Project 2 - Finding Similar News Articles
TDT4305 - Big Data Architecture (Spring 2024)

DUE: **March 15, 2024 11:59 PM**

Start Here

- **Collaboration policy:**
  - You are expected to comply with the University Policy on Academic Integrity and Plagiarism.
  - You are allowed to talk with other students on homework assignments.
  - You can share ideas but not code; you must submit your code individually or as a group of two.
  - All submitted codes will be compared against all codes submitted this semester and in previous semesters.
- **Programming:**
  - You are required to use Python 3.8 or newer versions for this assignment.
  - Ensure the code given runs on your environment before you start your implementation.
  - Do not use any external libraries for this project, all necessary libraries have been provided.
  - Ensure the dataset is in your project's "/data" directory.
  - There are some default empty methods in the code where you must implement your tasks. You can change the methods and add more if you need to, but don't change the way the code reads the inputs and gives the outputs.
  - Implement the missing code under ## *implement your code here* in the lsh.py file.
  - Since you will be dealing with string variables, use case-insensitive comparison if needed.
  - Ensure the Python file runs without errors, then save for submission.

- **Submissions:**
  - Note that if you submit your work multiple times, the most recent submission will be graded.
  - You can do this project individually or in a team of two people; if you are in a team, both members of the team must submit the same work separately on Blackboard.
  - Only include the Python file, Notebook and Report pdf in the .zip files; do not include anything else, such as dataset or binary files.
  - When you are done, save the Python file, Notebook and Report pdf in a single .zip file named 'student_id_Project2_LSH.zip' (use both members' student ID when in a team of 2) and upload the .zip file on BlackBoard via the submission link for the assignment. Also, include the names of team members in the notebook (or only your name if you are working on the assignment individually).

  Important: *Post all questions on Piazza and only send me an email in case of (rare) anomalies!*

Introduction

The overarching aim of this project is finding similar BBC news articles, and assuming that similar news articles talk about the same event, the aim is to use various methods covered during the lecture, such as Shingling, Jaccard Similarity, MinHash and Locality Sensitive Hashing (LSH) to find these similar articles with optimal computational speed and resource efficiency. To facilitate a seamless transition to the "BBC Big Dataset", we created two preliminary parts (Part 1 and Part 2) on a toy (small) data. These parts are crucial in providing an intuitive understanding of the algorithms and methods used in Part 3.

1 Part 1 - Small Data Warm Up (*20pts*)

In this part, we want to test our understanding of the concepts needed to accomplish this project on a toy dataset before delving into the BBC dataset in Part 3. This is analogous to saying - we want to learn the rudiments of walking before running.

To begin, we have curated four (4) 'relevant' TDT4305 sentences.

Sentence 1: "The Big Data platform for students is Blackboard"
Sentence 2: "Questions on MinHash project by NTNU students is on Piazza"
Sentence 3: "NTNU Big Data platform are Blackboard and Piazza"
Sentence 4: "The project data for students are on Blackboard not Piazza"

For the rest of this part, the above sentences will be the foundation for obtaining our results.

N.B: At this stage, ignore every code provided and focus on getting this done on pen and paper (or any writing material of your preference)

1.1 Extracting K-Shingles

Create a list of **word** k-shingles from each sentence with a k value of 1. To make this example more tractable, we will ignore all (definite or indefinite) articles, auxiliary verbs, prepositions and conjunctions. For example, "The TDT4305 Piazza platform is a bastion for exceptional students" transforms to

$$['tdt4305', 'piazza', 'platform', 'bastion', 'exceptional', 'students']$$

Provide the extracted k-shingles for all 4 sentences in your report. To avoid case sensitivity, convert all words to lowercase.

1.2 Creating Unique Words Dictionary

In this step, we want to create a word bank or dictionary of all the unique words that appear in our sentences. Again, provide these unique words in your report in lowercase.

1.3 Input Matrix of Sentences

Create an input matrix of all four (4) sentences from the list of the unique words. The input matrix contains a 1 if a word from the unique words list is present in a sentence; otherwise, assign 0. Provide this input matrix and the Jaccard Similarity of every possible sentence combination in your report. Which sentence pair is the most similar?

1.4 Calculating MinHash Signature matrix

In the previous step, we were able to compute the Jaccard Similarity between different sentence pairs. However, this is not efficient and requires large storage. For instance, the unique words dictionary in real-world applications can be up to a million, leading to an extremely large sparse input matrix. So, we need a way of computing similarities that don't scale with the size of the word dictionary. MinHash provides an elegant way of achieving this by creating a Signature matrix with dimensions influenced by the number of hash functions rather than the unique words dictionary.

Using the following permutations, construct a MinHash signature matrix for the input matrix obtained in the previous section.

H_1 = 5 6 7 8 9 10 1 2 3 4
H_2 = 9 6 3 10 7 4 1 8 5 2
H_3 = 10 7 4 1 8 5 2 9 6 3

From the MinHash signature matrix, compute the similarity of all sentence pairs by finding the fraction of the hash functions in which they agree. Which sentence pair is the most similar? Is this consistent with the sentence pair gotten from the Jaccard Similarity?

---

2 Part 2 - Coding MinHash for Small Data (*20pts*)

So far, we likely have blunted our pens while manually working through the Input and MinHash signature matrix, and now it's time to determine if we can similarly wear out our keyboards while translating this process into code. We have provided a Python notebook file with the name **MinHash.ipynb** in the code folder for this implementation.

2.1 Confirming the Unique Words

The core component of this part is creating the Input Matrix and using the MinHash algorithm to build a Signature Matrix. Feel free to manually enter the k-shingles obtained in Part 1 for each sentence or alternatively split each sentence. For easy comparison and grading of your solution, the unique words dictionary must be in lowercase and arranged alphabetically. We have hashed the correct answer for encryption and stored the hashed values in encrypted_dictionary.txt

To check if your unique words dictionary is correct, enter the entire unique word lists in the text box provided in the notebook.

2.2 Coding the Input Matrix

In this part, we want to create an Input Matrix using the sentences and unique words shingles gotten so far. You are allowed to code this out any way you deem fit. Our focus here is not on efficiency but on results. The final result from this step is an input matrix with zeros and ones similar to the input matrix in Part 1.

2.3 MinHash Algorithm

The nucleus of this part is coding the MinHash algorithm from scratch. It turns out that the salient statement in MinHash - "the index of the first (in the permuted order) row in which column C has value 1" is as difficult for a computer to interpret as it is for TDT4305 students to comprehend. Coding this out presents two apparent challenges. The first is that writing an algorithm to scan for random permutations starting from the lowest and checking the occurrence of 1 in the input matrix is far from ideal and efficient. The second is that until this time, we have been provided with lists of random permutations. For large dictionary of unique works with, say 1 million words and above 100 hash functions, storing this in memory is very demanding.

To solve the first limitation, we initialize our signature matrix $M$with infinite values and update until we arrive at the smallest possible value where column C has value 1. For the second limitation, we will create a list of hash functions h_n and only hash an index when needed in the MinHash algorithm.

Algorithm 1: MinHash Signature Matrix Algorithm

```
for each row r do
    for each row c do
        if c has 1 in row r then
            for each hash function h_i do
                if h_i(r) < M(i, c) then
                    M(i, c) ← h_i(r);
                end
            end
        end
    end
end
```

Code a MinHash function using the MinHash algorithm above that takes the input matrix and a list of hash functions as inputs and outputs a signature matrix.

Test this out on the input matrix from the previous section using the following hash functions.

$$H_1 = (x + 3) \mod 10 + 1$$
$$H_2 = (7x + 1) \mod 10 + 1$$
$$H_3 = (7x + 2) \mod 10 + 1$$

Coincidentally, the permutations from Part 1 were generated using these same hash functions. This implies that if your implementation is correct, you should get the same MinHash signature matrix as in Part 1.

---

3 Part 3 - Finding Similar BBC News Article with LSH (*60pts*)

Along the way, you have progressively developed enough expertise to tackle the 'Big Data' tasks. In other words, by now, you have both the dexterity and motivation to 'Go big'. But the thing with big data is that efficiency is needed in every pipeline of the implementation. To be specific, the toy example provided in Parts 1 and 2 comprises of 4 sentences and requires a total of 6 pairs for comparison. This brute force comparison will not scale nicely for this part with 2225 documents. More so, in reality, most of these pairs are dissimilar, leading to unnecessary comparisons. This necessitates the need for an algorithm that refines the search space to only pairs that share some level of similarity. This is what the Locality Sensitive Hashing (LSH) is about. We will see at the end of this part that we don't need to check all possible pairs of the 2225 documents but rather a few hundred candidate pairs.

Data

The data can be found in the .zip file you are given under the "code/data/bbc" folder. This folder contains 2225 .txt documents from the BBC news website corresponding to stories from 2004-2005.

Code

You are given a Python script in which the main methods of reading the data, reading the input parameters, and giving the results are implemented. The methods that are implemented are described as follows:

- **read_parameters():** Reads the input parameters from the "default_parameters.ini" parameter file. The input parameters are k, permutations, b, data, naive, t, described in Table 1. The method stores the input parameters in the "parameters_dictionary" dictionary, and they can be accessed as "parameters_dictionary['parameter_name']".

Table 1: *Parameter Descriptions*

Parameter Name	Type	Default	Description
k	int	5	The number of words for the k-shingles of each document
permutations	int	100	The number of permutations for simulation of MinHash
b	int	20	The number of bands used for the LSH algorithm
data	string	bbc	The directory of the input data should be under "data"
naive	boolean	false	Parameter that controls the run of the naive method
t	float	0.6	Similarity threshold

- **read_data(data_path)**: Reads the input data that are in the **data_path** directory. The documents in the input data are stored in the **document_list** dictionary, where the key of the dictionary is the document id (integer of the name of the file) and the value is the document text. The documents are then sorted by their document IDs.
- **naive()**: If the parameter **naive** is **true**, the **naive()** method is run. It calculates the similarities of all the combinations of documents and stores the similarities in the triangular array **naive_similarity_matrix**.
- **jaccard(doc1, doc2)**: Calculates the Jaccard similarity of two documents that are represented as sets of words.
- **get_triangle_index(i, j, length)**: Calculates the triangle index for the triangular array used in the **naive()** method.

3.1 Task 1: Create the k-Shingles

Implement the document representations using k-Shingles (words) for the documents in the data. Use the default k from the global parameter in your function.

3.2 Task 2: Compute the input matrix

Using the k-Shingles from the previous task, create the signature sets or input matrix for the documents.

3.3 Task 3: Generate the MinHash signature matrix

Using the k-Shingles Signatures sets from the previous task, implement the algorithm of the MinHash Signature Matrix of the documents. If your implementation in **Part 2** is correct, you don't need to do anything here. Just copy the function from the notebook to the lsh.py file.

However, now you need to create a function that generates an arbitrary number of hash functions using random **a**, **b** and **p** coefficients, with p being a prime number. Use the number of permutations and the size of the unique shingles as input parameters.

3.4 Task 4: Find candidate pairs using LSH

Every method executed until this time was intended to bring us closer to this significant final step. The LSH is a critical technique for reducing the necessary computation needed when finding similar items. Recall that we introduced MinHash to deal with the unique k-shingles in our document. While that saves storage, we still have to compare all document pairs, which consumes time and computation. We need a way of telling useful pairs in advance before computing actual similarity on the MinHash signature matrix. LSH provides an ingenious way to achieve this.

Using the MinHash Signature Matrix from the previous task as input parameter, implement the LSH method to create a set of candidate document pairs that are possibly similar. Use the number of band b from the parameter file in your function.

3.5 Task 5: Calculates the similarities above t

For the candidate document pairs from the previous task, calculate the document signature sets similarity using the fraction of the hash functions on which they agree, i.e.

$$\text{similarity}(d_1, d_2) = \frac{\#(h_i(d_1) == h_i(d_2))}{\text{permutations}}$$

From the similarities of the candidate document pairs, return the document pairs with similarity over the threshold input parameter t along with their similarities. Basically, this function takes as input the candidate document pairs and MinHash similarity matrix and returns a dictionary of all the pairs with similarities above the threshold with the similarity as a corresponding value.

Report

In your report, briefly describe the following:

1. Using parameter k = [1; 5; 10], and the default values for the rest of the parameters, how does the parameter k affect the total number of shingles? Is having more shingles better for identifying as many as possible document pairs that have similarities over the threshold?
2. Using permutations = [10; 20; 50; 100], and the default values for the rest of the parameters, how does the number of permutations affect the number of false positives?
3. Using b = [20; 50; 100], and the default values for the rest of the parameters, what is the optimal number of bands so that we make the least amount of comparisons? also, how does the b value affect the number of false positives?
4. Using the naive() method, parameter naive = true, count how many document pairs are checked for similarity for the naive() and how many for the LSH. How efficient is the LSH method compared to the simple naive method both in terms of document pairs and running time?
