CSC 502: Assignment 1

Due on: Wednesday, Feb 5 at 23:59 PST

Where: Brightspace (https://bright.uvic.ca/d2l/home/398483)

Instructions

- You must complete this assignment entirely on your own. In other words, you should come up with the solution yourself, write the code yourself, and test the code yourself. The university policies on academic dishonesty (a.k.a. cheating) will be taken very seriously. Check the syllabus for details on what is allowed and what is not.
  - You can have high-level discussions with your classmates about the course material.
  - You are also welcome to use Piazza or come to office hours and ask questions. If in doubt, please ask— we are here to help!
- You can use a maximum of **two (2) late days per assignment**, and a maximum of **five (5) days for the whole semester**. Any further accommodations must be announced and negotiated with the instructor at least seven (7) days before the deadline.
- If you think some things should be explained in detail, please include the explanations in the form of source code comments.
- We will try to grade your assignments within seven (7) days of the initial submission deadline.
- If you think there is a problem with your grade, you have one week to raise concerns after the grades go public. TA will be holding office hours during those seven days to address any such problems. After that, your grade is set in stone.
- You should only submit four Python/Jupyter files in `.py` or `.ipynb` format to Brightspace. Name them as `probN.py` (lowercase; `N` is a number from 1 to 5, inclusive).
  - **Warning:** Different file names might result in zero points. Our scripts expect the correct file names.
- Please indicate your V-number and name within a comment at the top of each submitted file.

Problem 1: Spark [20 points]

Write a Spark program that implements a simple social network friendship recommendation algorithm. The key idea is that if two people have a lot of mutual friends, then the system should recommend that they connect with each other.

Input

- The associated data file is `data/p1-users.txt`.
- The file contains the adjacency list and has multiple lines in the following format: `<User><TAB><Friends>`, where `<User>` is a unique integer ID corresponding to a unique user, and `<Friends>` is a comma-separated list of unique IDs corresponding to the friends of the user with the unique ID `<User>`. Note that the friendships are mutual (i.e., edges are undirected): if A is friend with B then B is also friend with A. The data provided is consistent with that rule as there is an explicit entry for each side of each edge.

Algorithm

Let us use a simple algorithm such that, for each user U, the algorithm recommends $N = 10$ users who are not already friends with U, but have the largest number of mutual friends in common with U.

Output

- The output should contain one line per user in the following format: `<User><TAB>` `<Recommendations>` where `<User>` is a unique ID corresponding to a user and `<Recommendations>` is a comma-separated list of unique IDs corresponding to the algorithm's recommendation of people that `<User>` might know, ordered in decreasing number of mutual friends.
- Even if a user has less than 10 second-degree friends, output all of them in decreasing order of the number of mutual friends. If a user has no friends, you can provide an empty list of recommendations. If there are recommended users with the same number of mutual friends, then output those user IDs in numerically ascending order.

Sketch

Please provide a description of how you used Spark to solve this problem. Don't write more than 3 to 4 sentences for this: we only want a very high-level description of your strategy to tackle this problem.

Tips

- Use Google Colab to use Spark seamlessly, e.g., copy and adapt the setup cells from Assignment 0.

- You can also use your own machine if you install Java 8 or 17 (newer versions do not work) and PySpark. There is no need to use PyDrive for this task. Make sure to set `os.environ["JAVA_HOME"]` in your code to a correct Java version!
- Before submitting a complete application to Spark, you may go line by line, checking the outputs of each step. Command `.take(X)` should be helpful, if you want to check the first `X` elements in the RDD.
- Beware of laziness: an error at line 15 might in cell 5 be related to line 2 in cell 2 because that stage is *only executed when needed* (in this case, that line in cell 2 won't execute until being needed by the cell 5). Make sure to inspect **the whole code** if there is an error.
- We recommend using either PySpark DataFrame and/or RDD syntax for this question.
- For sanity check, your top 10 recommendations for user ID 11 should be: 27552, 7785, 27573, 27574, 27589, 27590, 27600, 27617, 27620, 27667.
- Beware of `collect` calls: use them only when really needed!
- The pipeline will **not** be fast. It takes a minute on my laptop to complete; on Colab, it might take much longer.

What to submit?

1. Upload your code on Brightspace (name: `p1.ipynb` or `p1.py`). **Do not upload data files!**
2. Include in your notebook / source a short paragraph sketching your Spark pipeline.
3. Include in your writeup the recommendations for the users with the following user IDs: 11, 924, 8941, 8942, 9019, 9020, 9021, 9022, 9990, 9992, 9993.

Problem 2: Associations [20 points]

Association rules are frequently used for *market basket analysis* (MBA) by retailers to understand the purchase behaviour of their customers. This information can then be used for many different purposes, such as *product cross-selling* (action or practice of selling additional products or services to existing customers), up-selling, sales promotions, loyalty programs, store design, discount plans and many others.

Giving product recommendations is one of the examples of cross-selling that is frequently used by online retailers. One simple method to give product recommendations is to recommend products that are frequently browsed together by the customers.

Suppose we want to recommend new products to the customer based on the products they have already browsed online. Write a program using the Apriori algorithm to find products which are frequently browsed together. Fix the support to $s = 100$ (i.e. product pairs need to occur together at least 100 times to be considered frequent) and find itemsets of size 2 and 3.

Input

- The associated data file is `data/p2-baskets.txt`.
- Each line represents a browsing session of a customer. On each line, each string of 8 characters represents the ID of an item browsed during that session. The items are separated by spaces.

Algorithm

1. *[10 points]* Identify pairs of items (X, Y) such that the support of $\{X, Y\}$ is at least 100. For all such pairs, compute the *confidence* scores of the corresponding association rules: $X \rightarrow Y, Y \rightarrow X$. Sort the rules in decreasing order of *confidence* scores and list the top 5 rules. Break ties, if any, by lexicographically increasing order on the left-hand side of the rule.
2. *[10 points]* Then, identify item triples (X, Y, Z) such that the support of $\{X, Y, Z\}$ is at least 100. For all such triples, compute the confidence scores of the corresponding association rules: $(X, Y) \rightarrow Z$, $(X, Z) \rightarrow Y$, and $(Y, Z) \rightarrow X$. Sort the rules in decreasing order of *confidence* scores and list the top 5 rules. Order the left-hand side lexicographically and break ties, if any, by lexicographical order of the first and then the second item in the pair.

Output

- Please use the following output format: `<ITEM1> -> <ITEM2>: <CONFIDENCE>` (for 2 items) or `<ITEM1>,<ITEM2> -> <ITEM3>: <CONFIDENCE>` (for 3 items) for each line.
- Please provide at least five decimal points for your confidence scores.
- You must also follow the exact rules for ordering. We are not giving partial credits to coding when results are wrong.

Tips

- **This problem does NOT require Spark.**
- These sanity checks are provided to help you progress:
  1. There are 647 frequent items after 1st pass ($|L_1| = 647$).
  2. The top 5 pairs you should produce in the first part all have confidence scores greater than 0.985.
  3. The top 3 pairs you should produce in the second part all have confidence scores greater than 6.
- You cannot use non-standard Python libraries (such as `apyori` etc.). You need to implement Apriori from scratch yourself. Trust me—all Apriori libraries for Python are of low quality!

What to submit?

1. Upload your code on Brightspace (name: `p2.ipynb` or `p2.py`). **Do not upload data files!**

Problem 3: Neighbours [25 points]

In this problem, we study the application of LSH to the problem of finding approximate near neighbours. Assume we have a dataset A of n points in a metric space with distance metric $d(\cdot,\cdot)$. Let c be a constant greater than 1. Then, the (c,λ)-*approximate near neighbour* (ANN) problem is defined as follows:

Problem 1 ((c,λ)-ANN) *Given a query point z, assuming that there is a point x in the dataset with $d(x,z) \leq \lambda$, return a point x' from the dataset with $d(x',z) \leq c\lambda$ (this point is called a (c,λ)-ANN).*

The parameter c, therefore, represents the maximum approximation factor allowed in the problem.

Input

- A dataset of images is provided in `data/p3-patches.csv`. Each row in this dataset is a 20×20 image patch represented as a 400-dimensional vector.
- You are also provided with a half-completed starter code, `p3.py`, with all loading functionality already implemented for you. This file has all the locations where you need to contribute code marked with TODOs. In particular, you will need to use the functions `lsh_setup` and `lsh_search`, and implement your own linear search.
  - The default parameters $L = 10, k = 24$ to `lsh_setup` work for this exercise.

Tasks

We will use the L_1 distance metric on $\mathbb{R}^{400}$ to define the similarity of images. We would like to compare the performance of LSH-based approximate near neighbour search with that of linear search (i.e., comparing the query point z directly with every database point x).

1. *[5 points]* For each of the image patches in rows $100, 200, 300, \ldots, 1000$, find the top 3 near neighbours (excluding the original patch itself) using both LSH and linear search. What is the average search time for LSH? What about linear search?

2. *[10 points]* Assuming $\{z_j \mid 1 \leq j \leq 10\}$ to be the set of image patches considered above (i.e., z_j is the image patch in row $100j$), $\{x_{ij}\}_{i=1}^{3}$ to be the approximate near neighbours of z_j found using LSH, and $\{x_{ij}^*\}_{i=1}^{3}$ to be the (true) top 3 near neighbours of z_j found using linear search, compute the following error measure:

$$\text{error} = \frac{1}{10}\sum_{j=1}^{10}\frac{\sum_{i=1}^{3} d(x_{ij}, z_j)}{\sum_{i=1}^{3} d(x_{ij}^*, z_j)}.$$

   Plot the error value as a function of L (for $L = 10, 12, 14, \ldots, 20$, with $k = 24$). Similarly, plot the error value as a function of k (for $k = 16, 18, 20, 22, 24$ with $L = 10$). You can use matplotlib's `subplots` to stack the plots. Briefly comment on the two plots (one sentence per plot would be sufficient).

3. *[10 points]* Finally, plot the top 9 near neighbours found using LSH and linear search (using the default $L = 10, k = 24$) for the image patch in row 100 (i.e. index 99 when using

zero-indexing), together with the image patch itself. You may find the function `plot` and matplotlib's `imshow` useful. How do they compare visually?

Output

- For Task 1, output two lines: `avg(lsh) = <AVG1>` and `avg(linear) = <AVG2>` where each average is the average runtime in seconds rounded to two decimal points.
- For Task 2, output two plots and provide a short comment.
- For Task 3, output one image and two 3×3 image collages with 9 nearest neighbours each. Don't forget to include a short comment as well.

Tips

- Sometimes, the function `lsh_search` may return less than 3 nearest neighbours. You can use a while loop to check that `lsh_search` returns enough results.
- You may sometimes have less than 9 nearest neighbours in your results; you can use the above hack to bypass this problem.
- Mind the zero-based indexing in Python!

What to submit?

1. Upload your notebooks on Brightspace (name: `p3.ipynb`). **Do not upload data files!** Each notebook should contain all plots (inline, not as separate files) and comments.

Problem 4: Spar$k$-means [25 points]

This problem will help you understand the nitty-gritty details of implementing clustering algorithms on Spark. In addition, this problem will also help you understand the impact of using various distance metrics and initialization strategies in practice.

Let us say we have a set $\mathcal{X}$ of n data points in the d-dimensional space $\mathbb{R}^d$. Given the number of clusters k and the set of k centroids $\mathcal{C}$, we now proceed to define various distance metrics and the corresponding cost functions that they minimize.

Euclidean distance: Given two points A and B in d-dimensional space such that $A = [a_1, a_2, \ldots, a_d]$ and $B = [b_1, b_2, \ldots, b_d]$, the Euclidean distance between A and B is defined as:

$$\|a - b\| = \sqrt{\sum_{i=1}^{d}(a_i - b_i)^2}.$$

The corresponding cost function ϕ that is minimized when we assign points to clusters using the Euclidean distance metric is given by:

$$\phi = \sum_{x \in \mathcal{X}} \min_{x \in \mathcal{C}} \|x - c\|^2.$$

Note that in the cost function, the distance value is squared. This is intentional, as it is the squared Euclidean distance the algorithm is guaranteed to minimize.

Manhattan distance: Given two points A and B in d-dimensional space such that $A = [a_1, a_2, \ldots, a_d]$ and $B = [b_1, b_2, \ldots, b_d]$, the Manhattan distance between A and B is defined as:

$$|a - b| = \sum_{i=1}^{d} |a_i - b_i|.$$

The corresponding cost function ψ that is minimized when we assign points to clusters using the Manhattan distance metric is given by:

$$\psi = \sum_{x \in \mathcal{X}} \min_{x \in \mathcal{C}} |x - c|.$$

Input

The folder has 3 files:

1. `data/p4-data.txt` contains the dataset which has 4601 rows and 58 columns. Each row is a document represented as a 58-dimensional vector of features. Each component in the vector represents the importance of a word in the document.
2. `data/p4-c1.txt` contains k initial cluster centroids. These centroids were chosen by selecting $k = 10$ random points from the input data.
3. `data/p4-c2.txt` contains initial cluster centroids which are as far apart as possible, using Euclidean distance as the distance metric. (You can do this by choosing 1st centroid c_1 randomly, and then finding the point c_2 that is farthest from c_1, then selecting c_3 which is farthest from c_1 and c_2, and so on).

Algorithm

We learned the basic k-means algorithm in class which is as follows: k centroids are initialized, each point is assigned to the nearest centroid and the centroids are recomputed based on the assignments of points to clusters. In practice, the above steps are run for several iterations. We present the resulting iterative version of k-means in Algorithm 1.

Algorithm 1 Iterative k-means Algorithm

```
Select $k$ points as initial centroids of the $k$ clusters
for iterations := 1 to MAX_ITER do
    for all point $p$ in the dataset do
        Assign point $p$ to the cluster with the closest centroid
    end for
    Calculate the cost for this iteration
    for all cluster $c$ do
        Recompute the centroid of $c$ as the mean of all the data points assigned to $c$
    end for
end for
```

Your job is to implement iterative k-means using Spark.

Set the number of iterations (`MAX_ITER`) to 20 and the number of clusters k to 10 for all the experiments carried out in this question. Your driver program should ensure that the correct amount of iterations are run.

When assigning points to centroids, if there are multiple equidistant centroids, choose the one that comes first in lexicographic order (please consider vectors as vectors of numerical values instead of strings, that is to say, $[1, 2]$ should be placed before $[1, 12]$ instead of the opposite).

The running time for this problem should be less than 1 minute for each question.

Tasks

This problem should be implemented in Spark. You should not use the Spark MLlib clustering library for this problem. You may store the centroids in memory if you choose to do so.

The following two tasks should be done once for each distance measure (Eucledian or Manhattan):

1. [10 points] Compute the cost function $\phi(i)$ (or $\psi(i)$) for every iteration i. This means that, for your first iteration, you'll be computing the cost function using the initial centroids located in one of the two text files. Run the k-means on `p4-data.txt` using `p4-c1.txt` and `p4-c2.txt`. Generate a graph where you plot the cost function $\phi(i)$ (or $\psi(i)$) as a function of the number of iterations $i = 1 \ldots 20$ for `p4-c1.txt` and also for `p4-c2.txt`. How does the error change as a function of iterations?

2. [10 points] What is the percentage change in cost after 10 iterations of the k-means algorithm when the cluster centroids are initialized using `p4-c1.txt` vs. `p4-c2.txt` with the chosen distance metric? Is random initialization of k-means using `p4-c1.txt` better than initialization using `p4-c2.txt` in terms of cost $\phi(i)$ (or $\psi(i)$)? Explain your reasoning.

Output

- Task 1 should just output a single plot with two lines. Don't forget to include your commentary on the plot.
- Task 2 should include 4 lines in the following format: `Change in c1 (Eucledian) = <PP>%`, where `c1` and `Eucledian` vary based on the problem. `<PP>` is an integer between 0 and 100 (no decimal points). Again, don't forget to include your commentary.

Tips

- Note that you do not need to write a separate Spark job to compute $\phi(i)$ (or $\psi(i)$). You should be able to calculate costs while partitioning points into clusters.
- In Task 2, the percentage refers to `(cost[0]-cost[10]) / cost[0]`; here, `cost[0]` is the total cost attained using the initial centroids provided in the two text files.
- Sanity check: `cost[2]` with Eucledian distance for initialization using `p4-c1.txt` is between 4×10^8 and 5×10^8.
- Sanity check: `cost[2]` with Manhattan distance for initialization using `p4-c1.txt` is between 4×10^5 and 5×10^5.
- A better initialization results in a lower final cost. Please keep an integer for your final result for the percentage change, meaning that your answer should be in the form of xy%.

What to submit?

1. Upload your notebooks on Brightspace (name: `p4.ipynb`). Each notebook should contain all plots (inline, not as separate files) and comments. **Do not upload data files!**