ESE 545 Project 1

Report by **Saumya Shah** (58598022)

(Project in collaboration with
Megha Mishra (70763660))

1 Data loading and pre-processing

The json file was loaded into a pandas dataframe and the reviewID and reviewText columns were retained. As required for the problem 1, the data was pre-processed using the following steps:

- The alphabets were converted to all lower cases.
- The punctuation marks were removed using the extensive list of 32 punctuation marks.
- The reviews were split on spaces, the stop words were removed and the words were joined again with spaces.

2 Binary matrix representation

The main trick used here was using the number theory to convert the shingle to an integer index. Thus, a dictionary of 38 elements which included the following:
26 alphabets + 1 space character + 10 numbers + 1 '@' symbol to pad the reviews of length less than the shingle length chosen.

Thus, the spaces were considered as a character for the shingles and the reviews of lengths shorter than the shingle length were padded using a new character '@'.Following this, each of these characters was keyed to a number from 0 through 37 in the dictionary.
Therefore, the dictionary made was {'0': 0, '1': 1, '2': 2, '3': 3, '4': 4, '5': 5, '6': 6, '7': 7, '8': 8, '9': 9, 'a': 10, 'b': 11, 'c': 12, 'd': 13, 'e': 14, 'f': 15, 'g': 16, 'h': 17, 'i': 18, 'j': 19, 'k': 20, 'l': 21, 'm': 22, 'n': 23, 'o': 24, 'p': 25, 'q': 26, 'r': 27, 's': 28, 't': 29, 'u': 30, 'v': 31, 'w': 32, 'x': 33, 'y': 34, 'z': 35, ' ': 36, '@': 37}

And finally the shingles were represented as indices by weighing the least significant bit, i.e. the last letter with the weight of 38^0 and the most significant bit with the weight of 38^k where k is the length of the shingle. This may be thought of as an extended version of the Hexadecimal number system or in fact, any number system.
Thus, for example, if the shingle is "abc" it's index will be $10 * 38^2 + 11 * 38^1 + 12 * 38^0 = 14870$

After this, it was more of an implementation task. The string was looped over to check for the shingles and the corresponding index generated was stored in a list, one for each of the documents. The indices were then used as the row index and the document index was used as the column index and finally, the list of row indices and the corresponding column indices was passed through the coo sparse matrix (coo because the columns were sparse) initialization to generate the final required binary matrix representing the shingle set for each of the documents.

The value of k=4 seemed to work fine but to distinguish the pairs better, k=5 was used. Higher values of k weren't used because it may have decreased the chances of preserving similarity as it would have exaggerated the uniqueness and also increased the size of possible shingles exponentially which further would

have added to the computational complexity.

3 Random sample analysis

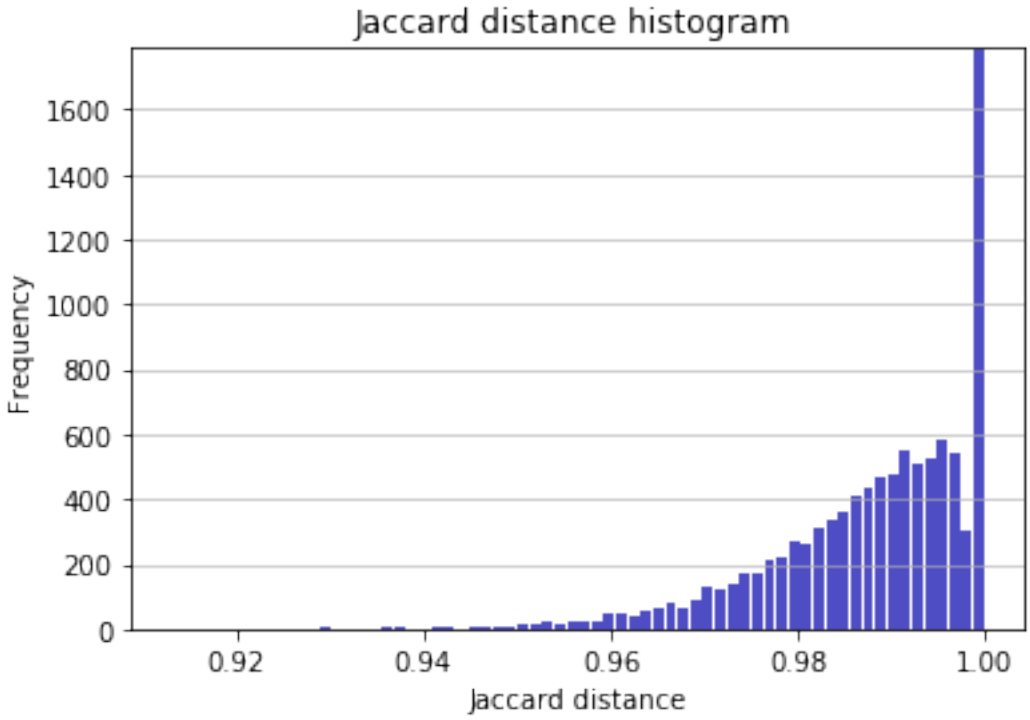

Figure 1: Histogram of Jaccard distance between 10000 random pairs of reviews

Average Jaccard distance: 0.9878543866538226
Minimum Jaccard distance: 0.9130434782608696

Thus as seen, the histogram has virtually zero number of pairs with Jaccard distance less than 0.9. This represent how less the number of pairs with Jaccard distance less than 0.2 would be as compared to the total number of pairs of reviews possible.
Almost 1700 out of the 10000 pairs have the Jaccard distance of 1, which means they had nothing in common. This gives us the sense of how small is the set of the similar reviews (Jaccard distance less than 0.2) is as compared to the set of all the possible pairs.

4 Hashing : "Efficient Data structure"

Since the sparse, yet huge, binary matrix is not the best representation to work with, a more efficient data structure was used. Locally Sensitive hashing, Min hashing was used for the purpose. For generating the random permutation, $(a * r + b)\%R$ was used where a and b were randomly chosen integers and R was the prime number after the length of the binary representation (or the number of possible shingles). The operations were vectorized with the help of numpy and use of broadcasting which could apply the permutation on all the reviews at once and thus, the implementation was done much faster. The number of hashes to generate the signature matrix was determined from the probability curves such that no true positives at the cost of possibly more false positives. The choice for m and the results are further elaborated in the next section.

5 Results:

Thus, a signature matrix was generated using the min-hashing process described in the previous section to efficiently represent the shingles in a more compact form. Now to find the similar pairs from the signature matrix, we hash the signature matrix again. A form of combination of "and" and "or" operations was used to control the number of true positives with the cost of the false positives.
The signature matrix was broken into b bands each containing r rows. Each of these rows in a band were hashed using a function and the sum of hashes was used to approximate the "and" operation as the sum of the hashes is likely to be the same only if all the hashes were equal. After this, the "or" operation was performed such that if at least one of the bands resulted in same value for the sum of hashes, then the corresponding reviews were classified as similar.

The implementation of this was done as follows:

1. For each band, r permutations were generated and finally r hash values were obtained for all the reviews in parallel using numpy broadcasting.
2. All the reviews resulting in the same value of the sum of hashes were paired together.
3. After the above steps were implemented for each of the bands, we finally manage to get unique groups of reviews which are later further processed to find the True positive pairs and filter the False positive pairs and since the number of pairs are lesser, the brute-force check takes only about a minute (64 seconds) to filter.

The values in the legend correspond to the values of r for the corresponding curve. Plots for various values of band width r for different number of hashes m using $p = 1 - (1 - s^r)^{m/r}$:

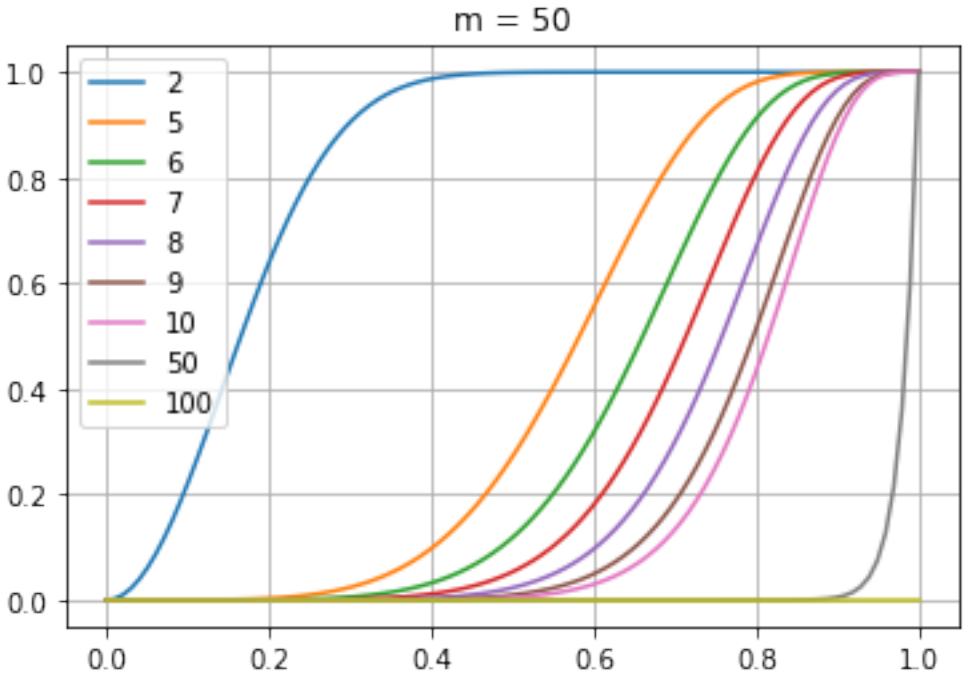

Figure 2: Probability vs Jaccard similarity curve for m=50

For m=50, the optimal value of r to capture all the True positives was 5 indicated by the yellow curve.

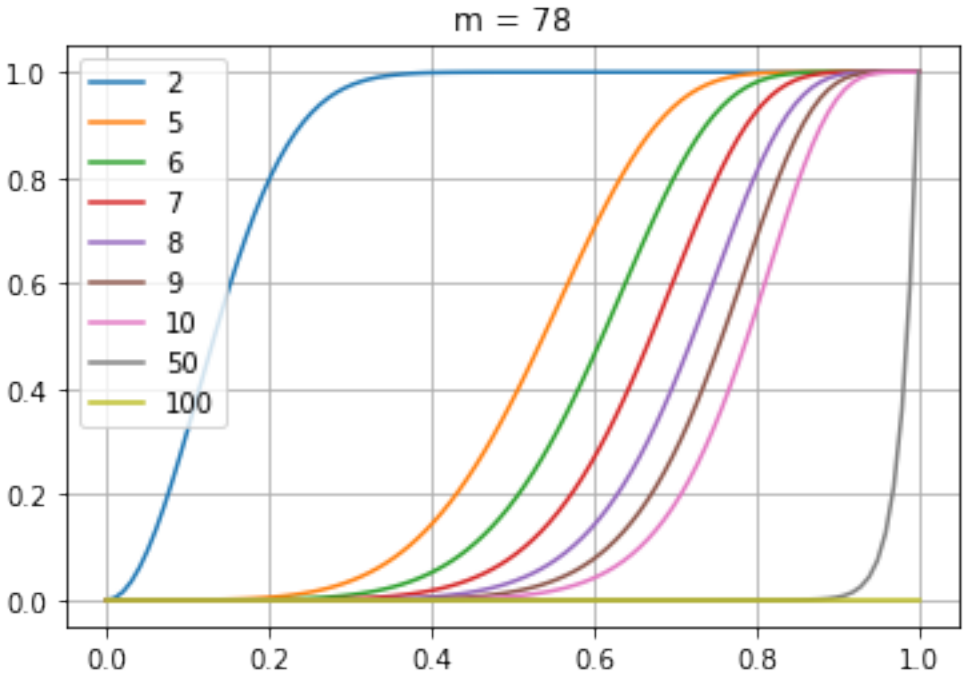

Figure 3: Probability vs Jaccard similarity curve for m=78

For m=78, the optimal value of r to capture all the True positives was 6 indicated by the green curve.

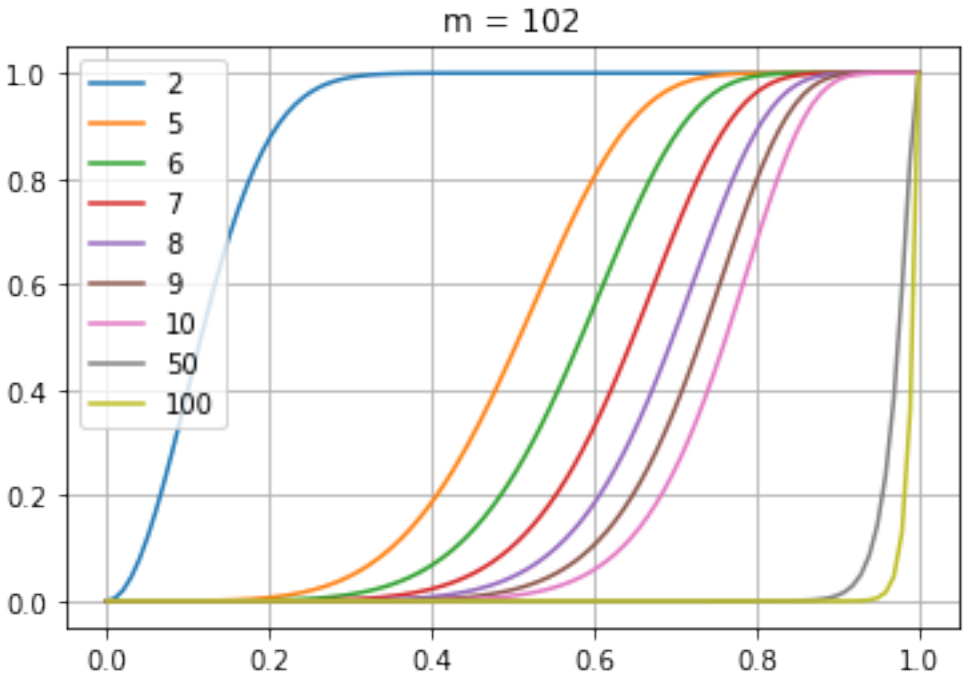

Figure 4: Probability vs Jaccard similarity curve for m=102

For m=102, the optimal value of r to capture all the True positives was 6 indicated by the green curve.

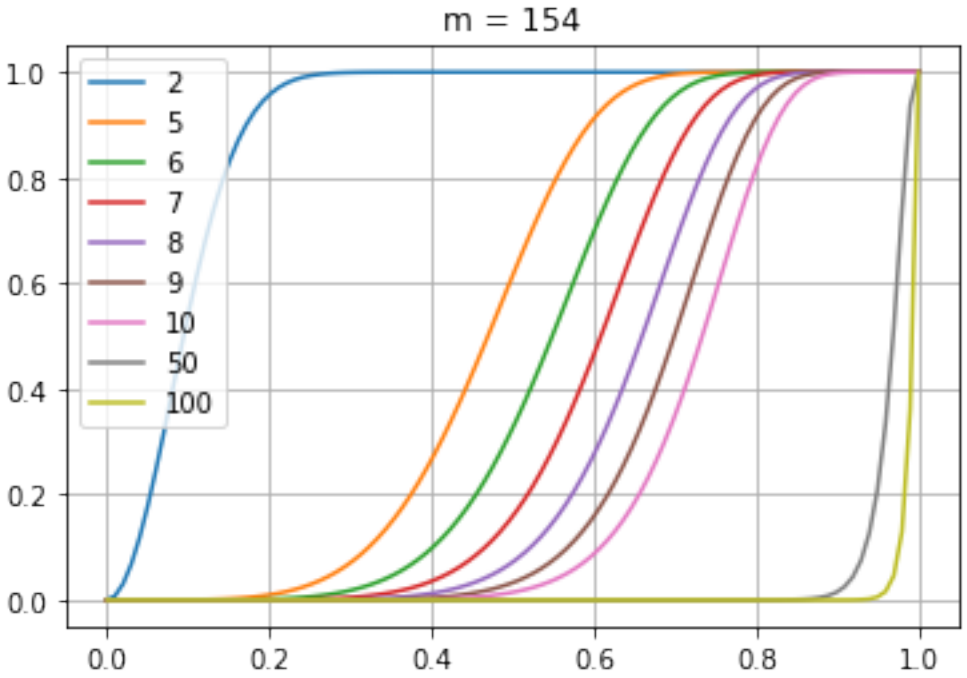

Figure 5: Probability vs Jaccard similarity curve for m=154

For m=154, the optimal value of r to capture all the True positives was 7 indicated by the red curve.

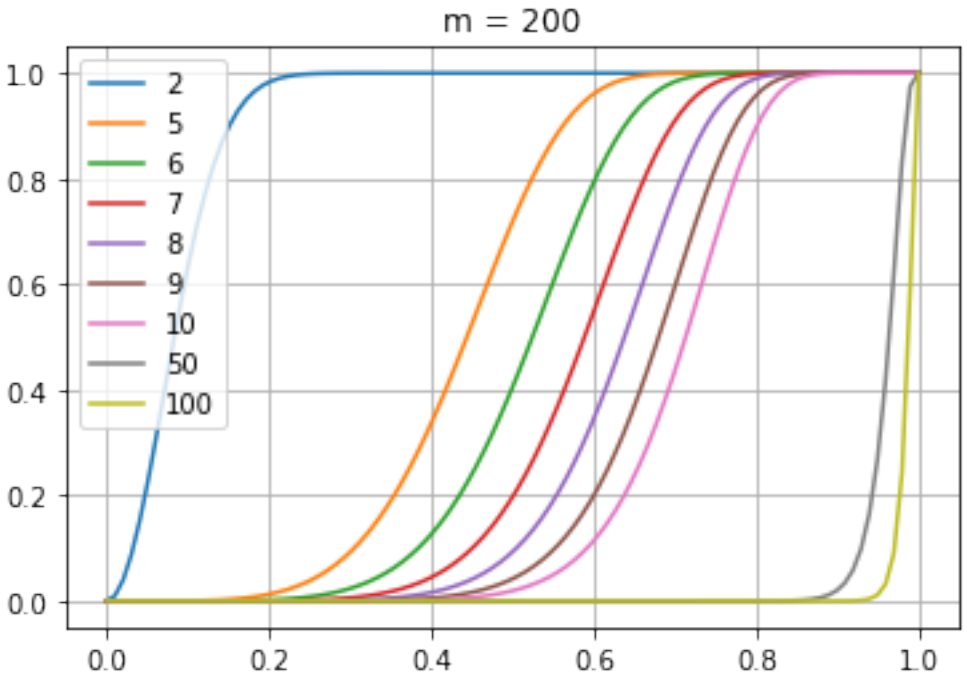

Figure 6: Probability vs Jaccard similarity curve for m=200

For m=200, the optimal value of r to capture all the True positives was 8 indicated by the violet curve.

The following is the tabulation of the results obtained for various values of m with the corresponding optimal value of r i.e. the number of rows per band.

	With null reviews		Without null reviews
	True Positives	False Positives	True Positives
m = 50 r = 5	13061	39080	1126
m = 78 r = 6	13058	41887	1123
m = 102 r = 6	13060	54050	1125
m = 154 r = 7	13061	65152	1126
m = 200 r = 8	13062	68369	1127

The term "null reviews" refer to the reviews which are either null by default or had only stop words which after pre-processing of reviews resulted in null reviews.

The column under "Without null review" refer to the number of True Positives of the actual non-null reviews.

The null reviews could have been filtered in the very beginning but weren't as the reviews with only stop words are essentially similar to each other as they convey almost nothing substantial and are thus also similar to the reviews which were empty originally. The results of these aren't stored in the .csv since we only wished to display the non-null reviews. Although the null reviews may also be stored by just using the pairs in "TP" instead of pairs in "TP_nonzero" in the code.

The results are finally stored in "results_102_6_5.csv" file. 102, 6, 5 are the values of m (number of hashes in signature matrix per review), r(size of the bands), k (length of the shingles) respectively.

The values of r were selected for any given m from the curve such that virtually no similar pairs should be missed. As the value of m increases the curve gets steeper and more flatter on the upper side indicating that the number of True negatives shall be lower. This is also seen in the table above as the number of true positives increases although by a very small amount, but is an indicative of the fact that the True negatives are reduced.

For the final submission, the values of m and r are 102 and 6 respectively as the time required for the computation vs the cost of performance seemed optimal for those values.

6 Test Your Own String

There are 2 ways to test your own string as follows (also explained in the README.md):
1. You may write your string before hand in the main file in the variable "str_val" and in the end, the main code writes all the similar reviews to the given review in the "matches_for_your_string.csv" file.

2. All the intermediate results from the previous steps can be stored in a .json file by commenting the code block from line 141 (class NumpyEncoder) to line 148 (json dump). Note that this is going to be a huge file of about 900 MB. Comment all the code following this and you may then use "get_similar_reviews.py" file to find matches for your string by setting its str_val variable appropriately. It shall save the results similarly

as in step one. This may help save time when testing for multiple strings with minimal changes in the code since you do not have to run the whole code in the main file as the results are pre-computed and stored.

This may be an interesting application to check for plagerism if pre-computed (or "trained") over a corpus of documents and using the "get_similar_reviews.py" to check (or "test") any given document for plagerism.

7 Complexity analysis

Following are the notations for the complexity analysis:
k = length of the shingle
n = number of reviews
m = number of hashes for the signature matrix
r = size of each band
kp = 38^5, which is the total number of possible shingles

7.1 Naive Algorithm

The Naive Algorithm would be to check the Jaccard similarity of each of the pairs from the shingles in the binary representation for each of the review, with each of the other reviews. If the Jaccard similarity is more than 0.8 as per our definition of similar pairs, it would be classified as similar. The total number of pairs would be O(n^2) and to check the Jaccard similarity for one pair, the computational complexity would be O(kp) and thus, the total computational complexity would be O($kp * n^2$).

7.2 Our implementation

1. Binary matrix representation:
   This step is probably common to both the methods. Each of the review is traversed over and shingles are processed to find the corresponding index for binary representation. This takes about 299 seconds for the given data set which includes the string pre-processing step.

2. Generating Signature Matrix:
   The Signature Matrix is made by Hashing each of the values in the binary representation of every review. Due to vectorization and broadcasting features of numpy, this takes only about 876 seconds for about 100 hash values. Though the actual computational complexity of this would be O(n*kp) since each of the shingle is permuted and then a minimum is found for min hashing for each of the reviews. It probably could be the most expensive step but still less expensive as compared to Naive Algorithm by an order of n. Although this, in practice, was cut down by a lot due to the usage of sparse matrix and vectorization.

3. Hashing the hashes and finding the similar pairs:
   This was done by summing the hash values of each of the rows in individual bands and checking for similar pairs since similar reviews would have same sum in at-least one of the bands. Processing similar pairs from one band was probably of the order of O(n) and was made faster using list comprehension. Thus, for b bands, it would have been of the order of O(b*n). For our case (m=102, r=6, number of bands=17), it took about 342 seconds.

Thus, we can safely conclude that our implementation is faster by orders of magnitude as compared to the naive implementation. Although it comes at the cost of some False Negatives and False Positives due to the probabilistic nature of the algorithm. The False positives can be filtered in much lesser time by brute-force filtering and the false negatives can be reduced by controlling the number of bands or the size of bands.