DEPARTMENT OF COMPUTER SCIENCE, UNIVERSITY OF GEORGIA

CSCI6900 Assignment 3: Clustering on Spark

DUE: Friday, Oct 2 by 11:59:59pm

Out Friday, September 18, 2015

1 OVERVIEW

Clustering is a data mining technique for identifying subgroups of data that are, in some sense, "similar." Its use is often synonymous with "unsupervised," meaning that ground-truth information regarding the origin of the data is missing or otherwise does not exist. As such, unsupervised learning typically involves choosing some sort of similarity metric with which to compare the data, and a stopping criterion for determining the number of discrete clusters in the data. K-means clustering [1] is probably the most popular clustering algorithm. It involves two steps: first, the data points are assigned to a cluster based on which centroid they are closest to; second, the centroids are re-computed using the new data-cluster labelings. The two-step process is repeated until some convergence threshold or maximum number of is reached.

In this assignment, your goal will be to cluster documents containing information on NIPS abstracts, NY Times news articles, and PubMed abstracts. For converting the documents into feature vectors, we'll be using TF-IDF. Your code will need to perform the following steps:

1. Read the provided vocabulary file.
2. Read each document, building a *sparse* TF-IDF vector representation of it.
3. Perform K-means clustering on the vectors to identify clusters of similar content.

[1] `https://en.wikipedia.org/wiki/K-means_algorithm`

2 Data

The data consists of text documents from the UCI Machine Learning Repository [2]. These data are provided in the "bag-of-words" format, where word order and sentence structure have been discarded. Furthermore, stop words and words with frequencies under a certain threshold have also already been discarded.

For each document, there are two files. Let's start with the NIPS data (the smallest): `docword.nips.txt` and `vocab.nips.txt`. The `vocab` text file is very simple: it contains the entire vocabulary (i.e. unique words), one word per line. The integer ID of the word corresponds to the line number, so the first word "a2i" has an integer ID of 1, the second word "aaa" has an integer ID of 2, and so on.

The `docword` file is more complicated. The first three lines of the file are global header values: `D`, `W`, and `NNZ`, respectively.

1. `D` is the number of documents in the dataset (for NIPS, this is roughly 1.9 million).
2. `W` is the size of the vocabulary (should correspond with the number of lines in the `vocab` file!).
3. `NNZ` is the sum of all the word counts in the documents.

After these three lines, each subsequent line follows the pattern: `docID wordID count`, where each of the three variables is an integer, separated by a space. Each line gives a unique word with how many times that word appeared in the given document. Thus, the `docword` text file looks like this:

```
D
W
NNZ
docID wordID count
docID wordID count
docID wordID count
...
docID wordID count
```

These are the six text files:

- `vocab.nips.txt` (0.1 MB)
- `docword.nips.txt` (8.5 MB)
- `vocab.nytimes.txt` (1.35 MB)

[2]`https://archive.ics.uci.edu/ml/datasets/Bag+of+Words`

- `docword.nytimes.txt` (1,004 MB)
- `vocab.pubmed.txt` (1.46 MB)
- `docword.pubmed.txt` (**7,811 MB**)

Each of them can be found at this URL: `http://ugammd.blob.core.windows.net/bagofwords/`. There is also a `readme.txt` file at the same URL which describes the data and its format.

3 Apache Spark

This assignment will function as an introduction to Apache Spark. It is your decision whether to use Spark's Python, Scala, Java, or R bindings (Scala can invoke Java packages and routines. However, the syntax is slightly different). I would highly recommend either Scala or Python.

3.1 Feature Transformation API

The first thing you'll need to do before running your clustering code is to convert the data to feature vectors. We are using TF-IDF as document feature vectors, and Spark has utilities for computing this. You'll need to read the Spark documentation to see how to use the feature extractors for the given language you are using: `https://spark.apache.org/docs/latest/mllib-feature-extraction.html#tf-idf`

Also, you will need to put the data in the correct format for the Spark feature extractors to work on them.

3.2 Pseudocode

Your code will take three parameters:

- *k*, an integer number of clusters
- *x*, an integer number of maximum iterations (set to 100)
- *t*, a float convergence tolerance (set to 1.0)

Once you have converted the documents to TF-IDF format, you will choose *k* initial centroids randomly from the dataset.

```
tfidf_vectors = ... # there should N of these, where N is number of documents
# each vector should be of length W, where W is the size of the vocabulary
# (may want to use a sparse vector representation!!!)

old_centroids = tfidf_vectors.takeSample(k)
iterations = 0
```

```
converge = 1.0
while iterations < x and converge > t:
        # e-step: assign each point to a cluster, based
        # on smallest Euclidean distance

        # m-step: update centroids by averaging points
        # in each cluster

        converge = sum( (old_centroids - new_centroids) ** 2 )
        old_centroids = new_centroids
        iterations += 1 # increment iterations

print 'Took {} iterations to converge.'.format(iterations)
print '{} centroid residual left.'.format(converge)
print old_centroids
```

4 Deliverables

Create a folder in your repository named `assignment3`. **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. In addition, please provide answers to the following questions.

1. Run your code either on Azure (if you have credits) or on a UGA machine with at least 8 cores. In either case, run your code with $k = 10$ and $k = 50$ clusters for each of the three datasets.
   If you are running on Azure:
   - Run your program on 1 node. Record the runtime.
   - Run your program on 8 nodes. Record the runtime.

   If you are on a workstation:
   - Run your program on 1 core: `--master local`
   - Run your program on all cores: `--master local[*]`

Compare the two runtimes you recorded. Is there a change in the runtime? If so, is it proportional to the number of cores / executors? How does the number of clusters k impact the runtime?

2. Hopefully you were able to implement K-means on your own; given its role as the prototypical clustering algorithm, there are plenty of K-means implementations out there. For this question, re-run the analysis on all three datasets with $k = 10$ (use 8 nodes or all cores, depending on your environment), but this time instead of your own K-means, substitute the implementation in MLlib [3]. Record the runtime. How does the MLlib implementation runtime compare to yours?

3. If the runtime between your implementation and that in MLlib is radically different, cluster initialization is likely a big reason. We discussed in lecture how random initialization is the worst possible choice. Canopy Selection, described in `http://cs.uga.edu/~squinn/mmd_f15/articles/canopy.pdf`, can be used as a rapid centroid initializer. Please read the paper. Describe the workflow (in terms of Spark API pseudocode) to incorporate the Canopy Selection with your K-means code to generate initial cluster centroids.

4. Another clustering variant is Kernel K-means. Kernel K-means attempts to blend K-means with the use of nonlinear distance metrics (i.e. non-Euclidean) without incurring the computational expense of performing a full eigendecomposition *a la* spectral clustering. Read section 2.1 in `http://cs.uga.edu/~squinn/mmd_f15/articles/kernel.pdf`. TF-IDF lends itself to cosine distance between vectors; thus, Kernel K-means is a perfect fit for document clustering. However, exhaustive pairwise comparisons is $\mathcal{O}(n^2)$ operations, intractable for any n that is remotely large. Design a heuristic (in terms of Spark API pseudocode) that computes a small fraction of the total n^2 possible pairwise comparisons but could still feasibly provide a "good" kernel matrix estimate. Justify your definition of a "good estimate."

5 Bonus Questions

1. Implement Canopy Clustering to generate your initial centroids and feed them into the K-means code that you wrote. Run your K-means code with $k = 50$ clusters on the *small* dataset (using 8 nodes / all cores). Run it again. Run it three more times (for a total of five). Report the *average number of iterations to convergence*. Now re-run your original code (random centroid initialization) with $k = 50$ on the small dataset (using 8 nodes / all cores) five times. Average the number of iterations each took to converge (maximum of 100). Is there any difference? Does Canopy Clustering help to speed up convergence?

2. Implement Kernel K-means, using cosine distance to generate the kernel matrix with whatever heuristic you proposed to avoid performing all n^2 pairwise comparisons. Run with $k = 10$ clusters on the small dataset (using 8 nodes / all cores). Select a centroid

[3]`https://spark.apache.org/docs/latest/mllib-clustering.html#k-means`

at random. Plot a histogram of the pairwise cosine angles between the TF-IDF vector representing that centroid to the other nine. Describe the results and explain what you see.

6 Marking breakdown

- Code correctness (commit messages, program output, and the code itself) **[50 points]**
- Question 1 **[15 points]**
- Question 2 **[15 points]**
- Question 3 **[10 points]**
- Question 4 **[10 points]**
- Bonus 5.1 **[25 points]**
- Bonus 5.2 **[25 points]**

7 Other stuff

Microsoft Azure builds its HDInsight platform against specific builds of Apache Spark, and unfortunately the latest version available is 1.2.0 (Spark 1.5 was just released a couple weeks ago). However, the K-means and TF-IDF APIs are all in place for both Scala and Python as of 1.2.0. Furthermore, there is an unofficial `spark_azure` script package for setting up custom Spark instances on Azure. I have not tested it, but you are welcome to give it a try: `http://spark-packages.org/package/sigmoidanalytics/spark_azure`.

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 [4]).

Spark has fantastic documentation for all its APIs: `https://spark.apache.org/docs`

[4] `http://spark.apache.org/docs/latest/tuning.html`