ESE 545 Project 1

Yibo Yang[a,b]
[a]Department of Mechanical Engineering and Applied Mechanics,
[b]Penn Institute for Computational Science,
University of Pennsylvania, 3401 Walnut Wing A, 518,
ybyang@seas.upenn.edu

Abstract

This is the report of project 1 of ESE 545 data mining. In this report, we will discuss how we would be able to load and parse "amazonReviews.json" data set. Later on, we implement the locally sensitive hashing [1] together with permutation trick (min-hashing) to determine the similarity between data pairs. Specifically, Jaccard distance is introduced to represent the similarity for the data pair. In order to find similar pairs in the whole data set, another hashing is used to assign the partitioned signature matrix into different buckets for clustering [3]. In the end, the application of all the previous results is shown for finding the nearest neighbor of a given new review text. The codes are written into two files. The first one is main.py which is taking care of all the questions. The second is find_nearest_reviewID.py that is used to directly find the nearest review of a given new review. One should run the main.py first to see all the results and provide the new reviews in find_nearest_reviewID.py (optional) to find the closest review.

Keywords: Data mining, Locally sensitive hashing, Min-hashing, Jaccard distance, similarity in text data set.

1 Clarification

Before we start, there are something we should clarify:

- The submission consists of two functions: main.py and find_nearest_reviewID.py
- All the results could be seen by simply running the function main.py
- After running main.py it will automatically generate an 800Mb file storing all the hash function for the buckets in order to make the question 6 solved more efficient. One can see how fast it is for finding nearest reviewID and content by running find_nearest_reviewID.py after running main.py. In order to efficiently save and load the hashing (essentially this is a list of 300 hashing,) we import json (but we only use it to do this part). We should notice that the storing hashing part is not necessary in main.py, one can just comment line 530 - line 550 in main.py and it should also work well and show all the results. But if you would like to see how fast it can do for question 6, please be a little bit patient and follow the procedure to run main.py first and then find_nearest_reviewID.py
- The code for generating the binary matrix in question 2 is commented in line 140 - line 166. Because we do not need to use it later, we do not want to loss much memory to store it. Actually we are not using sparse matrix to store it, so it works well on the author's computer but it will blow up the memory on some other computer (in some cases). In this case, we just comment these part. If one want to see the results and check, just un-comment this part and run to see the results.
- The whole project including the codes and report is completely finished by the author himself (thanks for the stimulating discussions with all the TAs).

2 Problem 1: Parsing the data

We load the data using pandas. If the length of the string for each review is less than 5 (which could be too small), we just discard that review.
For removing the punctuation, we first create a list storing all the possible punctuation. Then, we go through all the reviews letter by letter and find if it is with in the punctuation list, we replace it by nothing. In this way, we would be able to remove all the punctuation.
For removing the stop words, we already write down all the stop words in an txt file that is: "stop_words_list.txt". Loading this file and store all the words inside into a list called "stop_words_list". using split function and go through each review words by words, we would be able to find if the current word is in the stop word list. If so, just remove it.
Finally, we calculate the total number of reviews in this data set and note it as num_sentences.

3 Problem 2: Represent the reviews in terms of binary matrix

In order to construct the binary matrix, we first need to know how many shingles do we have given the length of each shingles k_num. It could be easily get that a small number for k_num would loss representative power and leak of ability to capture "word order" information and most review will be similar, while a large number for k_num would lead to confusion that even similar documents may have share small number of shingles. As it is well known that the average word length in English language is 4.7 characters (very close to 5). In order to capture the local similarity of text files (within each words and between words) and consider the size of the review we have in the data set, we would think k_num around 5 is a good choice. Also, given k_num = 5, the total number of possible shingles would be around 27^5 which is much higher than a review's characters length would be. In this case, we choose k_num = 5. For large documents, such as research articles, choice k = 9 is considered safe. More information and discussion could be found in [3].
To do so, we build up a dictionary called k_shingles_dict where we store all the shingles we find in the whole data set. Simultaneously, we label the index of each shingle with this dictionary (essentially a hashing function). At the end, we note the total number of shingle as R.
We initialize a big zeros matrix whose row number is the number of total shingles R and column number is the number of reviews in the data set num_sentences. Then, we go through the whole data set review by review for each review we fill in the element correspond to the shingles it contains. Roughly speaking, each review contains around hundred to thousand which is much smaller than the number of total shingles R. This will enable us to construct a new way to store the data in a set of list for each review and help to reduce the computational complexity and storage space significantly. We will discuss about it in the later sections.
Here we should mention that the code works well on a Macbook Pro with 9Th-generation 8-Core Intel Core i9 Processor. But as we discuss some drawbacks of this data structure and will not use it later, I just comment the code for generating this binary matrix in the main.py. If one is interested to see the outcome, just uncomment this part and run.

4 Problem 3: Jaccard distance in a small data set of 10,000 pairs

We first note that the Jaccard distance is defined as following:

$$\text{Jaccard}(d_1, d_2) = 1 - \frac{|d_1 \cap d_2|}{|d_1 \cup d_2|}, \tag{1}$$

where d_1 and d_2 are representing the binary vectors correspond to two reviews. On the implementation side, we can directly use element wise multiplication and take summation after that to find the $|d_1 \cap d_2|$, in other words: $\sum(d_1 * d_2)$. For $|d_1 \cup d_2|$, we would use $\sum d_1 + \sum d_2 - |d_1 \cap d_2|$. These two operations can significantly accelerate the speed of the process. For this question, we randomly sample 10,000 pairs of reviews and compute the Jaccard distance between them. We show the histogram of the results in figure 1. Also, we would be able to compute the lowest Jaccard distance as 0.9 for these 10,000 pairs. Finally, we compute the average Jaccard distance through out this simulation result as Jaccard_distance = 0.987 which is fairly large and close to 1. This reflect the high dimensional structure of the data. But as there exist low dimensional latent representation of the context file, there are some small distance.

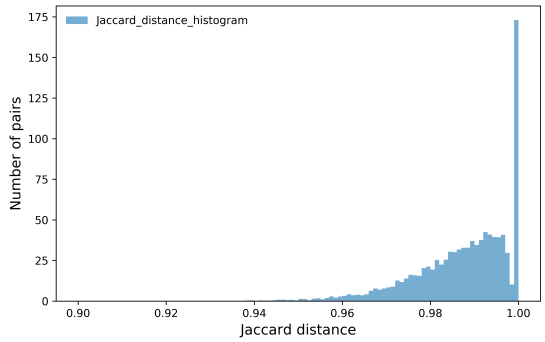

Figure 1: Histogram of Jaccard distance of 10,000 randomly selected pairs of reviews.

5 Problem 4: A better way to represent the data than binary matrix

In this section, we argue that using the binary matrix to represent the data is not a very good option. The matrix size for this binary representation is around ($R\times$ num_sentences). As R is a very large number, this matrix will take huge space to store and will be slow to extract some specific data from this matrix.
An alternative choice could be we store each review will a list that only contains the non-zero indices of a given review. This is benefited by the fact that we store the shingle index in a dictionary where the searching complexity is of $O(1)$. In this case, the final structure that we use to store the whole data set is a list of num_sentences small list where each small list has length around the average number of shingles in each review.
This representation is having at least two benefits:
(1) We significantly reduce the storage size.
(2) This makes it very efficient to do the permutation of locally sensitive hashing. Because we are using the $\pi_{a,b}$ directly working on the non-zero indices.

6 Problem 5: Determine similar data pairs in the whole data set using locally sensitive hashing techniques

We separate this task into 3 steps:
(1) Constructing the signature matrix of the whole data set.
(2) Do the hashing throughout each band of the reviews and calculate the buckets that each review is assigned.
(3) Go through to detected pairs and compute the Jaccard distance of each pairs to make conclusion on the similarity.

6.1 Computing signature matrix

In order to work with large data set, we need a large number of permutations M around hundred to thousand. We choose $M = 600$ and plot the probability of hit as a function of similarity in figure 2 following the function:

$$\Pr(\text{hit}) = 1 - (1 - s^r)^b \tag{2}$$

where different line represent different choice of band B and the number of data in each band r. We can see that $M = r \times B$. In order to keep the similarity as 0.8 (Jaccard distance 0.2), we would like to choose $r = 10$ and $B = 60$ to avoid false negative. This operation will definitely increase the false positive, but we can do post-processing to eliminate that parts. In this case, we would conduct a $M \times 2$ to represent the permutation we need to compute the signature matrix. Also, we need to find the smallest prime number that is larger than R to make sure the permutation we conduct is a true permutation. Here we note this prime number as R^* The permutation rule we follow is:

$$h(r) = (a * r + b) \mod R^* \tag{3}$$

where (a, b) represent the permutation $\pi_{a,b}$ and r denote the non-zero indices of a review. After conducting the M permutations for each review, we finally get the signature matrix having dimension $M\times$ num_sentences.

6.2 Hashing throughout each band of the reviews using the partition of signature matrix

In order to avoid directly loop through the data set twice and compare the similarity of two given reviews, we would consider to use hashing functions that assign (map) the reviews into several buckets and directly work on the results of the buckets to make rough conclusion of the similarity. In figure 3, we show a graph to illustrate the process. The detail process would be able to find in the code. Here, the rough idea is to loop through each band of all the reviews, we conduct another permutation on each

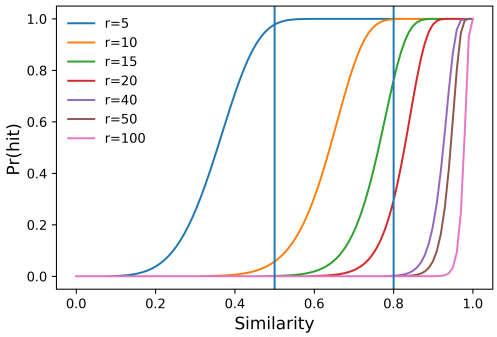

Figure 2: Probability of hit as a function of similarity where we have different choice of $r = 5, 10, 15, 20, 40, 50, 100$ together with two vertical lines representing similarity $= 0.5$ and 0.8.

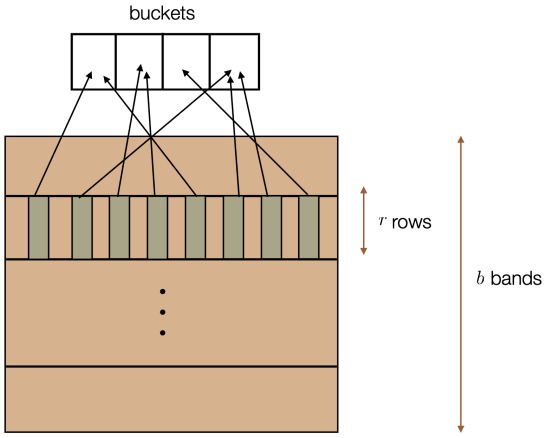

Figure 3: Partition of signature matrix.

element within the band of interest:

$$h_j(v) = (a_j * v + b_j) \mod P \tag{4}$$

$$h(v) = \sum_{j=1}^{r} h_j(v) \tag{5}$$

where P is a very large prime number. Here we should note that from the previous section, we choose $r = 10$ and $B = 60$. In this case, we generate a new permutation matrix of dimension $r \times 2$ for one band. And we use this throughout all the bands.
After the computation, if two reviews happen to be assigned to 1 bucket in any of all the bands, we would store them and set them as potential candidate as similar pairs.

6.3 Post processing for determining the true similar pairs in the whole data set

After conducting the operations we mentioned in the previous section. We would result into around 10 thousand candidate pairs. It is completely affordable to compute the Jaccard distance pairwisely. To this end, we implement the code for computing the Jaccard distance and finally come to around one thousand pairs whose Jaccard distance is less than 0.2 in the whole data set. The csv file including the summary could be found in close_pairs.csv. Notice that we found around 1189 pairs that have Jaccard distance less than 0.2. We are also showing the Jaccard distance of the identified (accepted) pairs in the histogram figure 4. We should notice that by

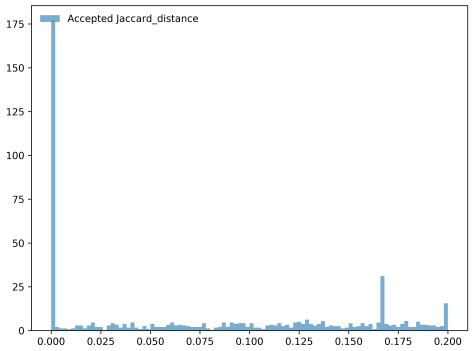

Figure 4: Histogram of the Jaccard distance for the identified (accepted) pairs that has similarity greater than 0.8.

observing the final similar reviews, we found some reviewers just submit the same review many times (with some small difference or even do not change anything). For these cases, we still keep them as similar pairs. Also, because we remove the stop words and the punctuation, some reviews turns out to be nearly the same, but essentially they could be different in some minor parts.

7 Problem 6: Function for finding an approximate nearest neighbor of a given queried review

7.1 Methods

We noticed that the number of similar pairs (under the constraint of Jaccard distance is less than 0.2) is too hard. In this case, in order to find a reasonable nearest neighbor of a given new queried

review, we would like to relax the constraint a little bit more to Jaccard distance less than 0.8. In this case, we need to reset the $r = 2$ and $B = 300$. Following the same procedure as the previous questions on the whole data set, we would obtain the buckets assignment discussed as before. For more information, one would refer to [2]. This will truly need some additional computational cost to recompute the new mapping information on the signature matrix.

Given a new queried review, we define a function called find_nearest_reviewID.py which takes a new queried review as input and provide its approximate nearest neighbor (reviewerID) together with its content. The function first parse the review by removing the punctuation and the stop words. Then it convert the given review into a list that is storing the non-zero indices together with a binary vector in order to compute the Jaccard distance. Than, we do the operation of permutations twices. First to compute the corresponding signature vector of the given queried review, if some new shingle appears to be not in our original training data set, we choose to ignore it. Second, based on the signature vector we obtained, we would use the second permutation on each band of this signature vector and assign the bucket values for this review. Finally, we check if there are some other data from the original data set happens to appear in the same bucket with the new review. Store the identified reviews in the original data set as candidates. Noting that the buckets information of the original data set are already computed and stored. We do not need to deal with the permutation of the whole data set for this implementation.

If there are no reviews in the data set has similarity less than 0.2, we would print "Cannot find any similar review having Jaccard distance less than 0.8 with the given one".

For better and efficient implementation and user friendly, we write down a function called find_nearest_reviewID.py, which could be used easily. At the beginning of this function, we first load all the dictionaries, data and parameters stored from running the main.py. Then, we can sufficiently test a new queried review. Noting that the loading data part is just a one time cost and would not need further computational cost. In this case, the testing would be extremely fast, around 10 second per queried review. The warm up processes (loading all the data, dictionary, etc) will take around 30 seconds.

7.2 Tests

Here are four test cases we try. For the first two cases, we put some reviews already exist into the function and try to find similar ones and the most nearest one. For the last three cases, we provide the function with two sentences we write randomly and see if it could work.

(1) Input: "good quality dental chew less give one day freshens dogs breath quality edible ingredients".
Output: A1OV3FT8XYDTSZ (exactly the original review ID).
Output review: good quality dental chew less give one day freshens dogs breath quality edible ingredients.

(2) Input: "dog loved birthday loves chance havent gotten see durability yet since days floating far hasnt chewed".
Output: AI0131QZO047J (exactly the original review ID).
Output review: dog loved birthday loves chance havent gotten see durability yet since days floating far hasnt chewed.

(3) Input: "I have a nice good cat walk around me."
Output: A18LH43H7N3J5I.
Output review: "dogs dont like absorbent walk around squat rare"

(4) Input: "I like to have enough time walking my dog run for long time."
Similar review ID: A3RT3640Z9FAE9.
Output: A3RT3640Z9FAE9.
Output review: "love use time walking dogs around house comfortable light waist"

Noting that this could change for each run because the permutation function we generate maybe different each time.

7.3 Discussion

In this case, we would conclude that if you pick up some review that already exists, the algorithm will definitely be able to locate itself together with some possible similar reviews. If we input some sentence that are similar like a review, the algorithm will return us some similar one (not very similar) e.g. case (4). If we write some sentence unlike a review, the algorithm will not be able to locate any good fit and try to provide some review that it thinks should be similar e.g. case (3). And for case (3), if we want to find a similar review in the original data set that has Jaccard distance less than 0.5 to the case (3), we will fail.

8 Problem 7: Discussion of the complexity and comparison to the naive way

8.1 Discussion and computational complexity

The complexity for the naive approach is very large and expensive. The number of review is noted as num_sentences in the previous section. The complexity for comparing each pairs would be $O(\text{num\_sentences}^2)$ (also we need to consider the complexity for compare the data pair). But in our case, we only need to loop through the whole data set once and perform hashing M times on each review. Although after the hashing and buckets clustering, we need to go over the candidate pairs we got and do a post processing for checking the similarity, it will not take too long. In this case, the complexity is around $O(\text{num\_sentences} + \text{num\_candidates}) \sim O(\text{num\_sentences})$. This is a huge improvement.
Also for finding the nearest reviewID, we do not need to go through all the data in the data set. Instead, we only need to spend very small computational time to parse the new input data and compute its min-hashing values. By comparing the assigned values in each buckets (complexity of $O(1)$ because we store the bucket using hash function.) Finally, we would come to around few candidates of the original data set (we can also set the threshold for the similarity to change this ratio). Looping through this small candidate data set, we would be able to reduce the computational cost. If we set the threshold of similarity to be 0.8, we would be able to further reduce the computational cost. In summary, using min-hashing twice (first on the raw data and second on the bands) would significantly reduce the pool size for comparison and further reduce the computational cost. Again, after the loading of data and dictionaries, we only need less than 10 seconds to identify the nearest reviewID together with its content for each new queried review.

8.2 True computational cost

The whole code finishes in around 15 minutes on a Macbook Pro with 9Th-generation 8-Core Intel Core i9 Processor. We should also mention that the last part on the main.py function that is computing the buckets information with $r = 2$ and $b = 300$ do not necessarily need to appear. This could be computed beforehand (stored and could be used directly in find_nearest_reviewID.py). In this case, the computational processes could be even faster.

Acknowledgement

We would like to thank professor Hassani and the TAs of this class for stimulating discussions. The codes and report is completely finished by the author himself.

References

[1] M. Datar, N. Immorlica, P. Indyk, V. S. Mirrokni, Locality-sensitive hashing scheme based on p-stable distributions, in: Proceedings of the twentieth annual symposium on Computational geometry, 2004, pp. 253–262.

[2] S. Har-Peled, P. Indyk, R. Motwani, Approximate nearest neighbor: Towards removing the curse of dimensionality, Theory of computing 8 (1) (2012) 321–350.

[3] A. Rajaraman, J. D. Ullman, Mining of massive datasets, Cambridge University Press, 2011.