EECS 476: Data Mining (Winter 2021)

Project 2 Solutions

Hashing, Clustering, Outlier detection

EARLY DUE date: Mar 14, 2021 11:50 pm (for 5 BONUS points)
FINAL DUE date: Mar 19, 2020 11:50pm

Part 1. [35 points] Duplicate Detection

In this part you will find pairs of near-duplicate documents by leveraging the pipeline that we discussed in class: Shingling → Minhashing → LSH.

Note that for the universal hash function in minhashing, the row index should start from 0.

Code Requirements. You should write serial code in Python 3.7. You may use any packages that are available via an Anaconda installation. A list of such packages can be found at: https://docs.anaconda.com/anaconda/install/linux/.

Data. A set of 75 text documents, some of which are near duplicates of each other.

- Each document is stored in its own file and has a corresponding ID value, which is used as its filename. For example, the document with ID 3, will be named "`3.txt`".
- Data can be downloaded from `/var/eecs476w21/`
- Store the path to all the input files in a file called "`manifest.txt`". For example, if your input files are stored in a directory named"`data`", then the `manifest` file will look like:

```
data/0.txt
data/1.txt
data/2.txt
...
data/74.txt
```

  The manifest file won't change in the autograder, so you can do whatever processing to be able to use the manifest file.

Detecting Near-Duplicate Text

1. **[15pts]** Write a serial program that finds near-duplicate documents in the data listed in the "`manifest.txt`" file. Your program should leverage the three steps that we discussed in class:
   (a) shingling (*without* converting the shingles into tokens) to turn the documents into sets
   (b) minhashing to obtain a short signature per document
   (c) LSH to avoid the quadratic number of signature comparisons.

   - For minhashing, use the universal hash functions: $h_{a,b}(r) = ((ar+b)\ mod\ p)\ mod\ m$, where a, b are random integers, r is the row number, m is the number of rows of the document matrix, and $p > m$ is a prime number.

- Based on the candidate pairs found by LSH, you should return as near-duplicates the pairs of documents for which the signatures' Jaccard similarity exceeds $(>)$ the **similarity threshold** $\tau = 0.7$.

Your program, which we refer to as *LSH*, should accept the following command line arguments:

```
--input_path  : The path that leads to the manifest file.
                e.g.: --input_path manifest.txt

--output_path : The path to the output file.
                e.g.: --output_path out.csv

-k            :  The size of the shingles.
                 e.g.: -k 9

-n            :  The number of minhash functions to use
                 e.g.: -n 6

-p            :  The prime number used in the minhash functions.
                 e.g.: -p 35401

-r            :  The number of rows per band in LSH
                 e.g.: -r 2

-b            :  The number of bands in LSH
                 e.g.: -b 3
```

The **output** should be a "`.csv`" containing the near-duplicate documents that you found. Each line of the csv file should be dedicated to one document: the first column should contain the document ID, followed by the sorted **space separated** list of documents which are suspected near-duplicates of it (in increasing order of ID), as in the following example. The rows should be sorted in increasing order of the first column.

```
0,1 2
1,0 2
2,0 1
3,-1
4,5
5,4
```

Note that for each near-duplicate pair of documents, the output should encode this information in both lines corresponding to the documents (e.g., documents 0, 1 and 2 are duplicates in the example above, so that is indicated in **both** the line beginning with 0, 1 **and** 2. If you find that a document has no near-duplicates (e.g., document 3 in the example), you should set the value after the comma to -1.

Remark 1. Make sure that your output has 75 lines (one per input document).

Remark 2. If you use the `random` library, to generate the hashing functions, or for anything else, the random seed can influence your final results. For consistency, when you submit your final output file, ***set the random seed to 3****. You can use this code at the beginning of the files that contain random. You can also experiment with other seeds to see how/if results change.*

Remark 3. Make sure the items in the universal set of shingles are sorted in the order the shingles are read from the file. For example, if you have 2 documents: "time is" and "time not" with the following 2 lists of shingles: [['ti', 'im', 'me', 'e ', ' i', 'is'], ['ti', 'im', 'me', 'e ', ' n', 'no', 'ot']], the universal set of shingles will be: ['ti', 'im', 'me', 'e ', ' i', 'is', ' n', 'no', 'ot']. Before submitting to autograder, I recommend to check your code on this example.

```
import random
random.seed(3)
```

2. **[5pts]** Write a serial program that finds near-duplicate documents in the data listed in the "`manifest.txt`" file through the brute-force algorithm: apply shingling to represent each document with its k-shingles and compute the Jaccard similarity between all pairs of documents. Return all the pairs of documents with similarity that exceeds the threshold $\tau = 0.7$. We will refer to this set of near-duplicate documents as **ground-truth**.

   Your program, which we refer to as *Naïve*, should accept the following command line arguments:

```
--input_path    : The path that leads to the manifest file.
                  e.g.: --input_path manifest.txt

--output_path : The path to the output file.
                  e.g.: --output_path out.csv

-k             :  The size of the shingles.
                  e.g.: -k 9
```

   The output should be the same as in the previous question.

3. **[3pts]** Run the following *LSH* (Q1) and *Naïve* (Q2) variants and report their runtimes and their number of candidate pairs:

   - *LSH-1:* $k = 9$, $n = 100$, $p = 31159$, $r = 5$, $b = 20$
   - *LSH-2:* $k = 9$, $n = 100$, $p = 31159$, $r = 10$, $b = 10$
   - *LSH-3:* $k = 9$, $n = 100$, $p = 31159$, $r = 50$, $b = 2$
   - *LSH-4:* $k = 5$, $n = 100$, $p = 13259$, $r = 5$, $b = 20$

- *Naïve-1:* $k = 9$
- *Naïve-2:* $k = 5$

What do you observe?

4. **[5pts]** Consider the **candidate pairs** returned by the *LSH* variants in Q3 (**before** the final pass that returns only the near-duplicates for which the signature similarity exceeds the similarity threshold). Report the number of false positives and false negatives by comparing the candidate pairs to the corresponding ground-truth pairs.

   Specifically, you'll need to compare the candidate pairs of variants *LSH-1*, *LSH-2* and *LSH-3* to the ground-truth pairs of *Naïve-1*, and the results of variant *LSH-4* to the ground-truth defined by *Naïve-2*.

5. **[4pts]** For each of the *LSH* variants in Q3, give the expected probability of false negatives and the probability of false positives. Compare these to your results in Q4, and discuss your key observations.

6. **[3pts]** (more open-ended question) Read the near-duplicate document pairs returned by the two *Naïve* variants (for $k = 9$ and $k = 5$). Do you observe any differences between these two settings? Explain your observations. How would you set the value k to improve the results of your near-duplicate detection algorithm?

To Submit

- Autograder: Submit your source code for Q1 to the autograder. All files should be in the format: "*_duplicate.py" with your main function in "duplicate.py". All code will be copied to a top level directory and run with the command "python3.7 duplicate.py ..."
- Autograder: Submit your source code for Q2 to the autograder. All files should be in the format: "*_duplicate_naive.py" with your main function in "duplicate_naive.py". All code will be copied to a top level directory and run with the command "python3.7 duplicate_naive.py ..."
- Write-up: Your answers to questions Q3-Q6.

Part 2. [40 points + 5 bonus points] K-means on MapReduce

Instructions: For the following questions, include answers (e.g., plots, short-answer questions) in the `writeup.pdf` with clear numbering.

- For this part, you are asked to write MapReduce code and run it on the Hadoop cluster. Make sure you run your job on the cluster by setting the queue name as "eecs476". The datasets are under `/var/eecs476w21/` in the HDFS.
- Start early, start early, start early!! The computations in this exercise will take significant amount of time.

In this question, your task is to implement k-means (based on different distance functions) using MapReduce. Let $X_1, \ldots, X_N$ be d-dimensional real points (i.e., in $\mathbb{R}^d$), composing a set $\mathcal{X}$. In class we saw that we can use many different distance functions between points. Below we define two very popular distance measures and their corresponding cost functions for a given set of k clusters and their set of k centroids $\mathcal{C}$.

L_2 distance (or Euclidean distance): Given two points $\mathbf{x}$ and $\mathbf{y} \in \mathbb{R}^d$ such that $\mathbf{x} = [x_1, x_2, \ldots, x_d]$ and $\mathbf{y} = [y_1, y_2, \ldots, y_d]$, the L_2 distance between $\mathbf{x}$ and $\mathbf{y}$ is defined as:

$$||\mathbf{x} - \mathbf{y}||_2 = \sqrt{\sum_{i=1}^{d} (x_i - y_i)^2} \tag{1}$$

The corresponding cost function ψ that is minimized when we assign points to clusters is given by:

$$\psi = \sum_{p \in \mathcal{X}} \min_{c \in \mathcal{C}} ||p - c||_2^2 \tag{2}$$

L_1 distance (or Taxicab distance): Given two random points $\mathbf{x}$ and $\mathbf{y} \in \mathbb{R}^d$ such that $\mathbf{x} = [x_1, x_2, \ldots, x_d]$ and $\mathbf{y} = [y_1, y_2, \ldots, y_d]$, the L_1 distance between $\mathbf{x}$ and $\mathbf{y}$ is defined as:

$$||\mathbf{x} - \mathbf{y}||_1 = \sum_{i=1}^{d} |x_i - y_i| \tag{3}$$

The corresponding cost function χ that is minimized when we assign points to clusters is given by:

$$\chi = \sum_{p \in \mathcal{X}} \min_{c \in \mathcal{C}} ||p - c||_1 \tag{4}$$

The iterative k-means algorithm that we learned in class is described in Algorithm 1.

Algorithm 1 k-means

procedure ITERATIVE K-MEANS
    Initialize the k clusters by picking k centroids (e.g., randomly).
    $N \leftarrow MAX\_ITER$
    for iterations $i = 1$ to N **do**
        for each data point $\mathbf{x} \in \mathcal{X}$ **do**
            Cluster $\mathbf{x}$ to nearest centroid
        for each cluster $c \in \mathcal{C}$ **do**
            Update the centroid as the mean of all the data points in c.
    Return $\mathcal{C}$

Remarks:

- When you are assigning points to clusters, it is possible that you might have a point which is equally close to 2 or more clusters. In this event, you should **break ties** by assigning your data point to the cluster whose centroid has the smallest magnitude or length (i.e., Euclidean norm).
- It is also worth noting that algorithm description starts with **initializing the k clusters** via some method. The initialization will be provided via a command line argument.

Data. The dataset is available at `/var/eecs476w21/`.

- `data.txt`: Document matrix with 480 rows (documents) and 300 columns (features). Each row is a document represented as a 300-dimensional vector of features. The rows represent the same word embeddings you used in the previous exercise.
- `randomC.txt`: Initial set of $k = 10$ cluster centroids that were randomly selected points from the input data.
- `selectC.txt`: Initial cluster centroids selected such that they are as far apart as possible from each other.

An example of a valid input data file with three documents and 3 features (e.g. data.txt):

```
2.5,3.4,4.6
3.7,4.1,5.7
5.9,6.8,7.1
```

This file would represent three documents (data points) in $\mathbb{R}^3$: $(2.5, 3.4, 4.6)$,$(3.7, 4.1, 5.7)$, and $(5.9, 6.8, 7.1)$.

An example of a valid centroid file (e.g. `randomC.txt` or `selectC.txt`) would be:

```
4.03,4.76,5.8
1.03,8.36,2.1
```

This example file represents two centroids, one per line (i.e., in this case, we are looking for two clusters). Note that centroids do not need to consist of integers, because they are the mean of their cluster's data points.

Q1. [15pts] Iterative k-Means clustering on Hadoop

In this task, you are asked to implement k-means on Hadoop. You should use the following command line arguments to set variables in your code:

```
--inputPath   : Path to the file containing your data to be clustered.

--centroidPath: Path to the file containing the initial centroid assignments.

--outputScheme: Output scheme that your jobs will use.
             * Make sure that your final output is a directory beginning with
             this argument followed by the number of your final job. That is,
             if you run 20 jobs with --outputScheme output, your final output
             should be in the directory "output20".

--norm: 1 or 2. The "l-norm" for this run of k-means.
        Their definitions can be found above.

-k: The number of clusters you will find.
    This will match the number of clusters defined in --centroidPath.

-n: The number of iterations that your code will run.
```

Your program should take data points as described above via the inputPath argument, and centroids from the centroidPath argument. A description of their formats is listed above. You program should output the new centroids at the end of one iteration of the k-means algorithm. For example, if after one iteration of k-means, you find that the new centroids are located at $(1, 4, 2)$ and $(3, 6, 1)$, your program should produce the file "outputScheme1/part-r-00000":

1 ,4 ,2
3 ,6 ,1

The order of your centroids does not matter, but the order of coordinates within a centroid does. That is, you could switch the line order of the above example, but not the order of numbers within a line.

Hint: A single step of MapReduce completes one iteration of the k-means algorithm that is described above. In other words, in order to run k-means for i iterations, you will have to run a chain of i MapReduce jobs.

To Submit

(a) Autograder: Source code for your Hadoop k-means implementation. Make sure that the main function of your implementation is in Kmeans.java and any additional java files follow the pattern "*_Kmeans.java"

(b) **DO NOT** submit any data.

Q2. [15 pts] Euclidean distance in k-means

In this question, you will use the L_2 distance as the distance measure.

1. **[3pts]** Using the L_2 distance (Equation 1 and –norm 2) as the distance measure and `randomC.txt` as –centroidsPath, compute the cost function $\psi(i)$ (Equation 2) for every iteration i. Plot the cost function $\psi(i)$ as a function of the number of iterations $i = 1, 2, \ldots, 25$.

2. **[3pts]** Repeat Q2(1) by using "`selectC.txt`" as initial centroids.

   Hint: For the first MapReduce job iteration, you will be computing the cost function using the initial centroids located in text files. You do not need to write a separate MapReduce job to compute $\chi(i)$. Instead, you can simply include this computation into the Mapper or Reducer.

3. **[3pts]** Report the percentage change in cost after 15 iterations of the k-means algorithm when the cluster centroids are initialized using "`randomC.txt`" and the percentage change for "`selectC.txt`". Is random initialization "`randomC.txt`" better than initialization using "`selectC.txt`" in terms of cost ψ? Briefly explain your answer.

4. **[3pts]** How many clusters would you pick? (Assume that you are doing random initialization of the centroids. Briefly explain your answer.

5. **[3pts]** Prove (with arguments) that the cost ψ is a non-increasing function of i(i *represents the number of iterations*).

To Submit

- Write-up: The plot of the cost function as a function of the number of iterations for input centroids `randomC.txt`
- Write-up: The plot of the cost function as a function of the number of iterations for input centroids `selectC.txt`
- Write-up: The percentage difference in cost after 15 iterations of the k-means algorithm when using the `randomC.txt` vs. `selectC.txt` as well as an explanation of why this is your answer.
- Write-up: Answer and justification for Q2.4.
- Write-up: Proof for Q2.5.

Q3. [10 pts] Taxicab distance in k-means

In this question, you will use the L_1 distance as the distance measure.

1. **[3pts]** Using the L_1 distance (Equation 3 and –norm 1) as the distance measure and `randomC.txt` as –centroidsPath, compute the cost function $\chi(i)$ (Equation 4) for every iteration i. Plot the cost function $\chi(i)$ as a function of the number of iterations $i = 1, 2, \ldots, 25$.

2. **[3pts]** Repeat Q3(1) using "`selectC.txt`" as initial centroids.

   Hint: The same hints provided in Q2 apply here too.

3. **[4pts]** Report the percentage change in cost after 15 iterations of the k-means algorithm when the cluster centroids are initialized using "`randomC.txt`" vs. "`selectC.txt`". Is random initialization "`randomC.txt`" better than initialization using "`selectC.txt`" in terms of cost χ? Briefly explain your answer.

4. **[5pts Bonus]** [Analysis in Python] t-SNE (Distributed Stochastic Neighbor Embedding) is a dimensionality reduction technique for visualizing high-dimensional data. Using the corresponding python package for t-SNE, plot the embeddings given in `data.txt` in 2 or 3 dimensions, using a different color per cluster (e.g., all the points that belong to cluster 1 should be blue, all the points that belong to cluster 2 should be orange, etc.). Create two plots based on the results of Q2.1 and Q3.1. What do you observe? Which method is better?

To Submit

- Write-up: The plot of the cost function as a function of the number of iterations for input centroids "randomC.txt"
- Write-up: The plot of the cost function as a function of the number of iterations for input centroids "selectC.txt"
- Write-up: The percent difference in cost after 15 iterations of the k-means algorithm when using the "randomC.txt" vs "selectC.txt" as well as an explanation of why this is your answer.
- Write-up (optional): Two plots visualizing the embeddings and clustering results in two dimensions, and description of your observations.

Part 3. [25 points] Outlier Detection

Fake news is playing an increasingly dominant role in spreading misinformation by influencing people's perceptions or knowledge to distort their awareness and decision-making. The growth of social media and online forums has spurred the spread of fake news causing it to easily blend with truthful information. In this project, you will work with a dataset that contains both fake and real short news articles and you will classify them using an outlier detection method.

Data. The `fake news` dataset contain 480 text articles, which can be either fake or legit.

- Each document is stored in its own file and has a corresponding ID value, which is used as its filename. That is, the document with ID 3, will be named "3.txt"
- The labels: fake/ 1 or legit/ 0, together with the corresponding article ID, are provided in labels.csv
- The word embeddings for each article ID are stored in "`embeddings.csv`". All the information from the .txt articles is compressed here.
- Data can be downloaded from `/var/eecs476w21/`

What are word embeddings? A word embedding is a learned representation for text (learned vector of floats) that preserves the semantic information of words. That is, when projected in a vector space, the distance between semantically similar words will be small.

- You can see some examples of equalities that hold in the embedding space in Figure 3. For example, the embedding of the word "king" minus the embedding of the word "man" plus the embeddings of the word "woman" is similar to the embedding of the work "queen".
- There are multiple methods to compute word embeddings. You can find the one that we used here.

$$king - man + woman \approx queen$$

$$german + airlines \approx lufthansa$$

$$\vdots$$

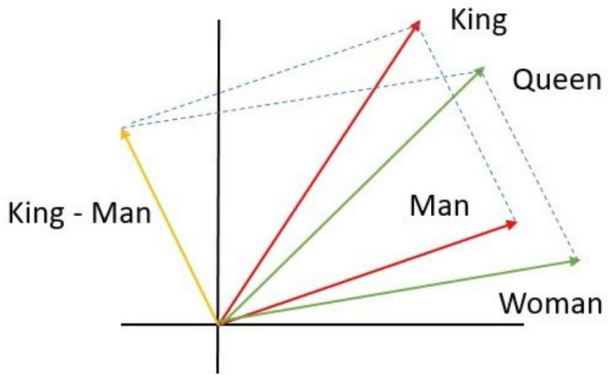

Figure 1: Math with words!

Locally Outlying Factors (LOF) for Outlier Detection

In this question, you will implement the Locally Outlying Factors (LOF) method for outlier detection and apply it on the `fake news` dataset. The hope is that fake articles will be identified as outliers.

In LOF, a point is considered an outlier if it has a much lower *local* density than that of its neighbors. The method is based on the notion of *k-nearest neighborhood* of a given point, which is defined as the set of data points that are at most as far as the k^{th} nearest neighbor to that point. More formally, given (1) the set of all data points D, (2) a distance function $d(x, y)$ for two points x and y (e.g., Euclidean distance), and (3) the distance $V^k(x)$ to the k^{th} nearest neighbor, we define the *k-nearest neighborhood* of point x as:

$$L_k(x) = \{y \in D | d(x, y) \leq V^k(x)\}$$

Recall that $L_k(x)$ can be larger than size k if there are various data points equal to $V^k(x)$.

To measure how outlying a data point is from another in its k-nearest neighborhood, we can define a "reachability" distance:

$$R_k(x, y) = max\{d(x, y), V^k(y)\}$$

which indicates how "reachable" x is from y. Then, the average "reachability" of x from *all* points $y \in L_k(x)$ is given as:

$$AR_k(x) = avg_{y \in L_k(x)}(R_k(x, y))$$

Now that we have an average reachability score we can actually define outliers using the Locally Outlying Factor, LOF. We define the LOF score of node x as:

$$LOF_k(x) = avg_{y \in L_k(x)} \left(\frac{AR_k(x)}{AR_k(y)} \right)$$

We would expect that the LOF score of data points within a cluster would be close to 1. This is because we expect the average reachability of x to be very similar to the average reachability of x's k nearest neighbors. Conversely, we would expect the LOF score of an outlier to be much larger than 1. This is because if x is an outlier, it is not in the neighborhood of its own nearest neighbors.

1. **[15pts]** Implement the Locally Outlying Factors method in Python 3.7 using only packages which are available from Anaconda. Your program should accept the following command line arguments:

```
--input_path  : The path that leads to the input file embeddings.csv.
                e.g.: --input_path data/embeddings.csv

--output_path : The path to the output file formatted as shown in Table 2.
                e.g.: --output_path data/out.csv
```

```
-k              :  The minimum size of the nearest neighborhoods considered.
                   e.g.: -k 1

-cutoff         :  The LOF cutoff you will use to determine whether a data
                   point is an outlier. A data point is considered an
                   outlier if its score is *strictly greater* than the cutoff.
                   e.g.: -cutoff 3
```

Your **output** should contain one line for every article in the input data. A sample output might look like:

1 ,0
2 ,0
3 ,1
4 ,0

where the first column corresponds to the article ID and the second column to its label (fake/1, legit/0).
The example indicates that articles 1, 2, and 4 are **not** outliers, while 3 **is** an outlier. It also indicates that data point 3 is a **fake** news articles and the rest are legit.

You should make no assumptions about what the command line inputs will be when ran on the autograder. **Do not hardcode** in any values, not even for the filenames.

2. **[7pts]** Run your code on the `embeddings.csv` dataset with `k` $= 1$, `cutoff` $= 3$. Generate the output file with the outliers. Evaluate your result by computing the precision and recall of your solution, which we define below.

   The result of LOF can be represented in a $N \times 1$ binary vector $\mathbf{b}$ (with i^{th} entry b_i), where N is the number of records in the input file (in this specific case, $N = 480$ articles). Similarly, you can represent `labels.csv` as another binary vector, $\mathbf{r}$ with r_i as the i^{th} real label for record i (i.e, the ground-truth).

   Precision measures "over all the predicted fake articles in your output file, how many of them are actually fake?". The definition of precision is given in Equation 5.

$$\textbf{Precision} = \frac{\sum_{i=1}^{N} b_i \cdot r_i}{\sum_{i=1}^{N} b_i} \tag{5}$$

   Recall measures "over all the ground-truth fake articles, how many of them are predicted correctly in your output file?". The definition of recall is given in Equation 6.

$$\textbf{Recall} = \frac{\sum_{i=1}^{N} b_i \cdot r_i}{\sum_{i=1}^{N} r_i} \tag{6}$$

3. **[3pts]** You might notice that these parameters perform quite poorly. Why do you think this set of parameters performed so poorly? Suggest a set of parameters "-k" and "-cutoff" that might perform better and explain why.

To Submit

- Autograder: Source code for this problem. All code should follow the pattern "*_lof.py" and the main function should be in a file called "lof.py". All code will be copied to a top level directory and run with the command "python3.7 lof.py ...".
- Write-up: Your precision and recall from your run of LOF on embeddings.csv.
- Write-up: Explanation for the performance and proposal of better parameters.