generate
     */
    def fraction (n: Int): Double =
        var count = 0
        for i <- 0 until n do if grain.gen.normSq <= 1.0 then count += 1
        count / n.toDouble
    end fraction

eend GrainDrop

The fraction method counts the number of generated grains that that are inside the unit circle and divides that count by the total number of grains generated. For a good random number generator, this fraction should correspond to the ratio of the areas for the unit circle versus the bounding square \(\{(x, y) : x \in [-1, 1], y \in [-1, 1]\}\). The area of the square is 4 so the fraction times 4 should provide an estimate for \(\pi\).

@main def grainDropTest (): Unit =

    for stream <- 0 to 5 do
        val bn = new GrainDrop (stream)
        banner (s"Grain Drop for stream = $stream")
        for k <- 1 to 8 do
            val n = 10~^k
            println (s"for n = $n: pi = ${4 * bn.fraction (n)}")
        end for
    end for

eend grainDropTest
14.6.4 Simulation of the Monty Hall Problem

Imagine you are a contestant on the *Let’s Make a Deal* game show and host, Monty Hall, asks you to select door number 0, 1 or 2, behind which are two worthless prizes and one luxury car. Whatever door you pick, he randomly opens one of the other non-car doors and asked if you want to stay with your initial choice or switch to the remaining door. What are the probabilities of winning if you (a) stay with your initial choice, or (b) switch to the other door? Finish the code below to validate your results.

```scala
@main def montyHall (): Unit =

  val stream = 0 // random number stream (0 to 999)
  val rg = Randi (0, 2, stream) // door selection (0, 1 or 2) random generator
  val coin = Bernoulli (stream + 1) // coin flip generator
  val stream = 0 // random number stream, try up to 999
  var winStay = 0 // count wins with stay strategy
  var winSwitch = 0 // count wins with switch strategy

  for it <- 1 to 100000 do // test the strategies 100,000 times
    // car randomly placed behind this door
    // contestant randomly picks a door
    // Monty Hall shows other non-car door (if choice, make randomly)
    if pick == car then winStay += 1 // stay with initial pick
    else winSwitch += 1 // switch to the other door
  end for

  println (s"winStay = $winStay")
  println (s"winSwitch = $winSwitch")

end MontyHall
```

Note, since the opened door never has the car behind it, the car must be behind either the originally picked door (stay) or the remaining door (switch). Hence, the form of the above *if then else* statement.

14.6.5 Exercises

1. The hands in Five-Card Draw Poker are the following: (1) high card, (2) pair, (3) two pair, (4) three-of-a-kind, (5) straight, (6) flush, (7) full house, (8) four-of-a-kind, (9) straight flush, and (10) royal flush.

   Use probability theory to determine the probabilities of each Poker hand. Do the same thing using Monte Carlo simulation and compare the results. Let the number of repetitions (hands drawn) increase until the estimates stabilize. Hint: use a large number of samples.

2. Use Monte Carlo simulation to integrate the CDF for the Standard Normal Distribution at 1, \( F_y(1) \).

   Note, the distribution is symmetric around zero, so the following integral may be computed.

   \[
   \text{area} = \int_0^1 \frac{1}{\sqrt{2\pi}} e^{-y^2/2} \]

   469
The solution for $F_y(1)$ will be $\frac{1}{2} + \text{area}$. Check your answers by calling `CDF.normalCDF(1)` in the `random` package.

3. Finish the coding of the Simulation of the Monte Hall Problem. Determine the winning percentages for the Stay and Switch strategies as the number of game simulations increases. Do the they converge? Explain why one strategy is better than the other.

4. Estimate the probability distribution for rolling three 6-sided dice and taking their sum

$$y = x_1 + x_2 + x_3$$

where $x_i \sim \text{Randi}(1, 6)$, i.e., $p_{x_i}(k) = 1/6$ for $k = 1, 2, 3, 4, 5, 6$. The probability mass function (pmf) for $y$ has a range from 3 to 18. Use Monte Carlo Simulation to estimate $p_y(k) = ?$ for $k = 3, 4, \ldots, 17, 18$

Draw/plot the pmf $p_y(k)$ vs. $k$.

5. When rolling $n_d$ six-sided dice, how many ways can four be rolled, for $n_d = 1, 2, 3, 4$ dice.

<table>
<thead>
<tr>
<th>$n_d$</th>
<th>list of ways</th>
<th>$n_{ways}$</th>
<th>configurations</th>
<th>ratio</th>
<th>probability $p_y(k)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[4]</td>
<td>1</td>
<td>6</td>
<td>1/6</td>
<td>.1667</td>
</tr>
<tr>
<td>2</td>
<td>[1,3], [2,2], [3,1]</td>
<td>3</td>
<td>36</td>
<td>1/12</td>
<td>.08333</td>
</tr>
<tr>
<td>3</td>
<td>[1,1,2], [1,2,1], [2,1,1]</td>
<td>3</td>
<td>216</td>
<td>1/72</td>
<td>.01389</td>
</tr>
<tr>
<td>4</td>
<td>[1,1,1,1]</td>
<td>1</td>
<td>1296</td>
<td>1/1296</td>
<td>.007880</td>
</tr>
</tbody>
</table>

Use Monte Carlo simulation to estimate the number of $n_{ways}$ and the probability mass function $p_y(k)$ for $n_d = 1, 2, 3, 4$ dice.

6. The number of ways $n_{ways}$ can also be solved using the following recursive function for the sum $s$ from $n_d$ to $6n_d$. 470
\[ n_{\text{ways}}(n_d, s) = \sum_{k=1}^{6} n_{\text{ways}}(n_d - 1, s - k) \]

The base case for the recursion is when \( n_d = 1 \), in which case \( n_{\text{ways}}(1, s) = 1 \) for \( s = 1, \ldots, 6 \).

Write a program to calculate \( n_{\text{ways}} \) for \( n_d = 1, 2, 3, 4 \) dice. Note, the recursive function may be computed more efficiently using dynamic programming.
14.7 Hand Simulation

Before using or developing software to perform simulation, it is instructive to carry out a simple simulation by hand.

This can be done as follows: Generate random variates for inter-arrival and service times for \( m = 10 \) customers. Fill in these two columns in Table 14.2. Use \( \text{Exponential}(1/\lambda) \) for inter-arrival times and \( \text{Exponential}(1/\mu) \) for service times. Let \( \lambda = 10 \) and \( \mu = 12 \) per hour, giving means of 6 and 5 minutes, respectively. Also, fill in the zeroth row, for a non-existent customer, with all zeros.

**Table 14.2: Hand Simulation of M/M/1 Queue**

<table>
<thead>
<tr>
<th>customer</th>
<th>iarrival</th>
<th>arrival</th>
<th>begin-service</th>
<th>waiting</th>
<th>service</th>
<th>departure</th>
<th>system</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
<td>.</td>
<td>.</td>
<td>5</td>
<td>.</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>.</td>
<td>.</td>
<td>6</td>
<td>.</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>.</td>
<td>.</td>
<td>4</td>
<td>.</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>.</td>
<td>.</td>
<td>3</td>
<td>.</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>.</td>
<td>.</td>
<td>5</td>
<td>.</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>8</td>
<td>.</td>
<td>.</td>
<td>7</td>
<td>.</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>.</td>
<td>.</td>
<td>2</td>
<td>.</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>.</td>
<td>.</td>
<td>4</td>
<td>.</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>.</td>
<td>.</td>
<td>5</td>
<td>.</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>6</td>
<td>.</td>
<td>.</td>
<td>8</td>
<td>.</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td>total</td>
<td>56</td>
<td>.</td>
<td>.</td>
<td>49</td>
<td>.</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td>mean</td>
<td>5.6</td>
<td>.</td>
<td>.</td>
<td>4.9</td>
<td>.</td>
<td>.</td>
<td></td>
</tr>
</tbody>
</table>

Notice that the sample mean inter-arrival time and sample mean service time shown in the last row should correspond the theoretical means of 6 and 5 (keeping the mind the inaccuracy of small samples).

The hand simulation part now begins. Filling in the table row-by-row requires the event logic to be followed. A customer cannot begin service until the previous customer has departed. They must wait in the queue until the server is available. The time between arrival and beginning of service is the wait time. The inter-arrival time indicates the time gap for the next arriving customer.

Given the inter-arrival (iarrival) times \( (\tau_i) \) and service times \( (s_i) \), the equations below may be applied to provide values for the empty columns.

\[
\begin{align*}
    a_i &= a_{i-1} + \tau_i & \text{arrival time} \\
    b_i &= \max(a_i, d_{i-1}) & \text{begin service} \\
    w_i &= b_i - a_i & \text{waiting time} \\
    d_i &= b_i + s_i & \text{departure time} \\
    t_i &= d_i - a_i & \text{time in system}
\end{align*}
\]

Note, the columns in **bold** correspond to events. Before looking at the completed table, try to fill in the previous one.
Table 14.3: Completed Hand Simulation of M/M/1 Queue

<table>
<thead>
<tr>
<th>customer</th>
<th>iarrival t</th>
<th>arrival t</th>
<th>begin-service t</th>
<th>waiting t</th>
<th>service t</th>
<th>departure t</th>
<th>system t</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>0</td>
<td>5</td>
<td>11</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>9</td>
<td>11</td>
<td>2</td>
<td>6</td>
<td>17</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>14</td>
<td>17</td>
<td>3</td>
<td>4</td>
<td>21</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>18</td>
<td>21</td>
<td>3</td>
<td>3</td>
<td>24</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>21</td>
<td>24</td>
<td>3</td>
<td>5</td>
<td>29</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>8</td>
<td>29</td>
<td>29</td>
<td>0</td>
<td>7</td>
<td>36</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>34</td>
<td>36</td>
<td>2</td>
<td>2</td>
<td>38</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>41</td>
<td>41</td>
<td>0</td>
<td>4</td>
<td>45</td>
<td>4</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>50</td>
<td>50</td>
<td>0</td>
<td>5</td>
<td>55</td>
<td>5</td>
</tr>
<tr>
<td>10</td>
<td>6</td>
<td>56</td>
<td>56</td>
<td>0</td>
<td>8</td>
<td>64</td>
<td>8</td>
</tr>
<tr>
<td>total</td>
<td>56</td>
<td>278</td>
<td>291</td>
<td>13</td>
<td>49</td>
<td>340</td>
<td>62</td>
</tr>
<tr>
<td>mean</td>
<td>5.6</td>
<td>27.8</td>
<td>29.1</td>
<td>1.3</td>
<td>4.9</td>
<td>34.0</td>
<td>6.2</td>
</tr>
</tbody>
</table>

Define $\Sigma_q$, $\Sigma_s$ and $\Sigma_y$ to be the sums of the waiting, service and system times, respectively. As shown in the table, the average times in the Queue, Service and System are given by the formulas below.

$$T_q = \frac{\Sigma_q}{m} = 1.3$$  \hspace{1cm} Number in Queue  \hspace{1cm} (14.40)

$$T_s = \frac{\Sigma_s}{m} = 4.9$$  \hspace{1cm} Number in Service  \hspace{1cm} (14.41)

$$T_y = \frac{\Sigma_y}{m} = 6.2$$  \hspace{1cm} Number in System  \hspace{1cm} (14.42)

### 14.7.1 Little’s Law

The start of simulation is 0.0, while the end of simulation is 64.0 (for a duration $\tau$ of 64.0 time units/minutes). During this time $m = 10$ customers arrived, giving an effective arrival rate of

$$\lambda_e = \frac{m}{\tau} = \frac{10}{64.0} = 0.1563 \text{ per minute} = 9.375 \text{ per hour}$$  \hspace{1cm} (14.43)

Little’s Law relates time averages to occupancy averages. As waiting queues get longer, one would expect the waiting time to increase. Using Little’s Law (see the section on Markov Chains for more details), the occupancy (number in the system) $L_y$ is proportional to the time in the system $T_y$ (and the same is true for sub-components).

$$L_y = \lambda_e T_y$$  \hspace{1cm} (14.44)

The proportionality constant is $\lambda_e$. Little’s Law allows the following summary results for our M/M/1 Queue simulation to be collected into Table 14.4.
### Table 14.4: Formulas for M/M/1 Queueing Models

<table>
<thead>
<tr>
<th>part</th>
<th>length</th>
<th>result</th>
<th>time</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Queue</td>
<td>( L_q = )</td>
<td>0.203</td>
<td>( T_q = 1.3 ) minutes</td>
<td></td>
</tr>
<tr>
<td>Service</td>
<td>( L_s = )</td>
<td>0.766</td>
<td>( T_s = 4.9 ) minutes</td>
<td></td>
</tr>
<tr>
<td>System</td>
<td>( L_y = )</td>
<td>0.969</td>
<td>( T_y = 6.2 ) minutes</td>
<td></td>
</tr>
</tbody>
</table>

#### 14.7.2 Event Times

Rather than computing the number in the Queue (q), Service (s) or System (y) using Little’s Law, they may be computed directly as the areas under the curves of \( L_q(t) \), \( L_s(t) \) and \( L_y(t) \), where for example,

\[
L_y = \frac{1}{\tau} \int_0^\tau L_y(t) \, dt \quad (14.45)
\]

Notice that the function \( L_y(t) \) can only change at event times (the state of the system can only change at these times). The event times are the following:

\[ \text{VectorD}(0.0, 6.0, 9.0, 11.0, 14.0, 17.0, 18.0, 21.0, 24.0, 29.0, 34.0, 36.0, 38.0, 41.0, 45.0, 50.0, 55.0, 56.0, 64.0) \]

Event time 0.0 has the start simulation event, times 6.0 and 9.0 have arrival events, time 11.0 has a departure event, times 21.0 and 29.0 have both arrival and departure events, time 64.0 has the last departure event. As there is 1 start simulation event, 8 arrival events, 8 departure events and 2 dual events, there should be a total of 19 event times.

For the above simulation, \( L_y(t) \) takes on the values 0, 1 or 2. When \( L_y(t) = 0 \) the server is idle, \( L_y(t) = 1 \) there is one customer in service and none waiting, and \( L_y(t) = 2 \) there is one customer in service and one waiting.

Start with \( L_y(0) = 0 \), then ":" means no change, + means add 1, - means subtract 1. In this way the value of \( L_y(t) \) can be determined for all event times.

\[ \begin{align*}
0: & \quad 6+, \ 9+, \ 11-, \ 14+, \ 17-, \ 18+, \ 21+: \ 24-, \ 29:, \\
34+, \ 36-, \ 38-, \ 41+, \ 45-, \ 50+, \ 55-, \ 56+, \ 64- \\
\end{align*} \]

Check that the number of arrivals (+) equals the number of departures (-). Tracing through the list, one can deduce the occupancy for the time intervals between the events.

- \( L_y(t) = 0 \): [0, 6], [38, 41], [45, 50], [55, 56]
- \( L_y(t) = 1 \): [6, 9], [11, 14], [17, 18], [24, 34], [36, 38], [41, 45], [50, 55], [56, 64]
- \( L_y(t) = 2 \): [9, 11], [14, 17], [18, 24], [34, 36],

Plot \( L_y(t) \) vs. \( t \) from 0.0 to 64.0 and use it to determine the area under the curve. The subtotals for each are 15, 36, 13 the integral sums to \( 0 \cdot 15 + 36 \cdot 1 + 13 \cdot 2 = 62 \). Similar calculations yield results for \( L_q \) and \( L_s \).

\[
\begin{align*}
L_q &= \frac{\sum q}{\tau} = \frac{13}{64} = 0.203 \\
L_s &= \frac{\sum s}{\tau} = \frac{49}{64} = 0.766 \\
L_y &= \frac{\sum y}{\tau} = \frac{62}{64} = 0.969 \\
\end{align*}
\]
Note, for 15 minutes there were no customers in the system, so the server *idle time* is 15 minutes, while the server *busy time* is 64 - 15 = 49 minutes.

### 14.7.3 Spreadsheet Simulation

To allow a longer simulation with more customers, essentially the same approach can be carried out using spreadsheet software (*spreadsheet simulation*). To reproduce the above hand simulation, copy the first table into a spreadsheet. This will have the `arrival-t` and `service-t` columns filled in. The top row (row 0) will contain all zeros. Use spreadsheet formulas for filling in all the rest of columns for row 1. Now use copy-paste to fill in the rest of the *m* rows. Compute all of the column sums. Below the column sums row, compute all the sample averages (divide by *m*). Make a row below the sample averages row for time averages (divide by τ). Your spreadsheet should look like the hand simulation table, with one more row for the time averages.

Spreadsheet software provides Random Number Generators (RNGs) such as RAND in Excel. Several Random Variate Generators (RVGs) can easily be produced using the Inverse Transform Method (ITM). Try creating Exponential random variates using ITM and then use these to fill in the pre-filled columns (`arrival-t` and `service-t`). These column can now be filled in one row at a time. Letting the arrival rate λ = 10 per hour and the service rate µ = 12 per hour, run the spreadsheet simulation for *m* = 10, 20 and 100 customers/entities.

A slightly more automated approach is provided by the SCALA'Tion’s *tableau* package. To handle more complex event logic, the event-scheduling paradigm should be followed. Process-interaction and agent-based simulation are higher-level, more resource intensive alternative ways to develop simulation models.

### 14.7.4 Exercises

1. Use a spreadsheet to plot $L_q(t)$, $L_s(t)$ and $L_y(t)$ versus time from 0.0 to τ for the M/M/1 Queue. Directly compute the area under the curve and then the average height.

2. Do the same plot for the M/M/1 Queue using PlotM from SCALA'Tion’s mathstat package.

3. Suppose there are now two severs and the arrival rate λ = 20 per hour. Redo all the calculations for this M/M/2 Queue.

4. Redo the plots for the M/M/2 Queue simulation.

5. Write code that takes the arrival and departure columns and produces all the event times in time order. Hint: set up an i-cursors for the arrival list and j-cursor for the departure list, if the values at i and j are the same, copy that value into the event list and advance both cursors, otherwise copy the smaller value and advance its cursor. Do this in a *while* loop.

6. Recall that $\lambda_e = m/\tau$. For the definitions given in this section for $L_q, L_s, L_y$, and $T_q, T_s, T_y$, show the following forms of Little’s Law hold.

\[
L_q = \lambda_e T_q \quad (14.46) \\
L_s = \lambda_e T_s \quad (14.47) \\
L_y = \lambda_e T_y \quad (14.48)
\]
7. Spreadsheet Simulation: A Small Fast Food Restaurant has two servers and enough space for three customers to wait (at most five customers total at any given time). For the case of a single queue, perform a spreadsheet simulation for \( m = 20 \) customer arrivals. Assume each server can process \( \mu = 30 \) customers per hour and that the customer arrival rate \( \lambda = 75 \) customers per hour (assume Exponential distributions). Each completed order gives a net profit (before paying the servers) of 2.00 dollars. Each server makes 11.00 dollars per hour. Should the restaurant hire a third server? Explain in terms of profit (after paying the servers) per hour. Give the simulation table and summary results produced by the spreadsheet.
14.8 Tableau-Oriented Simulation

In tableau-oriented simulation models, each simulation entity’s event times are recorded in a row of a matrix/tableau. For example in a Bank simulation, each row would store information about a particular customer, e.g., when they arrived, how long they waited, their service time duration, etc. If 20 customers are simulated, the matrix will have 20 rows (actually 24 since there are always 4 special rows). Average waiting and service times can be easily calculated by summing columns and dividing by the number of customers. This approach is similar to, but not as flexible as Spreadsheet simulation. The complete code for this example may be found in Ex_Bank.scala in the scalation.simulation.tableau package.

```scala
@main def runEx_Bank (): Unit =

  val stream   = 0 // random number stream (0 to 999)
  val lambda   = 6.0 // customer arrival rate (per hour)
  val mu       = 7.5 // customer service rate (per hour)
  val maxCusts = 20 // stopping rule: simulate maxCusts

  val iArrivalRV = Exponential (HOUR / lambda, stream)
  val serviceRV = Exponential (HOUR / mu, (stream + 1) % N_STREAMS)

  // Run the simulation of the model Bank.
  val mm1 = new Model ("Bank", maxCusts, Array (iArrivalRV, serviceRV))
  mm1.simulate ()
  mm1.report ()
```


Note that it is important that the various random variate generators use “different random number streams” to keep them independent. The stream number (0 to 999) specifies the combinations of seeds to use for the random number generator. In this model, iArrivalRV uses stream, while serviceRV uses stream + 1.

14.8.1 Iterating through Tableau Equations

The logic of the hand simulation is coded into equations for computing the value of each column. The tableau/matrix called tab is built up row-by-row.

- The zeroth row is a placeholder for the previous non-existent entity (the values are all zero).
- Rows 1 to m are for the entity timings, i.e., row i records times for the ith entity.
- The last three rows hold the column sums, sample averages and time averages, respectively.

The simulate method is used to evaluate the equations, row-by-row. A basic set of equations is provided in the Model class that works for a collection of simple, related models. Other models will require the simulate method to be overridden.
for i <- 1 to m do tab(i, 0) = i // ID-0

def simulate (startTime: Double = 0.0): Unit =
    for i <- 1 to m do
        tab(i, 1) = rv(0).gen // IArrival-1
        tab(i, 2) = tab(i-1, 2) + tab(i, 1) // Arrival-2
        tab(i, 3) = tab(i, 2) max tab(i-1, 6) // Begin-3
        tab(i, 4) = tab(i, 3) – tab(i, 2) // Wait-4
        tab(i, 5) = rv(1).gen // Service-5
        tab(i, 6) = tab(i, 3) + tab(i, 5) // Departure-6
        tab(i, 7) = tab(i, 6) – tab(i, 2) // Total-7
    end for
end simulate

The columns are the same as those given in the Hand Simulation section.

14.8.2 Reproducing the Hand Simulation

The example from the hand simulation example is replicated by the following tableau model.

@main def runQueue_MM1 (): Unit =

    val iArrivalArr = Array[Double] (6, 3, 5, 4, 3, 8, 5, 7, 9, 6)
    val serviceArr = Array[Double] (5, 6, 4, 3, 5, 7, 2, 4, 5, 8)

    val maxCusts = 10 // stopping rule: at maxCusts
    val iArrivalRV = Known (iArrivalArr) // inter-arrival time random variate
    val serviceRV = Known (serviceArr) // service time random variate

    // Run the simulation of the model Queue_MM1.

    val mm1 = new Model("Queue_MM1", maxCusts, Array(iArrivalRV, serviceRV))
    mm1.simulate ()
    mm1.report () // show the table/matrix
    mm1.summary () // show summary performance statistics

end runQueue_MM1

The Known Random Variate Generator (RVG) simply repeats the given sequence of numbers.

14.8.3 Customized Logic/Equations

For models where the event logic is different, as mentioned, the simulate method will need to overridden. For example, in an Ex_CallCenter model with a single server (tele-service representative) and no call-waiting, the logic now requires an if statement to check if the phone line is busy. It requires the departure time (call hang up) of the last completed call (call 1) to be less than or equal to the arrival time of the current call (call i).
override def simulate (startTime: Double): Unit = 
    var l = 0 // last established call
    for i <- 1 to m do
        tab(i, 1) = rv(0).gen // IArrival-1
        tab(i, 2) = tab(i-1, 2) + tab(i, 1) // Arrival-2
        if tab(i, 6) <= tab(i, 2) then // call established
            tab(i, 3) = tab(i, 2); l = i // Begin-3
            tab(i, 4) = tab(i, 3) - tab(i, 2) // Wait-4
            tab(i, 5) = rv(1).gen // Service-5
            tab(i, 6) = tab(i, 3) + tab(i, 5) // Departure-6
            tab(i, 7) = tab(i, 6) - tab(i, 2) // Total-7
        end if
    end for
end simulate

Model developers may wish to copy the base simulate method code from Model and make minimal modifications to the equations. The only changes above are the introduce of the variable l, the addition of the if statement, and modification to the Begin-3 equation.

14.8.4 Tableau.scala

The Model class support tableau-oriented simulation models in which each simulation entity’s events are recorded in tabular form (in a matrix). This is analogous to Spreadsheet Simulation (http://www.informs-sim.org/wsc06papers/002.pdf).

Class Methods:

@param name the name of simulation model
@param m the number entities to process before stopping
@param rv the random variate generators to use
@param label_ the column labels for the matrix

class Model (name: String, m: Int, rv: Array [Variate], label_ : Array [String])
    extends Modelable:

    def simulate (startTime: Double = 0.0): Unit =
    def report (): Unit =
    def summary (): Unit =
    def timeLine (): (VectorD, VectorD) =
    def save (): Unit =

The report method displays the tab matrix and should be called after simulate. The last two rows display useful averages such as average inter-arrival, waiting, service and system times. The summary method displays averages for lengths of queues and waiting times, etc. It summarizes the number and time in the queue (q), service (s) and system (y), i.e., L_q, L_s, L_y, T_q, T_s, T_y. The Model.occupancy
The `mml.timeline()` line gives the event times and corresponding values for \( L_q(t) \). The `save` method saves the matrix in a `.csv` file that may be loaded into a spreadsheet for further processing.

The `report` method for the hand simulation problem outputs the following table where the last three rows show the sums, sample averages and time averages.

<table>
<thead>
<tr>
<th>ID</th>
<th>IArrival-1</th>
<th>Arrival-2</th>
<th>Begin-3</th>
<th>Wait-4</th>
<th>Service-5</th>
<th>Departure-6</th>
<th>Total-7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>1.000</td>
<td>6.000</td>
<td>6.000</td>
<td>6.000</td>
<td>0.000</td>
<td>5.000</td>
<td>11.000</td>
<td>5.000</td>
</tr>
<tr>
<td>2.000</td>
<td>3.000</td>
<td>9.000</td>
<td>11.000</td>
<td>2.000</td>
<td>6.000</td>
<td>17.000</td>
<td>8.000</td>
</tr>
<tr>
<td>3.000</td>
<td>5.000</td>
<td>14.000</td>
<td>17.000</td>
<td>3.000</td>
<td>4.000</td>
<td>21.000</td>
<td>7.000</td>
</tr>
<tr>
<td>4.000</td>
<td>4.000</td>
<td>18.000</td>
<td>21.000</td>
<td>3.000</td>
<td>3.000</td>
<td>24.000</td>
<td>6.000</td>
</tr>
<tr>
<td>5.000</td>
<td>3.000</td>
<td>21.000</td>
<td>24.000</td>
<td>3.000</td>
<td>5.000</td>
<td>29.000</td>
<td>8.000</td>
</tr>
<tr>
<td>6.000</td>
<td>8.000</td>
<td>29.000</td>
<td>29.000</td>
<td>0.000</td>
<td>7.000</td>
<td>36.000</td>
<td>7.000</td>
</tr>
<tr>
<td>7.000</td>
<td>5.000</td>
<td>34.000</td>
<td>36.000</td>
<td>2.000</td>
<td>2.000</td>
<td>38.000</td>
<td>4.000</td>
</tr>
<tr>
<td>8.000</td>
<td>7.000</td>
<td>41.000</td>
<td>41.000</td>
<td>0.000</td>
<td>4.000</td>
<td>45.000</td>
<td>4.000</td>
</tr>
<tr>
<td>9.000</td>
<td>9.000</td>
<td>50.000</td>
<td>50.000</td>
<td>0.000</td>
<td>5.000</td>
<td>55.000</td>
<td>5.000</td>
</tr>
<tr>
<td>10.000</td>
<td>6.000</td>
<td>56.000</td>
<td>56.000</td>
<td>0.000</td>
<td>8.000</td>
<td>64.000</td>
<td>8.000</td>
</tr>
<tr>
<td>55.000</td>
<td>56.000</td>
<td>278.000</td>
<td>291.000</td>
<td>13.000</td>
<td>49.000</td>
<td>340.000</td>
<td>62.000</td>
</tr>
<tr>
<td>5.500</td>
<td>5.600</td>
<td>27.800</td>
<td>29.100</td>
<td>1.300</td>
<td>4.900</td>
<td>34.000</td>
<td>6.200</td>
</tr>
<tr>
<td>0.859</td>
<td>0.875</td>
<td>4.344</td>
<td>4.547</td>
<td>0.203</td>
<td>0.766</td>
<td>5.313</td>
<td>0.969</td>
</tr>
</tbody>
</table>

**14.8.5 Exercises**

1. Run the bank simulation for many more customers and see if the averages begin to stabilize.

2. Suppose there are now two servers and the arrival rate \( \lambda = 20 \) per hour. Override the `simulate` method for this M/M/2 Queue. The logic will need to handle the fact that there are now two statistically identical servers. What do the new results indicate?

3. Now suppose the second server works 20% faster than the first server. What do the new results indicate?

4. Rework the previous exercise using a spreadsheet.

5. An M/M/1/1 Queue has one server and a system capacity of one (no space for waiting). Develop and run a Tableau simulation for \( \lambda = 10 \) per hour and \( \mu = 12 \) per hour. Redo for an M/M/2/2 Queue and \( \lambda = 20 \) per hour.
Chapter 15

State Space Models

In dynamic models, the state of a system may be described by a state vector \( x(t) \). For example, a particle may be tracked over time. In two-dimensional space, one might be interested in the height and down range distance of the particle over time,

\[
x(t) = [x_0(t), x_1(t)]
\]  

(15.1)

where \( x_0(t) \) is the height of the particle and \( x_1(t) \) is the horizontal distance travelled at time \( t \).

The dynamics of the particle may be described by a \( n \)-dimensional vector-valued function of time.

\[
f : \mathbb{R}^+ \to \mathbb{R}^n
\]

(15.2)

Such a function may be developed using physical laws that are expressed as a system of Ordinary Differential Equations (ODEs) consisting of first-order time derivatives. When the system is linear, the time derivative \( \dot{x}(t) \) equals an affine transformation of the current state \( x(t) \), i.e., the sum of a linear transformation of the current state and a constant vector.

\[
\dot{x}(t) = Fx(t) + c
\]

(15.3)

where the dot signifies a time derivative \( \frac{d}{dt} \), and \( F \in \mathbb{R}^{n \times n} \) is a matrix of coefficients and \( c \in \mathbb{R}^n \) is a vector of constants.
15.1 Example: Trajectory of a Ball in One-Dimensional Space

Consider the application of Newton’s Laws of Motions to determine the trajectory of a ball in one dimension. Suppose someone hits a golf ball with a driver straight up in the air and wishes to know how high it will go and how long it will be in the air. Let \( x(t) = [y(t), v(t)] \) be the height of the ball \( y(t) \) and its velocity \( v(t) \) at time \( t \). Also, let the initial conditions be \([0 \text{ m}, 60 \text{ m/s}]\), corresponding to almost hitting the ball from the ground with initial upward velocity of 60 mps (approximately 134.2 mph).

15.1.1 Ordinary Differential Equations

Newton’s Second Law of Motion (force = mass times acceleration), \( F = ma \) and the Law of Gravity (force = mass times minus the gravitational constant, \( F_G = -mg \)) yield a system of Ordinary Differential Equations (ODEs),

\[
\begin{align*}
\dot{y}(t) &= v(t) & \text{time derivative of position} \\
\dot{v}(t) &= -g & \text{time derivative of velocity}
\end{align*}
\]

where the gravity of Earth \( g = 9.807 \text{ m/s}^2 \) and \( \dot{v}(t) = a \) (constant acceleration).

The system of differential equations may be written using vector and matrix notation, with all the columns being treated as column vectors.

\[
\begin{bmatrix}
\dot{y}(t) \\
\dot{v}(t)
\end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} x(t) \\ -g \end{bmatrix}
\]

This system of differential equations may be solved by integration. Integrating both sides of the second equation/row produces,

\[
v(t) - v(0) = \int_0^t -g \, d\tau = -gt
\]

Thus, \( \dot{y}(t) = v(0) - gt \), so

\[
\begin{align*}
y(t) &= y(0) + v(0)t - \frac{1}{2}gt^2 \\
v(t) &= v(0) - gt
\end{align*}
\]

Substituting in the initial conditions, \( x(0) = [y(0), v(0)] = [0 \text{ m}, 60 \text{ m/s}] \), gives

\[
\begin{align*}
y(t) &= -\frac{1}{2}gt^2 + 60t \\
v(t) &= -gt + 60
\end{align*}
\]
15.1.2 Discretization

Typically, it is not so easy to solve a system of ODEs, so iterative algorithms (called integrators, e.g., Euler, Verlet, Runge-Kutta, Dormand-Prince) are used to provide approximate solutions. Using the Verlet Method \[107, 35, 90\] with a time gap of $\Delta t$, the ODEs can discretized to the following form,

\[y(t) = y(t - \Delta t) + v(t - \Delta t)\Delta t - \frac{1}{2}g(\Delta t)^2\]
\[v(t) = v(t - \Delta t) - g\Delta t\]

where the current value is computed from the previous value. This also allows a continuous-time system to treated as a discrete-time system. Here to maintain notational compatibility with the notation previously used for time-series analysis, $t$ in $x_t$ has two interpretation: (1) time index and (2) the actual discrete time/timestamp. Therefore, the above equation can written as follows.

\[y_t = y_{t-1} + v_{t-1}\Delta t - \frac{1}{2}g(\Delta t)^2\]
\[v_t = v_{t-1} - g\Delta t\]

15.1.3 Trajectory Simulation

The trajectory of the ball can be simulated using SCALATiON’s DynamicEq class in the scalation.dynamics package.

```scala
def f (t: Double): VectorD = VectorD (-0.5 * g * t~^2 + mps * t, -g * t + mps)
```

```scala
val dyn = new DynamicEq (f)
```

To determine the maximum height, simply set the velocity to zero ($x_1(t) = 0$), solve for the time $\tau$ when velocity becomes zero, and calculate the height $x_0(t)$ at time $\tau$.

\[\tau = \frac{60}{g} = 6.118 \text{ s}\]
\[x_0(t) = -0.5 \times g \times 6.118^2 + 60 \times 6.118 = 183.5 \text{ m}\]

Can a golf ball hit with the swing speed of an average golfer really go so high? To seek the answer a theorist and an experimentalist may be consulted. The theorist explains that the model ignores other forces (e.g., drag due to air resistance) and therefore, the state equation needs to have an error term. The experimentalist explains that measurements (e.g., using RADAR/LIDAR) need to be taken and of course there will be measurement errors (another error term). These errors can be modeled as noise and, as such,
turn the deterministic state vector into a random one $x(t)$. Furthermore, since measurements are not made continuously, discrete time is introduced. The measurements may be recorded as time-series data $y_t$.

Now there are two stochastic processes: $\{x(t) \mid t \in [0, t_e]\}$, the actual process, and $\{y_t \mid t \in \{0, t_e\}\}$ the observed process. Simplicity argues for merging the two stochastic processes. Unfortunately, it is commonly the case that the actual process is only partially observable (some of the state variables are not directly measurable). Therefore, merging the two may result in less accurate and/or less explainable models.

Dynamic models consisting of state equations and observation/measurement equations come in several varieties:

<table>
<thead>
<tr>
<th>state</th>
<th>state time</th>
<th>model type</th>
</tr>
</thead>
<tbody>
<tr>
<td>continuous</td>
<td>continuous</td>
<td>Kalman-Bucy Filter</td>
</tr>
<tr>
<td>continuous</td>
<td>discrete</td>
<td>Kalman Filter</td>
</tr>
<tr>
<td>discrete</td>
<td>continuous</td>
<td>CT Hidden Markov Chain</td>
</tr>
<tr>
<td>discrete</td>
<td>discrete</td>
<td>Hidden Markov Model</td>
</tr>
</tbody>
</table>

Note: Kalman-Bucy Filter has continuous-time for the state and discrete time for the observations. Similarly, CT Hidden Markov Model has continuous-time for the state and discrete time for the observations. The next section discusses Dynamic Linear Models as Kalman Filters where the forcing/control vector is missing. Kalman filters are models that are particularly useful for dealing with/filtering out noise.

15.1.4 Exercises

1. The diameter and mass of modern golf balls are approximately 4.268 cm and 45.93 grams, respectively. Combine Newton’s Second Law of Motion ($F = ma$) and the Law of Gravity ($F_G = -mg$) to deduce the following equation:

$$\dot{v}(t) = -g$$

2. Consider the effect of wind/air resistance (drag) as another force, drag force $F_D$ (in addition to gravity $F_G$)

$$F_D = \frac{\rho C_D A}{2} v(t)^2$$

where $\rho$ = density of air (1.225 kg/m$^3$), $C_D$ = drag coefficient (0.4), and $A$ = cross sectional area of the golf ball (14.3 cm$^2$).

Recompute the time in the air and maximum height considering the effects of both $F_G$ and $F_D$. 
15.2 Markov Chains

A Markov Chain [94] is a simple type of Markov Model where one is interested in tracking the state of a system over time. Consider the following Markov Chain with $n = 6$ states, corresponding to the number of dollars one currently has. At each discrete time point, flip a coin, heads (with probability $p$) gives a dollar, while tails (with probability $q = 1 - p$) takes a dollar. There are two terminal states, 0 (lose) and 5 (win). One pays $x_0 \in \{1, 2, 3, 4\}$ dollars to start the game. The Markov Chain that models this game is shown in Figure 15.1.

Consider a discrete-valued, discrete-time stochastic processes that represents the state of a system over time $t$.

$$\{x_t : t \in \{0, 1, \ldots\}\} \quad \text{where} \quad x_t \in S$$  \hspace{1cm} (15.4)

The state space $S$ is discrete (finite with $n$ states or countable infinite).

Since $x_t$ is a stochastic process, its trajectory needs to be described probabilistically. For tractability and because it often suffices, the assumption is made that the state $x_t$ is only significantly influenced by its previous state $x_{t-1}$.

$$P(x_t|x_{t-1}, x_{t-2}, \ldots, x_0) = P(x_t|x_{t-1})$$  \hspace{1cm} (15.5)

This is the Markov Property: the transitions from state to state are governed by a discrete-time Markov chain and characterized by a state-transition probability matrix $A = [a_{ij}] \in [0,1]^{n \times n}$, where

$$a_{ij} = P(x_t = j|x_{t-1} = i)$$  \hspace{1cm} (15.6)

The transition from state $i$ to state $j$ is depicted in Figure 15.2.

The Markov Property can be generally stated as the future given the present is conditionally independent of the past.

485
15.2.1 Probability Mass Function

The probability mass function (pmf) for the state at time \( t \) as a vector may be given as follows:

\[
\pi_t = [P(x_t = 0), P(x_t = 1), \ldots, P(x_t = n - 1)]
\]  
(15.7)

Due to the Markov Property, the next state vector (as a row vector) may be computed by vector-matrix multiplication.

\[
\pi_t = \pi_{t-1}A
\]  
(15.8)

Suppose \( p = .6 \) and the initial state is 3 (3 dollars to enter the game). Probabilistically the initial state is given by \( \pi_0 = [0, 0, 0, 1, 0, 0] \), so the RHS of the above equation is

\[
\begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 \\
.4 & 0 & .6 & 0 & 0 & 0 \\
0 & .4 & 0 & .6 & 0 & 0 \\
0 & 0 & .4 & 0 & .6 & 0 \\
0 & 0 & 0 & .4 & 0 & .6 \\
0 & 0 & 0 & 0 & .4 & 0 \\
\end{bmatrix}
\]

Then probabilistically the next state is \( \pi_1 = \pi_0A = [0, 0, .4, 0, .6, 0] \). The dot product of the row vector \( \pi_0 \) with each column of \( A \) is used to compute \( \pi_1 \).

In SCALATION, advancing to the next state is carried out by the `next` method.

```scala
def next (pi: VectorD): VectorD = pi *: a
```

where the right-associative `*:` operator is for vector-matrix multiplication (and complements the left associate `*` operator for matrix-vector multiplication).

This recurrence can be unfolded to yield the following equation,

\[
\pi_t = \pi_{t-1}A \\
= \pi_{t-2}A^2 \\
= \pi_0A^t
\]

i.e., \( \pi_0 \) times \( A \) to the \( t^{th} \) power.

The only choice in optimizing one’s game is to determine the number of dollars to spend to start the game to, for example, maximize expected earnings, depending on the weight (value of \( p \)) of the coin. If \( p = 1 \), always start with one dollar, but if \( p = 0 \), do not play the game. In the first case, the expected earning are 4 dollars per game, while in the second playing the game will ensure some level of loss, so 0 dollars of expected earnings is optimal.

For other values of \( p \), a simple way to estimate the expected earnings is to simulate playing the game many times for a given start state, and determining the average earnings. See `MarkovChainTest2` and `MarkovChainTest3` in the `scalation.simulation.state` package for two ways to estimate the earnings.
15.2.2 Reducible Markov Chains

State $j$ is reachable from state $i$ if

$$a_{ij}^{(t)} > 0$$

(15.9)

for some discrete time $t$, where $a_{ij}^{(t)}$ is the $i,j$ element in $A^t$.

Based on reachability a Markov Chain may decomposed into multiple subschains, where all the states in each subchain are reachable from each other. If states $i$ and $j$ are mutually reachable, they are said to communicate. Since communication forms an equivalence class (reflexive, symmetric and transitive), each subchain is an equivalence class. The six-state Markov Chain shown in the figure has three equivalence classes.

1. $S_1 = \{0\}$ Lost
2. $S_1 = \{1, 2, 3, 4\}$ Still Playing the Game
3. $S_1 = \{5\}$ Won

States 0 and 5 are absorbing states, since once in such a state, it will never be left. A Markov Chain that has just one communication/equivalence class is called an Irreducible Markov Chain.

A state $i$ if said the recurrent if the probability of returning some time in the future is one, otherwise it said to be transient (may never return). A recurrent state is either positive (finite return time) or null recurrent (infinite return time).

15.2.3 Limiting/Steady-State Distribution

For an Irreducible, Positive Recurrent Markov Chain, $\pi_t$ may converge to a limiting probability distribution,

$$\pi = \lim_{t \to \infty} \pi_t$$

(15.10)

After convergence, $\pi$ may be substituted for both $\pi_t$ and $\pi_{t-1}$, so the previous boxed equations becomes,

$$\pi = \pi A$$

(15.11)

where $\pi$ is the probability vector (non-negative and sums to 1) and $A$ is the transition probability matrix. When this equation has a solution, $\pi$ is the limiting/steady-state probability vector.

Interpretation

If the Markov Chain is

- *aperiodic*, $\pi$ can be viewed as a long-term probability for $t$ large enough, otherwise, if it is

- *periodic*, $\pi$ is interpreted as as an average over time.

For example, a Markov Chain that oscillates between two states, the long-term probabilities will depend on the initial conditions.
Solving for the Limiting Probabilities

Subtracting \( \pi \) from the above boxed equation \( \pi = \pi A \) yields,

\[
\pi A - \pi = 0
\]

(15.12)

with \( 0 \) as a row vector. Using an identity matrix \( I \), this may be rewritten,

\[
\pi (A - I) = 0
\]

(15.13)

Taking the transpose produces

\[
(A - I)^\top \pi^\top = 0^\top
\]

(15.14)

The vector \( \pi \) is the eigenvector solution to the left eigenvalue problem for eigenvalue \( \lambda = 1 \) (see the chapter on Linear Algebra). This can be solved by computing the nullspace of \((A - I)^\top\). In ScalaTion, the limiting distribution can be found using QR Factorization (see the exercises).

```scala
def limit: VectorD =
val fac = new Fac_QR ((a - eye (a.dim, a.dim)).transpose, true)
fac.nullspace (a.dim-1)(?, 0).toProbability
end limit
```

The solution for \( \pi \) also may be found by first computing \((A - I)^\top\) and then solving for \( \pi \) using for example LU factorization. One of the equations will turn out to be redundant and can be replaced with the normalization equation (probabilities sum to 1). The example below illustrates three ways to solve for \( \pi \).

Example: Computing the Limiting Probabilities

Consider the example problem from *Introduction to Probability Models*, 3rd Ed., Ross, p. 146 having the following transition probability matrix \( A \). Solve for the stationary (steady-state) distribution three ways:

1. Start with state probability vector \( \pi = [\pi_0, \pi_1, \pi_2] = [.5, .5, 0] \) and repeatedly compute \( \pi A \).

2. Solve the eigenvector problem \( \pi = \pi A \), i.e.,

\[
[\pi_0, \pi_1, \pi_2] = [\pi_0, \pi_1, \pi_2]
\begin{bmatrix}
.5 & .4 & .1 \\
.3 & .4 & .3 \\
.2 & .3 & .5 \\
\end{bmatrix}
\]

Since the \( A \) matrix is stochastic, one of the equations is redundant and may be replaced with the normalization equation \( \|\pi\|_1 = 1 \).

\[
\begin{align*}
\pi_0 &= .5\pi_0 + .3\pi_1 + .2\pi_2 \\
\pi_1 &= .4\pi_0 + .4\pi_1 + .3\pi_2 \\
\pi_2 &= 1 - \pi_0 - \pi_1
\end{align*}
\]
3. This may be rewritten as an augmented matrix and solved using LU Factorization.

\[
\begin{pmatrix}
-0.5 & 0.3 & 0.2 & 0 \\
0.4 & -0.6 & 0.3 & 0 \\
1 & 1 & 1 & 1
\end{pmatrix}
\]

Note, the matrix above is simply \((A - I)^T\) with the last row replaced with 1s from the normalization equation.

**Software Solution**

The following SCALATION code uses all three solution techniques.

```scala
/** The 'markovChainTest4' function tests the 'MarkovChain' class. 
 * @see Introduction to Probability Models, 3rd Ed., Ross, p. 146. 
 * > runMain scalation.simulation.state.markovChainTest4 
 */
@main def markovChainTest4 (): Unit =

val a = MatrixD ((3, 3), .5, .4, .1, // 3-by-3 matrix
    .3, .4, .3,
    .2, .3, .5)
val mc = new MarkovChain (a)
println ("Discrete-Time Markov Chain mc = " + mc + "\n")
banner ("Discrete-Time Markov Chain: transient solution:")

var pi = VectorD (.5, .5, 0)
println ("on epoch 0, \pi = " + pi)
for k <- 1 to 10 do
    pi = mc.next (pi)
    println (s"on epoch $k, \pi = $pi")
end for

banner ("eigenvector solution for steady-state \pi = " + mc.limit)
val aa = MatrixD ((3, 3), -.5, .3, .2,
    .4, -.6, .3,
    1, 1, 1 )
val b = VectorD (0, 0, 1)
val lu = new Fac_LU (aa).factor ()
banner ("lu factorization for steady-state \pi = " + lu.solve (b))

end markovChainTest4
```

489
15.2.4 Continuous-Time Markov Chains

Continuous-Time Markov Chains are analogous to their discrete-time counterparts, except that transitions can occur at any time. This makes the notion of transition probability hard to define, so it is replaced with the notion of transition rate. Therefore, the transition probability matrix $A$ (denoted $P$ in some textbooks) is replaced with a transition rate matrix $Q = [q_{ij}] \in (\mathbb{R}^+)^{n \times n}$.

Consider a Markov model for a single-server queue with room for four customers. Arrivals are assumed to come at a rate of $\lambda$ customers per unit time and can be serviced at a rate of $\mu$ customers per unit time. If the system is empty, the server is idle, while if it is full of customers, the new arrival is turned away. This can be modeled as an $n = 5$ state Continuous-Time Markov Chain, as shown in Figure 15.3.

![Figure 15.3: State Transition Diagram for Five-State Continuous-Time Markov Chain](https://example.com/fig15_3)

Self-loops are not included, since the state is unchanged until there is an out transition. One may view this as an event that changes the state (here there are arrival and service completion events).

The transient solution for Continuous-Time Markov Chains can be found by solving Kolmogorov differential equations, see Ross, Chapter 6 [86].

15.2.5 Limiting/Steady-State Distribution

The limiting/steady-state solution can be given by,

$$
\pi Q = 0
$$

See [https://mast.queensu.ca/~stat455/lecturenotes/set5.pdf](https://mast.queensu.ca/~stat455/lecturenotes/set5.pdf) for a derivation (where $Q$ is called the generator matrix $G$).

When the system (single-server queue) opens, there will be no customers, but they will arrive at rate $\lambda$ until the system close. The server can process customers at rate of $\mu$ per unit time. The stationary/steady-state solution can be found by solving the following equations.

$$
\begin{bmatrix}
\pi_0 & \pi_1 & \pi_2 & \pi_3 & \pi_4 \\
-\lambda & \lambda & 0 & 0 & 0 \\
\mu & -(\lambda + \mu) & \lambda & 0 & 0 \\
0 & \mu & -(\lambda + \mu) & \lambda & 0 \\
0 & 0 & \mu & -(\lambda + \mu) & \lambda \\
0 & 0 & 0 & \mu & -\mu
\end{bmatrix} = \begin{bmatrix}0 & 0 & 0 & 0 \end{bmatrix}
$$

The diagonal elements are set so that the rows of matrix $Q$ sum to 0 (inflow = output).
Total Balance Equations

Multiplying the vector $\pi$ by the matrix $Q$ produces the following five equations.

\[-\lambda \pi_0 + \mu \pi_1 = 0\]
\[\lambda \pi_{j-1} - (\lambda + \mu) \pi_j + \mu \pi_{j+1} = 0 \quad \text{for } j = 1, 2, 3\]
\[\lambda \pi_3 - \mu \pi_4 = 0\]

These equations are referred to as the total balance equations.

Partial Balance Equations

In this case a simpler approach is possible, as the solution can be developed using partial balance equations that equates up-flow with down-flow, so

\[\mu \pi_j = \lambda \pi_{j-1} \quad \text{for } j = 0, 1, 2, 3, 4 \quad (15.16)\]

Therefore, $\pi_1 = \frac{\lambda}{\mu} \pi_0$, $\pi_2 = \frac{\lambda}{\mu} \pi_1$, $\pi_3 = \frac{\lambda}{\mu} \pi_2$, and $\pi_4 = \frac{\lambda}{\mu} \pi_3$.

**Traffic Intensity**: The traffic intensity is the ratio of the arrival rate to the service rate. The higher the traffic intensity, the more the chain is pushed to the right (toward more congestion).

\[\rho = \frac{\lambda}{\mu} \quad (15.17)\]

The partial balance equations can be expressed with $\rho$ replacing $\lambda$ and $\mu$.

\[\pi_j = \rho \pi_{j-1} \quad \text{for } j = 0, \ldots, 4 \quad (15.18)\]

This recursive equation can be unfolded to give,

\[\pi_j = \rho^j \pi_0 \quad (15.19)\]

Furthermore, the probabilities must add to one (normalization equation), so

\[\left[1 + \rho + \rho^2 + \rho^3 + \rho^4\right] \pi_0 = 1 \quad (15.20)\]

Solving for $\pi_0$ (see the exercises) yields

\[\pi_0 = \frac{1 - \rho}{1 - \rho^n} \quad (15.21)\]

Finally, the state-probabilities may be determined.

\[\pi_j = \frac{1 - \rho}{1 - \rho^n} \rho^j \quad (15.22)\]

This is the solution for an M/M/1/K Queue (with $K = 4$ and $n = K + 1 = 5$). The notation M/M/1/K means the arrival process is Markovian (Poisson or Exponential inter-arrival times), the service distribution is Exponential, the number of servers is 1 and customer capacity is K (one in service and the rest waiting).
15.2.6 Queueing Models

The solution for an M/M/1/K Queue may be written

\[ \pi_j = \frac{1 - \rho}{1 - \rho^{K+1}} \rho^j \]  

(15.23)

This solution works when \( \rho = 0 \) \((\pi_0 = 1), \rho \in (0, 1), \rho = 1 \) (via L'Hospital's Rule) and when \( \rho > 1 \).

As the waiting capacity \( K \) (the therefore the number of states \( n \)) goes to infinity, stability requires traffic intensity \( \rho < 1 \). In which case \( \rho^{K+1} \) goes to zero. Therefore, the solution for an M/M/1 Queue can be obtained.

\[ \pi_j = (1 - \rho)\rho^j \]  

(15.24)

Note that \( \pi_0 = 1 - \rho \) is the probability the server is idle.

**Expected Number in the System**

Given that \( \pi_j = P(x = j) \) indicates the probability there are \( j \) customers in the system, the expected number in the system is given as follows.

\[ E[x] = \sum_{j=0}^{\infty} j\pi_j = (1 - \rho) \sum_{j=0}^{\infty} j\rho^j \]  

(15.25)

As indicated in the exercises, the result becomes,

\[ L = E[x] = \frac{\rho}{1 - \rho} \]  

(15.26)

The number in service \( L_s \) corresponds to probability the server is busy \( = 1 - \pi_0 = \rho \). Therefore, the expected length of the queue is

\[ L_q = L - L_s = \frac{\rho}{1 - \rho} - \rho = \frac{\rho^2}{1 - \rho} \]  

(15.27)

**Application of Little's Law**

The expected time in service \( T_s = \frac{L}{\mu} \) is one over the service rate, so

\[ L_s = \lambda T_s = \frac{\lambda}{\mu} = \rho \]  

(15.28)

This relationship between length (number in) and time carries over to the queue

\[ L_q = \lambda T_q \]  

(15.29)

and the system.

\[ L = \lambda T \]  

(15.30)

**Summary of Results**

In summary, the formulas for an M/M/1 Queue are collected into Table 15.2.
Table 15.2: Formulas for M/M/1 Queueing Models

<table>
<thead>
<tr>
<th>part</th>
<th>length</th>
<th>formula</th>
<th>time</th>
<th>formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>Queue</td>
<td>$L_q$</td>
<td>$\frac{\rho^2}{1-\rho}$</td>
<td>$T_q$</td>
<td>$\frac{\rho/\mu}{1-\rho}$</td>
</tr>
<tr>
<td>Service</td>
<td>$L_s$</td>
<td>$\rho$</td>
<td>$T_s$</td>
<td>$\frac{1}{\mu}$</td>
</tr>
<tr>
<td>System</td>
<td>$L$</td>
<td>$\frac{\rho}{1-\rho}$</td>
<td>$T$</td>
<td>$\frac{1/\mu}{1-\rho}$</td>
</tr>
</tbody>
</table>

15.2.7 Exercises

1. For the given six-state discrete-time Markov Chain, advance the state probability $\pi_t$ over the next 20 time points.

2. For the six-state Markov Chain, what happens to the probabilities of states 1 to 4 as time increases.

3. Consider the middle subchain, states 1 to 4. Do they form a irreducible Markov Chain and admit a stationary/steady state solution? Compute the value for this solution $\pi$.

4. A square matrix is stochastic if all its elements are non-negative and all the columns sums equal 1. Show that a stochastic matrix has an eigenvalue equal to 1. Hint: see [textbooks.math.gatech.edu/ila/stochastic-matrices.html](https://textbooks.math.gatech.edu/ila/stochastic-matrices.html)

5. For discrete-time Markov chains, explain how the limit method works for computing $\pi$.

6. Show the following formula holds.

$$s_n = \sum_{j=n}^{\infty} \rho^j = \frac{\rho^n}{1-\rho}$$

Use this to deduce that

$$[1 + \rho + \rho^2 + \rho^3 + \rho^4] \pi_0 = [s_0 - s_5] \pi_0 = 1$$

and finally that

$$\pi_0 = \frac{1-\rho}{1-\rho^5}$$

7. Develop the steady-state solution $\pi_j$ for an M/M/c/K Queue where $c$ is the number of servers and $K$ is the system capacity.

8. Derive the formula for the expected number in the system $L$ for an M/M/1 Queue. Hint: $\frac{d}{d\rho}\rho^j = j \rho^{j-1}$.

9. The relationship $L = \lambda T$ is called Little’s Law. Sketch Stidham’s proof of the law. See [95]
15.3 Dynamic Linear Models

As with a Hidden Markov Model (HMM), a Dynamic Linear Model (DLM) may be used to represent a system in terms of two stochastic processes, the state of the system at time $t$, $x_t$, and the observed values from measurements of the system at time $t$, $y_t$. The main difference from an HMM is that the state and its observation are treated as continuous quantities. For time series analysis, it is natural to treat time as discrete values.

As background, consider the following system of homogeneous ODEs that only includes a linear transformation (no constant term). The transition matrix $F^{(c)}$ has been renamed to emphasize that it is for the continuous time problem.

$$\dot{x}(t) = F^{(c)}x(t)$$  \hfill (15.31)

The corresponding discrete-time system is defined as

$$x_t = e^{\Delta t F^{(c)}} x_{t-\Delta t}$$  \hfill (15.32)

where $\Delta t$ is time gap between consecutive time points. It is assumed here that the time gaps are uniform. The matrix exponential can be calculated using the ? algorithm [?]. We define matrix $F$ as follows:

$$F = e^{\Delta t F^{(c)}}$$  \hfill (15.33)

Substituting in the matrix $F$ gives,

$$x_t = F x_{t-1}$$  \hfill (15.34)

Again to maintain notational compatibility with the notation previously used for time-series analysis, $t$ in $x_t$ has two interpretation: (1) time index and (2) the actual discrete time/timestamp. Therefore, the above equation can written as follows.

$$x_t = F x_{t-1}$$  \hfill (15.35)

To deal with uncertainty in the system a noise term may be added. In addition, the state and observation equations distinguished.

For a basic DLM, the dynamics of the system are described by two equations: The State Equation indicates how the next state vector $x_t$ is dependent on the previous state vector $x_{t-1}$ and a process noise vector $w_t \sim \text{Normal}(0, Q)$

$$x_t = F x_{t-1} + w_t$$  \hfill (15.36)

where $Q$ is the covariance matrix for the process noise. If the dynamics are deterministic, then the covariance matrix is zero, otherwise it can capture uncertainty in the relationships between the state variables (e.g., simple models of the flight of a golf ball often ignore the effects due to the spin on the golf ball).

The Observation/Measurement Equation indicates how at time $t$, the observation vector $y_t$ is dependent on the current state $x_t$ and a measurement noise vector $v_t \sim \text{Normal}(0, R)$

$$y_t = H x_t + v_t$$  \hfill (15.37)
where $R$ is the covariance matrix for the measurement noise/error. The process noise and measurement noise are assumed to be independent of each other. The state transition matrix $F$ indicates the linear relationships between the state variables, while the $H$ matrix establishes linear relationships between the state of system and its observations/measurements.

### 15.3.1 Example: Traffic Sensor

Consider the operation of a road sensor that records traffic flow (vehicles per 15 minutes) and average speed (km per hour). Let $x_t = [x_{t0}, x_{t1}]$ be the flow of vehicles $x_{t0}$ and their average speed $x_{t1}$ at time $t$. Assume that the flow is high enough that it can be treated as a continuous quantity and that the covariance matrices are diagonal (uncertainty of flow and speed are independent). The dynamics of the system then may be described by the following state equations:

\[
x_{t0} = f_{00}x_{t-1,0} + f_{01}x_{t-1,1} + w_t\0
x_{t1} = f_{10}x_{t-1,0} + f_{11}x_{t-1,1} + w_t\1
\]

The sensor tries to capture the dynamics of the system, but depending on the quality of the sensor there will be measurement errors. The observation/measurement variables $y_t = [y_{t0}, y_{t1}]$ may correspond to the state variables in a one-to-one correspondence or by some linear relationship. The observation of the system then may be described by the following observation equations:

\[
y_{t0} = h_{00}x_{t0} + h_{01}x_{t1} + v_t\0
y_{t1} = h_{10}x_{t0} + h_{11}x_{t1} + v_t\1
\]

Further assume that estimates for the $F$ and $H$ parameters of the model have been found (see the subsection on Training).

**State Equations**

\[
x_{t0} = 0.9x_{t-1,0} + 0.2x_{t-1,1} + w_t\0
x_{t1} = -0.4x_{t-1,0} + 0.8x_{t-1,1} + w_t\1
\]

\[
\begin{bmatrix} x_{t0} \\ x_{t1} \end{bmatrix} = \begin{bmatrix} 0.9 & 0.2 \\ -0.4 & 0.8 \end{bmatrix} \begin{bmatrix} x_{t-1,0} \\ x_{t-1,1} \end{bmatrix} + w_t
\]

These state equations suggest that the flow will be a high percentage of the previous flow, but that higher speed suggests increasing flow. In addition, the speed is based on the previous speed, by higher flow suggests that speeds may be decreasing (e.g., due to congestion).

**Observation/Measurement Equations**

\[
y_{t0} = 1.0x_{t0} - 0.1x_{t1} + v_t\0
y_{t1} = -0.1x_{t0} + 1.0x_{t1} + v_t\1
\]
\[ y_t = \begin{bmatrix} y_{t0} \\ y_{t1} \end{bmatrix} = \begin{bmatrix} 1.0 & -0.1 \\ -0.1 & 1.0 \end{bmatrix} x_t + v_t \]

These observation/measurement equations suggest that higher speed makes it more likely for a vehicle to pass the sensor without being counted and higher flow makes under-estimation of speed to be greater.

15.3.2 Exercises

1. For a DLM, consider the case where \( m = n = 1 \). The state equations and measurement equations become

\[
\begin{align*}
x_t &= a x_{t-1} + w_t \\
y_t &= c x_t + v_t
\end{align*}
\]

where \( w_t \sim \text{Normal}(0, \sigma^2_q) \) and \( v_t \sim \text{Normal}(0, \sigma^2_r) \). Compare this model with an AR(1) model.

2. For the Traffic Sensor Example, let \( Q = \sigma^2_q I \) and \( R = \sigma^2_r I \). Develop a DLM model using ScalaTion and try low, medium and high values for the variances \( \sigma^2_q \) and \( \sigma^2_r \) (9 combinations). Let the initial state of the system be \( x_{00} = 100.0 \) vehicles per 15 minutes and \( x_{01} = 100.0 \) km per hour. How does the relative amount of process and measurement error affect the dynamics/observation of the system?

3. Consider the state and observation equations given in the Traffic Sensor Example and assume that the state equations are deterministic (no uncertainty in system, only in its observation). Reduce the DLM to a simpler type of time series model. Explain.

4. Use the Traffic Sensor Dataset (traffic.csv) to estimate values for the 2-by-2 covariance matrices \( Q \) and \( R \).

5. Use the Traffic Sensor Dataset (traffic.csv) to estimate values for the parameters of a DLM model, i.e., for the \( F \) and \( H \) 2-by-2 matrices.
15.4 Kalman Filter

A Kalman Filter (KF) is a Dynamic Linear Model that incorporates an outside influence on the system. If a driving force or control is applied to the system, an additional term $G\mathbf{u}_t$ is added to the state equation,

$$\mathbf{x}_t = F\mathbf{x}_{t-1} + G\mathbf{u}_t + \mathbf{w}_t$$  \hspace{1cm} (15.38)

where $\mathbf{u}_t$ is the (assumed deterministic) force/control vector and the $B$ matrix establishes as linear relationships between the force/control vector and the state vector. An example of a force/control is the constant vector that includes the force of gravity as discussed in the golf ball trajectory problem,

$$\mathbf{u}_t = [-g]$$

The observation/measurement equation remains the same.

$$\mathbf{y}_t = H\mathbf{x}_t + \mathbf{v}_t$$

The process noise $\mathbf{w}_t$ and the measurement noise $\mathbf{v}_t$ also remain the same. The Kalman Filter model, therefore includes five matrices.

<table>
<thead>
<tr>
<th>matrix</th>
<th>dimensions</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$F$</td>
<td>$n$-by-$n$</td>
<td>state transition matrix</td>
</tr>
<tr>
<td>$G$</td>
<td>$n$-by-$l$</td>
<td>state-control matrix</td>
</tr>
<tr>
<td>$H$</td>
<td>$m$-by-$n$</td>
<td>observation matrix</td>
</tr>
<tr>
<td>$Q$</td>
<td>$n$-by-$n$</td>
<td>process noise covariance matrix</td>
</tr>
<tr>
<td>$R$</td>
<td>$m$-by-$m$</td>
<td>measurement noise covariance matrix</td>
</tr>
</tbody>
</table>

The Kalman Filter model at time $t$ includes five vectors:

<table>
<thead>
<tr>
<th>vector</th>
<th>dimension</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mathbf{x}_t$</td>
<td>$n$</td>
<td>state vector</td>
</tr>
<tr>
<td>$\mathbf{u}_t$</td>
<td>$l$</td>
<td>force/control vector</td>
</tr>
<tr>
<td>$\mathbf{y}_t$</td>
<td>$m$</td>
<td>measurement vector</td>
</tr>
<tr>
<td>$\mathbf{w}_t$</td>
<td>$n$</td>
<td>process noise vector</td>
</tr>
<tr>
<td>$\mathbf{v}_t$</td>
<td>$m$</td>
<td>measurement noise vector</td>
</tr>
</tbody>
</table>

15.4.1 Example: Golf Ball Trajectory

Returning to problem posed at the beginning of the chapter of modeling the trajectory of a golf ball hit straight up with an initial velocity of 60 meters per second (mps), the discretized form of the system of Ordinary Differential Equations is copied below.
\[ y_t = y_{t-1} + v_{t-1} \Delta t - \frac{1}{2} g (\Delta t)^2 \]
\[ v_t = v_{t-1} - g \Delta t \]

In matrix-vector form, the equation for \( x_t \) may be written as

\[
\begin{bmatrix} y_t \\ v_t \end{bmatrix} = \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x_{t-1} \\ \Delta t \end{bmatrix} + \begin{bmatrix} \frac{1}{2} (\Delta t)^2 \\ -g \end{bmatrix}
\]

When noise is added, the variables will become random variables (blue font). The discrete-time system then serves as the basis to formulate the Kalman filter state equations \cite{80}.

\[
\begin{bmatrix} x_t \\ v_t \end{bmatrix} = \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x_{t-1} \\ \Delta t \end{bmatrix} + \begin{bmatrix} \frac{1}{2} (\Delta t)^2 \\ -g \end{bmatrix} + \begin{bmatrix} w_t \end{bmatrix}
\]

In this case, the force/control is the one-dimensional vector \( u_t = [-g] \). Again the initial conditions are \( x_0 = [0 \text{ m}, 60 \text{ m/s}] \). For simplicity, the process noise \( w_t \) is assumed to be on scale with the measurement noise (see below).

\[ Q = \mathbb{C}[w_t] = 0.05 I = \begin{bmatrix} 0.05 & 0.0 \\ 0.0 & 0.05 \end{bmatrix} \]

The measurement equation indicate what variables are measured and how they relate to the state variables. When are the state variables are directly measurable and one if interested all the state variables, the observation matrix will be the 2-dimensional identity matrix \( H = I \).

\[ y_t = I x_t + v_t \]

The last step is to determine the covariance \( R \) of the measurement noise \( v_t \)

\[ v_t \sim \text{Normal}(0, R) \]

Modern golf ball tracking devices have standard deviations as low as \( \sigma_0 = 0.25 \text{ meters} \) for position in a particular dimension and \( \sigma_1 = 0.2 \text{ meters per second} \) for velocity. These are rough estimates based on sources such as \cite{55}. These may be used to for the variances (diagonal elements) in the covariance matrix. Remaining is to determine the correlation \( \rho_{01} \) between \( v_{t0} \) and \( v_{t1} \) to get the covariance \( \sigma_{01} = \rho_{01} \sigma_0 \sigma_1 \). Such information is hard to come by, but it makes sense that they would be positively correlated, so let \( \rho_{01} = 0.5 \).

\[
R = \mathbb{C}[v_t] = \begin{bmatrix} \sigma_0^2 & \sigma_{01} \\ \sigma_{01} & \sigma_1^2 \end{bmatrix} = \begin{bmatrix} 0.0625 & 0.00125 \\ 0.00125 & 0.04 \end{bmatrix}
\]

See the next section for alternative approaches for estimating \( Q \) and \( R \).

Note that if only the height is of interest, then observation matrix \( H = [1, 0] \).
15.4.2 Training

The main goal of training is to minimize the error in estimating the state. At time $t$, a new measurement $y_t$ becomes available. The errors before and after this event are the differences between the actual state $x_t$ and the estimated state before $\hat{x}_t^{-}$ (predicted) and after $\hat{x}_t$ (corrected) [110].

$$e_t^{-} = x_t - \hat{x}_t^{-}$$
$$e_t = x_t - \hat{x}_t$$

Since $w_t$ has a zero mean, the covariance matrices ($P_t^{-}$ and $P_t$) for the before and after state errors may be computed as expectations of their outer products.

$$P_t^{-} = C[e_t^{-}] = E[e_t^{-} \otimes e_t^{-}] \quad \text{before errors} \quad (15.39)$$
$$P_t = C[e_t] = E[e_t \otimes e_t] \quad \text{after errors} \quad (15.40)$$

The essential insight by Kalman was that the after estimate should be the before estimate adjusted by a weighted difference between the actual measured value $y_t$ and its before estimate $H\hat{x}_t^{-}$.

$$\hat{x}_t^{-} = F\hat{x}_{t-1} + Gu_t \quad \text{predicted} \quad (15.41)$$
$$\hat{x}_t = \hat{x}_t^{-} + K_t[y_t - H\hat{x}_t^{-}] \quad \text{corrected} \quad (15.42)$$

The $n$-by-$m$ $K_t$ matrix is called the Kalman Gain and the above equations may be referred to as the Kalman state update equations. If the actual measurement is very close to its predicted value, little adjustment to the predicted state value is needed. On the other hand, when there is a disagreement, the adjustment based upon the measurement should be tempered based upon the reliability of the measurement. A small gain will dampen the adjustment, while a high gain may result in large adjustments. The trick is to find the optimal gain $K_t$.

Using a Minimum Variance Unbiased Estimator (MVUE) for parameter estimation for a Kalman Filter means that the trace of the error covariance matrix should be minimized (see exercises for details).

$$\mathbb{V}[\|e_t\|] = E[\|e_t\|^2] = \text{trace } C[e_t] \quad (15.43)$$

Plugging the Kalman Gain equation into the above equation gives the following optimization problem:

$$\min \text{trace } E[(\hat{x}_t^{-} + K_t[y_t - H\hat{x}_t^{-}]) \otimes (\hat{x}_t^{-} + K_t[y_t - H\hat{x}_t^{-}])] \quad (15.44)$$

This optimization will produce (see exercises) the following equation that can be used to update the Kalman Gain.

$$P_t^{-} = FP_t^{-1}F^t + Q \quad (15.45)$$
$$K_t = P_t^{-}H'[HP_t^{-}H^t + R]^{-1} \quad (15.46)$$
$$P_t = [I - K_tH]P_t^{-} \quad (15.47)$$
15.4.3 Exercises

1. Suppose that fog negatively affects traffic and speed. Use the Traffic Sensor with Fog Dataset (traffic_fog.csv) to estimate values for the 2-by-2 covariance matrices $Q$ and $R$.

2. Use the Traffic Sensor with Fog Dataset (traffic_fog.csv) to estimate values for the parameters of a Kalman Filter model, i.e., for the $F$, $G$ and $H$ 2-by-2 matrices.

3. Show that if $\hat{y}$ is an unbiased estimator for $y$ (i.e., $E[\hat{y}] = E[y]$) then the minimum error variance $\mathbb{V}[\|y - \hat{y}\|]$ is

$$
\mathbb{E}[\|y - \hat{y}\|^2] = \text{trace} \mathbb{E}[(y - \hat{y}) \otimes (y - \hat{y})]
$$

4. Explain why minimizing the trace of the covariance $\mathbb{C}[e_t]$ leads to optimal Kalman Gain $K$. 
15.5 Extended Kalman Filter

When some of the relationships between state variables are nonlinear, the simplest option is to use an Extended Kalman Filter (EKF). The linear combinations in the equations for Kalman Filters are now replaced with differentiable (nonlinear) vector functions \( f \) and \( h \).

**State Transition Function**

The dynamics of the state are governed by a (nonlinear) state transition function \( f : \mathbb{R}^n \rightarrow \mathbb{R}^n \) and specified in the state equation,

\[
x_t = f(x_{t-1}, u_t) + w_t
\]

where \( u_t \) is the (assumed deterministic) control vector, if relevant.

**Observation/Measurement Function**

The observation/measurement equation may also include a (nonlinear) observation function \( h : \mathbb{R}^n \rightarrow \mathbb{R}^m \) of the state.

\[
y_t = h(x_t) + v_t
\]

The process noise \( w_t \) and the measurement noise \( v_t \) should be close to Gaussian (Normally distributed). They are also assumed to be additive, otherwise they need to be incorporated into the \( f \) and \( h \) functions.

<table>
<thead>
<tr>
<th>function</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f ) : ( \mathbb{R}^n \rightarrow \mathbb{R}^n )</td>
<td>state transition function</td>
</tr>
<tr>
<td>( h ) : ( \mathbb{R}^n \rightarrow \mathbb{R}^m )</td>
<td>observation function</td>
</tr>
</tbody>
</table>

For the example in the next subsection of a *SEIHRD* epidemic/pandemic model, the state vector is 6-dimensional \((n = 6)\), while the measurement/observation vector is 4-dimensional \((m = 4)\).

15.5.1 Training

Extended Kalman Filters operate much like Kalman filters [80]. The only change to the Kalman update equations is to use the \( f \) and \( h \) functions in place of the multiplications involving the \( F, G \) and \( H \) matrices (for simplicity in previous section, these were taken to be constant matrices, but in a more general treatment they would vary with time \( F_t, G_t \) and \( H_t \)).

The Kalman state update equations for EKF are as follows:

\[
\hat{x}_t^- = f(\hat{x}_{t-1}, u_t) \quad \text{predicted} \quad (15.50)
\]

\[
\hat{x}_t = \hat{x}_t^- + K_t[y_t - h(\hat{x}_t^-)] \quad \text{corrected} \quad (15.51)
\]

The Kalman Gain for EKF may be updated as follows:
\[ P_t^- = F_{t-1}P_{t-1}F_{t-1}^T + Q \]  
\[ K_t = P_t^-H_t^T[H_tP_t^-H_t^T + R]^{-1} \]  
\[ P_t = [I - K_tH_t]P_t^- \]  

where now the matrices \( F_t \) and \( H_t \) are slopes of the \( f \) and \( h \) functions, respectively.

\[ F_t = \frac{\partial f}{\partial x_t} \]  
\[ H_t = \frac{\partial h}{\partial x_t} \]

In other words, at each discrete time step \( t \), the nonlinear vector-valued functions \( f \) and \( h \) are approximated by their local slopes at the predicted state \( \hat{x}_{t-1} \), computed as Jacobian matrices. Note, \( F_t^T \) and \( H_t^T \) are the transposes of \( F_t \) and \( H_t \), respectively.

Recall that the Jacobian of a vector function \( f : \mathbb{R}^n \rightarrow \mathbb{R}^l \) is an \( l \)-by-\( n \) matrix.

\[
\begin{bmatrix}
\frac{\partial f_0}{\partial x_0} & \frac{\partial f_0}{\partial x_1} & \cdots & \frac{\partial f_0}{\partial x_n} \\
\frac{\partial f_1}{\partial x_0} & \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\
\vdots & \vdots & \ddots & \vdots \\
\frac{\partial f_{l-1}}{\partial x_0} & \frac{\partial f_{l-1}}{\partial x_1} & \cdots & \frac{\partial f_{l-1}}{\partial x_n}
\end{bmatrix}
\]

### 15.5.2 Example: SEIHRRD Model

This subsection presents a detailed example of how an Extended Kalman Filter (EKF) can be used to model epidemics and pandemics. An SEIHRRD model as an extension of the SEIR model [58] can be used to forecast the number of infections, hospitalizations and deaths in epidemics.

The EKF model will derived from a system of Ordinary Differential Equations (ODEs). The time varying variables/functions are specified below. At this point, time is treated continuously.

1. The number of Susceptible individuals at time \( t \) is denoted by \( S(t) \).
2. The number of Exposed individuals at time \( t \) is denoted by \( E(t) \).
3. The number of Infected and not Hospitalized individuals at time \( t \) is denoted by \( I(t) \).
4. The number of Infected and Hospitalized individuals at time \( t \) is denoted by \( H(t) \).
5. The number of Recovered individuals at time \( t \) is denoted by \( R(t) \).
6. The number of Deaths time \( t \) is denoted by \( D(t) \).

To make this more clear, suppose the study is the spread of COVID-19 during the year 2020 in the United States. The population of individuals is given by \( N = 330 \) million.
State Transition Rates

The dynamics of the system are governed by the state transition rates. These will become parameters in the differential equations to be estimated from the data. Assuming the birth rate matches the COVID-19 death rate, the population not change over time (a short time span approximation).

\[ N = S(t) + E(t) + I(t) + H(t) + R(t) \]  \hspace{1cm} (15.57)

The model is begun when there is a minimal level of infection. Assuming that the whole population is susceptible, the initial conditions are \( S(0) = N - I(0) \). For this model, time 0 is set to when United States has at least one infection. Each of the other variables at time 0 is 0.

The state transition rates are explained in the table below.

<table>
<thead>
<tr>
<th>Rate</th>
<th>From</th>
<th>To</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( q_0 )</td>
<td>( S )</td>
<td>( E )</td>
<td>exposure rate</td>
</tr>
<tr>
<td>( q_1 )</td>
<td>( E )</td>
<td>( I )</td>
<td>infection rate</td>
</tr>
<tr>
<td>( q_2 )</td>
<td>( I )</td>
<td>( H )</td>
<td>hospitalization rate</td>
</tr>
<tr>
<td>( q_3 )</td>
<td>( I )</td>
<td>( R )</td>
<td>recovery rate for mild cases</td>
</tr>
<tr>
<td>( q_4 )</td>
<td>( H )</td>
<td>( R )</td>
<td>recovery rate for severe cases</td>
</tr>
<tr>
<td>( q_5 )</td>
<td>( H )</td>
<td>( D )</td>
<td>death rate</td>
</tr>
<tr>
<td>.</td>
<td>( R )</td>
<td>( S )</td>
<td>birth = death rate = ( D(t) )</td>
</tr>
</tbody>
</table>

State Transition Diagram

The state transition diagram is shown in Figure 15.4. If there is a significant reinfection rate, an edge can be added from state \( R \) to state \( S \).

![State Transition Diagram for SEIHRD Models](image)

Ordinary Differential Equations

The six time variables are interrelated through ordinary differential equations, where a dot above the variable denotes a time derivative,
\[ \dot{S}(t) = \dot{D}(t) - q_0 S(t) \]

Susceptible

\[ \dot{E}(t) = q_0 S(t) - q_1 E(t) \]

Exposed

\[ \dot{I}(t) = q_1 E(t) - (q_2 + q_3) I(t) \]

only Infected

\[ \dot{H}(t) = q_2 I(t) - (q_4 + q_5) H(t) \]

Infected and Hospitalized

\[ \dot{R}(t) = q_3 I(t) + q_4 H(t) \]

Recovered

\[ \dot{D}(t) = q_5 H(t) \]

Died

The LHS is the rate of change of the variable, while the RHS is the sum of incoming edges minus the sum of the outgoing edges. From epidemiology, the first transition rate \( q_0 \) can be further dissected. Exposure depends on members in \( S(t) \) coming in contact with an infected individual in \( I(t) \) or \( H(t) \) in terms of their fraction of the population \( N \) and is proportional to the new parameter \( \alpha \).

\[ q_0 = \alpha \frac{I(t) + H(t)}{N} \]

Replacing \( q_0 \) and \( \dot{D}(t) \) in the differential equations results in

\[ \dot{S}(t) = q_5 H(t) - \alpha \frac{I(t) + H(t)}{N} S(t) \]

Susceptible

\[ \dot{E}(t) = \alpha \frac{I(t) + H(t)}{N} S(t) - q_1 E(t) \]

Exposed

\[ \dot{I}(t) = q_1 E(t) - (q_2 + q_3) I(t) \]

only Infected

\[ \dot{H}(t) = q_2 I(t) - (q_4 + q_5) H(t) \]

Infected and Hospitalized

\[ \dot{R}(t) = q_3 I(t) + q_4 H(t) \]

Recovered

\[ \dot{D}(t) = q_5 H(t) \]

Died

In order for the six equations to be useful for forecasting, the six parameters, \( \alpha, q_1, q_2, q_3, q_4, q_5, q_6 \), must be estimated from data.

Notice that a further simplification is possible: One of the variables may be removed since all the variables sum up to \( N \).

Discretization

As was done with the golf ball example, the ODEs may be discretized. In this case, since all the differential equations are first-order, the performance of the Euler Method should be acceptable, although more advanced numerical integration methods could be applied. (Note that Newton’s Second Law is a second-order ODE (converted into two coupled first order equations) so the Verlet Method was superior to the Euler Method for the golf ball trajectory problem.) Recall that \( \Delta t \) is the time gap in the discrete-time system and time can now be used as subscript since it is discrete.
\[ S_t = S_{t-1} + \Delta t \left[ q_5 H_{t-1} - \alpha \frac{I_{t-1} + H_{t-1} S_{t-1}}{N} \right] \]  
\text{Susceptible}

\[ E_t = E_{t-1} + \Delta t \left[ \frac{I_{t-1} + H_{t-1}}{N} S_{t-1} - q_1 E_{t-1} \right] \]  
\text{Exposed}

\[ I_t = I_{t-1} + \Delta t \left[ q_1 E_{t-1} - (q_2 + q_4) I_{t-1} \right] \]  
\text{only Infected}

\[ H_t = H_{t-1} + \Delta t \left[ q_2 I_{t-1} - (q_4 + q_5) H_{t-1} \right] \]  
\text{Infected and Hospitalized}

\[ R_t = R_{t-1} + \Delta t \left[ q_3 I_{t-1} + q_4 H_{t-1} \right] \]  
\text{Recovered}

\[ D_t = D_{t-1} + \Delta t \left[ q_5 H_{t-1} \right] \]  
\text{Died}

Similar discretized system of ODEs are given in the literature for SEIR models and their extensions [15].

Formulation as an Extended Kalman Filter

The discretized system of ODEs may be now formulated as an Extended Kalman Filter model. For this model, there is no forcing/control vector, i.e., \( u_t \) is removed. Also, the observation function \( h \) is assumed to be linear. Therefore, the state and observation/measurement vector equations take the following form:

\[ x_t = f(x_{t-1}) + w_t \]
\[ y_t = Hx_t + v_t \]

State Equations

The state vector includes random variables for each of the variables in the discrete-time system of equations.

\[ x_t = [S_t, E_t, I_t, H_t, R_t, D_t] \]

For this model, the nonlinear function \( f \) is quadratic so it can be written in matrix form.

where \( x_{t-1}^t \) is the transpose of \( x_{t-1} \) (making it a row vector), \( q_{23} = q_2 + q_3 \) and \( q_{45} = q_4 + q_5 \).

Observation/Measurement Equations

The observation/measurement vector includes random variables for the observable state variable, i.e., those variables for which time-series data exists.

\[ y_t = [I^o_t, H^o_t, R^o_t, D^o_t] \]
For this model, there is a direct relationship between the state and measurement variables, so the observation/measurement vector is as follows:

\[
\begin{bmatrix}
I^o_t \\
H^o_t \\
R^o_t \\
D^o_t
\end{bmatrix} =
\begin{bmatrix}
0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 1
\end{bmatrix} x_t + v_t
\]

**Jacobian Matrices**

\( F_t \in \mathbb{R}^{6 \times 6} \) is given by the Jacobian matrix for the state transition function \( f \).

\[
F_t = \frac{\partial f}{\partial x_t}
\]

\[
F_t =
\begin{bmatrix}
1 - \alpha \frac{h_{t-1} + h_{t}}{N} \Delta t & 0 & \frac{s_{t-1}}{N} \Delta t & (q_5 + \frac{s_{t-1}}{N}) \Delta t & 0 & 0 \\
\alpha \frac{h_{t-1} + h_{t}}{N} \Delta t & 1 - q_1 \Delta t & \frac{s_{t-1}}{N} \Delta t & \frac{s_{t-1}}{N} \Delta t & 0 & 0 \\
0 & q_1 \Delta t & 1 - q_{23} \Delta t & 0 & 0 & 0 \\
0 & 0 & q_2 \Delta t & 1 - q_{45} \Delta t & 0 & 0 \\
0 & 0 & q_3 \Delta t & q_4 \Delta t & 1 & 0 \\
0 & 0 & 0 & 0 & q_5 \Delta t & 0 & 1
\end{bmatrix}
\]

Similarly, \( H_t \in \mathbb{R}^{4 \times 6} \) is given by the Jacobian matrix for the observation function \( h \).

\[
H_t = \frac{\partial h}{\partial x_t}
\]

\[
H_t =
\begin{bmatrix}
0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 1
\end{bmatrix}
\]

**Noise Covariance Matrices**

The covariance of the process noise \( Q \in \mathbb{R}^{n \times n} \) and the covariance of the measurement noise \( R \in \mathbb{R}^{m \times m} \) can be challenging to determine. These covariance matrices feed into the calculation of the Kalman Gain \( K \), so that reliance on process predictions versus measured quantities is partially based on how low their respective covariances are.

There are three basic approaches for assigning values to \( Q \) and \( R \): (1) Use knowledge of the process for \( Q \) and of the measurement devices for \( R \). The covariance of the process noise \( Q \) may be roughly determined based on inherent uncertainty in the model (unpredictably changing wind conditions) or missing model elements (e.g., the force of drag). The covariance of the measurement noise \( R \) may be roughly determined based on characteristics of the measurement device(s). (2) Use hyper-parameters and Hyper-Parameter Optimization (HPO) as discussed below. (3) Use Adaptive (Extended) Kalman Filters that adjust \( Q_t \) and \( R_t \) at each step based on calculated errors/residuals/innovations \([?]\).
As approach (1) is very problem specific and approach (3) is complicated, approach (2) is a good alternative for a quick start. This alternative uses tunable hyper-parameters $\lambda_Q$ and $\lambda_R$ as multipliers of identity matrices.

$$ Q = \lambda_Q I $$
$$ R = \lambda_R I $$

where $I_Q$ is an $n$-by-$n$ identity matrix and $I_R$ is an $m$-by-$m$ identity matrix. The hyper-parameters are problem specific, but $\lambda = 0.1$ or $1.0$ are reasonable ballparks for starting grid searches or more efficient HPO techniques.

Note that approach (1) is used in the golf ball trajectory problem given in the last section and approach (3) is explored in the exercises.

** Initialization**

Some initialization is needed before starting an Extended Kalman Filter. Due to lack of sensitivity, the initial covariance of the process error can be set to an $n$-by-$n$ identity matrix, i.e., $P_0 = I$. The initial state $x_0 = [329999999, 0, 1, 0, 0, 0]$. The time gap/increment $\Delta t = 1$ day.

### 15.5.3 Exercises

1. Use the ScalaTion COVID-19 datasets found at [https://github.com/scalation/data](https://github.com/scalation/data) to train an Extended Kalman Filter (EKF) and make four-week ahead forecasts. It is composed of data collected from multiple datasets stored at [https://github.com/CSSEGISandData/COVID-19/tree/master/csse_covid_19_data](https://github.com/CSSEGISandData/COVID-19/tree/master/csse_covid_19_data). In particular, use the dataset that contains data about COVID-19 cases, hospitalizations, recoveries, and deaths in the United States for one full year since the first confirmed case on January 22, 2020. The dataset is in a 367 row by 6 column CSV file containing the data indicated in the table below. There are 367 rows since the first row contains column headers and the year 2020 was a leap year.

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$d$</td>
<td>date: January 22, 2020 to January 21, 2021</td>
</tr>
<tr>
<td>$t$</td>
<td>time: 0 to 365</td>
</tr>
<tr>
<td>$C_t$</td>
<td>total (cumulative) number of confirmed cases by day $t$</td>
</tr>
<tr>
<td>$H_t$</td>
<td>total (cumulative) number of hospitalized individuals by day $t$</td>
</tr>
<tr>
<td>$R_t$</td>
<td>total (cumulative) number of recovered individuals by day $t$</td>
</tr>
<tr>
<td>$D_t$</td>
<td>total (cumulative) number of deaths by day $t$</td>
</tr>
<tr>
<td>$I_t$</td>
<td>deduced current number of infected individuals on day $t$ (not counting hospitalizations)</td>
</tr>
<tr>
<td>$I_t^H$</td>
<td>deduced current number of hospitalized individuals on day $t$</td>
</tr>
</tbody>
</table>

The data cover the last four states of the State Transition Diagram. The data also include the total number of confirmed cases $C_t$ and the total number of hospitalizations $H_t$ that are not directly part of the SEIHRD model, but are used to compute $I_t$ and $H_t$.  

507
\[ I_t = C_t - H_t \]
\[ H_t = H_t - R_t - D_t \]

2. **Early Stage Approximation.** In the early stage of an epidemic/pandemic, the number of Susceptible individuals changes slowly, so that \( \frac{S_{t-1}}{N} \) is nearly a constant that can be rolled into \( \alpha \). This allows the first two difference equations to be rewritten.

\[
S_t = S_{t-1} + \Delta t \left[ q_5 H_{t-1} - \alpha (I_{t-1} + H_{t-1}) \right] \quad \text{Susceptible}
\]
\[
E_t = E_{t-1} + \Delta t \left[ \alpha (I_{t-1} + H_{t-1}) - q_1 E_{t-1} \right] \quad \text{Exposed}
\]

The state equations are now linear (and the observation/measurement equations have been linear), so an ordinary Kalman Filter may be used. Use the SCALATION COVID-19 dataset for the year 2020 to train a Kalman Filter (KF) and make four-week ahead forecasts. Compare with the results of using the EKF.

3. \( S_t \) and \( E_t \) are hard to measure. Use the following identity to eliminate \( S_t \) from the model.

\[
S_t = N - E_t - I_t - H_t - R_t
\]

Use the SCALATION COVID-19 dataset for the year 2020 to train a Kalman Filter (KF) and make four-week ahead forecasts. Compare with previous results.

4. Based on the last exercise, there is only one unobserved variables \( E_t \). Use the early stage approximation to eliminate \( E_t \). Now all the state variables are observable \((n = m = 4)\). Create a new Kalman Filter where both \( F \) and \( H \) are 4-by-4 matrices. Use the SCALATION COVID-19 dataset for the year 2020 to train this new Kalman Filter (KF) and make four-week ahead forecasts. Compare with previous results.

5. When all the state variables are observable and the state transition and observation functions are linear, a Vector Auto-Regressive VAR\((p, n)\) model may be applied. Use the SCALATION COVID-19 dataset for the year 2020 to train a VAR model and make four-week ahead forecasts. In particular, create a \( \text{VAR}(1, 4) \) model and a \( \text{VAR}(2, 4) \) model as shown below,

\[
y_t = \delta + \Phi^{(0)} y_{t-2} + \Phi^{(1)} y_{t-1} + \epsilon_t
\]

where \( y_t = [I_t, H_t, R_t, D_t] \), \( p = 2 \) and \( n = 4 \), with two parameter matrices: \( \Phi_0 \in \mathbb{R}^{4 \times 4} \) and \( \Phi_1 \in \mathbb{R}^{4 \times 4} \). Compare with previous results.

6. Create an \( \text{AR}^*(1, 4) \) model and a \( \text{AR}^*(2, 4) \) model. Recall that an \( \text{AR}^*(p, n) \) model is a \( \text{VAR}(p, n) \) model where all the parameter matrices are diagonal. Compare with previous results.
7. As the state transition function $f$ or the observation function $h$ are no longer well locally approxi-
mately by linear functions or the noise is no longer well approximated by Gaussian distributions, it is
recommended to use an Unscented Kalman Filter [108]. Use the ScalaTion COVID-19 dataset
for the year 2020 to train an Unscented Kalman Filter (UKF) and make four-week ahead forecasts.
Compare with previous results.

8. As the dimensionality of the problem becomes very large, such as in Numerical Weather Prediction
(NWP), it is recommended to use an Ensemble Kalman Filter [27, 49]. Use the ScalaTion
COVID-19 dataset for the year 2020 to train an Ensemble Kalman Filter (EnKF) and make four-week ahead forecasts.
Compare with previous results.

9. There are two techniques for estimating noise covariance matrices $Q_t$ and $R_t$ for Adaptive (Extended)
Kalman Filters, one based innovations (errors before correction) the other based on residuals (errors
after correction). Read the following two papers [13, 3] and write a short essay on how the two
techniques work.

10. The discretization of the system of Ordinary Differential Equations (ODEs) for the $SEIHRD$ model
used the Euler Method. For a system of ODEs, the vector equation is of the form:

$$ \dot{y}(t) = f(t, y(t)) $$

Discretization using the explicit, first-order Euler Method gives

$$ y_t = y_{t-1} + \Delta t f(t - \Delta t, y_{t-1}) $$

Try using an explicit, second-order Runge-Kutta Method (RK2) instead

$$ \tilde{y}_t = y_{t-1} + \Delta t f(t - \Delta t, y_{t-1}) \quad \text{predicted} $$

$$ y_t = y_{t-1} + \frac{\Delta t}{2} \left[ f(t - \Delta t, y_{t-1}) + f(t, \tilde{y}_t) \right] \quad \text{corrected} $$

11. Mobility Aware $SEIHRD$ Model

This next more realistic, although more complex, model looks at regions (e.g., states, counties) and
considers mobility/traffic between regions.

**FIX**

**Mobility Aware Differential Equations**

(a) The number of Susceptible individuals in region $k$ at time $t$ is denoted by $S_{kt}$.
(b) The number of Exposed individuals in region $k$ at time $t$ is denoted by $E_{kt}$.
(c) The number of Infected and not Hospitalized individuals in region $k$ at time $t$ is denoted by $I_{kt}$.
(d) The number of Infected and Hospitalized individuals in region $k$ at time $t$ is denoted by $H_{kt}$.
(e) The number of Recovered individuals in region $k$ at time $t$ is denoted by $R_{kt}$.
The number of Deaths in region \( k \) at time \( t \) is denoted by \( D_{kt} \).

Since infections in one region can cause infections in other regions, use of mobility data can improve forecasts. A simple approach is to add a connectivity/mobility matrix that indicates the amount of travel between regions.

\[
C_{jk} = \text{number of individuals traveling from region } j \text{ to region } k \text{ in one time unit} \quad (15.59)
\]

The diagonal in the matrix is minus the number of individuals leaving the region. The issue is to determine, for each state the number of individuals entering region \( k \) in one time unit. For example, for susceptibility the travel adjusted number is

\[
\tau(S_{kt}) = S_{kt} + \sum_{j=0}^{l-1} S_{jt} C_{jk} \quad (15.60)
\]

Therefore, the six space-time variables are now interrelated through a new set of ordinary differential equations,

\[
\begin{align*}
\dot{S}_{kt} & = q_5 \tau(H_{kt}) - \alpha \frac{\tau(I_{kt}) + \tau(H_{kt})}{n_k} \tau(S_{kt}) \quad \text{Susceptible} \\
\dot{E}_{kt} & = \frac{\alpha \tau(I_{kt}) + \tau(H_{kt})}{n_k} \tau(S_{kt}) - q_1 \tau(E_{kt}) \quad \text{Exposed} \\
\dot{I}_{kt} & = q_1 \tau(E_{kt}) - (q_2 + q_3) \tau(I_{kt}) \quad \text{only Infected} \\
\dot{I}_H^{kt} & = q_2 \tau(I_{kt}) - (q_4 + q_5) \tau(H_{kt}) \quad \text{Infected and Hospitalized} \\
\dot{R}_{kt} & = q_3 \tau(I_{kt}) + q_4 \tau(H_{kt}) \quad \text{Recovered} \\
\dot{D}_{kt} & = q_5 \tau(H_{kt}) \quad \text{Died}
\end{align*}
\]

Parameter Estimation

A common way to estimate the parameters is to use Maximum Likelihood Estimation \[57\]. The likelihood function is ...

Basic Difference Equations Approximation

The basic six differential equation can be approximated using six difference equations, where for example \( \Delta S_{kt} = S_{kt} - S_{k,t-1} \). One time unit will correspond to one day.
$$\Delta S_{kt} = q_5 H_{k,t-1} - \alpha \frac{I_{k,t-1} + H_{k,t-1}}{n_k} S_{k,t-1}$$ \quad \text{Susceptible}

$$\Delta E_{kt} = \alpha \frac{I_{k,t-1} + H_{k,t-1}}{n_k} S_{k,t-1} - q_1 E_{k,t-1}$$ \quad \text{Exposed}

$$\Delta I_{kt} = q_1 E_{k,t-1} - (q_2 + q_3) I_{k,t-1}$$ \quad \text{only Infected}

$$\Delta H_{kt} = q_2 I_{k,t-1} - (q_4 + q_5) H_{k,t-1}$$ \quad \text{Infected and Hospitalized}

$$\Delta R_{kt} = q_3 I_{k,t-1} + q_4 H_{kt}$$ \quad \text{Recovered}

$$\Delta D_{kt} = q_5 H_{k,t-1}$$ \quad \text{Died}

Again, the six parameters to estimate are $\alpha, q_1, q_2, q_3, q_4, q_5$, and $q_6$. Each of the fifty states in the United States may be solve separately or pooled together.

Note that the approximation above only uses first lags, utilizing more lags may lead to better results.

The dataset will consist of three time-series, $I_{kt}$ infected cases, $H_{kt}$ hospitalizations, and $D_{kt}$ deaths.

**Mobility Aware Difference Equations Approximation**

TBD.
15.6 ODE Parameter Estimation

\[ y = x(t) + \epsilon \]

\[ \frac{dx(t)}{dt} = f(x(t); b) \]

Nonlinear Least Squares (NLS)
Least Squares Approximation (LSA)
Chapter 16

Event-Oriented Models

The simulation modeling techniques discussed so far center around specifying a set of equations. The structure or operation rules of actual systems may require logic that is hard to express in this fashion. A very flexible way of expressing such logic is to focus on what happens that may affect the state of the system. This is the approach that is followed by Event-Oriented Models. These are also called Event Scheduling Models that suggest how the simulation engine would need to work. The engine needs to provide a means for creating, scheduling, and processing events over time.

Before discussing event-oriented models in more detail, several simulation modeling paradigms will be highlighted.

16.1 A Taxonomy/Ontology for Simulation Modeling

The most recent version of the Discrete-event Modeling Ontology (DeMO) lists five simulation modeling paradigms or world-views for simulation (see the bullet items below). These paradigms are briefly discussed below and explained in detail in [96].

- **State-Oriented Models.** State-oriented models focus states and state transitions. State-oriented models include Markov Chains and their generalizations such as Generalized Semi-Markov Processes (GSMPs). A GSMP can be defined using three functions,

  - an activation function \( \{ e \} = a(x(t)) \),
  - a clock function \( t' = c(x(t), e) \),
  - a state-transition function \( x(t') = d(x(t), e) \).

In simulation, advancing to the current state \( x(t) \) causes a set of events \( \{ e \} \) to be activated according to the activation function \( a \). Events occur instantaneously and may affect both the clock and transition functions. The clock function \( c \) determines how time advances from \( t \) to \( t' \) and the state-transition function determines the next state \( x(t') \). One can also tie in the input and output vectors. The input vector \( u \) is used to initialize a state at some start time \( t_0 \) and the response vector \( y \) can be a function of the state sampled at multiple times during the execution of the simulation model.
• **Event-Oriented Models.** State-oriented models may become unwieldy when the state-space becomes very large. One option is to focus on state changes that occur by processing events in time order. An event may indicate what other events it causes as well as how it may change the state. Essentially, the activation and state transition functions are divided into several simpler functions, one for each event $e$:

$$
\begin{align*}
\{e\} &= a_e(x(t)), \\
x(t') &= d_e(x(t)).
\end{align*}
$$

Logic for each event type implements the $a_e$, what other event to trigger and $d_e$ how to change (or transition) the state. Time advance is simplified to just setting the time $t'$ to the time of the most imminent event on a Future Event List (activated events are placed on this list in time order).

• **Process-Oriented Models.** One of the motivations for process-oriented models is that event-oriented models provide a fragmented view of the system or phenomena. As combinations of low-level events determine behavior, it may be difficult to see the big picture or have an intuitive feel for the behavior. Process-oriented or process-interaction models aggregate events by putting them together to form a process. An example of a process is a customer in a store. As the simulated customer (as an active entity) carries out behavior it will conditionally execute multiple events over time. A simulation then consists of many simultaneously active entities and may be implemented using coroutines (or threads/actors as a more heavyweight alternative). Typically, there is one coroutine for each active entity. The overall state of a simulation then includes a combination of the states of each active entity (each coroutine/thread has its own stack). The global shared state also includes a variety of resources types.

• **Activity-Oriented Models.** There are many types of activity-oriented models including Petri-Nets and Activity-Cycle Diagrams. The main characteristics of such models is a focus on the notion of activity. An activity (e.g., customer checkout) corresponds to a distinct action that occurs over time and includes a start event and an end event. Activities may be started because time advances to its start time or a triggering condition becomes true. Activities typically involve one or more entities. State information is stored in activities, entities and the global shared state.

• **System Dynamics Models.** System dynamics models have been added to DeMO, since hybrid models that combine continuous and discrete aspects are becoming more popular. One may consider modeling the flight of a golf ball once struck by a golf club. Let the response vector $y = [y_0 \ y_1]$ where $y_0$ indicates the horizontal distance traveled, while $y_1$ indicates the vertical height of the ball. Future positions of $y$ depend on the current position and time $t$. Using Newton’s Second Law of Motion, $y$ can be estimated by solving a system of Ordinary Differential Equations (ODEs) such as

$$\dot{y} = f(y, t), \ y(0) = y_0.$$

The Newtons2nd object uses the Dormand-Prince ODE solver to solve this problem. More accurate models for estimating how far a golf ball will carry when struck by a driver can be developed based on inputs/factors such as club head speed, spin rate, smash factor, launch angle, dimple patterns, ball compression characteristics, etc. There have been numerous studies of this problem, including [14].
In addition to these main modeling paradigms, ScalaTion support a simpler approach called **Tableau-Oriented Models** that can be thought of as an automated analog to *Spreadsheet Simulation*.

One may also classify models as supporting **Object-Oriented Simulation** where the advantages of object-oriented programming are utilized. For example, some forms of process-oriented simulation (e.g., GPSS) require each process to strictly follow a sequence of blocks. This makes it easier to create models (and they can be fully specified in a GUI), but reduces the flexibility with which simulation can be built (see [SI] for details). In ScalaTion, all of the simulation modeling techniques take advantage of object-orientation, especially the process-interaction models. The importance of each actor running concurrently is the reason it falls in the process-oriented category. In the sense that it has active entities that are objects and concurrent, puts it in the lineage of Simula-67.
16.2 List Processing

Event-Oriented Models will require events and entities to be maintained in various types of lists or queues.

16.2.1 FCFS Queue

The most common type of queue is a First-Come, First-Serve (FCFS) Queue also known as a First-In, First-Out (FIFO) Queue. As a data structure, they may be implemented as an array or linked list. Efficient access to the front (head) and back (tail) of the queue are needed, as well as efficient methods for adding a item to the back of the queue (enqueue) and removing an item from the front of the queue (dequeue).

The Queue class in the scala.collection.mutable package provides these capabilities. It extends the ArrayDeque class from the same package. The ArrayDeque class is implemented using a resizable circular array that allows efficient operations to both ends (front and back) of the queue. Deque stands for double ended queue, see Figure 16.1.

![Figure 16.1: Efficient Access to the Front and Back of ArrayDeque](image)

The following methods are provided by ArrayDeque for adding and removing items from the ends of a queue.

- **prepend** (alias +:) - add an item to the front of the queue.
- **addOne** (alias +=) - add an item to the back of the queue.
- **removeHead** - remove the item at the front of the queue.
- **removeLast** - remove the item at the back of the queue.

The Queue class implements its enqueue (with alias +=) and its dequeue methods as follows:

```scala
def enqueue (elem: A): this.type = this += elem // join the back of the line (addOne)

def dequeue (): A = removeHead () // remove from front
```

Note, specifying this.type means q.enqueue (elem) returns the type of q which could be Queue or a subclass of Queue, e.g., MyQueue extends Queue.

See [http://scalada.blogspot.com/2008/02/thistype-for-chaining-method-calls.html](http://scalada.blogspot.com/2008/02/thistype-for-chaining-method-calls.html) for an example.
16.2.2 LCFS Queue

Another type of queue is a Last-Come, First-Serve (LCFS) Queue also known as a Last-In, First-Out (LIFO) Queue. As a data structure, they may also be implemented as an array or linked list. Efficient access to the front (top) of the queue is needed, as well as efficient methods for adding a item to the front of the queue (push) and removing an item from the front of the queue (pop).

The Stack class in the `scala.collection.mutable` package provides these capabilities. It also extends the ArrayDeque class from the same package.

The Stack class implements its push and its pop methods as follows:

```scala
def push (elem: A): this.type = prepend (elem) // join at the front
def pop () : A = removeHead () // remove from front
```

16.2.3 Priority Queue

Priority Queues are essential for simulation engines as events need to be efficiently placed in a data structure in time order. The time being when the event (e.g., next arrival or service completion) is to occur. How far in the future the event is to occur determines its placement in the priority queue. In this way events are processed in time order. The most imminent event is the one at the front of the queue. When it is removed, the simulation clock is advanced to the time of this event.

ScalaTion uses the PriorityQueue class in the `scala.collection.mutable` package for scheduling events. This class supports adding and removing items in logarithmic time by having an interval heap data structure. A heap may be implemented using a resizable array for which the heap order is maintained, i.e., `heap(i) <= heap(2*i)` and `heap(i) <= heap(2*i+1)`. New events are added to the `eventList` as follows:

```scala
eventList += anEvent
```

Note, `+=` is alias for the `addOne` method. The most imminent event is removed from the `eventList` as follows:

```scala
nextEvent = eventList.dequeue ()
```

16.2.4 Time Advance Mechanism

ScalaTion supports two types of event-oriented simulation modeling paradigms: Event Scheduling and its extension, called Event Graphs. For both paradigms, the state of the system only changes at discrete event times with the changes specified via event logic. Time is advanced (jumps forward) to the event time of the next event, so unlike discrete-time models, discrete-event models have non-constant time increments.

(1) Event Scheduling models encode the event logic in classes that extend the Event class. The event logic is written in each such class’ customized `occur` method. A scheduler within the model will execute the events in time order. A time-ordered priority queue is used to hold the future events and is often referred to as a Future Event List (F.E.L.), see Figure 16.2.
(2) Event Graph models capture the event logic related to triggering other events in causal links. In this way, Event Graph models are more declarative (less procedural) than Event Scheduling models. They also facilitate a graphical representation and animation. They can also serve as a design diagram for event scheduling as they depict the relationships between events.
16.3 Event Scheduling

A simple, yet practical way to develop a simulation engine to support discrete-event simulation is to implement event-scheduling. This involves creating the following three classes: Event, Entity and Model. An Event is defined as an instantaneous occurrence that can trigger other events and/or change the state of the simulation. An Entity, such as a customer in a bank, flows through the simulation. The Model serves as a container/controller for the whole simulation and carries out scheduling of event in time order. The centerpiece class for event scheduling and the one model developers will be most concerned with is the Event class.

16.3.1 Event Class

The abstract Event class provides a framework for defining types of simulation events. A subclass extending Event needs to be created for each type of event.

A subclass (e.g., Arrival) of Event must provide event-logic in the implementation of its occur method. The Event class also provides methods for comparing activation times (actTimes for events and converting an event to its string representation. Note: unique identification is mixed in via the Identifiable trait.

Class Methods:

- @param entity the entity involved in this event
- @param director the controller/scheduler that this event is a part of
- @param delay the time delay before this event’s occurrence
- @param stat the object for collecting statistics about delay times
- @param proto the prototype (serves as node in animation) for this event

abstract class Event (val entity: Entity, director: Model, delay: Double = 0.0, stat: Statistic = null, val proto: EventNode = null)
extends Identifiable with Ordered [Event]:

- def compare (ev: Event): Int = ev.actTime compare actTime
- def cancel (): Unit = { _live = false }
- def live: Boolean = _live
- def occur (): Unit
  override def toString: String = entity.toString + "\t" + me

An Event must be defined inside a Model referenced by director and have an Entity that is involved in this event.

An important field in the Event class is actTime, which indicates the activation/occurrence time for the event.

The methods in this class perform the following functions:

- The compare method compares the actTime of two events, thus allowing events to be placed in time order in the F.E.L. (eventList). Since Scala’s PriorityQueue class is organized as Highest Priority First (HPF), the logic of the above compare method is reversed.

519
- The `cancel` method allows scheduled events to be cancelled by marking them as not live.
- The `live` method returns whether an event has been cancelled.
- The `occur` method must be implemented in each subclass and it captures the event logic for a particular type of event (e.g., `Arrival`). The method may (1) schedule other events and (2) specify state changes.
- The `toString` method converts internal information about an event into a string.

A frequently used method from the `Model` class is `schedule`. It will place the `event` on the `eventList` in time order. The `eventList` is managed by the `Model`.

```scala
def schedule (event: Event): Unit =
```

Much of what is needed to develop simply event scheduling models has now been covered.
16.3.2 Example: Bank Model

To create a simple bank simulation model, one could use the classes defined in the event-scheduling engine to create

- two subclasses of Event, called Arrival and Departure, and
- one subclass of Model, called BankModel.

The complete code for this Bank simulation example may be found in the scalation.simulation.event package in Bank.scala.

The event logic is coded in the occur method which in general triggers future events and updates the current state. It indicates what happens when the event occurs.

Arrival Class

The Arrival case class extends the abstract Event class. The class constructor takes two parameters: The entity involved in the event and the time delay (how far in the future this event is to occur). These parameters are passed into the base class (Event) along with the Model director (this) and a Statistic object for recording statistics on inter-arrival times t_ia_stat.

```scala
@param customer the entity that arrives, in this case a bank customer
@param delay the time delay for this event’s occurrence

case class Arrival (customer: Entity, delay: Double)
  extends Event (customer, this, delay, t_ia_stat):

  def occur (): Unit = ???

end Arrival
```

Before implementing the logic of the occur methods, it is useful to create an Event Graph design diagram (even if Event Graphs are not used for the implementation). Each type of event is depicted as a node and the directed edges indicate event causality. For the Bank Model, two event types will suffice (Arrival and Departure). There are three causal links (directed edges): an arrival event triggers the next arrival and it may trigger a service completion/departure event, and a departure event may trigger the next service completion event. Typically, the directed edges have conditions on them (i.e., the event is only triggered when the condition is true). The events are also triggered to occur in the future based on the transition/delay times associated with edges. Figure 16.3 depicts the event graph to the Bank Model (the delays time are not placed in the graph, but in the caption to avoid clutter).

The first condition \( n_{Arr} < n_{Stop} - 1 \) will be true when the simulation stopping rule becomes true, the second condition \( n_{In} = 0 \) being true allows the arriving customer to go directly into service as the server is not busy, and the third condition allows a customer in the queue to begin service, by scheduling the end the service activity.
The `occur` method will schedule the next arrival event (up to the limit), check to see if the teller is busy. If so, it will place itself in the Wait Queue (W.Q.), otherwise it schedules its own departure to correspond to its service completion time. Finally, it adjusts the state by incrementing both the number of arrivals \(n_{Arr}\) and the number in the system \(n_{In}\).

```java
def occur (): Unit =
    if nArr < nStop-1 then
        val toArrive = Entity (iArrivalRV.gen, serviceRV.gen, BankModel.this)
        schedule (Arrival (toArrive, toArrive.iArrivalT))
    end if
    if nIn == 0 then
        schedule (Departure (customer, customer.serviceT))
    else
        waitQueue.enqueue (customer) // collects time in Queue statistics
    end if
    nArr += 1 // update the current state
    nIn += 1
end occur
```

Suppose \(n_{Stop} = 3\) and that three arrival events, \(A_0, A_1, A_2\), with corresponding customers \(C_0, C_1, C_2\) occur before any departure events \(D_0, D_1, D_2\). The table below shows the situation at the end of each `occur` method.

<table>
<thead>
<tr>
<th>Event</th>
<th>F.E.L.</th>
<th>W.Q.</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>([A_0])</td>
<td>[]</td>
<td>([0, 0])</td>
</tr>
<tr>
<td>(A_0).occur</td>
<td>([A_1, D_0])</td>
<td>[]</td>
<td>([1, 1])</td>
</tr>
<tr>
<td>(A_1).occur</td>
<td>([A_2, D_0])</td>
<td>([C_1])</td>
<td>([2, 2])</td>
</tr>
<tr>
<td>(A_2).occur</td>
<td>([D_0])</td>
<td>([C_1, C_2])</td>
<td>([3, 3])</td>
</tr>
<tr>
<td>(D_0).occur</td>
<td>([D_1])</td>
<td>([C_2])</td>
<td>([3, 2])</td>
</tr>
<tr>
<td>(D_1).occur</td>
<td>([D_2])</td>
<td>[]</td>
<td>([3, 1])</td>
</tr>
<tr>
<td>(D_2).occur</td>
<td>[]</td>
<td>[]</td>
<td>([3, 0])</td>
</tr>
</tbody>
</table>
Departure Class

For the Departure class, the occur method will check to see if there is another customer waiting in the queue and if so, schedule that customer's departure. It will then signal its own departure by updating the state; in this case decrementing nIn and incrementing nOut.

```scala
@param customer the entity that departs, in this case a bank customer
@param delay the time delay for this event's occurrence

case class Departure (customer: Entity, delay: Double)
    extends Event (customer, this, delay, t_s_stat):

    def occur (): Unit =
        leave (customer) // collects time in sYstem statistics
        if ! waitQueue.isEmpty then // nIn > 1
            val nextService = waitQueue.dequeue () // first customer in queue
            schedule (Departure (nextService, nextService.serviceT))
        end if
        nIn -= 1 // update the current state
    end occur

end Departure
```

BankModel Class

The BankModel class defines a simple Event-Scheduling model of a Bank where service is provided by one teller and models an M/M/1 queue. The parameters to the constructor are the name of the simulation model, the number of independent replications to run (see the Simulation Output Analysis Chapter for details), the number of entities to create before stopping the simulation, and the base random number stream to use. Each random variate should use a difference stream for independence. The models need to initialize the model constants, create the random variates and state variables, specify the event logic in subclasses for Event, in this case, the Arrival and Departure events defined in the previous subsections, and finally, start the simulation after scheduling the first priming event. Once the simulation stops, reports and summarizes may be output.

```scala
@param name the name of the simulation model
@param reps the number of independent replications to run
@param nStop the number arrivals before stopping
@param stream the base random number stream (0 to 999)

class BankModel (name: String = "Bank", reps: Int = 1, nStop: Int = 100, stream: Int = 0)
    extends Model (name, reps):

    // Initialize Model Constants

    val lambda = 6.0 // customer arrival rate (per hr)
    val mu = 7.5 // customer service rate (per hr)
```

523
// Create Random Variables (RVs)

val iArrivalRV = Exponential (HOUR / lambda, stream)
val serviceRV = Exponential (HOUR / mu, (stream + 1) % N_STREAMS)

// Create State Variables

var nArr = 0.0 // number of customers that have arrived
var nIn = 0.0 // number of customers in the bank

val t_ia_stat = new Statistic ("t_ia") // time between Arrivals statistics
val t_s_stat = new Statistic ("t_s") // time in Service statistics
val waitQueue = WaitQueue (this) // waiting queue that collects stats
addStats (t_ia_stat, t_s_stat)

// Specify Logic for each Type of Simulation Event

case class Departure ...

case class Arrival ...

// Start the simulation after scheduling the first priming event

val firstArrival = Entity (iArrivalRV.gen, serviceRV.gen, this)
schedule (Arrival (firstArrival, firstArrival.iArrivalT)) // first priming event
simulate () // start simulating

report ("nArr", nArr), ("nIn", nIn)
reportStats ()
waitQueue.summary (nStop)

end BankModel

Note, to aid with debugging, it may be useful to add the following state variable.

var nOut = 0.0 // number of customers that departed

The three code segments may now be merged together and compiled. The following imports are required: scalation.mathstat.Statistic, scalation.random.Exponential, and scalation.random.RandomSeeds.N_STREAMS. Outside of SCALATION for example in my.scalation, the following import is also required: scalation.simulation.event.

Model Execution

The BankModel can be invoked by calling a main function, such as runBank.

@main def runBank (): Unit = new BankModel ()

This function may be executed in sbt using runMain

> runMain scalation.simulation.event.example_1.runBank
Defining Simulation Scenarios

Some of the model preamble (before the Event subclasses) may be moved out of the BankModel to allow trying multiple combinations of model parameters/ constants.

```scala
@main def runBank2 (): Unit =
  val mu = 7.5
  for lambda <- 5 to 8 do new BankModel2 (lambda, mu)
end runBank2

class BankModel2 (lambda: Double = 6.0, mu: Double = 7.5,
agency String = "Bank", reps: Int = 1, nStop: Int = 100, stream: Int = 0)
  extends Model (name, reps):
```

The scenario specification can also be extended to include the random variates as well, allowing different arrival and service distributions to be readily tested.

Statistic Class

In order to collect statistical information, the constructor of the Event class calls the tally method from the Statistic class in the scalation.mathstat package to obtain statistics on

- the time in queue \( T_q \),
- the time in service \( T_s \), and
- the time in system \( T_y \).

```scala
if stat != null then stat.tally (delay)
```

Class Methods:

```scala
@def set (n_: Int, sum_: Double, sumAb_: Double, sumSq_: Double, 
  minX_: Double, maxX_: Double): Unit =
  def reset (): Unit =
  def tally (x: Double): Unit =
  inline def num: Int = n
  inline def nd: Double = n.toDouble
  inline def min: Double = if n == 0 then 0.0 else minX
  inline def max: Double = maxX
  def mean: Double = if n == 0 then 0.0 else sum / nd
  def variance: Double =
  def stdev: Double = sqrt (variance)
  def ms: Double = sumSq / nd
```
Monitor Class

The Monitor class is used to trace the key actions in the execution of a model. This class works for both event oriented and process oriented model. Information collected by calling the trace method is saved in log file.

Class Methods:

```scala
@param project the project to be monitored
case class Monitor (project: String = "simulation"):

  def toggle (): Unit = ew.toggle ()
  def traceOff (): Unit = tracing = false
  def traceOn (): Unit = tracing = true
  def trace (who: Identifiable, what: String, whom: Identifiable, when: Double): Unit =
  def finish (): Unit = ew.finish ()
```

Model developers may call trace anywhere in their code. The Model class calls trace multiple times,

```scala
private [event] val log = Monitor ("simulation")
log.trace (this, "starts", this, _clock)
log.trace (this, s"executes ${nextEvent.me} on ${nextEnt.eid}" , nextEnt, _clock)
log.trace (this, "terminates", this, _clock)
```

The traces are are written into a file located in for example log/simulation directory. If this directory does not exist, it will need to be made for the model to run. Both scalation and my_scalation have this directory.
16.3.3 Example: Call Center Model

A simpler model is one for which there is no waiting queue. If a server is free/idle, service may begin, otherwise, it can be attempted later. A simple call center can be modeled in this way. The `CallCenter.scala` file contains event logic for the case of a call center with a staff of one. The Event Graph design diagram is similar the one for the Bank Model. The major difference is that upon departure, as there is no queue, the ending call cannot trigger the beginning of the next completed call, i.e., the `Departure` event does not trigger any other events, it just updates the state.

Figure 16.4: Call Center Event Graph: with edge delay times of $t_{ia}$ and $t_s$, going left to right

Arrival Class

The `Arrival` class replaces waiting in a queue with incrementing a counter of the number of lost calls (`nLost`).

```scala
@constructor("Arrival")
@constructor("Arrival")
case class Arrival(call: Entity, delay: Double) extends Event(call, this, delay, t_a_stat):
  def occur(): Unit =
    if nArr < nStop-1 then
      val toArrive = Entity(iArrivalRV.gen, serviceRV.gen, CallCenterModel.this)
      schedule(Arrival(toArrive, toArrive.iArrivalT))
    end if
    if nIn == 0 then
      schedule(Departure(call, call.serviceT))
    end if
    nArr += 1 // update the current state
    if nIn == 1 then nLost += 1 else nIn = 1
  end occur
end Arrival
```
**Departure Class**

The logic for the `Departure` class simply records information and updates counters.

```scala
@param call the entity that departs, in this case a call
@param delay the time delay for this event’s occurrence

case class Departure (call: Entity, delay: Double)
  extends Event (call, this, delay, t_s_stat):

  def occur (): Unit =
    leave (call) // collects time in sYstem statistics
    nIn = 0 // update the current state
    nOut += 1

end Departure
```

The remaining four of five classes used for creating simulation models following the Event Scheduling paradigm are discussed in the next four subsections.

### 16.3.4 Entity Class

Conceptually, entities are the dynamic objects that populate a simulation model, e.g., customers in a bank simulation or calls in a call center simulation. An instance of the `Entity` class represents a single simulation entity for event oriented simulation. For each instance, it maintains information about that entity’s arrival time and next service time.

**Class Methods:**

```scala
@param iArrivalT the time from the last arrival
@param serviceT the amount of time required for the entity’s next service
@param director the controller/scheduler that this event is a part of

case class Entity (val iArrivalT: Double, var serviceT: Double, director: Model)
  extends Identifiable:

  override def toString = "Entity-" + eid
```

Important fields in the `Entity` class are the entity id `eid`, the time at which the entity arrived `arrivalT = director.clock + iArrivalT`, and the time it begins waiting `startWait`, if applicable.

In the `BankModel`, a new entity (called `toArrive`) for a future arrival event is created before being passed to the event.

```scala
val toArrive = Entity (iArrivalRV.gen, serviceRV.gen, BankModel.this)
schedule (Arrival (toArrive, toArrive.iArrivalT))
```
16.3.5 WaitQueue Class

When entities are unable to begin service immediately, they are often placed in a wait queue. The `WaitQueue` class provides a First-Come, First-Served (FCFS) queue and is implemented by extending Scala’s `Queue` class. An entity is added to the back of the queue using the `enqueue` method and is removed from the front of the queue using the `dequeue` method. By default it has infinite capacity, but may be restricted by passing a value in the `cap` parameter. In this case, entities are barred from entering the queue when the queue is full. The number of times it is called is returned by the `barred` method.

```scala
override def enqueue (ent: Entity): WaitQueue.this.type =
  ent.startWait = director.clock
  if length <= cap then super.enqueue (ent) else _barred += 1
  this
end enqueue
```

The `dequeue` method collects both sample statistics (for $T_q$) and time-persistent statistics (for $L_q$) on queue occupancy, see the exercises.

```scala
override def dequeue (): Entity =
  val ent = super.dequeue ()
  val timeInQ = director.clock - ent.startWait
  waitTimes += timeInQ
  director.log.trace (director, s"records $timeInQ wait for ${ent.eid}", ent, director.clock)
  t_q_stat.tally (timeInQ)
  l_q_stat.accum (length + 1, director.clock)
  ent
end dequeue
```

The `summary` method returns statistics about waiting times in the queue.

```scala
override def summary (numEntities: Int): Unit =
```

Class Methods:

```scala
@param director the controller/scheduler that this event is a part of
@param ext the extension to distinguish the wait queues
@param cap the capacity of the queue (defaults to unbounded)

case class WaitQueue (director: Model, ext: String = "", cap: Int = Int.MaxValue)
  extends Queue [Entity]:
    def barred: Int = _barred
    def isFull: Boolean = length >= cap
    override def enqueue (ent: Entity): WaitQueue.this.type =
    override def dequeue (): Entity =
    def summary (numEntities: Int): Unit =
```

529
16.3.6 WaitQueue_LCFS Class

Again, entities in the model that are unable to begin service immediately are often placed in a wait queue. The WaitQueue_LCFS class provides a Last-Come, First-Served (LCFS) queue and is implemented by extending Scala’s Stack class. An entity is added to the front (top) of the queue using the enqueue method and is removed from the front (top) of the queue using the dequeue method. By default it has infinite capacity, but may be restricted by passing a value in the cap parameter. In this case, entities are barred from entering the queue, and need to stay where they are, go somewhere else or be lost to the simulation. The number of times entities are barred is returned by the barred method. The summary method returns statistics about waiting times in the LCFS queue.

Class Methods:

```scala
@param director the controller/scheduler that this event is a part of
@param ext the extension to distinguish the wait queues
@param cap the capacity of the queue (defaults to unbounded)

case class WaitQueue_LCFS (director: Model, ext: String = "", cap: Int = Int.MaxValue)
    extends Queue [Stack]:

    def barred: Int = _barred
    def isFull: Boolean = length >= cap
    def enqueue (ent: Entity): WaitQueue.this.type =
    def dequeue (): Entity =
    def summary (numEntities: Int): Unit =
```

See the exercises for how to add additional types of wait queues.

16.3.7 Model Class

As shown in the examples, the foundation class for event-oriented simulation models is the Model class. An application model will extend this class and create an instance called director. The director schedules events and implements the time advance mechanism for event-oriented simulation models. Two important methods in the Model class are the schedule and simulate methods. Scheduled events are placed in the Future Event List (F.E.L.) in time order. In SCA La Tion, the F.E.L. is called the eventList and is implemented using Scala’s priority queue.

- The schedule method is used to add events to the eventList so they may be processed in time order. To preempt a scheduled event, the cancel method may be called.

- The simulate method repeatedly removes and processes events in the eventList. The simulate method will cause the main simulation loop to execute, which will remove the most imminent event from the eventList and invoke its occur method.

The main loop in the Model class is the following while with the line controlling animation removed.
while simulating &amp; ! eventList.isEmpty do
    nextEvent = eventList.dequeue()
    if nextEvent.live then
        _clock = nextEvent.actTime
        nextEnt = nextEvent.entity
        log.trace(this, s"executes ${nextEvent.me} on ${nextEnt.eid}", nextEnt, _clock)
        debug("simulate", s"$nextEvent \t" + "%g".format(_clock))
        nextEvent.occur()
    end if
end while

The simulation will continue until a stopping rule evaluates to true (either the simulating flag becomes false or the eventList becomes empty). The nextEvent is removed from the front of the priority queue and checked to make sure it is still live (not cancelled). If it is live, the director’s simulation clock is set to the this event’s activation time actTime. For tracing purposes, this event’s associated entity is referenced. Then this event’s occur method is called that carries out the event logic based on the type of event it is.

Methods to getStatistics and report statistical results are also provided.

Class Methods:

@!param name the name of the model
@!param animation whether to animate the model (only for Event Graphs)

class Model (name: String, animating: Boolean = false)
    extends Modelable with Identifiable:

    def addStats (stat: Statistic*): Unit = for st <- stat do stats += st
    def schedule (event: Event): Unit =
    def cancel (event: Event): Unit = event.cancel ()
    def leave (entity: Entity): Unit = t_y_stat.tally (clock - entity.arrivalT)
    def simulate (startTime: Double = 0.0): Unit =
    def report (vars: (String, Double)*): Unit =
    def reportStats (): Unit =
    def getStatistics: ListBuffer [Statistic] = stats
    def animate (who: Identifiable, what: CommandType, color: Color, shape: Shape, at: Array [Double]): Unit =
    def animate (who: Identifiable, what: CommandType, color: Color, shape: Shape, from: Event, to: Event, at: Array [Double] = Array ())

The animate methods are used with Event Graphs (see the next section).

As a starting point for developing simulation models following the event scheduling paradigm, a model developer may copy the Ex_Template.scala file in the scalation.simualation.event package.

### 16.3.8 Example: Machine Shop Model

Before ending this section, consider the following more complex model: Parts arrive at a two-stage machine shop with an arrival rate of \( \lambda = 10 \text{ hr}^{-1} \). The first machine reshapes the part, while the second machine
polishes the part. The service rate for the first machine is $\mu_1 = 12 \text{ hr}^{-1}$ and is $\mu_2 = 15 \text{ hr}^{-1}$ for the second machine. Both machines have space to store three parts waiting to be machined. When there is a backup of parts, flow from other departments must continue, so these parts are removed to be sold for scrap. The company wishes to conduct a simulation study to see which policy is better.

- **Policy One**: Block machine 1 when machine 2’s queue is full.
- **Policy Two**: Do not block machine 1 when machine 2’s queue is full, rather send the partially finished part to scrap.

The cost of a raw part is 100 dollars, the value of partially finished part is 50 dollars and value of finished part is 200 dollars. The operational cost of machine is 60 dollars an hour and is 30 dollars for machine 2.

The management has argued that machine 1 should be blocked since 50 dollars are lost whenever a partially finished part is sold for scrap, while if it a raw part it can be resold for 100 dollars. Others say stopping/forced idling of machine 1 is costly.

Figure 17.1 shows the flow of parts through the machine shop. Use simulation to estimate performance characteristics and determine the better policy. How sensitive is the decision to the relative service rates of the two machines.

![Machine Shop Schematic Diagram](image)

**Figure 16.5**: Machine Shop Schematic Diagram: Each Queue Has Capacity Three

The Machine.scala file in the event.example_1 package contains a partial implementation of the machine shop model. The Event Graph design diagram now requires three event types: Arrival, FinishMachine1, and FinishMachine2. The Event Graph is depicted in Figure 16.6.

![Machine Shop Event Graph](image)

**Figure 16.6**: Machine Shop Event Graph
The state variable are the following:

- the number of part arrivals at machine 1, $n_{Arr}$,
- the current number of parts at machine 1 (queue + service), $n_{In1}$,
- the current number of parts at machine 2, $n_{In2}$,
- the number of completed parts, $n_{Out}$,
- the number of scrapped raw parts, $n_{Scrap1}$
- the number of scrapped partially finished parts, $n_{Scrap2}$

**Blocking.** Note that one policy may require machine station 1 to be blocked due to no room in machine station 2. The blocking occurs when machine 1 finishes a part and it is ready to move onto machine station 2, but cannot. This part will be stuck at machine 1 and machine 1’s operator will be idle, until machine 2 completes the part it is working on. An additional edge should be added to the Event Graph to indicate this causal connection. As the blocked part is not in a queue and not in the F.E.L., it needs to be held somewhere, e.g., in model variable called $heldAtMachine1$. 


16.4 Event Graphs

Event Graphs operate in a fashion similar to Event Scheduling. Originally proposed as a graphical conceptual modeling technique (Schruben, 1983) for designing event oriented simulation models, modern programming languages now permit more direct support for this style of simulation modeling.

In ScalaTion, the simulation engine for Event Graphs consists of the following seven classes: The first five classes are shared with Event Scheduling.

1. An **Entity**, such as a customer in a bank, flows through the simulation.

2. An **Event** represents an instantaneous occurrence that affects the simulation.

3. An **WaitQueue** allows entities to wait for service in a FCFS Queue.

4. An **WaitQueue LCFS** allows entities to wait for service in an LCFS Queue.

5. The **Model** serves as a container/controller for the whole simulation.

The last two classes are specific to Event Graphs.

1. An **EventNode** (subclass of **Event**), defined as an instantaneous occurrence that can trigger other events and/or change the state of the simulation, is represented as a *node* in the event graph.

2. A **CausalLink** emanating from an event/node is represented as an outgoing directed *edge* in the event graph. It represents causality between events. One event can conditionally trigger another event to occur some time in the future.

16.4.1 Example: Bank Model

For example, to create a simple bank simulation, one could use the four classes provided by the Event Graph simulation engine to create subclasses of **EventNode**, called **Arrival** and **Departure**, and one subclass of **Model**, called **BankModel**. The complete code for this example may be found in **Bank2.scala**. In more complex situations, one would typically define a subclass of **Entity** to represent the customers in the bank.

```scala
class BankModel2 (name: String, nStop: Int, iarrivalRV: Variate, serviceRV: Variate) extends Model (name, true) // true => animation on

The Scala code below was made more declarative than typical code for event-scheduling to better mirror event graph specifications, where the causal links specify the conditions and time delays. For instance,

```scala
() => nArr < nStop-1
``` 

is a anonymous function/closure returning **Boolean** that will be executed when arrival events are handled. In this case, it represents a stopping rule; when the number of arrivals exceeds a threshold, the arrival event will no longer schedule the next arrival. The **serviceRV** is a random variate to be used for computing service times.

In the **BankModel** class, the logic in the **Event** classes is simplified somewhat due to the specification of **CausalLinks**. Before the **Event** classes are specified, the following four types for definitions are required: statistical accumulators, event nodes, causal links/edges, and the state variables.

First the statistical accumulators are defined, one for inter-arrivals and one for service. A wait queue is given and it maintains its own statistics.
val t_ia_stat = new Statistic ("t_ia") // time between Arrivals statistics
val t_s_stat = new Statistic ("t_s") // time in Service statistics
addStats (t_ia_stat, t_s_stat)
val waitQueue = WaitQueue (this) // waiting queue that collects stats

For animation of the event graph, a prototype for each type of event is created and displayed as a node. Locations are required for each EventNode.

val aLoc  = Array (150.0, 200.0, 50.0, 50.0) // Arrival event node location
val dLoc  = Array (450.0, 200.0, 50.0, 50.0) // Departure event node location
val aProto = new EventNode (this, aLoc) // prototype for all Arrival events
val dProto = new EventNode (this, dLoc) // prototype for all Departure events

The edges connecting these prototypes represent the casual links. The aLink array holds two causal links emanating from Arrival, the first a self link representing triggered arrivals and the second representing an arrival finding an idle server, so it can schedule its own departure. The dLink array holds one causal link emanating from Departure, a self link representing the departing customer causing the next customer in the waiting queue to enter service (i.e., have its departure scheduled).

val aLink = Array (CausalLink ("l_A2A", this, () => nArr < nStop-1, aProto),
    CausalLink ("l_A2D", this, () => nIn == 0, dProto))
val dLink = Array (CausalLink ("l_D2D", this, () => nIn > 1, dProto))
aProto.displayLinks (aLink)
dProto.displayLinks (dLink)

The state variables, nArr, nIn and nOut, are defined as vars since they will change during the simulation.

var nArr  = 0.0 // number of customers that have arrived
var nIn   = 0.0 // number of customers in the bank
var nOut  = 0.0 // number of customers that finished & left

An animation of the Event Graph consisting of two EventNodes Arrival and Departure and three CausalLinks is depicted in Figure 16.7

![Event graph animation](image)

Figure 16.7: Event Graph Animation of a Bank.
The main thing to write within each subclass of Event is the occur method. To handle arrival events, the occur method of the Arrival class first checks the aLinks to see if it needs to trigger additional events. It then updates the current state by incrementing both the number of arrivals (nArr) and the number in the system (nIn).

```scala
case class Arrival (customer: Entity, delay: Double)
  extends Event (customer, this, delay, t_ia_stat, aProto):
  def occur (): Unit =
    if aLink(0).condition () then
      val toArrive = Entity (iArrivalRV.gen, serviceRV.gen, BankModel2.this)
      schedule (Arrival (toArrive, toArrive.iArrivalT))
    end if
    if aLink(1).condition () then
      schedule (Departure (customer, customer.serviceT))
    else
      waitQueue.enqueue (customer) // collects time in Queue statistics
    end if
    nArr += 1 // update the current state
    nIn += 1
  end occur

end Arrival
```

To handle departure events, the occur method of the Departure class first checks the dLink to see if it needs to trigger additional events. It then updates the state by decrementing the number in the system (nIn) and incrementing the number of departures (nOut).

```scala
case class Departure (customer: Entity, delay: Double)
  extends Event (customer, this, delay, t_s_stat, dProto):
  def occur (): Unit =
    leave (customer) // collects time in System statistics
    if dLink(0).condition () then
      val nextService = waitQueue.dequeue () // first customer in queue
      schedule (Departure (nextService, nextService.serviceT))
    end if
    nIn -= 1 // update the current state
    nOut += 1
  end occur

end Departure
```

Four of the classes used for creating simulation models following the Event Scheduling paradigm can be used for Event Graphs, namely Entity, Event, Model, and WaitQueue. In addition, EventNode is also required as they form the nodes in the Event Graphs. An edge in the Event Graph is an instance of the CausalLink class. These two new classes (EventNode and CausalLink) are described in the subsections below.
16.4.2 EventNode Class

The EventNode class provides facilities for defining simulation events. Subclasses of Event provide event-logic in their implementation of the occur method. The main purpose of EventNode is to associate a type of event with a node in the event graph.

Class Methods:

@param director the controller/scheduler that this event node is a part of
@param at the location of this event node

class EventNode (director: Model, at: Array [Double] = Array ())
    extends Event (EventNode.makePrototype (director), director, -1.0, null):

    def occur (): Unit = throw new NoSuchMethodException ("this occur should not be called")
    def displayLinks (links: Array [CausalLink]): Unit =

16.4.3 CausalLink Class

The CausalLink class provides casual links between events. Before an event updates the state, it checks its causal links to schedule/cancel other events. Events graphs make the linkage between types of events more explicit, e.g., an arrival triggers the next arrival some time in the future. If a server is available, it can also trigger the entity’s own service completion/departure event.

Class Methods:

@param label the name/label of the causal link
@param director the controller/scheduler that this causal link is a part of
@param condition the condition under which the link is triggered
@param causedEvent the event caused by this causal link

case class CausalLink (label: String, director: Model, val condition: () => Boolean, causedEvent: Event)
    extends Identifiable:

    def display (from: Event, to: Event): Unit =
    override def toString: String = s"CausalLink(${getClass.getSimpleName ()}, $name)"

Events graphs support a more declarative means for specifying a simulation, allow the relationships between events to be seen visually and provide basic animation of simulation model execution.
16.5 Exercises

1. It is common practice to implement an `eventList` as a Heap-Based Priority Queue. The most imminent event is at the top of the heap data structure in a priority queue. Processing this event involves removing and returning this event, swapping in the event in the last position in the heap, and reestablishing the heap order. Draw pictures to illustrate what happens to the heap.

2. As mentioned, the `dequeue` in the `WaitQueue` class collects both sample and time-persistent statistics. The `Statistic` class in the `mathstat` package is used to collect via `tally` sample statistics, while the `TimeStatistic` class is used to collect `accum` time-persistent statistics.

   ```kotlin
   @param name the name for this statistic (e.g., 'numberInQueue' or 'tellerQ')
   @param _lastTime the time of last observation
   @param _startTime the time observation began
   
   class TimeStatistic (override val name: String = "p-stat",
   private var _lastTime: Double = 0.0,
   private var _startTime: Double = 0.0)
   extends Statistic (name):
   
   How do the `tally` and `accum` methods differ? Why is it necessary to have two such methods?

3. Explain what each line of the main `while` loop does in the `simulate` method of the `Model` class.

4. Draw an event graph for the simulation of an M/M/c/K Queue. Let the arrival rate $\lambda = 7$ per hour and the service rate $\mu = 8$ per hour. Write and execute an event scheduling model for this Queue System. Compare the results with the theoretical results from Queueing Theory. Consider the following cases: $c = 1, K = 1; c = 1, K = \infty; c = 2, K = 10; c = 2, K = \infty$.

5. **Wendy’s vs. McDonald’s Lines.** Given two servers, is it better to have a line/queue for each server or one common line for both servers. Analyze the waiting times and standard deviation of the waiting times. Let $\lambda = 20$ hr$^{-1}$ and $\mu = 12$ hr$^{-1}$.

   Hint: To compare the two waiting times, compute the adjusted waiting time. For example, with a single queue, suppose $W_q$ is the average time in queue for the $n_q$ customers that had to wait. The overall adjusted waiting time is then

   $$T_q = \frac{n_qW_q + (m-n_q)0}{m}$$

   When there are two queues with waiting times $W_{q1}$ and $W_{q2}$, the formula becomes

   $$T_q = \frac{n_{q1}W_{q1} + n_{q2}W_{q2} + (m-n_{q1}-n_{q2})0}{m}$$

6. **Machine Shop.** Explain the edge conditions given in the Event Graph design diagram for the Machine Shop. Specify the random variates for the transition/delay times for the edges in the Event Graph.

7. **Machine Shop.** Complete the implementation of the Machine Shop simulation and determine which policy is better.
8. **Two-Stage Queueing System Simulation.** Consider modeling a system with two stages of service: in stage one a patient registers, and in stage two the patient receives treatment. The line for registration is unbounded, while patients waiting for treatment must be within a room with a total capacity of $K$. The first stage has one server, while the second stage has two. Based on the cases in the previous exercise, simulate a two-stage service system where the first stage has a queue with $c = 1, K = \infty$ and the second stage has a queue with $c = 2, K = 10$. When the second queue is full, the server in the first stage will be blocked. (i.e., must be idle until space is available).


10. **ScalaTion** uses Scala’s Priority Queue class for its time ordered F.E.L. (eventList), but that class could also be used for priority based waiting queues. Add a new class to the `event` package called `WaitQueuePQ` to provide this capability.

11. Use the new `WaitQueuePQ` class along with `WaitQueue` and `WaitQueueLCFS` to test various job scheduling algorithms: FCFS, LCFS, Shortest Job First (SJF) and Highest Priority First (HPF).

12. Event Scheduling (ES) Simulation: A **Small Fast Food Restaurant** has two servers and enough space for three customers to wait (at most five customers total at any given time). For the case of a single queue, perform an ES simulation for $m = 20$ customer arrivals. Assume each server can process $\mu = 30$ customers per hour and that the customer arrival rate $\lambda = 75$ customers per hour (assume Exponential distributions). Each completed order gives a net profit (before paying the servers) of $2.00$ dollars. Each server makes $11.00$ dollars per hour. Should the restaurant hire a third server? Explain in terms of profit (after paying the servers) per hour. **Note:** this simulation problem is posed in the Hand/Spreadsheet Simulation section of the Simulation Foundations Chapter.

   (i) Use ScalaTion’s Known Random Variate Generator (RVG) to make ES reproduce the results of the Spreadsheet Simulation. Give the Event Graph, code for the `Event` subclasses including their `occur` methods, and the summary results.

   (ii) Replace the Known RVG with ScalaTion’s Exponential RVG in order to run the simulation longer to obtain better results. Also, run multiple replications to produce multiple estimates for the final profit for having two servers vs. three servers. To make the replications independent, make sure each replication uses a different base random number stream.

   (iii) Explain how Confidence Intervals can be used to make more informed decisions.

   (iv) **[Bonus]** Create 95% Confidence Intervals for the two and three server simulations. Let the number of replications be 10 and use the Student’s t Distribution.
Chapter 17

Process-Oriented Models

A process can be thought of an unit of execution. Conceptually, an active entity in a process-oriented simulation may execute as its own process.

Computers with multiple cores and hyper-threading may execute many processes at once. A system with 16 cores and hyper-threading (2 threads run per core), would allow 32 processes to run in parallel. Actually, since several processes will be waiting on input, the number of processes is much larger that this number would suggest. Still, process-oriented simulation typically will require many more processes, so traditional heavy-weight processes are out of the question. Typically, the choice is between threads and coroutines.
17.1 Base Traits and Classes for Process-Oriented Models

The simulation package contains several base traits and classes that can be used by several types of simulation models, and are especially useful for process-oriented simulation models.

17.1.1 Identifiable Trait

The Identifiable trait provides unique identification for simulation components, entities and events. Includes a mandatory id and an optional name.

```scala
trait Identifiable:
  def name: String = _name
  def name_=(name: String): Unit =
  def simType: String = getClass.getSimpleName()
  def me: String = s"$simType.$_name.$id"
  override def equals(that: Any): Boolean =
  override def hashCode: Int = id
```

17.1.2 Locatable Trait

The Locatable trait provides location information/coordinates for objects in simulation models (e.g., Components).

```scala
trait Locatable:
  def at: Array[Double] = _at
  def at_= (at: Array[Double]): Unit =
```

17.1.3 Modelable Trait

The Modelable trait defines the notions of clock and simulate, common to many types of ScalaTion simulation models.

```scala
trait Modelable:
  def clock = _clock
  def simulate(startTime: Double): Unit
```

17.1.4 Temporal Trait

The Temporal trait adds time (actTime) and temporal ordering to the Identifiable trait.

```scala
trait Temporal extends Identifiable:
  def compare(other: Temporal): Int = { actTime compare other.actTime }
  override def toString: String = s"Temporal ($me, $actTime)"
```
17.2 Concurrent Processing of Actors

A simulation with multiple actors as active entities whose behaviors overlap in time are most naturally implemented using concurrent programming.

Traditionally, programming language support for concurrent programming has been limited and this makes providing support of process-oriented models more difficult. Typically, support for coroutines is sufficient for developing simulation engines of this type. Languages supporting coroutines include: Simula, Smalltalk, Modula-2, Ruby, Julia, Go, and Kotlin.

On the other hand, many languages support threads, notably Java and therefore all Java based languages including Scala. Although threads are capable of getting the job done, they introduce two problems: Unusual transfer of control between threads can lead to challenging bugs to eliminate. There is also more overhead (time and space) required to use threads rather than coroutines. To deal with the first problem, SCALATION implements a Coroutine class using Java’s Runnable interface and Thread class. Then users of SCALATION can avoid the complexity and bugs associated with threads. Improvement on the problem of overhead will be provided by Java 18 in the form of VirtualThreads. These run in user space with reduced overheads and can allow may more actors to run concurrently.

17.2.1 Java’s Thread Class

The PingPong Java class provides a simple example of how Java threads work. The class implements the Runnable interface allowing the run method of object instances of this class to be executed concurrently in multiple Threads. In this case, two instances are created: the first instance that writes ”ping” and sleeps for 333 milliseconds, and the second instance that writes ”PONG” and sleeps for 1000 milliseconds.

The following are some of the commonly used methods available in Java Threads: currentThread (the currently running thread), interrupt (receive an InterruptedException), join (waits for thread to terminate), run (execute the overridden method concurrently) sleep (temporarily cease execution for the given number of milliseconds), start (the thread begins execution and calls the run method), and yield (this threads offers to give another thread a chance to run). See the on-line Java API Documentation for more details.

```java
import static java.lang.System.*;

public class PingPong implements Runnable
{
    private String word;
    private int delay;

    /***********************************************
    * Construct a ping or pong object.
    */
    public PingPong (String whatToSay, int delayTime)
    {
        word = whatToSay;
        delay = delayTime;
    } // PingPong
```
/*******************************************************
* Run method for the ping/pong object.
*/
public void run ()
{
    for ( ; ; ) {
        out.println (word);
        try {
            Thread.sleep (delay);
        } catch (InterruptedException ex) {
            return;
        } // try
    } // for
} // run

/*******************************************************
* Main method for invoking the application.
* @param args Command-line arguments
*/
public static void main (String [] args)
{
    new Thread (new PingPong ("ping", 333)).start ();
    new Thread (new PingPong ("PONG", 1000)).start ();
} // main

} // PingPong

This code can be compiled typing the following:

$ javac PingPong.java
$ java PingPong

Notice the interleaved execution. Had the run been called directly, rather than indirectly via calling the start method, no interleaving would be seen, since this would require the first thread to finish before the second one can begin. Try change ”start” to ”run” in the code above. Note, to terminate the program, type ”Ctrl C”.

17.2.2 SCALATion’s Coroutine Class

The Coroutine class supports (one-at-a-time) quasi-concurrent programming. A coroutine runs/acts until it yields control from ’this’ to ’that’ coroutine. When resumed, a coroutines continues its execution where it left off. As in the PingPong class, the Coroutine class implements (extends in Scala) Runnable interface. The run method keeps track of the number of coroutines started and terminated as well as delegates to the act method. This abstract method must be implemented by the simulation model developer. The start method causes the run method to be executed concurrently, which delegates to the act method (that contains the simulation logic).

@param label the label for the class of coroutines to be created.
abstract class Coroutine (label: String = "cor")
    extends Runnable:

    def counts: (Int, Int, Int) = (nCreated, nStarted, nTerminated)
    def run (): Unit =
    def act (): Unit
    def yyield (that: Coroutine, quit: Boolean = false): Unit =
    def start (): Future[_] =
    def interrupt (): Unit =

The Coroutine also uses the following from the java.util.concurrent package: Executors, ExecutorService, Future, Semaphore, and ThreadPoolExecutor. See Coroutine.scala in the scalation.simulation package for details.
17.3 Process Interaction

Many discrete-event simulation models are written using the process-interaction world view, because the code tends to be concise and intuitively easy to understand. Take for example the process-interaction model of a bank (BankModel a subclass of Model) shown later. Following this world view, one simply constructs the simulation components and then provides a script for entities (cimActors) to follow while in the system. In this case, the act method for the customer class provides the script (what entities should do), i.e., enter the bank, if the tellers are busy wait in the queue, then receive service and finally leave the bank.

The development of a simulation engine for process-interaction models is complicated by the fact that concurrent (or at least quasi-concurrent) programming is required. Various language features/capabilities from lightweight to middleweight include continuations, coroutines, fibers, actors, virtual-threads and threads. Heavyweight concurrency via OS processes is infeasible, since simulations may require a very large number of concurrent entities. The main requirement is for a concurrent entity to be able to suspend its execution and be resumed where it left off (its state being maintained on a stack). Since preemption is not necessary, lightweight concurrency constructs are ideal. Presently, ScalaTion uses the Coroutine class.

ScalaTion includes several types of model components: Gate, Junction, Resource, Route, Sink, Source, Transport, WaitQueue and WaitQueue_LCFS. A model may be viewed as a directed graph with several types of nodes:

- **Gate**: a gate is used to control the flow of entities, they cannot pass when it is shut.
- **Junction**: a junction is used to connect two transports.
- **Resource**: a resource provides services to entities (typically resulting in some delay).
- **Sink**: a sink consumes entities.
- **Source**: a source produces entities.
- **WaitQueue**: a FCFS wait-queue provides a place for entities to wait, e.g., waiting for a resource to become available or a gate to open.
- **WaitQueue_LCFS**: an LCFS wait-queue provides a place for entities to wait, e.g., waiting for a resource to become available or a gate to open.

These nodes are linked together with directed edges (from, to) that model the flow entities from node to node. A Source node must have no incoming edges, while a Sink node must have no outgoing edges.

- **Route**: a route bundles multiple transports together (e.g., a two-lane, one-way street).
- **Transport**: a transport is used to move entities from one component node to the next.

The model graph includes coordinates for the component nodes to facilitate animation of the model. Coordinates for the component edges are calculated based on the coordinates of its from and to nodes. Small colored tokens move along edges and jump through nodes as the entities they represent flow through the system.

Formally, it is not required to be a graph since entities can move from node to node without going along an edge (e.g., WaitQueue to Resource). These graph-like structures are referred to as Network Diagram. ScalaTion’s Agent-Based Simulation (see the next section), however, does require the model to be based on an underlying Property Graph.
17.3.1 Model Template

All process-interaction simulation models in SCALATiON are of the following basic form. The file called Ex_Template.scala in the scalation.simulation.process package may be used as a starting template for the development of specific process-interaction simulation models.

```scala
@param name the name of the simulation model
@param reps the number of independent replications to run
@param animating whether to animate the model
@param aniRatio the ratio of simulation speed vs. animation speed
@param nStop the number arrivals before stopping
@param stream the base random number stream (0 to 999)

class SOMEModel (name: String = "SOME", reps: Int = 1, animating: Boolean = true,
    aniRatio: Double = 8.0, nStop: Int = 100, stream: Int = 0)
    extends Model (name, reps, animating, aniRatio):

    // Initialize Model Constants
    val lambda = 6.0 // customer arrival rate (per hour)

    // Create Random Variables (RVs)
    val iArrivalRV = Exponential (HOUR / lambda, stream)

    // Create Model Components
    val entry = Source ("entry", this, () => SOMEActor (), 0, nStop, iArrivalRV, (100, 290))
    val exit = Sink ("exit", (600, 290))

    addComponent (entry, exit)

    // Specify Scripts for each Type of Simulation Actor
    case class SOMEActor () extends SimActor ("s", this):
        def act (): Unit =
            println ("SOMEActor: please write the script for this actor")
            exit.leave ()
        end act
    end SOMEActor

    simulate ()
    waitFinished ()
    Model.shutdown ()

end SOMEModel
```
Specification of a process-interaction involves the four steps: (1) Initialize Model Constants, (2) Create Random Variables (RVs), (3) Create Model Components, and (4) Specify Scripts for each Type of Simulation Actor. The \texttt{SOMEModel} class can be invoked as follows:

```scala
> runMain scalation.simulation.process.runSOME
@main def runSOME (): Unit = new SOMEModel ()
```

### 17.3.2 Component Trait

The basic structure of a simulation is given by the simulation components. The \texttt{Component} trait provides basic common features for simulation components. A component may function either as a node or edge. Entities/sim-actors interact with component nodes and move/jump along component edges. All components maintain sample/duration statistics (e.g., time in waiting queue) and all except \texttt{Gate}, \texttt{Source} and \texttt{Sink} maintain time-persistent statistics (e.g., number in waiting queue).

The most important component is the containing \texttt{Model}. For example, in \texttt{BankModel} discussed next, the instance of this class will serve as the director. Its role is similar to that in the \texttt{event} package, but rather than executing \texttt{occur} methods, it transfers control between the \texttt{SimActors}.

---

#### Class Methods:

```scala
trait Component
extends Identifiable with Locatable:

def initComponent (label: String, loc: Array[Double]): Unit =

def director: Model = _director

def director_= (director: Model): Unit =

def composite: Boolean = subpart.size > 0

protected def initStats (label: String): Unit =

def aggregate (): Unit =

def display (): Unit =

def tally (duration: Double): Unit = _durationStat.tally (duration)

def accum (value: Double): Unit = _persistentStat.accum (value, _director.clock)

def durationStat: Statistic = _durationStat

def persistentStat: TimeStatistic = _persistentStat
```

---

### 17.3.3 Example: BankModel

Consider an simple bank simulation model where customers wish to utilize one of several tellers, but may need to wait their turn in a shared queue. Figure 17.1 depicts the flow of customers (SimActors) through the bank (entry to door/exit) and corresponds closely with the code required to implement the model in the process-interaction paradigm.
Such a BankModel (see the example_1 package) may be developed as follows:

- **Initialize Model Constants**: In this case customer arrival (\(\lambda\)) and service (\(\mu\)) rates need to be specified. In addition, the number of service units (\(nTellers\)) needs to be specified.

- **Create Random Variables (RVs)**: This model will have three random variates: one for the inter-arrival times (\(iArrivalRV\)), one for service times (\(serviceRV\)), and one for movement (\(moveRV\)) along Transports.

- **Create Model Components**: A key step is to define the component nodes entry, tellerQ, teller, and door. Then two edge components, toTellerQ and toDoor, are defined. These six components are added to the BankModel using the `addComponent` method. Note, the endpoint nodes for an edge must be added before the edge itself.

```scala
class BankModel (name: String = "Bank", reps: Int = 1, animating: Boolean = true, aniRatio: Double = 8.0, nStop: Int = 100, stream: Int = 0)
extends Model (name, reps, animating, aniRatio):
    // Initialize Model Constants
    val lambda = 6.0 // customer arrival rate (per hour)
    val mu = 7.5 // customer service rate (per hour)
    val nTellers = 1 // the number of bank tellers (servers)

    // Create Random Variables (RVs)
    val iArrivalRV = Exponential (HOUR / lambda, stream)
    val serviceRV = Exponential (HOUR / mu, (stream + 1) % N_STREAMS)
    val moveRV = Uniform (4 * MINUTE, 6 * MINUTE, (stream + 2) % N_STREAMS)

    // Create Model Components
    val entry = Source ("entry", this, () => Customer (), 0, nStop, iArrivalRV, (100, 290))
    val tellerQ = WaitQueue ("tellerQ", (330, 290))
    val teller = Resource ("teller", tellerQ, nTellers, serviceRV, (350, 285))
    val door = Sink ("door", (600, 290))
    val toTellerQ = Transport ("toTellerQ", entry, tellerQ, moveRV)
    val toDoor = Transport ("toDoor", teller, door, moveRV)

    addComponent (entry, tellerQ, teller, door, toTellerQ, toDoor)
```

Figure 17.1: Bank Model Schematic Diagram
Specify Scripts for each Type of Simulation Actor: Finally, an inner case class called Customer is defined where the act method specifies the script for bank customers to follow. The act method specifies the behavior of concurrent entities (SimActor) and is analogous to the run method for Java/Scala Threads.

```scala
// Specify Scripts for each Type of Simulation Actor

case class Customer () extends SimActor ("c", this):

  def act (): Unit =
    toTellerQ.move ()
    if teller.busy then tellerQ.waitIn () else tellerQ.noWait ()
    teller.utilize ()
    teller.release ()
    toDoor.move ()
    door.leave ()
  end act

end Customer

simulate ()
waitFinished ()
Model.shutdown ()

end BankModel
```

The script for the actor consists of the following steps:

1. Upon creation by the Source at entry, the actor will move to the teller queue.
2. The actor will check whether all the tellers are busy and if all are busy, will wait in the queue tellerQ which is a WaitQueue. Note, the call to noWait is just for statistics collection.
3. The actor will utilize one of the tellers in the teller Resource, for a period of time corresponding to its service time.
4. After service is finished, the actor will then release the teller. This allows a waiting actor to begin service and triggers the collection of statistics.
5. The actor now moves to the door/exit.
6. Upon arrival at the door, a Sink, the actor will leave the bank/simulation and overall statistics will be collected.

The last three method calls will run the simulation using multiple threads/coroutines (simulate), wait until all the threads/coroutines are finished (waitFinished), and then safely shut down concurrent execution (Model.shutdown).
17.3.4 Executing the Bank Model

The BankModel class can be invoked as follows:

```scala
> runMain scalation.simulation.process.example_1.runBank

@main def runBank (): Unit = new BankModel ()
```

To make the animation easier to follow, try changing the aniRatio to 50.0.

17.3.5 Network Diagram

A Network Diagram for the Bank Model shown in Figure 17.2 displays four nodes (Source in green, WaitQueue in cyan, Resource in orange, and Sink in purple) and two edges (both Transports in blue).

![Network Diagram of Bank Model](image)

Figure 17.2: Animation of Network Diagram of Bank Model

An active entity (SimActor) representing a Customer is shown as token (small circle) frozen in its motion along the first Transport. Like an Event Graph, a Network Diagram can be used for both simulation model design and animation.

17.3.6 Comparison to Event Scheduling

In comparison, the Bank Model for event-scheduling did not include time delays and events for moving tokens along transports (although these could be added). The BankModel in the example_MIR package reduces the impact of transports by (1) using the transport’s jump method rather than its move method and (2) reducing the time through the transport by an order of magnitude. The jump method has the tokens jumping directly to the middle of the transport, while the move method simulates smooth motion using many small hops.

The example_1 package provides several example process-interaction models based on the OSS method, including BankModel, CallCenterModel, EmerDeptModel, LoopModel, MachineModel, OneWayStreetModel, RoadModel, and TrafficModel.

The example_MIR and example_MBM packages include a subset of these models.
17.3.7  SimActor Class

The SimActor abstract class represents entities that are active in the model. The act abstract method, which specifies entity behavior, must be defined for each subclass. Each SimActor extends the Coroutine class and may be roughly thought of as running in its own thread. The script for entities/sim-actors to follow is specified in the act method of the subclass as was done for the Customer case class in the BankModel.

For example, a customer in the BankModel will enter the bank, move to a teller, if there is a line/queue will wait in the queue, be served by the teller and finally leave the bank. The director will transfer control to the actor (bank customer) which will execute code to get to the next step and transfer control back to the director. In this way the entity progresses through time processing multiple events over its lifetime. A SimActor is created by a Source and terminated by a Sink. The act method encodes the logic of the actor’s script.

Class Methods:

@param label the label/name of the entity (‘SimActor’)
@param director the director controlling the model

abstract class SimActor (label: String, director: Model)
extends Coroutine (label) with Temporal with Ordered [SimActor] with Locatable:

  def trajectory: Double = _trajectory
  def trajectory_=(trajectory: Double): Unit =
  def compare (actor2: SimActor): Int = actor2.actTime compare actTime
  def act (): Unit
  def schedule (delay: Double): Unit =
     actTime = director.clock + delay
     director.reschedule (this)
  def yieldToDirector (quit: Boolean = false): Unit =
     director.log.trace (this, "resumes", director, director.clock)
     yyield (director, quit)
  override def toString: String = s"SimActor ($me at $actTime)"

Two of the key methods involved in transferring control between actors (via the director) are schedule and yieldToDirector.

    def schedule (delay: Double): Unit =
       actTime = director.clock + delay
       director.reschedule (this)
    end schedule

The schedule method places this actor in agenda (like a future event list) effectively specifying when the actor (a coroutine) will be reactivated. The delay parameter indicates how far into the future this will be.

    def yieldToDirector (quit: Boolean = false): Unit =
       director.log.trace (this, "resumes", director, director.clock)
       yyield (director, quit)
    end yieldToDirector
When the actor is has completed a step (conceptually like an embedded event) and either placed itself in a queue or the agenda, it is ready to let another actor to execute. It does this by yielding control to the director so the director can take the next action via the `yieldToDirector` method. The `quit` parameter is a flag indicating whether this actor has completed its last step.

### 17.3.8 Source Class

The `Source` class is used to periodically inject entities (SimActors) into a running simulation model (and a token into the animation). It may act as an arrival generator. A `Source` is both a simulation `Component` and a special `SimActor`, and therefore can run concurrently.

The `act` method loops over time through the creation of `units` entities. After making an entity using the `makeEntity` function, it schedules itself to run in the future by an amount of the given by `iArrivalTime.gen`, and it transfers control back to the `director` by calling `yieldToDirector`. A `Source` may create entities of different subtypes. For example, in the `TrafficModel` the `subtype` indicates which of the four directions cars will traveling as they approach an intersection. The `act` method has an outer loop over replications and an inner scheduling loop shown below.

```scala
breakable {
  for i <- 1 to units do // minor loop - make actors
    debug ("act", s"make $i SimActor")
    if director.stopped then break () // terminate source, simulation ended
    val actor = makeEntity () // make new actor
    actor.mySource = this // actor's source
    actor.subtype = esubtype // set the entity subtype
    director.numActors += 1 // number of actors created by all sources
    director.log.trace (this, "generates", actor, director.clock)
    director.animate (actor, CreateToken, randomColor (actor.id), Ellipse (),
      Array (at(0) + at(2) + RAD / 2.0, at(1) + at(3) / 2.0 - RAD))
    actor.schedule (0.0)
    if i < units then
      val duration = iArrivalTime.gen
      tally (duration)
      schedule (duration)
      yieldToDirector () // yield and wait duration time units
    end if
  end for
} // breakable
```

For conciseness the `Source.group` method may be used to create multiple sources for a model.

```scala
def group (director: Model, makeEntity: () => SimActor, units: Int, xy: (Int, Int),
  src: (String, Int, Variate, (Int, Int))*)(: List [Source] =
  val sourceGroup = new ListBuffer [Source] ()
  for s <- src do sourceGroup += Source (s._1, director, makeEntity, s._2, units, s._3,
    (xy._1 + s._4._1, xy._2 + s._4._2))
  sourceGroup.toList
end group
```

553
For animation, the location of a **Source** node is specified by `loc`.

### Class Methods:

@param name the name of the source  
@param director the director controlling the model  
@param makeEntity the function to make entities of a specified type  
@param esubtype indicator of the subtype of the entities to be made  
@param units the number of entities to make  
@param iArrivalTime the inter-arrival time distribution  
@param loc the location of the source (x, y, w, h)

class Source (name: String, director: Model, makeEntity: () => SimActor,  
esubtype: Int, units: Int,  
iArrivalTime: Variate, loc: Array [Double])  
  extends SimActor (name, director) with Component:

def this (name: String, director: Model, makeEntity: () => SimActor, esubtype: Int,  
units: Int, iArrivalTime: Variate, xy: (Double, Double)) =
def display (): Unit =
def act (): Unit =

### 17.3.9 Sink Class

The **Sink** class is used to terminate entities (**SimActors**) when they are finished. This class will remove the token from the animation and collect important statistics about the entity. A sink has a name and a location in the animation. Unlike a **Source**, a **Sink** is not active, rather is bundles logic to be executed by an actor when they are ready to leave the simulation. They do this by calling the `leave` method, which will tally the actors’ time in the system.

   tally (director.clock - actor.arrivalT)

Note, when a **SimActor** is no longer referenced (e.g., in director’s **agenda** or in a wait queue) it becomes available for garbage collection (memory reclamation).

### Class Methods:

@param name the name of the sink  
@param at the location of the sink (x, y, w, h)

class Sink (name: String, at: Array [Double])  
  extends Component:

def this (name: String, xy: (Double, Double)) =
def display (): Unit =
def leave (): Unit =

554
17.3.10 Transport Class

The Transport class provides a pathway between two other component nodes. The Components in a Model conceptually form a graph in which the edges are Transport objects and the nodes are other Component objects. An edge may be either a Transport or Route. A Transport is directional connecting a from component to a to component. When flow is required in both directions, two transports are required.

A SimActor may utilize a Transport by calling either the move or jump methods. The move method is intended for smooth animation, while the jump transports the entity quickly to the next component.

Class Methods:

```scala
@param name the name of the transport
@param from the first/starting component
@param to the second/ending component
@param motion the speed/trip-time to move down the transport
@param isSpeed whether speed or trip-time is used for motion
@param bend the bend or curvature of the transport (0 => line)
@param shift1 the x-y shift for the transport’s first endpoint (from-side)
@param shift2 the x-y shift for the transport’s second endpoint (to-side)

class Transport (name: String, val from: Component, val to: Component, motion: Variate, isSpeed: Boolean = false, bend: Double = 0.0, shift1: R2 = new R2 (0.0, 0.0), shift2: R2 = new R2 (0.0, 0.0))
extends Component:

override def at: Array[Double] =
def display (): Unit =
def jump (): Unit =
def move (): Unit =
```

17.3.11 Resource Class

The Resource class provides services to entities (SimActors). The service provided by a resource typically delays the entity by an amount of time corresponding to its service time. The Resource may or may not have an associated waiting queue. It provides a number of service units (e.g., tellers) and a busy method to determine if all the servers are busy. A SimActor typically calls utilize () to go into service for an amount time determined by serviceTime.gen. Alternatively, if the actor knows the length of its service time, it may call utilize (duration). When finished, the actor should release the server. Finally, it is possible change the number service units available (e.g., a teller goes on break) by calling changeUnits.

Class Methods:
class Resource (name: String, line: WaitQueue, private var units: Int, serviceTime: Variate, 
at: Array[Double])
    extends Component:

    def this (name: String, line: WaitQueue, units: Int, serviceTime: Variate, 
             xy: (Double, Double))
    def changeUnits (dUnits: Int): Unit =
    def display (): Unit =
    def busy: Boolean = inUse == units
    def utilize (): Unit =
    def utilize (duration: Double): Unit =
    def release (): Unit =

17.3.12 WaitQueue Class

The WaitQueue class is a wrapper for Scala’s Queue class, which supports FCSC Queues. It adds monitoring capabilities and optional capacity restrictions. If the queue is full, entities (SimActors) attempting to enter the queue are barred. At the model level, such entities may be (1) held in place, (2) take an alternate route, or (3) be lost (e.g., dropped call/packet). An entity on a WaitQueue is suspended for an indefinite wait. The actions of some other concurrent entity will cause the suspended entity to be resumed (e.g., when a bank customer finishes service and releases a teller).

When seeking service, an actor should check whether the servers are busy, and if so call waitIn. Although it is not necessary to call noWait, it is preferred to make this call to get the average waiting time for all actors, not just those that had to wait (of course, if that is the real interest of the study, the noWait call may be left out).

    if teller.busy then tellerQ.waitIn () else tellerQ.noWait ()

Class Methods:

@param name the name of the wait-queue
@param at the location of the wait-queue (x, y, w, h)
@param cap the capacity of the queue (defaults to unbounded)

class WaitQueue (name: String, at: Array[Double], cap: Int = Int.MaxValue)
    extends Queue[SimActor] with Component:
17.3.13 WaitQueue_LCFS Class

The \texttt{WaitQueue_LCFS} class is a wrapper for Scala’s \texttt{Stack} class, which supports Last-Come, First-Serve (LCFS) Queues. It adds monitoring capabilities and optional capacity restrictions. If the queue is full, entities (\texttt{SimActor}s) attempting to enter the queue are ’barred’. At the model level, such entities may be (1) held in place, (2) take an alternate route, or (3) be lost (e.g., dropped call/packet).

Class Methods:

@param name \hspace{1em} the name of the wait-queue
@param at \hspace{1em} the location of the wait-queue (x, y, w, h)
@param cap \hspace{1em} the capacity of the LCFS queue (defaults to unbounded)

class WaitQueue_LCFS (name: String, at: Array [Double], cap: Int = Int.MaxValue)
    extends Stack [SimActor] with Component:

    def this (name: String, xy: (Double, Double), cap: Int) =
    def isFull: Boolean = length >= cap
    def barred: Int = _barred
    def display (): Unit =
    def waitIn (): Boolean =
    def noWait (): Unit = tally (0.0)

17.3.14 Junction Class

The \texttt{Junction} class provides a connector between two transports/routes. Since \texttt{Lines} and \texttt{QCurves} have limitation (e.g., hard to make a loop back), a junction may be needed, see the \texttt{LoopModel} for an example. Statistics are collected about entities at junctions.

For traffic simulations, junctions may also indicate locations for road sensors. The statistics collected at the junction can then be compared with the data recorded by a real-life sensor and used to calibrate/validate the simulation model. For example CalTrans records and makes available sensor data in its Performance Measure System (PeMS) \url{https://dot.ca.gov/programs/traffic-operations/mpr/pems-source}

Class Methods:

557
17.3.15 Gate Class

The Gate class models the operation of gates that can open and shut. When a gate is open, entities can flow through and when shut, they cannot. When shut, the entities may wait in a queue or go elsewhere. A gate can model a traffic light (green ⇒ open, red ⇒ shut). The onTime indicates how long the gate will be open (green light), while offTime indicated how long the gate will be shut (red light).

Class Methods:

- @param name the name of the gate
- @param director the model/container for this gate
- @param line the queue holding entities waiting for this gate to open
- @param units number of units/phases of operation
- @param onTime distribution of time that gate will be open
- @param offTime distribution of time that gate will be closed
- @param loc the location of the Gate (x, y, w, h)
- @param shut0 'Boolean' indicating if the gate is initially opened or closed
- @param cap the maximum number of entities that will be released when the gate is opened

```scala
class Gate (name: String, director: Model, line: WaitQueue, units: Int, onTime: Variate, offTime: Variate, loc: Array[Double], shut0: Boolean = false, cap: Int = 10)
  extends SimActor (name, director) with Component:

  def shut: Boolean = _shut
  def display (): Unit =
  def release (): Unit =
  def act (): Unit =
  def gateColor: Color = if _shut then red else green
```

558
def flip (): Unit = _shut = ! _shut
def duration: Double = if _shut then offTime.gen else onTime.gen

17.3.16 Route Class

The Route class provides a multi-lane pathway between two other node components. The Components in a Model conceptually form a graph in which the edges are Transports/Route and the nodes are other components. A route is a composite component that bundles several transports. The route will have k lanes that are stored in an array of Transports.

val lane = Array.ofDim [Transport] (k)

See the RoadModel which uses two Route objects, one for West-bound traffic and the other for East-bound traffic. Each route has two lanes, making the road a four-lane road overall.

Class Methods:

@param name the name of the route
@param k the number of lanes/transports in the route
@param from the starting component
@param to the ending component
@param motion the speed/trip-time to move down the transports in the route
@param isSpeed whether speed or trip-time is used for motion
@param angle angle in radians of direction (0 => east, Pi/2 => north, Pi => west, 3Pi/2 => south)
@param bend the bend or curvature of the route (0 => line)

class Route (name: String, k: Int, from: Component, to: Component,
  motion: Variate, isSpeed: Boolean = false,
  angle: Double = 0.0, bend: Double = 0.0)
  extends Component:

def selector: Variate = lane(0).selector
def selector_= (selector: Variate): Unit = lane(0).selector = selector
override def at: Array [Double] = lane(0).at
def display (): Unit =

17.3.17 Model Class

The Model class maintains a list of components making up the model and controls the flow of entities (SimActors) through the model, following the process-interaction world-view. A simulation model must extend the Model class and create an instance object that serves as the director. The director maintains a time-ordered priority queue called agenda to activate/re-activate each of the entities. The metaphor is that
the director directs the actors in the play (i.e., simulation model). Each entity (SimActor) is implemented as a Coroutine and may be roughly thought of as running in its own thread. Control is transferred back and forth between the director and the actors in the play.

Class Methods:

```scala
@param name the name of the simulation model
@param reps the number of independent replications to run
@param animating whether to animate the model
@param aniRatio the ratio of simulation speed vs. animation speed
@param full generate a full report with both sample and time-persistent statistics

class Model (name: String, val reps: Int = 1, animating: Boolean = true, aniRatio: Double = 1.0, val full: Boolean = true)
  extends Coroutine (name) with Completion with Modelable with Component:
```

```scala
def addComponent (_parts: Component*): Unit = for p <- _parts do parts += p
def addComponents (_parts: List [Component]*): Unit = for p <- _parts; q <- p do parts += q
def theActor: SimActor = _theActor
def stopped: Boolean = ! simulating
def reset (): Unit =
def resetStats (rep: Int): Unit =
def simulate (_startTime: Double = 0.0): Unit =
def cleanup (): Unit =
def reschedule (actor: SimActor): Unit = agenda += actor
def act (): Unit =
def getStatistics: ListBuffer [Statistic] =
def display (): Unit = for p <- parts do p.display ()
def animate (who: Identifiable, what: CommandType, color: Color, shape: Shape, at: Array [Double]): Unit =
def animate (who: Identifiable, what: CommandType, color: Color, shape: Shape, from: Component, to: Component, at: Array [Double] = Array ()): Unit =
protected def fini (rep: Int): Unit =
private def report (): Unit =
private def reportf (): Unit = new StatTable (s"$name statistics", getStatistics)
```

Process-Interaction Simulation Engine

The operation of the process-interaction simulation engine can be understood by looking at the inner scheduling loop within the act method of the Model class. The director takes the first actor in the agenda and marks it as _theActor. Then it advances its _clock to the activation time of _theActor. The director then transfer control by yielding to the _theActor. After executing the next step in their logic, _theActor will transfer control back to the director. This will continue until the agenda becomes empty or the director is instructed to stop simulating.
while simulating && ! agenda.isEmpty do // INNER SCHEDULING LOOP
  _theActor = agenda.dequeue () // next from priority queue
  _clock = _theActor.actTime // advance the time
  log.trace (this, "resumes", _theActor, _clock)
  yyield (_theActor) // director yields to actor
end while

The outer loop in the act method is for replications.

for rep <- 1 to reps do // LOOP THROUGH REPLICATIONS
  For One-Shot Simulation (OSS), reps should be one and for the Method of Independent of Replications (MIR) it should be say 10 or greater. See the Simulation Output Analysis Chapter for details. The simulate method is called by the class extending Model to initialize the component parts of the model. Its call to start will make a new thread that begins executing the director’s act method.

  def simulate (_startTime: Double = 0.0): Unit =
    startTime = _startTime
    _clock = startTime
    log.trace (this, "starts", this, _clock)
    for p <- parts do
      log.trace (this, s"establish x = \${p.at(0)}, y = \${p.at(1)}", p, _clock)
      p.director = this
      for q <- p.subpart do q.director = this
      if p.isInstanceOf [Source] then reschedule (p.asInstanceOf [Source])
    end for
    start () // start the director thread/actor -> act ()
  end simulate

17.3.18 Vehicle Traffic Model

A Network Diagram for a Vehicle Traffic simulation model is shown in Figure 17.3. It models an intersection of two multi-lane roads, one East-West and the other North-South. The intersection is controlled by traffic lights. Each road is divided into four parts: two directions and two segments (before and after the light).

There are five random variates used in this simulation.

val iArrivalRV = Uniform (iaTime, stream)
val onTimeRV = Sharp (onTime, (stream + 1) % N_STREAMS)
val offTimeRV = Sharp (offTime, (stream + 2) % N_STREAMS)
val moveRV = Uniform (mvTime, (stream + 3) % N_STREAMS)
val laneRV = Bernoulli ((stream + 4) % N_STREAMS)

The components in the simulation model are prescribed as follows:

At the beginning of each route, a Source will generate car arrivals. As traffic flow changes during each day, a Non-Homogeneous Process-Process (NHPP) may be used to generate cars (this example just uses Uniform). The time-dependent arrival rate λ(t) may be fit to traffic flow data. For conciseness, the group
method is used to create all four Sources. The coordinates (800, 250) form a reference point; the rest are relative to the reference point.

```scala
val source = Source.group (this, () => Car (), nStop, (800, 250),
  ("s1N", 0, iArrivalRV, (0, 0)), // from North
  ("s1E", 1, iArrivalRV, (230, 200)),
  ("s1S", 2, iArrivalRV, (30, 400)),
  ("s1W", 3, iArrivalRV, (-200, 230)))
```

A place is needed for cars waiting for a stop light to change from red to green. Thus four WaitQueues are needed.

```scala
val queue = WaitQueue.group ((800, 430), ("q1N", (0, 0)), // before North light
  ("q1E", (50, 20)),
  ("q1S", (30, 70)),
  ("q1W", (-20, 50)))
```

The 4-way intersection requires four traffic lights to the control flow of cars. At any particular time, two of the lights should be red (closed Gate) and two should be green (open Gate). The onTimeRV gives the duration for the green light, while offTimeRV gives the duration of the red light. Both are Sharp distributions that give a constant value. The group method swaps the on and off times, based upon whether its number in the group is even or odd. Lights are positioned at the back of the intersection, e.g., the light for traffic coming for the North source "s1N" is the bottom left light in Figure 17.3.

```scala
val light = Gate.group (this, nStop, onTimeRV, offTimeRV, (800, 480),
  ("l1N", queue(0), (0, 0)), // traffic from North
  ("l1E", queue(1), (0, -30)),
  ("l1S", queue(2), (30, -30)),
  ("l1W", queue(3), (30, 0)))
```

After making it through the intersection, traffic continues to its designated Sink. For example, cars created by source s1N will be terminated by sink k1S.

```scala
val sink = Sink.group ((830, 250), ("k1N", (0, 0)),
  ("k1E", (200, 230)),
  ("k1S", (-30, 400)), // end for North traffic
  ("k1W", (-230, 200)))
```

For each Source, two Routes are created: one from the Source to the WaitQueue and the other from the Gate to the Sink.

```scala
val road = ListBuffer [Route] ()
for i <- source.indices do
  road += Route ("ra" + i, 2, source(i), queue(i), moveRV)
end for
for i <- source.indices do
  road += Route ("rb" + i, 2, light(i), sink((i + 2) % 4), moveRV)
end for
```

addComponents (source, queue, light, sink, road.toList)
In total, there are 16 nodes: four Sources, four WaitQueues, four Gates and four Sinks. In addition, there are 8 edges: all Routes with two lanes each (so underlying there are 16 Transports).

![Figure 17.3: Animation of Network Diagram of Vehicle Traffic Model](image)

For this simulation, Cars are the actors moving along the roads (may be thought of as autonomous vehicles or car-driver combinations). The behavior of a car depends on the direction it is traveling and is specified by its subtype.

```scala
case class Car () extends SimActor ("c", this):
  def act (): Unit =
    val i = subtype // from North (0), East (1), South (2), West (3)
    val l = laneRV.igen // randomly select lane l
    road(i).lane(l).move ()
    if light(i).shut then queue(i).waitIn ()
    road(i + 4).lane(l).move () // add 4 for next segment
    sink((i + 2) % 4).leave ()
  end act

end Car
```
The TrafficModel may be runs as follows:

```scala
> runMain scalation.simulation.process.example_1.runTraffic

@main def runTraffic (): Unit = new TrafficModel ()
```

### 17.3.19 Model_MBM Class

While the Model class works for both One-Shot Simulation (OSS) and the Method of Independent of Replications (MIR), the Model_MBM class is required for the Method of Batch Means (MBM). Although this method involves a single replication/simulation run, the single long run is divided into multiple batches in order to provide statistics. MIR and MBM provide complete statistics while OSS cannot, see the Simulation Output Analysis Chapter for details.

**Class Methods:**

* @param name the name of the simulation model
* @param batches the number of batches to run
* @param batchSz the size of each batch
* @param animating whether to animate the model
* @param aniRatio the ratio of simulation speed vs. animation speed
* @param full generate a full report with both sample and time-persistent statistics
*
```scala
class Model_MBM (name: String, val batches: Int = 10, batchSz: Int = 100, animating: Boolean = true, aniRatio: Double = 1.0, full: Boolean = true)
  extends Model (name, 1, animating, aniRatio, full):
override def act (): Unit =
```

Notice that the Model_MBM class extends the Model class reusing all of methods, except its act method.

### 17.3.20 Exercises

1. Explain why the act method cannot be just a regular method/function call.

2. Explain what happens in the inner scheduling loop of the Model class. For the BankModel, suppose there are three coroutines/threads, the director, customer1, and customer2. Using three vertical time lines, one for each coroutine, with the director in the middle, show the control transfers between them. Assume customer2 arrives before customer1 finishes its service.

3. **Wendy’s vs. McDonald’s Lines.** Given two servers, is it better to have a line/queue for each server or one common line for both servers. Create a Network Diagram for Wendy’s and another one for MacDonald’s.

4. Implement a Process-Interaction Simulation and analyze the waiting times and standard deviation of the waiting times for Wendy’s and MacDonald’s. Let $\lambda = 20 \text{ hr}^{-1}$ and $\mu = 12 \text{ hr}^{-1}$.
5. **Machine Shop.** Create a Network Diagram for the Machine Shop described in the Event-Oriented Simulation chapter.

6. Implement a Machine Shop simulation using Process-Interaction and determine which policy is better.

7. **Vehicle Traffic Simulation.** Create a Network Diagram for the stretch of US 101 at the Stanford exits from Willow Road to Oregon Expressway. See the Caltrans PeMS map.

8. Create a Process-Interaction Simulation of US 101 (Bayshore Freeway) at the Stanford exits. Data is recorded every five minutes at each of the sensors from Willow Road to Oregon Expressway giving 288 data points per day per sensor. Collect data for a portion the year 2021 for these sensors. Use it to calibrate and validate your models. Place **Sources** at the beginning of all road segments and on-ramps. Model the traffic inflow to the model using a Non-Homogeneous Poisson Process (NHPP) for each source that fits the data at that location. Place **Sinks** at the end of all road segments and off-ramps. Place **Junctions** at all road sensors. Use **Routes** for the road segments between sensors. There are typically four lanes Northbound and four lanes Southbound. Finally, use the data provided by Caltrans PeMS (traffic flow and speed) to measure the accuracy (sMAPE) of your simulation model. See [https://getd.lib.uga.edu/pdfs/peng_hao_201908_phd.pdf](https://getd.lib.uga.edu/pdfs/peng_hao_201908_phd.pdf) and [https://dot.ca.gov/programs/traffic-operations/mpr/pems-source](https://dot.ca.gov/programs/traffic-operations/mpr/pems-source).


10. Compare the results of **BankModel** for the event package and process package What happens as the number entities (customers) increases? What happens when the move method is replaced with the jump method. How do these results compare to those from Queueing Theory?
17.4 Agent-Based Simulation

Agent-Based Simulation (ABS) may be viewed as a cousin of Process-Interaction Simulation. An important enhancement provided by ABS is to provide a richer structure for actors to interact. For example, in a traffic simulation, it may be useful for a Car to know about the cars ahead and behind, as they may influence what the car does. Although this can done with process-interaction, it is up to model developer to create all the code to handle this. An ABS system should provide a framework that facilitates enriched interactions, reducing the burden of the simulation model developer. To reflect the enhanced capabilities of actors, including greater knowledge of its environment and other actors, they are typically named agents.

For simplicity, we focus on ABS as a form of time-based simulation (discrete-time or discrete-event) with event causality and a time-advance mechanism, but do not consider the more general Agent-Based Modeling (ABM) that may run multiple autonomous agents without controlled causal ordering of events, see [113] for a discussion.

In this context, the increased flexibility provided by Agent-Based Simulation partially derives from the capabilities/properties of agents. Desirable characteristics for agents include the following [61, 62, 60]:

1. An agent needs to be identifiable, self-contained, and active. Although actors in the process interaction paradigm share this, the event-scheduling paradigm does not as entities are passive and their logic/behavior specification is scattered among multiple event routines.

2. Agents have some level of autonomy. An agent should have the ability to sense its environment, make decisions and act accordingly. An example where this is not the case would be a SimActor whose script includes no parameterization or decision making (e.g., if statements). An agent should be able to Observe-Decide-Act [33].

3. Agents have the ability to interact with other agents. An example, in a Vehicle Traffic simulation, would be a car using a car-following rule/model which influences the driver’s speed and gap to the car in front. The agent must be aware of its neighborhood to put a car-following rule/model into play. Some models may require move detailed interaction between agents, e.g., may require a communication protocol.

4. An agent is situated in an environment and interacts with its local environment. As agents can move around in their environment typically they will be given coordinates. Although coordinates were given for process-interaction, they were added for animation and providing a real interpretation of the coordinates is up to the model developer. An ABS system should directly support this capability.

5. It is also useful to provide support for agents to learn, for example, the model developer, could provide a set of possible rules for a car to change lanes. Support for learning would mean that cars can collect information and analyze it to improve there decision making. This allows the agents to adapt as the simulation continues. Improvement implies goals, for example, the car would prefer less (not more) travel time.

6. Resources may exist in the environments or within agents. For example, a server is often thought of as part of the environment in process-interaction simulations, but a more realistic or detailed model could represent each teller as an agent that does other thinks besides just serving bank customers, e.g., they work a shift, have lunch and take breaks.
In ScalaTion agents may access information about the simulated world via a Knowledge Graph. In particular, a spatial Property Graph (`PGraph`) is used to set up the components in the simulated world. See the `Property Graph` section in the Data Management Chapter.

The vertices in the `PGraph` represent resources that agents can work with. An agent has a position in the simulated world with 2D or 3D coordinates. These coordinates may be transformed to screen coordinates for purposes of animation (see the next section). The edges in the `PGraph` represent one-way connections between vertices.

### 17.4.1 SimAgent

A SimAgent is a dynamic entity that moves to vertices and along edges as the simulation progresses. Its location is recorded in terms of its topological coordinates. An agent is thought to be at a vertex or on an edge. In theory, vertices are points, but ScalaTion allows them to take up space and measures distance as the distance from the center of the vertex. Distance along an edge is given by the distance from the beginning of the edge where it connects to the from vertex to the agent. In animation, an agent is represented as a token that moves along the graph. Due to the small size of vertices, tokens within may be represented collectively. Tokens on edges are represented as circles moving along edges. Topological coordinates are specified using the `Topological` trait.

```scala
@param elem the element in the graph (at a vertex or on an edge)
@param dist its distance along the segment

trait Topological (var elem: Element, var dist: Double) extends PartiallyOrdered [Topological]:

  def tryCompareTo [B >: Topological: AsPartiallyOrdered] (other: B): Option [Int] =
  def neighbors: VEC [Topological] = elem.tokens
  def neighbors (d: Double): VEC [Topological] =
  override def toString: String = s"Topological ($elem, $dist)"
```

While moving through the graph, an agent may interact with vertices as well as other agents. Agents moving along an edge may speed up, slow down or jump to a parallel edge (e.g., lane change in a vehicle traffic simulation).

```scala
val tokens = Set [Topological] () // topological objects/tokens at this edge
```

At a vertex, agents may passively wait (e.g., in wait queue), work with a server for a period of time, update their properties (e.g., the value of a part changes at each machine stage), wait for a traffic light to change color, choose which edge to follow upon leaving the vertex (e.g., the road to turn onto).

```scala
val tokens = Set [Topological] () // topological objects/tokens at this vertex
```
As expected, agents play a more central role in Agent-Based Simulation compared to Process-Interaction Simulation where much of the specification of behavior/decision making is often delegated to the components/blocks in the simulation.

A Property Graph (PGraph) consists of multiple vertex types (VertexType) and multiple edge types (EdgeTypes) as defined in the scalation.database.graph package.

17.4.2 Vertices

The vertices in a property graph are grouped into one or more vertex types. A vertex is Identifiable and situated is space using Spatial coordinates.

```scala
@param _name the name of this vertex ('name' from 'Identifiable')
@param prop maps vertex’s property names into property values
@param _pos the position (Euclidean coordinates) of this vertex ('pos' from 'Spatial)

class Vertex (_name: String, val prop: Property, _pos: VectorD = null)
    extends Identifiable (_name)
    with Spatial (_pos)
    with PartiallyOrdered [Vertex]
    with Serializable:
```

SCALATION’s ABS system defined in scalation.simulation.agent_based currently supports the following types of vertices:

- **Gate**: a gate is used to control the flow of agents, they cannot pass when it is shut.
- **Junction**: a junction is used to connect two transports.
- **Resource**: a resource provides services to agents (typically resulting in some delay).
- **Sink**: a sink consumes agents.
- **Source**: a source produces agents.
- **WaitQueue**: a FCFS wait-queue provides a place for agents to wait, e.g., waiting for a resource to become available or a gate to open.
- **WaitQueue_LCFS**: an LCFS wait-queue provides a place for agents to wait, e.g., waiting for a resource to become available or a gate to open.

These vertex types correspond to the component nodes in process-interaction. Their constructors and methods are more oriented towards allowing more flexibility and specificity from the agents.

17.4.3 Edges

The edges in a property graph are grouped into one or more edge types. An edge is Identifiable and situated in space using Spatial coordinates based one of the vertices it connects to.
@param _name the name of this edge ('name' from 'Identifiable')
@param from this edge’s source/from vertex
@param prop maps edge’s property names into property values
@param to this edge’s target/to vertex

class Edge (_name: String, val from: Vertex, val prop: Property, val to: Vertex)
  extends Identifiable (_name)
  with Spatial (if from == null then to.pos else from.pos)
  with Serializable:

SCALATion’s ABS system currently supports the following types of edges:

- **Link**: a link supports simple/quick movement of agents between closely connected vertices (e.g., a queue before a resource).
- **Route**: a route bundles multiple transports together (e.g., a two-lane, one-way street).
- **Transport**: a transport is used to move agents from one vertex to the next.

Note, closely connected nodes in the process-interaction were not required to have an edge between them, however, as SCALATion’s ABS system in built using PGraph, edges are required between vertices, hence the need for the **Link** class.

### 17.4.4 Bank Model

Specification of an Agent-Based Simulation model for a simple bank is not much different than for a Process-Interaction model.

**Initialize Model Constants.** As with any type of simulation model, constants or model parameters need to be specified, although in practice it is usually preferred to specify them in a scenario specification, e.g., in a `@main` top-level function.

```scala
val lambda = 6.0 // customer arrival rate (per hour)
val mu = 7.5 // customer service rate (per hour)
val nTellers = 1 // the number of bank tellers (servers)
```

**Create Random Variates (RVs).** A fourth random variate is added for jumping through the link between the wait queue and the resource containing the servers/tellers.

```scala
val iArrivalRV = Exponential (HOUR / lambda, stream)
val serviceRV = Exponential (HOUR / mu, (stream + 1) % N_STREAMS)
val moveRV = Uniform (4 * MINUTE, 6 * MINUTE, (stream + 2) % N_STREAMS)
val jumpRV = Uniform (0.4 * MINUTE, 0.6 * MINUTE, (stream + 3) % N_STREAMS)
```

**Create the Graph Model.** Specifying a graph model replaces the specification of the model components under process interaction. Notice the addition of the **Link** edge.

```scala
val entry_pos = Source.at (100, 290)
val cust_pos = VectorD (110, 290, 10, 10)
```
Specify Scripts for each Type of Simulation Agent. The script is similar to the specification of actor scripts for process interaction. The service, move and jump times may also pass directly into the `work`, `move` and `jump` methods. Notice the increased specification available to `SimAgent`s. For more complex models, this allows greater flexibility. In addition, an explicit `ping` method is required, as it is now not implicitly done by the resource, i.e., actions are now under control of the agent.

```scala
case class Customer () extends SimAgent("c", director.clock, this, cust_pos.copy):

def act (): Unit =
  toTellerQ.move (this)
  if teller.busy then tellerQ.waitIn (this) else tellerQ.noWait (this)
  toTeller.jump (this)
  teller.work (this)
  teller.release (this)
  tellerQ.ping ()
  toDoor.move (this)
  door.leave (this)
end act

end Customer
```

simulate ()
waitFinished ()
Model.shutdown ()

17.4.5 Vehicle Traffic Model

The advantages of Agent-Based Simulation are not apparent in models as simple a Bank Simulation Model. Vehicle Traffic Models, on the other hand, are complex enough to illustrate some advantages.

Accurate simulation models of vehicle traffic often use a car-following rule/model to determine speed and braking actions of a car (an agent) based on the car ahead. This seems simple enough, just maintain a reference to the car ahead. Several complexities may ensue:

- The car ahead exits: the exiting car may then link the car behind (you) to the car ahead of itself, as in deletion from a doubly linked list.
- Your car wants to change lanes: where is the new car to follow, it needs to be efficiently found, by searching the local road segment.
- Your car wants to turn onto another road: the car needs to know where it is in the graph. The intersection is a vertex, the chosen road is an edge and the car starts on a particular road segment.
The following Car class includes an ability to change lanes. This requires a more flexible move method, where the second parameter indicates the fraction of the length of a Route to move along. Midway, a new lane \( l_2 \) is determined and passed to the changeLane method. The car continues along lane \( l_2 \) to the traffic light. If the light is green, then it continues through, otherwise it waits in the queue. After getting through the light it continues to its Sink.

```scala
case class Car () extends SimAgent ("c", director.clock, this):
  def act (): Unit =
    val i = subtype // from North (0), East (1), South (2), West (3)
    val l = laneRV.igen // randomly select lane 1
    road(i).lane(l).move (this, 0.5) // move half way down lane 1
    val l2 = (l + 1) % 2 // index of other lane
    road(i).changeLane (this, l, l2) // change to lane l2
    road(i).lane(l2).move (this, 0.5) // move the rest of the way down lane 12
    if light(i).shut then queue(i).waitIn (this) // stop and wait for red light
    road(i + 4).lane(l2).move (this) // add 4 for next segment
    sink((i + 2) % 4).leave (this) // end at this sink
  end act
end Car
```

17.4.6 Hybrid Models

Hybrid simulation techniques, although more complex, allow the most appropriate modeling technique to be applied to parts of an overall simulation study. A comprehensive example is given in “Application of Mixed Simulation Method to Modelling Port Traffic,” by Ehiagwina Omoforma Augustine, 2021 [25]. [https://researchonline.ljmu.ac.uk/id/eprint/15592/1/2021AugustineEhiagwinaPhD20.pdf](https://researchonline.ljmu.ac.uk/id/eprint/15592/1/2021AugustineEhiagwinaPhD20.pdf)

17.4.7 Exercises

1. Compare the software available for Agent-Based Simulation.

2. Discuss the simulation world-views/paradigms used for modeling port traffic in the following dissertation, “Application of Mixed Simulation Method to Modelling Port Traffic.”

3. What data structures and algorithms can used in the changeLane method to efficiently find the car ahead?

4. Design an Agent-Based Simulation for Vehicle Traffic Forecasting.

5. Develop an Agent-Based Simulation for Vehicle Traffic Forecasting. Add the capabilities discussed in this section to provide a more realistic simulation compared to Process-Interaction. See

6. Design the Emergency Department Simulation as an Agent-Based Simulation.

7. Develop the Emergency Department Simulation as an Agent-Based Simulation and model doctors and nurses as agents rather than resources as was indicated for the process-interaction approach.

8. Design an Agent-Based Simulation for analyzing the COVID-19 Pandemic.


17.5 Animation

Process-Interaction and Agent-Based Simulation are ideally suited for animation. The environment can be given largely by displaying nodes and edges of a graph. The dynamics can be displayed by creating, moving and destroying tokens. The locations and actions of an actor or agent over its lifetime can be depicted in the animation.

17.5.1 2D Animation

In the JVM world (includes Java, Scala and several other languages), simple 2D animation can be accomplished using **awt** and **swing**. A richer graphics library is provided by **JavaFx**. Motion involves both space and time, so Java’s **Thread** class and **Runnable** interface are also used for animation.

Basics of 2D Animation

Java’s abstract window toolkit (**awt**) and **swing** packages support simple 2D animations. The example below illustrates this by drawing a large blue circle and having a small red ball continually trace the circle. The drawing **Canvas** is a class that extends **JPanel** that is an inner class within a **JFrame**. The **run** method (1) updates the ball coordinates, (2) sleeps for **tau** millisecond, e.g., 20 milliseconds corresponds to 50 frames per second (generally fast enough for humans to see motion as smooth), and (3) repaints the canvas by calling **paint**.

```scala
class SimpleAnimator (title: String)
  extends JFrame (title) with Runnable:
    private val dim = new Dimension (600, 500) // the size of the canvas
    private val tau = 20 // operate at 50 Hz
    private val circle = new Ellipse2D.Double (200, 200, 200, 200) // the circle to traverse
    private val ballPos = new Point2D.Double (0, 300) // ball position
    private val ball = new Ellipse2D.Double () // the moving ball

    getContentPane ().add (new Canvas ())
    setLocation (100, 100)
    setSize (dim)
    setVisible (true)
    setDefaultCloseOperation (EXIT_ON_CLOSE)
    new Thread (this).start ()

class Canvas extends JPanel:
  override def paintComponent (gr: Graphics): Unit = ???
end Canvas

def run (): Unit =
  var theta = 0.0
  while true do
    theta += 0.05
    ballPos.x = 300 + 100 * cos (theta)
```

573
ballPos.y = 300 + 100 * sin (theta)
println (s"ballPos = \$ballPos")

try Thread.sleep (tau)
catch case ex: InterruptedException => println ("SimpleAnimator.run: sleep failed")
end try
repaint ()
end while
end run
end SimpleAnimator

The call to repaint will cause execution of the paintComponent method. Based on the new ball coordinates determined by the angle theta, the ball will move a few pixels each time the canvas is repainted. The setFrame (x, y, w, h) method is used to reset the ball coordinates: x-coordinate, y-coordinate, width, and height of the ball. Notice that draw draws the shape’s boundary, while fill fills the shape in with color.

override def paintComponent (gr: Graphics): Unit =
  super.paintComponent (gr)
  val gr2 = gr.asInstanceOf [Graphics2D] // use hi-res

  gr2.setPaint (Color.blue) // blue circle
g2.draw (circle)

  gr2.setPaint (Color.red) // read ball
  ball.setFrame (ballPos.x - 10, ballPos.y - 10, 20, 20)
gr2.fill (ball)
end paintComponent

The x and y coordinates specify the top-left position for the bounding box of the ball. As the ballPos is intended to be the center of the ball, half the width/height (10) must be subtracted to center the ball on the large red circle.

This example animation may be run as follows:

@main def runSimpleAnimator (): Unit = new SimpleAnimator ("SimpleAnimator")

SCALAION avoids the direct use of any graphics framework to facilitate changing frameworks as the technology evolves. Consequently, the scala2d package is used to insulate code from the particulars of any graphics framework. The SimpleAnimation2 class shows the minimal changes required to switch from direct use of Java graphics libraries to the use of the scala2d package.

2D Animation in SCALAION

The Model class in the event, process, and agent_based packages import the following from the scalation.animation package: AnimateCommand, CommandType, and DgAnimator.

The AnimateCommand class provides a data structure for holding animation command specifications.
case class AnimateCommand (action: CommandType, eid: Int, shape: Shape, label: String,
   primary: Boolean, color: Color, pts: Array[Double], time: Double,
   from_eid: Int = -1, to_eid: Int = -1):

   def compare (command2: AnimateCommand) = time compare command2.time
   def show (array: Array[Double]) =
   override def toString =

   The CommandType enumeration specifies the types of commands passed from a simulation engine to the animation engine.

   enum CommandType (val name: String):

   case CreateNode extends CommandType ("CreateNode")
   case CreateEdge extends CommandType ("CreateEdge")
   case CreateToken extends CommandType ("CreateToken")
   case DestroyNode extends CommandType ("DestroyNode")
   case DestroyEdge extends CommandType ("DestroyEdge")
   case DestroyToken extends CommandType ("DestroyToken")
   case MoveNode extends CommandType ("MoveNode")
   case MoveToken extends CommandType ("MoveToken")
   case MoveToken2Node extends CommandType ("MoveToken2Node")
   case MoveTokens2Node extends CommandType ("MoveTokens2Node")
   case MoveToken2Edge extends CommandType ("MoveToken2Edge")
   case ScaleNode extends CommandType ("ScaleNode")
   case ScaleToken extends CommandType ("ScaleToken")
   case ScaleTokensAt extends CommandType ("ScaleTokensAt")
   case SetPaintNode extends CommandType ("SetPaintNode")
   case SetPaintEdge extends CommandType ("SetPaintEdge")
   case SetPaintToken extends CommandType ("SetPaintToken")
   case TimeDilation extends CommandType ("TimeDilation")

   end CommandType

   The DgAnimator class is an animation engine for animating graphs. It contains a Panel that provides a paintComponent method for displaying the nodes, edges and tokens of the graph.

   @param _title the title for the display frame
class DgAnimator (_title: String, fgColor: Color = black, bgColor: Color = white,
                   aniRatio: Double = 1.0)
    extends VizFrame (_title, null, 1200, 800) with Runnable:

    def saveImage (fname: String): Unit = writeImage (fname, this)
    def run (): Unit =
    def animate (tStart: Double, tStop: Double): Unit =
    def invokeNow (cmd: AnimateCommand): Unit =
    def getCommandQueue: ConcurrentLinkedQueue [AnimateCommand] = cmdQ

The run method repeatedly executes animation commands, sleeps and repaights the drawing canvas/panel.
The length of the sleep is determined by the time gap between commands based on their values from the
simulation clock as well as the aniRatio.

17.5.2 3D Animation

C++ and C# are commonly languages used for 3D animations. There are several libraries available in the
JVM world as well. JavaFx has limited support for 3D. Currently, LWJGL and LibGDX are widely used by
the JVM community.

Further Reading

• Introduction to Computer Graphics and Animation, by F. E. Ekpenyong, National Open University
  pdf

17.5.3 Exercises

1. Create an animation of the trajectory of a golf ball using the equations given in the State Space Models
   chapter. Add a tracer to see the path taken by the golf ball.

2. Redesign the scala2d package and call it scala2df to use JavaFx rather than awt and swing.

3. Design a scala3d package that uses JavaFx with its limited 3D capabilities.

4. Design a scala3d package that uses LWJGL.

5. Design a scala3d package that uses LibGDX.

6. Add the ability to zoom-in and zoom-out for animations.
Chapter 18

Simulation Output Analysis

Unlike some modeling techniques, a simulation run will produce one result, the next run another result. Individually these results may be misleading as results depend on the combination of random variates generated. Simulation true power comes from generating several results and then performing statistical analysis on these results. In other words, the simulation outputs need to be analyzed.

Many types of output may be analyzed. For example, many simulation studies are interested in reducing waiting times. For such studies, one may define,

\[ y_i = \text{waiting time of the } i^{th} \text{ customer} \] (18.1)

A goal of the simulation study would then be to produce accurate point and interval estimates of the customer waiting times.

18.1 Point and Interval Estimates

As discussed in the Probability Chapter, given a sample of size \( m \), the sample mean (\( \bar{\mu} = \hat{\mu} \)) is computed as follows:

\[ \bar{\mu} = \frac{1}{m} \cdot \sum_{i=0}^{m-1} y_i \] (18.2)

The sample variance is computed as follows:

\[ \hat{\sigma}^2 = \frac{||y - \bar{\mu}||^2}{m - 1} = \frac{1}{m - 1} \sum_{i=0}^{m-1} (y_i - \bar{\mu})^2 \] (18.3)

Typically, simulations are used to study average behavior, so the focus is on the sample mean. To collect statistics on average behavior, several means need to be collected and it is important that they be independent (or at least not highly correlated). There are two common methods used to achieve this, the Method of Independent Replications and the Method of Batch Means.

Both methods will produce \( n \) means: \( \bar{\mu}_0, \bar{\mu}_1, \ldots, \bar{\mu}_{n-1} \). From these a grand mean will be calculated.

\[ \bar{\bar{\mu}} = \frac{1}{n} \sum_{i=0}^{n-1} \bar{\mu}_i \] (18.4)
The grand mean can be used to estimate average behavior or performance characteristics, such as average waiting times. In addition to using the grand mean as a point estimate, it is common practice to obtain an interval estimate \([\bar{\mu} - ihw, \bar{\mu} + ihw]\), where \(ihw\) is the interval half width.

To create an interval estimate in the form of a confidence interval, we need to determine the variability or variance in the point estimate \(\bar{\mu}\).

\[
\mathbb{V}[\bar{\mu}] = \mathbb{V} \left[ \frac{1}{n} \sum_{i=0}^{n-1} \bar{\mu}_i \right] = \frac{1}{n^2} \sum_{i=0}^{n-1} \mathbb{V}[\bar{\mu}_i] = \frac{\sigma^2_{\bar{\mu}}}{n} \tag{18.5}
\]

Consequently, the following is an estimate for the variance of \(\bar{\mu}\).

\[
\hat{\sigma}^2_{\bar{\mu}} = \frac{\hat{\sigma}^2_{\mu}}{n} \tag{18.6}
\]

Therefore the interval half width is

\[
ihw = t^* \frac{\hat{\sigma}_{\bar{\mu}}}{\sqrt{n}} \tag{18.7}
\]

where \(t^*\) is a critical value from the Student’s t Distribution. The interval estimate is then

\[
[\bar{\mu} - t^*\frac{2\sigma_{\bar{\mu}}}{\sqrt{n}}, \bar{\mu} + t^*\frac{2\sigma_{\bar{\mu}}}{\sqrt{n}}] \tag{18.8}
\]

Suppose the true mean waiting time \(\mu\) can be computed (e.g., from Queueing Theory), then the probability the confidence interval will contain it corresponds to the confidence level chosen (and \(t^*\) is determined by this and the degrees of freedom).
18.2 One-Shot Simulation

One-shot simulation happens when there a single simulation run and that run is treated as a whole (i.e., it is not broken into multiple batches). This is the simplest way to run a simulation and it is good idea to convince yourself that the model is running correctly in this mode before moving on a method more useful for simulation output analysis.

The process package contains several example models that are placed in sub-directories according how they are set up for output analysis and validation.

- **example_1**: One-Shot Simulation (OSS)
  During model development, OSS may be used since model execution takes less time and animation is on by default, thus giving the model developer quick feedback on the correctness of the model. The default settings are 1 replication, animation on, uses move method, extends Model (no batching). The move method gives smooth motion for better animation.

- **example_MIR**: Method of Independent Replications (MIR)
  OSS will report a mean, but not a standard deviation, since that requires at least 2 simulation runs (or replications). The default settings are 10 replications, animation off, uses jump method, and extends Model (no batching). The jump method gives less overhead in Transports which for many models may provide more accurate statistics that correspond better those in the event package. The developer needs to decide depending on what they are modeling. For example, one should use move and not jump for traffic simulations, but use jump over move for tandem queues where the time to move from one queue to next is negligible. Note, using move with OSS would likely be preferred as it will make it easier to track the entities while viewing the animation.

- **example_MBM**: Method of Batch Means (MBM)
  Although MIR can provide more useful statistics, it tends to be less effective for studying long-term or steady-state behavior. MBM has one long simulation run (replication), but divides it up into several batches. Each batch produces a batch mean and since there are several of them, standard deviations can be computed. The default settings are 1 replication, animation off, uses jump method, extends Model_MBM (batching). The Model_MBM class extends the Model class with batching capabilities (see the section on MBM for details).
18.3 Method of Independent Replications (MIR)
18.4 Method of Batch Means (MBM)
18.5 Simulation Model Validation
18.6 Exercises
Appendices
Appendix A

Optimization in Data Science

As discussed in earlier chapters, when matrix factorization cannot be applied for determining optimal values for parameters, an optimization algorithm will often need to be applied. This chapter provides a quick overview of optimization algorithms that are useful for data science. Note that the notation in the optimization field differs in that we now focus on optimizing the vector $x$ rather than the parameter vector $b$.

Many optimization problems may be formulated as restricted forms of the following,

$$
\text{minimize } f(x) \\
\text{subject to } g(x) \leq 0 \\
h(x) = 0
$$

where $f(x)$ is the objective function, $g(x) \leq 0$ are the inequality constraints, and $h(x) = 0$ are the equality constraints. Consider the example below.

$$
\text{minimize } f(x) = (x_1 - 4)^2 + (x_2 - 2)^2 \\
\text{subject to } g(x) = [x_1 - 3, x_2 - 1] \leq 0 \\
h(x) = x_1 - x_2 = 0
$$

If we ignore all the constraints, the optimal solution is $x = [4, 2]$ where $f(x) = 0$, while enforcing the inequality constraints makes this solution infeasible. The new optimal solution is $x = [3, 1]$ where $f(x) = 2$. Finally, the optimal solution when all constraints are enforced is $x = [1, 1]$ where $f(x) = 10$. Note, for this example there is just one equality constraint that forces $x_1 = x_2$. 
A.1 Gradient Descent

One the simplest algorithms for unconstrained optimization is Gradient Descent. Imagine you are in a mountain range at some point \( x \) with elevation \( f(x) \). Your goal is to find the valley (or ideally the lowest valley). Look around (assume you cannot see very far) and determine the direction and magnitude of steepest ascent. This is the gradient.

Using the objective/cost function from the beginning of the chapter,

\[
\text{minimize } f(x) = (x_1 - 4)^2 + (x_2 - 2)^2
\]

the gradient of the objective function \( \nabla f(x) \) is the vector formed by the partial derivatives \( \left[ \frac{\partial}{\partial x_1}, \frac{\partial}{\partial x_2} \right] \)

\[
\nabla f(x) = [2(x_1 - 4), 2(x_2 - 2)]
\]

In its most elemental form the algorithm simply moves in the direction that is opposite to the gradient \(-\nabla f(x)\) and a distance determined by the magnitude of the gradient. Unfortunately, at some points in the search space the magnitude of the gradient may be very large and moving that distance may result in divergence (you keep getting farther away from the valley). One solution is to temper the gradient by multiplying it by a learning rate \( \eta \) (tunable hyper-parameter typically smaller than one). Using a tuned learning rate, update your current location \( x \) as follows:

\[
x = x - \eta \nabla f(x)
\]  \hspace{1cm} (A.1)

Repeat this process until a stopping rule signals sufficient convergence. Examples of stopping rules include stop when the change to \( x \) or \( f(x) \) becomes small or after the objective function has increased for too many consecutive iterations/steps.

A.1.1 Line Search

Notice that the gradient is re-evaluated at every iteration/step and that it is unclear how far to move in the direction opposite the gradient (hence the need/annoyance of tuning the learning rate). Adding a line search may help with these issues. The idea is that the gradient gives you a direction to follow that may work well for awhile. Using a line search, you may move in that direction (straight line) so long as it productive. The line search induces a one dimensional function that reproduces the value of the original objective function along the given line.

One approach is to move along the line so long as there is sufficient decrease. Once this stops, re-evaluate the gradient and start another major iteration. An example of such an algorithm is the Wolfe Line Search. An alternative when you are confident of the extent of line search (upper limit on the range to be considered) is to use Golden Section Search that iteratively narrows down the search from the original extent.

The problem of learning rate is still there to some degree as the line search algorithms have step size as hyper-parameter. Of course, more complex variants may utilize adaptive learning rates or step sizes.
A.1.2

Application to Data Science

The gradient descent algorithm may be applied to data science, simply by defining an appropriate objective/cost function. Since the goal is often to minimize the sum of squared errors sse or some similar Quality
of Fit (QoF) measure, it may be used for the objective function. For a Perceptron, the equation has been
developed for h(b)
h(b) = (y − f (Xb) · (y − f (Xb)
in which case the gradient is
∇h(b) = − X t [f 0 (Xb) ]

(A.2)

where  = y − f (Xb).
For each epoch, the parameter vector b is update according to equation 16.1.
b = b − η∇h(b)

A.1.3

(A.3)

Exercises

1. Write a ScalaTion program to solve the example problem given above.
// function to optimize
def f(x: VectoD): Double = (x(0) - 4)~^2 + (x(1) - 2)~^2
// gradient of objective function
def grad (x: VectoD): VectoD = VectorD (?, ?)
val x
= new VectorD (2)
val eta = 0.1

// vector to optimize
// learning rate

for (k <- 1 to MAX_ITER) {
x -= grad (x) * eta
println (s"$k: x = $x, f(x) = ${f(x)}, lg(x) = ${lg(x)}, p = $p, l = $l")
} // for
2. Add code to collect the trajectory of vector x in a matrix z and plot the two columns in the z matrix.
val z = new MatrixD (MAX_ITER, 2)
z(k-1) = x.copy
new Plot (z.col(0), z.col(1)

// store x’s trajectory

585


A.2 Stochastic Gradient Descent

Continuing for a Perceptron, starting with $h(b)$

$$h(b) = (y - f(Xb)) \cdot (y - f(Xb))$$

an estimate of the gradient is computed for a limited number of instances (a batch). Several non-overlapping batches are created simultaneously by taking a random permutation of the row indices of data/input matrix $X$. The permutation is split into $n_B$ batches. Letting $i_B$ be the indices for the $i^{th}$ batch and $X[i_B]$ be the projection of matrix $X$ onto the rows in $i_B$, the estimate for the gradient is simply

$$\nabla h(b) = -X[i_B] \cdot f'(X[i_B]b) \epsilon \quad (A.4)$$

where $\epsilon = y[i_B] - f(X[i_B]b)$. Using the definition of the delta vector

$$\delta = -f'(X[i_B]b) \epsilon$$

the gradient becomes

$$\nabla h(b) = X[i_B] \delta$$

For each epoch, $n_B$ batches are created. For each batch, the parameter vector $b$ is update according to equation 16.1, using that batch’s estimate for the gradient.

$$b = b - \eta \nabla h(b) = b - X[i_B] \delta \eta \quad (A.5)$$

The corresponding ScalaTion code is in the Optimizer_SGD object.

```scala
def optimize (x: MatrixD, y: VectorD, b: VectorD, 
  eta_: Double = hp.default("eta"), 
  bSize: Int = hp.default("bSize").toInt, 
  maxEpochs: Int = hp.default("maxEpochs").toInt, 
  f1: AFF = f_sigmoid): (Double, Int) =
{
  val idx = VectorI.range (0, x.dim1) // instance index range
  val permGen = PermutedVecI (idx, ranStream) // permutation vector generator
  val nB = x.dim1 / bSize // the number of batches
  val stop = new StoppingRule () // rule for stopping early
  var eta = eta_ // set initial learning rate
  for (epoch <- 1 to maxEpochs) { // iterate over each epoch
    val batches = permGen.igen.split (nB) // permute index, split into batches
    for (ib <- batches) b -= updateWeight (x(ib), y(ib)) // iteratively update vector b
    val sse = sseF (y, f1.fV (x * b)) // recompute sum of squared errors
    val (b_best, sse_best) = stop.stopWhen (b, sse)
    if (b_best != null) {
      b.set (b_best())
    }
  }
  (sse, epoch)
}
```
```
return (sse_best, epoch - UP_LIMIT)
} // if
if (epoch % ADJUST_PERIOD == 0) eta *= ADJUST_FACTOR // adjust the learning rate
} // for
```

The above code shows the double loop (over `epoch` and `ib`). The parameter vector `b` is updated for each batch by calling `updateWeight`. The rest of the outer simply looks for early termination based on a stopping rule and records the best solution for `b` found so far. The final part of the outer loop, increases the learning rate `eta` at the end of each adjustment period (as the algorithm get closer to an optimal solution, gradients shrink and may slow down the algorithm).

```java
def updateWeight (x: MatriD, y: VectoD): VectoD =
{
    val yp = f1.fV (x * b) // yp = f(Xb)
    val e = yp - y // negative error vector
    val d = f1.dV (yp) * e // delta vector

    val eta_o_sz = eta / x.dim1 // eta over current batch size
    x.t * d * eta_o_sz // return change in vector b
} // updateWeight
```

The above code shows the `updateWeight` nested method and the final line of the outer `optimize` method. The `updateWeight` method simply encodes the boxed equations from the Perceptron section: computing predicted output, the negative of the error vector, the delta vector, and a batch size normalized learning rate, and finally, returning the parameter vector `b` update. The final line of `optimize` simply returns the value of the objective function and number epochs used by the algorithm.
A.3 Stochastic Gradient Descent with Momentum

To better handle situations where the gradient becomes small or erratic, previous values of the gradient can be weighed in with the current gradient. Their contributions can be exponentially decayed, so that recent gradients have greater influence. These contributions may be collected via the gradient-based parameter updates to the parameter vector \( b \).

\[
b_{\text{up}} = \eta \nabla h(b) = X[i_{B}]^{t} \delta \eta \quad (A.6)
\]

Again for a Perceptron, one can include a momentum vector \( m_{o} \) to hold the weighted average of recent gradients. The momentum vector \( m_{o} \) is then simply the sum of the old momentum decayed by parameter \( \beta \) and \( b_{\text{up}} \).

\[
m_{o} = \beta m_{o} + b_{\text{up}} \quad (A.7)
\]

Then the parameters are updated as follows:

\[
b -= m_{o} \quad (A.8)
\]

If \( \beta \) is zero, that algorithm behaves the same as Stochastic Gradient Descent. At the other extreme, if \( \beta \) is 1, there is no decay and all previous gradients will weigh in, so eventually the new gradient value will have little impact and the algorithm will become oblivious to its local environment. In ScalaTion, \( \text{BETA} \) is set to 0.9, but can easily be changed.

To add momentum into the code, the `updateWeight` method from the last section needs to be slightly modified.

```scala
def updateWeight (x: MatrixD, y: VectorD): VectorD = 
{
    val yp = f1.fV (x * b) // yp = f(Xb)
    val e = yp - y       // negative of the error vector
    val d = f1.dV (yp) * e // delta vector for y
    val eta_o_sz = eta / x.dim1 // eta over the current batch size
    val bup = x.t * d * eta_o_sz // gradient-based change in input-output weights
    val mo = mo * BETA + bup // update momentum
    mo // return momentum
}
```

See the Optimizer `SGDM` for more coding details.
A.4 Method of Lagrange Multipliers

The Method of Lagrange Multipliers (or Lagrangian Method) provides a means for solving constrained optimizations problems. For optimization problems involving only one equality constraint, one may introduce a Lagrange multiplier \( \lambda \). At optimality, the gradient of \( f \) should be orthogonal to the surface defined by the constraint \( h(x) = 0 \), otherwise, moving along the surface in the opposite direction to the gradient \( -\nabla f(x) \) for minimization) would improve the solution. Since the gradient of \( h \), \( \nabla h(x) \), is orthogonal to the surface as well, this implies that the two gradients should only differ by a constant multiplier \( \lambda \).

\[-\nabla f(x) = \lambda \nabla h(x)\]  \hspace{1cm} (A.9)

In general, such problems can solved by defining the Lagrangian

\[L(x, \lambda) = f(x) - \lambda \cdot h(x)\]  \hspace{1cm} (A.10)

where \( \lambda \) is a vector of Lagrange multipliers. When there is a single equality constraint, this becomes

\[L(x, \lambda) = f(x) - \lambda h(x)\]

Taking the gradient of the Lagrangian w.r.t. \( x \) and \( \lambda \) yields a vector of dimension \( n + 1 \).

\[\nabla L(x, \lambda) = [\nabla f(x) - \lambda \nabla h(x), h(x)]\]

Now we may try setting the gradient to zero and solving a system of equations.

A.4.1 Example Problem

The Lagrangian for the problem given at the beginning of the chapter is

\[L(x, \lambda) = (x_1 - 4)^2 + (x_2 - 2)^2 - \lambda (x_1 - x_2)\]

Computation of the gradient \([\frac{\partial}{\partial x_1}, \frac{\partial}{\partial x_2}, \frac{\partial}{\partial \lambda}]\) of the Lagrangian yields the following three equations,

\[-2(x_1 - 4) = \lambda\]
\[-2(x_2 - 2) = -\lambda\]
\[x_1 - x_2 = 0\]

The first two equations are from the gradient w.r.t. \( x \), while the third equation is simply the constraint itself \( h(x) = 0 \). The equations may be rewritten in the following form.

\[2x_1 + \lambda = 8\]
\[2x_2 - \lambda = 4\]
\[x_1 - x_2 = 0\]

This is a linear system of equations with 3 variables \([x_1, x_2, \lambda]\) and 3 equations that may be solved, for example, by LU Factorization. In this case, the last equation gives \( x_1 = x_2 \), so adding equations 1 and 2 yields \( 4x_1 = 12 \). Therefore, the optimal value is \( x = [3, 3] \) with \( \lambda = 2 \) where \( f(x) = 2 \).
Adding an equality constraint is addressed by adding another Lagrange multiplier, e.g., 4 variables \([x_1, x_2, \lambda_1, \lambda_2]\) and 4 equations, two from the gradient w.r.t. \(x\) and one for each of the two constraints.

Linear systems of equations are generated when the objective function is at most quadratic and the constraints are linear. If this is not the case, a nonlinear system of equations may be generated.
A.5 Karush-Kuhn-Tucker Conditions

Introducing inequality constraints makes the situation a little more complicated. A generalization of the Method of Lagrange Multipliers based on the Karush-Kuhn-Tucker (KKT) conditions is needed. For minimization, the KKT conditions are as follows:

\[-\nabla f(x) = \alpha \cdot \nabla g(x) + \lambda \cdot \nabla h(x)\]  \hspace{1cm} (A.11)

The original constraints must also hold.

\[g(x) \leq 0 \quad \text{and} \quad h(x) = 0\]

Furthermore, the Lagrange multipliers for the inequality constraints \(\alpha\) are themselves constrained to be non-negative.

\[\alpha \geq 0\]

When the objective function is at most quadratic and the constraints are linear, the problem of finding an optimal value for \(x\) is referred to a Quadratic Programming. Many estimation/learning problems in data science are of this form. Beyond Quadratic Programming lies problems in Nonlinear Programming. Linear Programming (linear objective function and linear constraints) typically finds less use (e.g., Quantile Regression) in estimation/learning, so it will not be covered in this Chapter, although it is provided by ScalaTion.

A.5.1 Active and Inactive Constraints
A.6 Augmented Lagrangian Method

The Augmented Lagrangian Method (also known as the Method of Multipliers) takes a constrained optimization problem with equality constraints and solves it as a series of unconstrained optimization problems.

\[
\begin{align*}
\text{minimize } & f(x) \\
\text{subject to } & h(x) = 0
\end{align*}
\]

where \( f(x) \) is the objective function and \( h(x) = 0 \) are the equality constraints.

In penalty form, the constrained optimization problem becomes.

\[
\begin{align*}
\text{minimize } & f(x) + \frac{\rho_k}{2} \| h(x) \|_2^2 \\
\text{subject to } & h(x) = 0
\end{align*}
\]

where \( k \) is the iteration counter. The square of the Euclidean norm indicates to what degree the equality constraints are violated. Replacing the square of the Euclidean norm with the dot product gives.

\[
\begin{align*}
\text{minimize } & f(x) + \frac{\rho_k}{2} h(x) \cdot h(x) \\
\text{subject to } & h(x) = 0
\end{align*}
\]

The value of \( \rho_k \) increases (e.g., linearly) with \( k \) and thereby enforces the equality constraints more strongly with each iteration.

An alternative to minimizing \( f(x) \) with a quadratic penalty is to minimize using the Augmented Lagrangian \( L(x, \rho_k, \lambda) \).

\[
L(x, \rho_k, \lambda) = f(x) + \frac{\rho_k}{2} h(x) \cdot h(x) - \lambda \cdot h(x)
\]

where \( \lambda \) is the vector of Lagrange multipliers. After each iteration, the Lagrange multipliers are updated.

\[
\lambda = \lambda - \rho_k h(x)
\]

This method allows for quicker convergence without the need for the penalty \( \rho_k \) to become as large (see the exercises for a comparison of the Augmented Lagrangian Method with the Penalty Method). This method may be combined with an algorithm for solving unconstrained optimization problems (see the exercises for how it can be combined with the Gradient Descent algorithm). The method also can be extended to work inequality constraints.

A.6.1 Example Problem

Consider the problem given at the beginning of the chapter with the inequality constraint left out.

\[
\begin{align*}
\text{minimize } & f(x) = (x_1 - 4)^2 + (x_2 - 2)^2 \\
\text{subject to } & h(x) = x_1 - x_2 = 0
\end{align*}
\]

where \( x \in \mathbb{R}^2 \), \( f \) is the objective function and \( h \) is the single equality constraint. The Augmented Lagrangian for this problem is
\[ L(x, \rho_k, \lambda) = (x_1 - 4)^2 + (x_2 - 2)^2 + \frac{\rho_k}{2} (x_1 - x_2)^2 - \lambda (x_1 - x_2) \]  

(A.13)

The gradient of the Augmented Lagrangian \( \nabla L(x, \rho_k, \lambda) \) is made up of the following two partial derivatives.

\[
\frac{\partial}{\partial x_1} = 2(x_1 - 4) + \frac{\rho_k}{2} 2(x_1 - x_2) - \lambda \\
\frac{\partial}{\partial x_2} = 2(x_2 - 2) - \frac{\rho_k}{2} 2(x_1 - x_2) + \lambda
\]

The Lagrange multiplier updates becomes

\[
\lambda = \lambda - \rho_k (x_1 - x_2)
\]

The code in the exercises tightly integrates the Gradient Descent algorithm with the Augmented Lagrangian method by updating the penalty and Lagrange multiplier during each iteration.

### A.6.2 Exercises

1. Write a SCALATION program to solve the example problem given above.

   ```scala
   // function to optimize
   def f(x: VectoD): Double = (x(0) - 4)^2 + (x(1) - 2)^2
   // equality constraint to maintain
   def h(x: VectoD): Double = x(0) - x(1)
   // augmented Lagrangian
   def lg (x: VectoD): Double = f(x) + (p/2) * h(x)^2 - l * h(x)
   // gradient of Augmented Lagrangian
   def grad (x: VectoD): VectoD = VectorD (?, ?)
   val x = new VectorD (2) // vector to optimize
   val eta = 0.1 // learning rate
   val p0 = 0.25; var p = p0 // initial penalty (p = p0)
   var l = 0.0 // initial value for Lagrange multiplier
   for (k <- 1 to MAX_ITER) {
      l -= p * h(x) // comment out for Penalty Method
      x -= grad (x) * eta
      println (s"$k: x = $x, f(x) = ${f(x)}, lg(x) = ${lg(x)}, p = $p, l = $l")
      p += p0
   } // for
   
   2. Add code to collect the trajectory of vector \( x \) in a matrix \( z \) and plot the two columns in the \( z \) matrix.
val z = new MatrixD (MAX_ITER, 2)  // store x's trajectory
z(k-1) = x.copy
new Plot (z.col(0), z.col(1))

3. Compare the Augmented Langrangian Method with the Penalty Method by simply removing the Lagrange multiplier from the code.
A.7 Quadratic Programming

The `QuadraticSimplex` class solves Quadratic Programming (QP) problems using the Quadratic Simplex Algorithm. Given a constraint matrix $A$, constant vector $b$, cost matrix $Q$ and cost vector $c$, find values for the solution/decision vector $x$ that minimize the objective function $f(x)$, while satisfying all of the constraints, i.e.,

$$
\begin{align*}
\text{minimize} & \quad f(x) = \frac{1}{2} x \cdot Q x + c \cdot x \\
\text{subject to} & \quad g(x) = A x - b \leq 0
\end{align*}
$$

Before considering the type of optimization algorithm to use, we may simplify the problem by applying the KKT conditions.

$$
-\nabla f(x) = Q x + c = \alpha \cdot \nabla g(x) = \alpha \cdot A
$$

Adding the constraints gives the following $n$ equations and $2m$ constraints:

$$
\begin{align*}
Q x + c &= \alpha \cdot A \\
A x - b &\leq 0 \\
\alpha &\geq 0
\end{align*}
$$

These equations have two unknown vectors, $x$ of dimension $n$ and $\alpha$ of dimension $m$.

The algorithm creates an simplex tableau. This implementation is restricted to linear constraints $Ax \leq b$ and $Q$ being a positive semi-definite matrix. Pivoting must now also handle nonlinear complementary slackness.

**Class Methods:**

- `@param a` the $M$-by-$N$ constraint matrix
- `@param b` the $M$-length constant/limit vector
- `@param q` the $N$-by-$N$ cost/revenue matrix (second order component)
- `@param c` the $N$-length cost/revenue vector (first order component)
- `@param x_B` the initial basis (set of indices where $x_i$ is in the basis)

```scala
class QuadraticSimplex (a: MatrixD, b: VectorD, q: MatrixD, c: VectorD, 
                       var x_B: Array[Int] = null) 
  extends Error

  def setBasis (j: Int = N, l: Int = M): Array[Int] = 
  def entering (): Int = 
  def comple (l: Int): Int = 
  def leaving (l: Int): Int = 
  def pivot (k: Int, l: Int) 
```
def solve (): (VectorD, Double) =
def tableau: MatrixD = t
def primal: VectorD =
def dual: VectorD = null  // FIX
def objValue (x: VectorD): Double = (x dot (q * x)) * .5 + (c dot x)
def showTableau ()
A.8 Coordinate Descent
A.9 Conjugate Gradient
A.10 Quasi-Newton Method
A.11 Alternating Direction Method of Multipliers
A.12 Nelder-Mead Simplex
Appendix B

Functional Data Analysis

B.1 Basis Functions
B.2 Functional Smoothing
B.3 Functional Principal Component Analysis
B.4 Functional Regression
Appendix C

Parallel and Distributed Computing

C.1 MIMD - Multithreading
C.2 SIMD - Vector Instructions
C.3 Message Passing
C.4 Distributed Shared Memory
C.5 Microservices
C.6 Distributed Functional Programming
Bibliography


[92] Cosma Shalizi. Advanced data analysis from an elementary point of view, 2013.

612


