CSCI6900 Assignment 5: Spectral Clustering on Spark

DUE: Monday, November 16 by 11:59:59pm
Out November 9, 2015

1 OVERVIEW

Spectral clustering is a variant of unsupervised machine learning methods for identifying groups of similar data. Spectral clustering works by embedding the original data in a low-dimensional subspace where the data are more easily separable. Specifically, it operates by constructing a graph of the data based on pairwise similarity, and uses eigenvectors of the affinity matrix to project the data into this low-dimensional space.

Given data points \(\vec{x}_1, \vec{x}_2, \ldots, \vec{x}_n\) as row vectors of matrix \(X\), spectral clustering works by first constructing an \(n \times n\) affinity matrix \(A\), where \(A_{ij}\) is a non-negative number between 0 (completely dissimilar) to 1 (identical) that indicates the similarity between data points \(\vec{x}_i\) and \(\vec{x}_j\). There are many pairwise distance formulas that can be used to construct the affinity matrix; one of the most popular for spectral clustering is the heat kernel, or gaussian similarity, or the radial-basis function. For a given pair of points \(\vec{x}_i\) and \(\vec{x}_j\), their similarity \(A_{ij}\) is computed by:

\[
A_{ij} = \exp\left\{-\gamma ||\vec{x}_i - \vec{x}_j||^2\right\},
\]

where \(\gamma\) is a scaling factor, effectively the “width” of the gaussian kernel that modulates the size of the neighborhood around each data point. The larger the neighborhood, the stronger the connections will be to data points further away (and the harder it will be to differentiate them from closer data points).

Once the affinity matrix is determined, a graph Laplacian \(L\) can be computed:
\[ L = D^{-1/2}AD^{-1/2}, \]

where \( D \) is the diagonal degree matrix of the graph, computed by summing each row of \( A \):
\[ D_{ii} = \sum_{j=1}^{n} A_{ij}. \]
Once \( L \) is computed, we perform an eigen-decomposition of \( L \), and cluster its leading \( k \) eigenvectors as proxy data for \( X \).

2 PIC ON SPARK

Where in assignment 3 you implemented k-means clustering, here you will use Spark’s built-in power iteration clustering (PIC) to simulate an approximate variant of spectral clustering. Instead of explicitly computing the eigenvectors of the graph Laplacian \( L \), Spark uses power iterations to approximate the leading eigenvector of the data. Recall that power iterations work by multiplying a vector with the target matrix:
\[ \vec{v}_{t+1} = M \vec{v}_t, \]
where each iteration brings the vector \( \vec{v} \) closer to the leading eigenvector of \( M \).

Don’t worry, you won’t actually implement any of the above equations yourself. They’ve already been implemented in Spark in Python, Scala, and Java, and under the hood use Spark’s GraphX engine to perform the power iterations. You’ll need to do a little bit of programming to parse the data into the correct format before feeding it to the Spark API.

3 DATA

All the data for this assignment are two-dimensional Cartesian points in different configurations. One data point is presented per line, and the \( x \) and \( y \) coordinates for each data point are separated by whitespace. For instance, here are the first 5 lines for the \texttt{blobs.txt} file:

5.370422 -2.447152  
8.984267 -4.874497  
9.496494 -3.790297  
10.488484 -2.758582  
7.521321 -2.122666

There are three text files:

1. \texttt{blobs.txt}: 2,400 data points, three Gaussian isotropic clusters (800 points per cluster).
2. \texttt{circles.txt}: 2,400 data points, two concentric circles (1,200 points per circle).
3. \texttt{moons.txt}: 24,000 data points, two half-moons (12,000 points per half-moon).
These are synthetic data, generated on-the-fly using random number generators (specifically, the scikit-learn samples generators); they don't represent any “real” data, but were created solely for the purpose of this assignment.

The data are available via the read-only git repository mmd.cs.uga.edu:assignment5. So to access the data, you'll run:

git clone git@mmd.cs.uga.edu:assignment5.git a5_data

This will create a directory named a5_data in your local path, with the three text files inside. You can't write to this repository, so take care to ensure you don't accidentally add your code to this repository and try to commit it; it won't work!

4 Walkthrough

This section is to give you a head start on the assignment, hopefully avoiding some immediate pitfalls to streamline the overall process (given the one-week timeline and such).

The astute reader will notice a discrepancy between the format of the data as provided for this assignment, and the description of the expected data format in Spark's PIC documentation. For your convenience, I’ve copied the relevant information from the Spark PIC documentation here:

“MLlib includes an implementation of PIC using GraphX as its backend. It takes an RDD of (srcId, dstId, similarity) tuples and outputs a model with the clustering assignments. The similarities must be nonnegative.”

This means the first thing you'll need to do after reading in the raw data off the filesystem is compute the pairwise similarities of the data. We'll use the Gaussian heat kernel in Eq. 1.1 above. To compute the pairwise affinities, you'll need to have each data point \( \vec{x}_i \) matched up with every other data point \( \vec{x}_j \) for all \( j \neq i \). The \texttt{rdd.cartesian(rdd)} function will serve this purpose perfectly. If you have an RDD \( X \) of all your data points, you can create another RDD of all possible tuples of data points by calling:

\[
C = X\.cartesian(X)
\]

where each element in \( C \) is a tuple of \((xi, yi), (xj, yj)\). You'll still need to compute affinities from the tuples in \( C \) and make sure you know the row \( i \) and the column \( j \) the affinity you're computing should go (srcId and dstId in the above PIC documentation). Put another way, you should probably run \texttt{zipWithIndex} on your data even before running \texttt{cartesian}.

Once you've computed the pairwise affinities, you can feed those directly into the \texttt{run} method for \texttt{PowerIterationClustering}.

Your program should accept the following command-line parameters:
• \( k \), an integer indicating the number of clusters.

• \( x \), an integer indicating the upper bound on the number of power iterations to run.

• \( g \), a floating point value for \( \gamma \) in Eq. 1.1 (this can be a broadcast variable).

5 Deliverables

Create a folder in your repository named assignment5. Keep all your code for this assignment there; I will be using autograding scripts to check your progress, so make sure your repository is named correctly and precisely!

I will reference the code in that folder for partial credit. This needs to have a commit timestamp before the deadline to be considered on time. You should implement the algorithm by yourself instead of using any existing machine learning toolkit.

Please include a README file. In the README, document precisely how to run your program (I should be able to follow the instructions; this is how I will grade your assignment!) and any known bugs or problems you were unable to fix prior to the final submission.

You may use either Azure (if you have credits) or a UGA machine with at least 4 cores (8 threads). Use some sort of plotting library (e.g., matplotlib for Python; Breeze plot for Scala) to answer the following questions.

1. Run your program for \( k = 3, \ x = 10, \) and \( g = 1 \) on the data in blobs.txt. Create a scatter plot of the clustered data, where the color of each data point is determined by its cluster membership. Record the runtime.

2. Run your program for \( k = 3, \ x = 10, \) and \( g = 100 \) on the data in blobs.txt. Create a scatter plot of the clustered data, where the color of each data point is determined by its cluster membership. Record the runtime.

3. Run your program for \( k = 2, \ x = 10, \) and \( g = 50 \) on the data in circles.txt. Create a scatter plot of the clustered data, where the color of each data point is determined by its cluster membership. Record the runtime.

4. Run your program for \( k = 2, \ x = 10, \) and \( g = 50 \) on the data in moons.txt. Create a scatter plot of the clustered data, where the color of each data point is determined by its cluster membership. Record the runtime.

5. Comment on the values of \( g \) for each of the previous questions. What relationship does it have, if any, with the quality of the resulting clusters?

6. Did you notice anything in observing the runtime for the previous questions? Where do you think the bottleneck is, and why?
6 Bonus 1

Propose a fix for the bottleneck you identified in question 6 above. Sketch out Spark pseudocode that addresses and alleviates the bottleneck. Prove that your solution results in an increase in overall performance by comparing relative complexities of the operation you propose with the operation it replaces.

7 Bonus 2.1

Write a chunk of Spark code that, given the affinity matrix $A$, computes the graph Laplacian $L = D^{-1/2}AD^{-1/2}$.

8 Bonus 2.2

Use a distributed SVD solver for either Java, Scala, or Python to compute the SVD of the graph Laplacian $L$ and take the singular vectors associated with the $k$ largest singular values. Feed these $k$ vectors to Spark’s built-in k-means clustering algorithm; use the resulting clusters to plot the original data for one of the three datasets in Q1, Q2, or Q3 (submit the plot). Compare the results to what you obtained in the original questions; is there any difference?

9 Marking breakdown

- Code correctness (commit messages, program output, and the code itself) [40 points]
- Questions 1-6 [10 points each]
- Bonus 1 [15 points]
- Bonus 2.1 [20 points]
- Bonus 2.2 [25 points]

10 Other stuff

START EARLY. Yet again, there is not a lot of code to be written, but parsing all the math and truly grasping the concepts at work can take some time.

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). In particular, I highly recommend this chapter from Learning Spark, as it provides invaluable advice for partitioning your data in the most effective way possible so as to minimize network shuffling.