DEPARTMENT OF COMPUTER SCIENCE, UNIVERSITY OF GEORGIA

CSCI6900 Assignment 3: SVD on Spark

DUE: Tuesday, March 19 by 11:59:59pm

Out Thursday, March 5, 2015

1 OVERVIEW

Singular Value Decomposition (SVD) is a powerful dimensionality-reduction technique that is related to the problem of finding eigenvalues and eigenvectors. It is a method of principal components analysis (PCA) that, rather than explicitly computing the covariance matrix AA^T or $A^T A$ and performing a subsequent eigendecomposition, uses a closed-form expression to diagonalize any real $n \times m$ matrix A into three matrices:

$$A = U \Sigma V^T,$$

where U is the $n \times n$ matrix of left-singular vectors as columns, V^T is the $m \times m$ matrix of right-singular vectors as rows, and Σ is the $n \times m$ matrix of singular values along the main diagonal. For the relationship of SVD to the eigenvalue problem, see this page: `http://en.wikipedia.org/wiki/Singular_value_decomposition#Relation_to_eigenvalue_decomposition`.

SVD has many applications. In this assignment, we investigate its use in signal processing, specifically image denoising.

2 APACHE SPARK

This assignment will function as an introduction to Apache Spark. It is your decision whether to use Spark's Python, Scala, or Java bindings (Scala can invoke Java packages and routines. However, the syntax is slightly different; the example code below is pure Java and would need to be adapted to work in a Scala environment).

2.1 Dependencies

Regardless of the language you choose, there are two primary dependencies that must be met: a **method for reading PNG images**, and a **linear algebra library**.

2.1.1 Reading PNGs

Images in this assignment are black-and-white, 8-bit pixels. All of the languages available in Spark have utilities for reading and manipulating black-and-white PNG images.

- If you choose Scala or Java, you will need to import the `javax.imageio` package for reading PNGs.

```
import java.io.*;
import javax.imageio.*;

BufferedImage img = null;
try {
    img = ImageIO.read(new File("strawberry.jpg"));
} catch (IOException e) {}
```

- If you choose Python, you will either need to install SciPy/NumPy [1], or use Continuum's prebuilt Anaconda package [2].

```
import scipy.ndimage
img = scipy.ndimage.imread("strawberry.png")
```

2.1.2 Linear Algebra Libraries

You will also need a method for computing standard SVD (note: we are *not* yet using Spark's built-in distributed SVD solver; skip to the extra credit if you're interested in this), which often requires a specific matrix data structure. Again, all languages have libraries which can allow you to do this.

- If you choose Scala or Java, there are plenty of Java libraries to instantiate matrix objects from `double[][]` types that will comprise your image. Apache Commons is the most popular; however, ND4J and EJML are also exceptional [3].
- If you choose Python, the SciPy library has linear algebra functionality built in.

[1] `http://www.scipy.org/`
[2] `https://store.continuum.io/cshop/anaconda/`
[3] `http://en.wikipedia.org/wiki/List_of_numerical_libraries#Java`

3 Data

For this assignment, we are using a collection of images from EMPIAR–the Electron Microscopy Pilot Image Archive. This particular collection, EMPIAR-10015 [4], consists of 416,213 electron microscopy images of ribosomes in yeast, taken from various angles and orientations. Each image is 420×420 black and white (8-bit pixels), coming in at a total of 273GB in size. We will only be using the first 1000 images, available here:

`http://ridcully.cs.uga.edu/assignment3/yeast-ribosome-small.tar.gz`

You'll notice in looking at some of the images that they're very noisy; in fact, some may appear to be pure static. Given that each pixel is roughly 1 square angstrom (or $1 \times 10^{-10}\text{m}^2$), it is not surprising that there is a large amount of noise. Our task is implement a simple denoising application using dimensionality reduction.

4 Program

Your program will generate a low-dimensional representation of each image, theoretically removing some of the noise, and average these representations together to construct an "average" image. You will not be implementing a distributed dimensionality reduction algorithm; however, you will still need a dimensionality reduction library, depending on the language you use.

In effect, you will write a small program in Spark to parallelize the following pseudocode:

```
images = ... // List of images, parallelized as an RDD
k = ... // Small number of dimensions to retain
rows, cols = images[0].length, images[0][0].length
average = zeros(rows, cols)
for image in images:
    // Perform an SVD of the image using a linear algebra library
    U, S, Vt = svd(image)

    // Reconstruct the image using only the first k principal components
    R = U[:, :k] * S[:k, :k] * Vt[:k, :]

    // Keep a running increment of the reconstructed images
    average += R

// Normalize by the number of images to produce an average reconstructed image
average /= images.length
```

[4] `http://www.ebi.ac.uk/pdbe/emdb/empiar/entry/10015/`

You will also write a routine in Spark which computes the span of each pixel in the image dataset (see the questions in the next section).

5 Deliverables

When grading your assignments, I will use the **most recent commit in your repository *prior* to the submission deadline**. I should be able to clone your repository, change to the directory containing this assignment, and run your code. If my BitBucket username is `magsol`, my assignment should be in `magsol/assignment3`.

Please include a `README` file with 1) the command you issued to run your code, and 2) any known bugs you were unable to fix before the deadline. In addition, please provide answers to the following questions.

1. Run your program with

   ```
   --master local
   ```

   meaning only 1 core will be used. Record the runtime. Now run your program with

   ```
   --master local[*]
   ```

   meaning all cores will be used. Record the runtime. Compare the two runtimes, including how many cores your machine has. Is there a change in the runtime? If so, is it proportional to the number of cores? What would you expect if you used multiple cores across multiple machines?

2. Run your program using $k = 10$ (number of principal components) and save the final image reconstruction. Now run your program using $k = 50$ and $k = 100$ principal components (save the final image reconstructions). Finally, run your program but *comment out* the SVD step; simply compute the average image without any dimensionality reduction (again, save the final image). If we define A to be the average image with no dimensionality reduction, and A_k to be the average image using the top k principal components, compute the Frobenius norms $||A - A_k||_F$ for each value of k; this quantifies the reconstruction error between the "ground truth" image A and the reconstructed image A_k. Comment on the tradeoff of k versus reconstruction error. Is there an ideal choice of k?

3. If you read more on the yeast ribosomal EM dataset, you'll discover that many of the images are captured at different angles; thus, by averaging the images together, we are discarding important information related to the structure of the ribosome. If you're thinking "clustering might help here," you would be correct. In particular, since we are

dealing with images, spectral clustering would be the ideal candidate. As we've learned, generating the full pairwise affinity matrix is infeasible for large datasets, so instead we'll start with locality-sensitive hashing to subdivide the data into buckets that are *likely* to be similar. Recall the paper on distributed approximate spectral clustering [5], in particular the section on determine the *span* of each dimension of the data; here, the dimensions are pixels of the image. **Write a Spark routine that ranks the dimensions (pixels) of the dataset in order of their span**. Since each image is 420 × 420, there are 420^2 dimensions (176,400); list the indices of the 10 pixels with the largest span (you can identify the pixels either by their row and column in the original matrix, or by their flattened index; specify whether you flattened the image row-wise or column-wise).

4. Using the spans of the dimensions, provide an outline for how you can use this information to perform an approximate spectral clustering using locality-sensitive hashing.

If you run your code on AWS: Include your Spark logs in your submission. Also provide the details of your cluster architecture and discuss its performance as compared to running your program on your local machine in standalone mode.

6 Bonus Questions

The first two questions are dependent on each other and require additional coding. **The third question is standalone and requires no additional coding.**

6.1 Locality-sensitive hashing

Using the spans of the pixels, implement locality-sensitive hashing in Spark. Provide your code. For a given number of bins, report the distribution of images in each bin. Theoretically, each bin should contain images that are captured at a similar angle. Is the distribution of images across bins relatively uniform?

6.2 Bucket averaging

For each bucket of images, perform the image "averaging" from the original assignment. Do the averages from each bucket look different?

6.3 Distributed spectral clustering

Consider flattening each image to a single 176,400-element vector, and computing a sparse affinity matrix A using the locality-sensitive hashing method of the previous sections. Using the full EMPIAR-10015 dataset, this would give us a matrix with dimensions $416,000 \times 416,000$. Given that this represents over 173 billion floating-point values, this would have to be distributed. The common abstraction for distributed matrices is *row-distributed*; that is, each row of the matrix resides contiguously in memory of a compute node. The Scala API

[5] `http://cobweb.cs.uga.edu/~squinn/mmd_s15/papers/p223-hefeeda.pdf`

in Spark, for instance, has an `IndexedRowMatrix` that extends its base `DistributedMatrix` class; each row is keyed by an integer index, thus each record consists of a key, value pair where the key is the row number, and the value is the row vector.

Once we have the affinity matrix A, an important step is computing the normalized graph laplacian $L = D^{-1/2}AD^{-1/2}$, where D is a diagonal matrix where each D_{ii} is the sum of the i^{th} row of A. Since this is a diagonal matrix, we can store it as a vector $\vec{d}$ and either load it into the `DistributedCache` in Hadoop or use it as a broadcast variable in Spark.

Write pseudocode for a mapper (you're free to use Spark or Hadoop style) that takes as input a key, value pair where the value is a single row of A and the key is the integer row. The mapper contains $\vec{d}$ in memory. The output of the mapper is a key, value pair where the key is the same as the input key, and the output is the corresponding row of the normalized graph laplacian L.

7 Marking breakdown

- Code correctness (commit messages, program output, and the code itself) **[50 points]**
- Questions 1, 2 **[10 + 10 points]**
- Questions 3, 4 **[15 + 15 points]**
- Bonus 6.1 **[20 points]**
- Bonus 6.2 **[10 points]**
- Bonus 6.3 **[20 points]**

8 Other stuff

Spark's performance, while generally much better than Hadoop, can vary wildly depending on how effectively you are caching your intermediate results (and what your memory strategy is [6]). **You are not required to run your code on AWS**, but if you do, Spark includes scripts for easily setting up an AWS cluster [7]. The only requirement is to run Spark in standalone mode on your local machine; we will vary the number of cores used to "simulate" a distributed environment.

[6] `http://spark.apache.org/docs/latest/tuning.html`
[7] `http://spark.apache.org/docs/latest/ec2-scripts.html`