UNIVERSITY OF ATHENS
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

Big Data Mining Techniques

Student name: *Panagiotis Zazos, Athina Vekraki*
sdi: 7115112300009, 7115112300003

Course: *Big Data Mining Techniques (M161)*
Semester: *Fall Semester 2023*

Contents

1. Abstract

This study delves into the realm of text classification with a primary focus on news article categorization. Leveraging a dataset comprising 111,795 training and 47,912 test items, the research employs a blend of traditional (SVM, Random Forest) and advanced machine learning techniques (Neural Networks) to accurately classify articles into four distinct categories: Business, Entertainment, Health, and Technology.
The classification task is approached through the implementation of Support Vector Machines (SVM) and Random Forests, evaluated on the basis of Bag of Words (BoW) and Singular Value Decomposition (SVD) feature extraction methods.
To surpass established benchmarks, a deep learning architecture was developed, integrating Embedding, Bidirectional LSTM, Dropout, and Dense layers. Hyperparameter tuning was conducted through the Optuna optimization framework, ensuring the model's optimal performance.
Finally, this study also leveraged advanced computational resources, specifically utilizing the power of an RTX GeForce 3060 GPU for GPU acceleration through Keras and CUDA.

2. Pre-Processing

The preprocessing stage of this study was designed to refine and prepare the textual data for subsequent analysis. This process started with the application of regular expressions to cleanse the text, effectively removing non-alphanumeric characters and ensuring a uniform lowercase format for all words.
To enhance the quality of the input data, stopwords (commonly occurring words that offer little to no semantic value) were systematically excluded from the dataset.
Furthermore, the process of lemmatization was implemented to unify various inflected forms of words into their base or dictionary form, thereby simplifying the vocabulary and minimizing repetition.

3. WordClouds

In the visualization phase of this study, word clouds were generated to provide a graphical representation of the most frequent words within each article category. Individual word clouds were created for each distinct category found in the dataset. The process involved aggregating the text of all articles within a category and then generating a word cloud from this collective text.

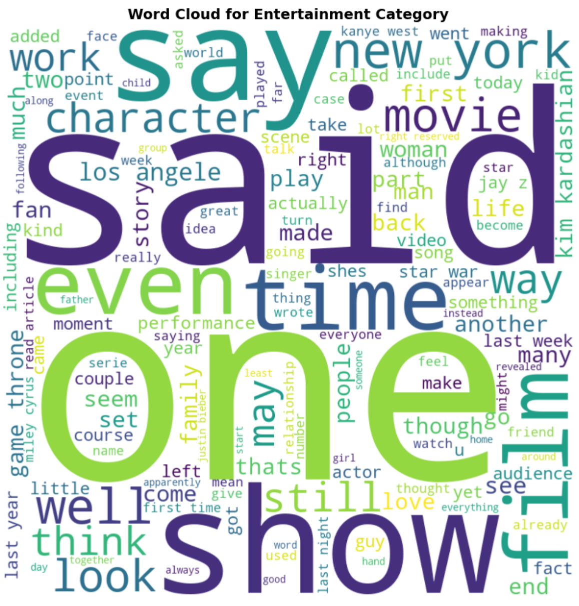

Figure 1: Entertainment

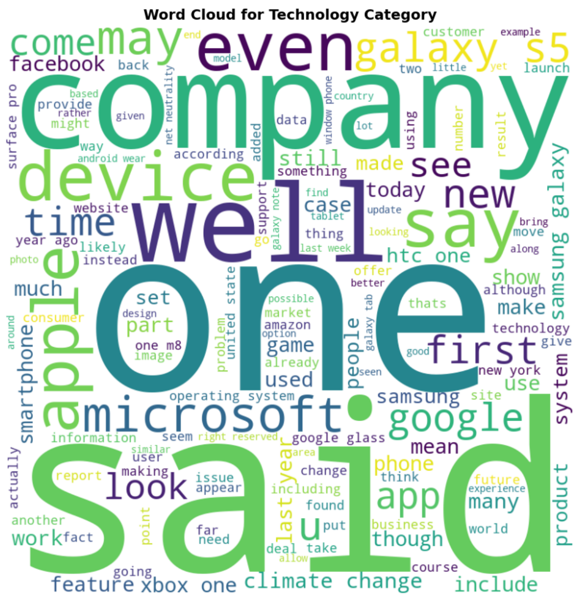

Figure 2: Technology

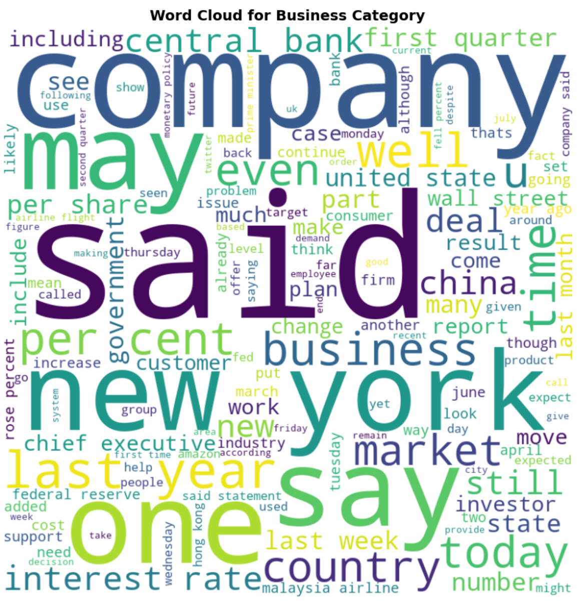

Figure 3: Business

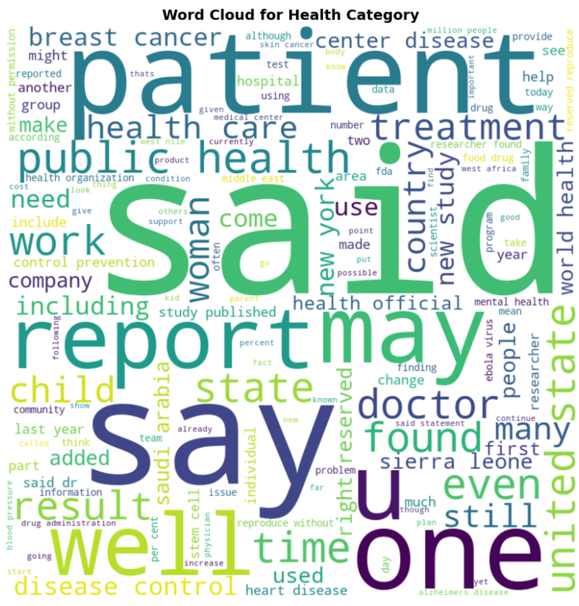

Figure 4: Health

4. SVM & Random Forest Implementation

4.1. Classification Pipelines

In the classification task of this study, a systematic approach was adopted by constructing four distinct pipelines, each designed to evaluate the performance of two classification algorithms—Support Vector Machines (SVM) and Random Forests—across two different feature extraction techniques: Bag of Words (BoW) via HashingVectorizer and dimensionality reduction using Truncated Singular Value Decomposition (SVD). The HashingVectorizer transforms text documents into a fixed-size numerical format (hash space) without the need for storing a vocabulary dictionary in memory. Unlike other methods (CountVectorizer and TF-IDF), HashingVectorizer avoids keeping the entire set of words in memory. Moreover, by limiting the number of features through the n_features parameter, the HashVectorizer helps mitigate the effects of the curse of dimensionality, which states that the volume of the space increases so rapidly that the available data becomes sparse.

- **SVM with BoW Pipeline (SVM BOW)**: This pipeline integrates the HashingVectorizer with 16,384 (2̂14) features to transform the text data into a high-dimensional sparse matrix, representing the Bag of Words model. This is followed by the LinearSVC classifier, configured to operate in its linear form with the dual formulation turned off for efficiency in large datasets.
- **Random Forest with BoW Pipeline (Random Forest BOW)**: Similar to the SVM BOW pipeline, this one also employs the HashingVectorizer for feature extraction. The classifier is a RandomForestClassifier, with 100 trees (n_estimators=100) and a maximum tree depth of 10 (max_depth=10), optimized for paraller computation across all available CPU cores (m_jobs=1).
- **SVM with SVD Pipeline (SVM SVD)**: This pipeline uses the same HashVectorizer for initial text transformation. Following this, a TruncatedSVD component reduces the dimensionality of the feauture space to 50 components, capturing the most significant variance. Lastly, this reduced data is classified using the LinearSVC classifier.
- **Random Forest with SVD Pipeline (Random Forest SVD)**: Finally, this pipeline mirrors the SVM SVD pipeline, utilizing the HashingVectorizer and the TruncatedSVD component for feature extraction and dimensionality reduction, respectively. The RandomForestClassifier is then applied to the reduced set, using the same hyperparameters as in the Random Forest BoW pipeline.

Each pipeline was evaluated using a 5-fold Cross Validation (CV) approach, utilizing the KFold method with shuffling enabled to ensure random distribution of data across folds. The performance of each pipeline was assessed based on three key metrics: **Accuracy**, **Precision** (macro-averaged), and **Recall** (macro-averaged).

4.2. Scores

Model	Accuracy	Precision	Recall
SVM BoW	96.43%	96.26%	95.98%
Random Forest BoW	75.21%	85.30%	67.18%
SVM SVD	89.23%	88.60%	87.14%
Random Forest SVD	87.96%	88.04%	84.88

Table 1: Performance metrics of text classification models

5. Beat The Benchmark

In the quest to surpass the benchmark performances established by traditional classification and feature extraction techniques, this study ventured into the realm of deep learning by building a Recurrent Neural Network (RNN) with Long Short-Term Memory (LSTM) units.
The decision to employ a Long Short-Term Memory (LSTM) Recurrent Neural Network (RNN) for the task of text classification was raised by the fact that news articles are inherently sequential; the meaning and sentiment conveyed by a sentence are not only dependent on the individual words but also on their order and contextual relationship with each other. LSTMs introduce ways to deal with limitations of RNNs, such as vanishing and exploding gradient problems. Standard RNNs fail to capture long range dependencies with the text. While not delving too deep into the 'hows' and 'whys' of LSTMs, they overcome these issues, allowing them to retain important historical context and forget irrelevant information.

5.1. Tokenization and Data Generators

5.1.1. Tokenization.
Utilizing the Tokenizer from Keras library, the news article content is being convertied into a format processable by the neural network. The num_words variable plays a critical role in this process. The maximum size of the tokenizer's vocabulary is decided to be of size 10,000, meaning that the tokenizer will only take into account the 10,000 most frequent words in the dataset. This allows the model to focus on the most relevant parts of the text, as well as mitigating the issue of overfitting by not incorporating biases coming from very rare words, although 10,000 words is still a large vocabulary of distinct and lemmatized words! The oov_token = <OOV> sets a token to be used for out-of-vocabulary (OOV) words.
The tokenizer, through the fit method, develops a mapping from words to integers, specifically the Content of the articles. This mapping is created based on word frequency, with the most common word getting the value of 1, the next getting value of 2, etc.
Once the tokenizer has been fitted to the Content data, the conversion to list of sequences take place. Each article in the Content column is transformed into a sequence of integers. Each word in the text is replaced by its corresponding integer index from the previously constructed mapping of words to integers.

5.1.2. Sequence Length.
To ensure that each input tensor has the same shape, which is required for batch processing in neural networks, a maximum length of sequence must be decided. The approach here was to take the average sequence length of all articles, instead of the maximum, to prevent the model from learning from a lot of empty paddings. Therefore, smaller articles will be padded with zeros at the end (padding='post') and larger articles will be truncated to the average sequence length. The average length of sequences (articles) is 249 as shown below.

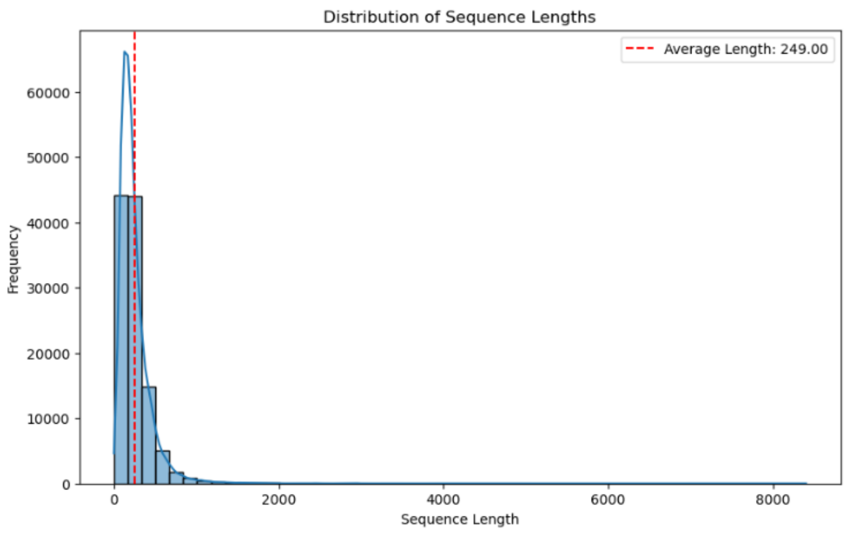

Figure 5: Average Sequence Length

5.1.3. Data Generators.
Large datasets, as in our case, can exceed the system's memory capacity. It's enfeasible to load the entire dataset into memory at once. These data generators offers by Keras, allow for dynamic loading and processing of data in batches during the model training. They yield batches of data only when needed, thus optimizing memory usage. Moreover, it was particularly beneficial in the case of this study, as the model was being trained using GPU, for the GPU was being fed with a constant stream of data without waiting for CPU operations to load and preprocess the next batch.

5.2. Neural Network Structure

The neural network's architecture was designed to leverage the sequential nature of text data, as mentioned before, embodying an Embedding layer, Bidirectional LSTM layers, a Dropout layer for regularization, and a Dense layer for output classification, as shown in **Table 2**.
The structure of this particular neural network commences with an embedding layer, designed to map the input vocabulary into a 200-dimensional vector space, containing 2,000,000 parameters. This layer was parameterized to handle a vocabulary size of 10,000 words, a significant increase from an initial setting of 1,000 words. This adjustment was informed by empirical evidence suggesting enhanced model accuracy without the risk of overfitting, resulted by the analysis of confusion matrices.
Following the embedding layer, is a bidirectional LSTM layer, which outputs a 512-dimensional vector for eac time step in the sequence, collectively consisting of 935,936 parameters. This structure allows the netword to learn from the sequence data in all directions, both forward and backward.
A dropout layer is then applied with no additional parameters, serving as a regularization technique to prevent overfitting by randomly omitting a subset of the features during training.
Following that is a standard LSTM layer with 787,456 parameters, which further processes the sequential data. Optuna provided a higher score when using this layer instead of only one LSTM layer.

The final layer is a dense layer with 1,028 parameters, which serves as the output layer of the network.

Layer (type)	**Output Shape**	**Param #**	
embedding (Embedding)	(None, 249, 200)	2,000,000	
bidirectional (Bidirectional)	(None, 249, 512)	935,936	
dropout (Dropout)	(None, 249, 512)	0	
lstm_1 (LSTM)	(None, 256)	787,456	
dense (Dense)	(None, 4)	1,028	
		Total params:	3,724,420
		Trainable params:	3,724,420
		Non-trainable params:	0

Table 2: Detailed architecture of the LSTM Neural Network

5.3. Optuna hyperparameters optimization

During the optimization process, Optuna's study evaluated ten trials, systematically exploring a predefined grid of hyperparameters, as shown in **Table 3**, to identify the most effective configuration for the LSTM neural network. The num_units parameter, which determines the complexity of the LSTM layers, was varied among 64, 128, and 256. The dropout_rate was tuned between 0.1 and 0.5 to find the optimal rate that would prevent overfitting while allowing sufficient learning. The choice of optimizer, critical for the convergence speed and stability of the training process, oscillated between 'adam' and 'rmsprop'. Although the number of LSTM layers was held constant at two (shown better performance in previous trials), the batch_size varied among 32, 64, and 128, impacting the gradient estimation and memory utilization during training. Lastly, the embedding_dim, controlling the dimensionality of the input word vectors, was chosen from 100, 200, and 300, influencing the expressiveness of the word representations.

Hyperparameter	**Values**
num_units	64, 128, 256
dropout_rate	0.1 to 0.5
optimizer	adam, rmsprop
num_lstm_layers	2 (fixed)
batch_size	32, 64, 128
epochs	10 (fixed)
embedding_dim	100, 200, 300

Table 3: Hyperparameter Grid for Optuna Optimization

Regarding the importance of hyperparameters, as illustrated in **Figure 5**, dropout_rate emerged as the most influential factor, significantly impacting model performance. This highlights the importance of regularization in model selection. The batch_size was the second most significant parameter, affecting both the computational efficiency and the stability of the learning process. Other parameters, such as optimizer, embedding_dim, and num_units, also contributed to the overall performance, but to a lesser extent, indicating a more nuanced influence on the model's behavior.

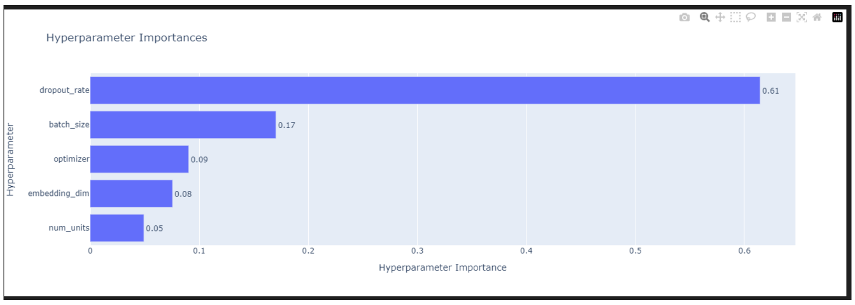

Figure 6: Hyperparameter Importance

5.4. Train and Validation

As mentioned before, the construction of the data generators, train_generator and validation_generator, proved mandatory to facilitate the efficient handling of the large-scale textual data. These generators dynamically load and preprocess the data in batches (BATCH_SIZE). The model was trained over 10 epochs, with each epoch iterating through batches of data. The number of steps per epoch (steps_per_epoch) was calculated to ensure that each sample in the training set was presented to the model once per epoch.

As shown in **Figure 7**, both accuracy and loss graphs indicate that the model is performing really well on both the training and the validation sets.
An important observation is that there are no apparent signs of overfitting or underfitting. Overfitting would be alerted when once observes a significant gap between training and validation accuracy or an increase in validation loss while training loss decreases.
The model, also, seems to have reached its learning capacity by the 4th epoch. Beyond that point the increase is minimal, therefore running more epochs may not lead to better performances and could potentially lead to overfit.

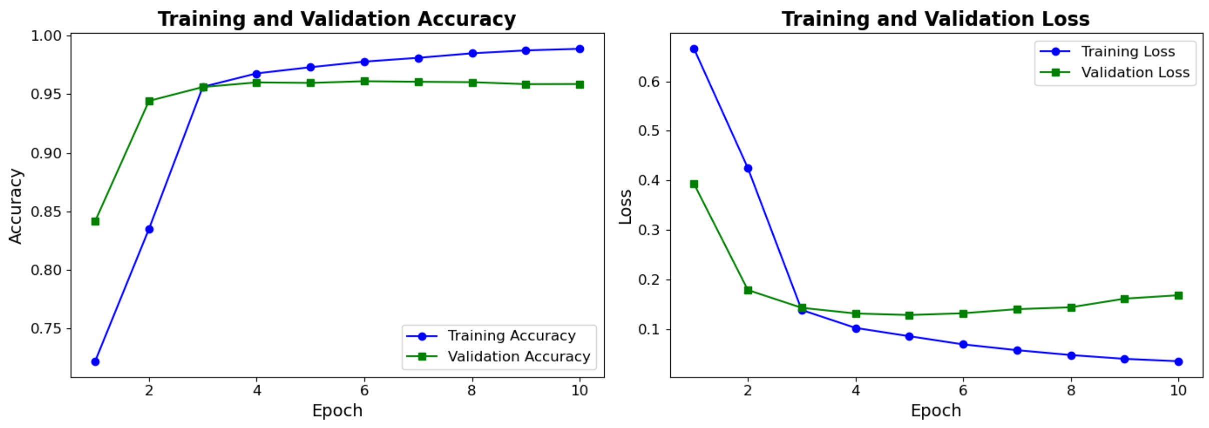

Figure 7: Accuracy and Loss over time

5.5. Scores

The macro-average was used to treat all classes equally when calculating the precision and recall metrics, particularly through the precision_recall_fscore_support function from Scikit-learn. Accuracy was determined using the accuracy_score function.

Category	Precision	Recall	F1-Score	Support
Business	0.93	0.93	0.93	4967
Entertainment	0.99	0.98	0.98	8967
Health	0.97	0.96	0.96	2404
Technology	0.94	0.95	0.94	6021
Accuracy	0.9587			
Precision (Macro)	0.9552			
Recall (Macro)	0.9549			

Table 4: Classification Performance Metrics

5.6. Confusion Matrix

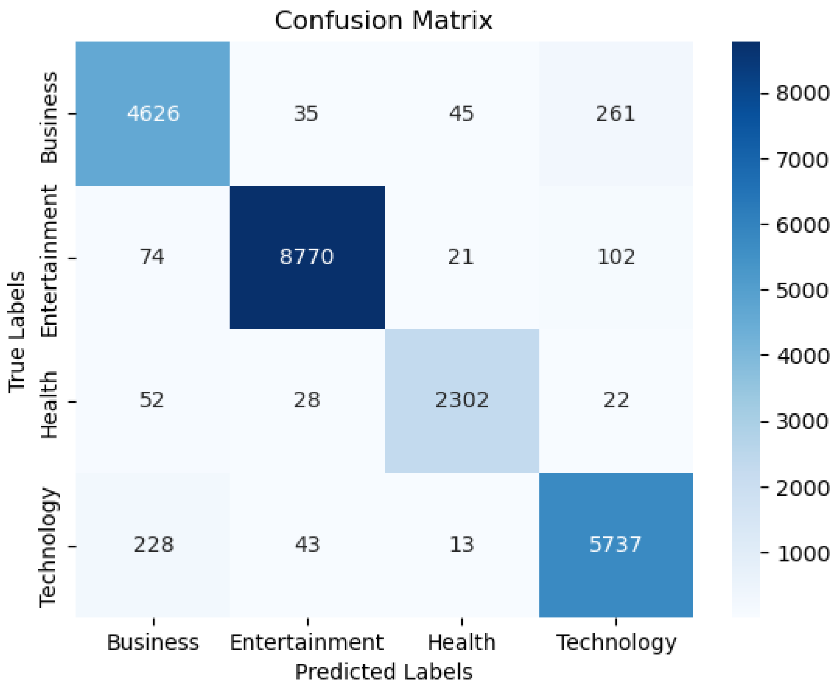

Figure 8: Confusion Matrix

As it is obvious when studying the confusion matrix above, the model performs best in classifying Entertainment, with the highest number of true positives and relatively low numbers of false positives and negatives. A slight misclassification occurs when trying to differentiate between Business and Technology, as indicated by the 228 instances where Technology was misclassified as Business.
The confusion matrix provided is based on the validation/test set (the 0,2 split of the dataset that the model has not seen during training, since we were not provided any specific validation dataset).

5.7. SVM, Random Forest and Our Model Scores

Model	Accuracy	Precision	Recall
SVM BoW	96.43%	96.26%	95.98%
Random Forest BoW	75.21%	85.30%	67.18%
SVM SVD	89.23%	88.60%	87.14%
Random Forest SVD	87.96%	88.04%	84.88%
Our Model	95.87%	95.52%	95.49%

Table 5: Performance metrics of text classification models

6. Speeding up K-NN Classification

The purpose of this project is to enhance the efficiency of the K-NN (K-Nearest Neighbors) classification method by leveraging the Locality Sensitive Hashing (LSH) technique. Specifically, the project aims to compare the performance of the traditional brute-force approach with an LSH-based approach for identifying candidate pairs of documents with high similarity.

6.1. Implementation Steps and Evaluation Metric

6.1.1. Preprocess. The preprocessing of the data in this project follows the same steps as outlined in Part 1 (Data Loading, Data Cleaning, Text Preprocessing).

Then the preprocessed content (X) and corresponding labels (y) are loaded from a DataFrame. The data are split into training and testing sets using an 80-20 ratio. For vectorization, the CountVectorizer is utilized to convert the text data into numerical representations.

6.1.2. Evaluation Metric. The Jaccard Similarity metric is utilized to measure the similarity between pairs of documents. The function takes two sparse matrices doc1 and doc2 as input and converts their rows into sets of indices. It then computes the intersection and union of these sets to calculate the Jaccard Similarity, defined as the ratio of the size of the intersection to the size of the union. If the union is empty (indicating no common elements between the documents), the function returns 0.

6.2. Brute-Force Method

In the brute-force approach, each document in the test set is compared to every document in the train set using the Jaccard Similarity metric. This method is straightforward but can be computationally expensive, especially for large datasets.

A KNeighborsClassifier is instantiated and fitted using the brute-force approach for similarity computation, utilizing the Jaccard Similarity metric. The classifier is configured to use 15 nearest neighbors (`n_neighbors=15`) and leverage parallel processing (`n_jobs=-1`). Then, an example of computing the brute-force Jaccard similarity for a single test instance is demonstrated. The kneighbors method is invoked on the classifier object to retrieve the 15 nearest neighbors for the given test instance, and the elapsed time for this operation is recorded using the time module.

6.3. Locality Sensitive Hashing (LSH) technique

The LSH technique is used to identify candidate pairs of documents where the expected similarity is above a certain threshold (initially set to τ=0.8). The Min-Hash LSH is utilized for this purpose, with the number of permutations set to 16, 32, 64. By using LSH, the goal is to reduce the number of comparisons needed, thereby improving the efficiency of the K-NN classification.

`minhash_document`: This function creates a MinHash signature for a given document represented as a vectorized form (e.g., TF-IDF vector). It iterates over the indices of non-zero elements in the vector and updates the MinHash with each index.

`build_lsh_index`: This function constructs an LSH index using the training data represented as TF-IDF vectors. It iterates over each document in the training set, generates its MinHash signature using the `minhash_document function`, and inserts it into the LSH index.

`find_nearest_neighbors_lsh`: This function finds the nearest neighbors for each document in the test set using the LSH index. It iterates over each test document, generates its MinHash signature, and queries the LSH index to retrieve candidate neighbors.

The code then proceeds with evaluating the LSH approach by comparing it with brute-force computation. It iterates over different numbers of permutations, builds the LSH index, and performs LSH queries on the test data. The true nearest neighbors are obtained using brute-force computation with Jaccard Similarity. The fraction of true K most similar documents that LSH method also returns is computed and stored in `fraction_correct_lsh`.

Finally, the evaluation results are printed, including the number of permutations, LSH index creation time, query time, total time, and the fraction of true K most similar documents returned by the LSH method. This information provides insights into the efficiency and effectiveness of using LSH for nearest neighbor search based on Jaccard Similarity.

6.4. Experiments

The plotted evaluation metrics offer insights into the performance characteristics of the Locality Sensitive Hashing (LSH) method for nearest neighbor search based on Jaccard Similarity. The index creation time generally increases with the number of permutations, reflecting the computational overhead of generating MinHash signatures. Similarly, query time exhibits a similar trend, highlighting the additional computational cost incurred during nearest neighbor retrieval. Consequently, the total time, comprising both index creation and query time, escalates with the number of permutations. Meanwhile, the fraction of true K most similar documents identified by LSH demonstrates its efficacy in approximating nearest neighbors, with higher fractions indicating better performance. These visualizations aid in understanding the trade-offs between computational efficiency and accuracy in LSH-based nearest neighbor search, facilitating informed decisions when configuring LSH parameters for specific applications.

Due to the large size of the dataset, we executed the code on a dataset comprising 10,000 documents.

The plots for threshold 0.8 and 10000 documents:

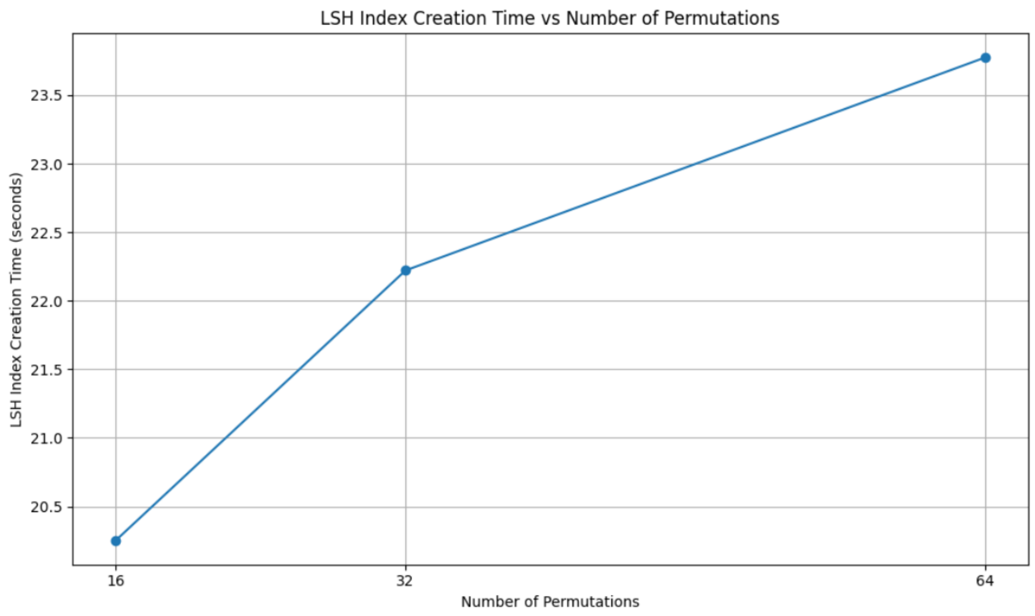

Figure 9: Creation Time - Number of Permutations

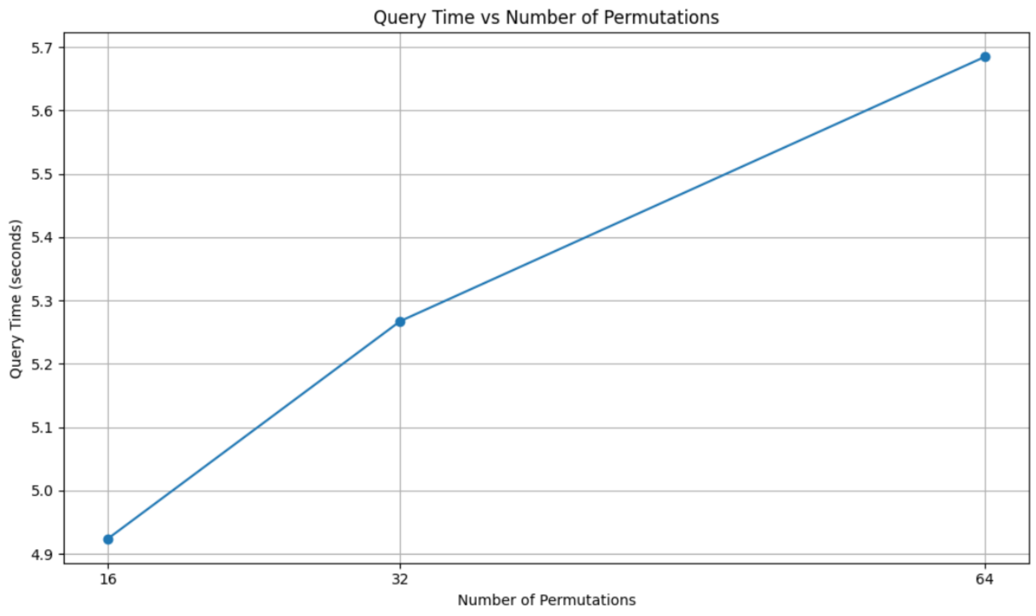

Figure 10: Query Time - Number of Permutations

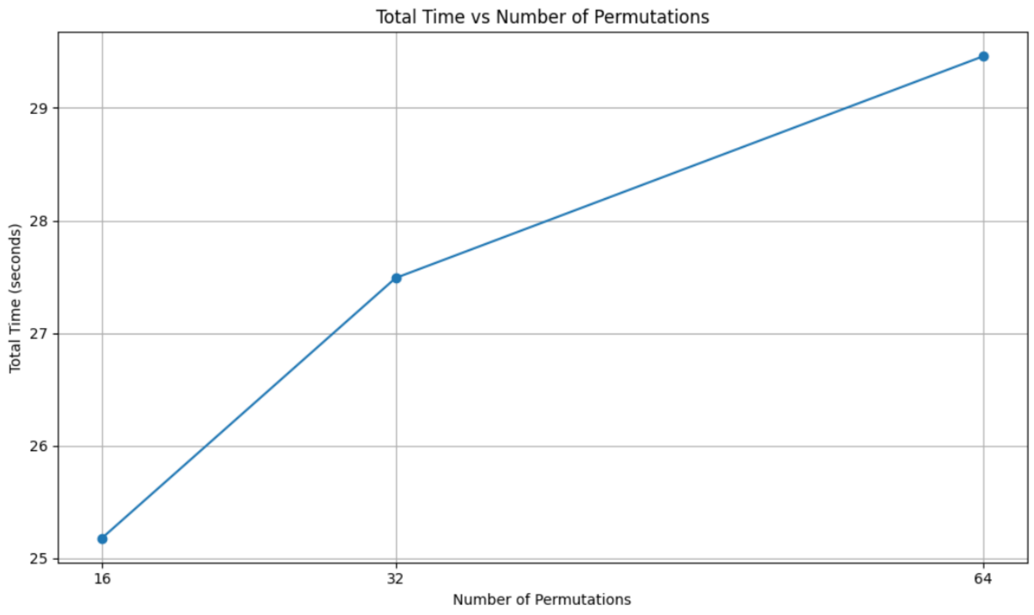

Figure 11: Total Time - Number of Permutations

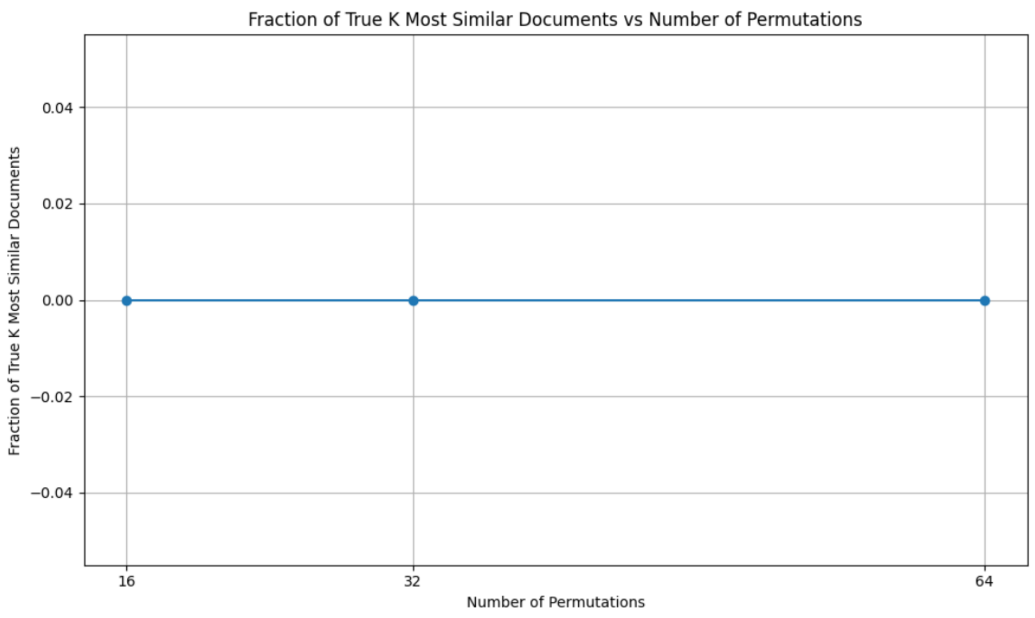

Figure 12: Fraction of True K Most Similar Documents - Number of Permutations

Type	BuldTime	QueryTime	TotalTime	Fraction of the true K most similar documents that are reported by LSH method as well	Parameters (different row for different K or for different number of permutations, etc)
Brute-Force-Jaccard		5106.27	5106.27	100%	-
LSH-Jaccard	20.25	4.92	19.95	0%	Perm=16
LSH-Jaccard	22.22	5.26	21.05	0%	Perm=32
LSH-Jaccard	23.77	5.68	23.04	0%	Perm=64

It was observed that the LSH method failed to identify any neighbors when employing a threshold of 0.8. This outcome suggests that the chosen threshold might have been too stringent for the given size of the dataset (10000 documents), resulting in a lack of candidate pairs deemed similar enough for further comparison.

Results:
LSH Results: [[], [], [], [], []] **True Similarities:** [2592, 6946, 6726, 1735, 7913, 3531, 5773, 2542, 607, 3309, 5746, 6291, 3383, 5752, 2815, 5502, 5506, 2340, 2374, 2342, 2344, 7305, 5454, 558, 5424, 5494, 5434, 2332, 574, 575, 640, 641, 1473, 1568, 3433, 5737, 6380, 943, 5552, 2865, 112, 2578, 1592, 2619, 4799, 3618, 7913, 5035, 652, 4589, 1710, 3627, 3440, 4307, 5045, 6998, 3420, 3512, 667, 6044, 4353, 4610, 450, 3489, 1473, 4073, 6891, 6380, 5037, 1902, 5387, 2096, 7895, 6652, 6111]
LSH Results: [[], [], [], [], []] **True Similarities:** [2592, 6946, 6726, 1735, 7913, 3531, 5773, 2542, 607, 3309, 5746, 6291, 3383, 5752, 2815, 5502, 5506, 2340, 2374, 2342, 2344,

7305, 5454, 558, 5424, 5494, 5434, 2332, 574, 575, 640, 641, 1473, 1568, 3433, 5737, 6380, 943, 5552, 2865, 112, 2578, 1592, 2619, 4799, 3618, 7913, 5035, 652, 4589, 1710, 3627, 3440, 4307, 5045, 6998, 3420, 3512, 667, 6044, 4353, 4610, 450, 3489, 1473, 4073, 6891, 6380, 5037, 1902, 5387, 2096, 7895, 6652, 6111]

LSH Results: [[], [], [], [], []] **True Similarities:** [2592, 6946, 6726, 1735, 7913, 3531, 5773, 2542, 607, 3309, 5746, 6291, 3383, 5752, 2815, 5502, 5506, 2340, 2374, 2342, 2344, 7305, 5454, 558, 5424, 5494, 5434, 2332, 574, 575, 640, 641, 1473, 1568, 3433, 5737, 6380, 943, 5552, 2865, 112, 2578, 1592, 2619, 4799, 3618, 7913, 5035, 652, 4589, 1710, 3627, 3440, 4307, 5045, 6998, 3420, 3512, 667, 6044, 4353, 4610, 450, 3489, 1473, 4073, 6891, 6380, 5037, 1902, 5387, 2096, 7895, 6652, 6111]

Then we reduced the threshold to 0.5.

The plots for threshold 0.5 and 10000 documents:

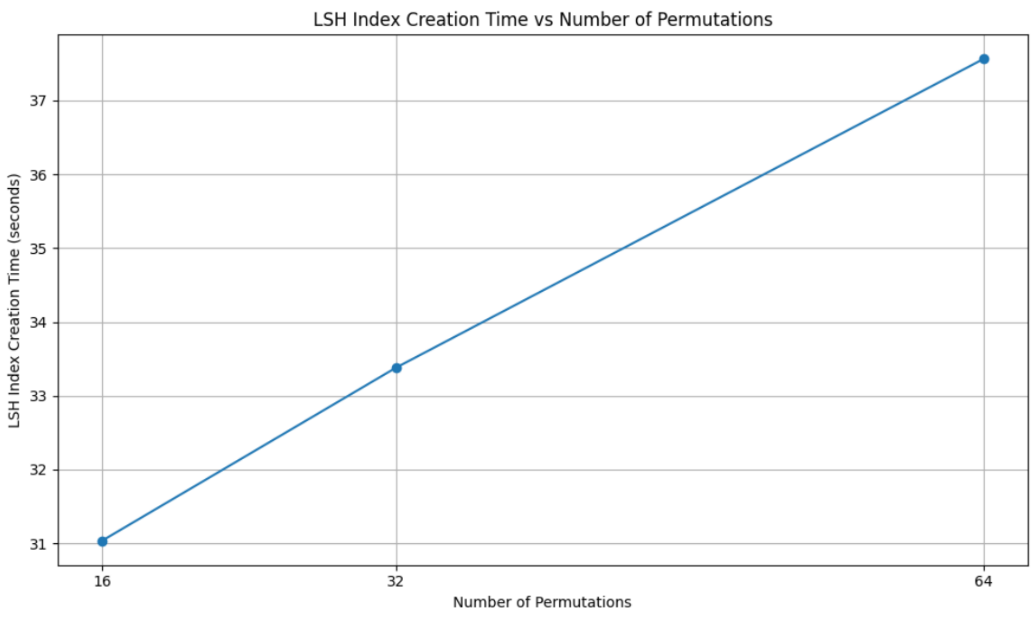

Figure 13: Creation Time - Number of Permutations

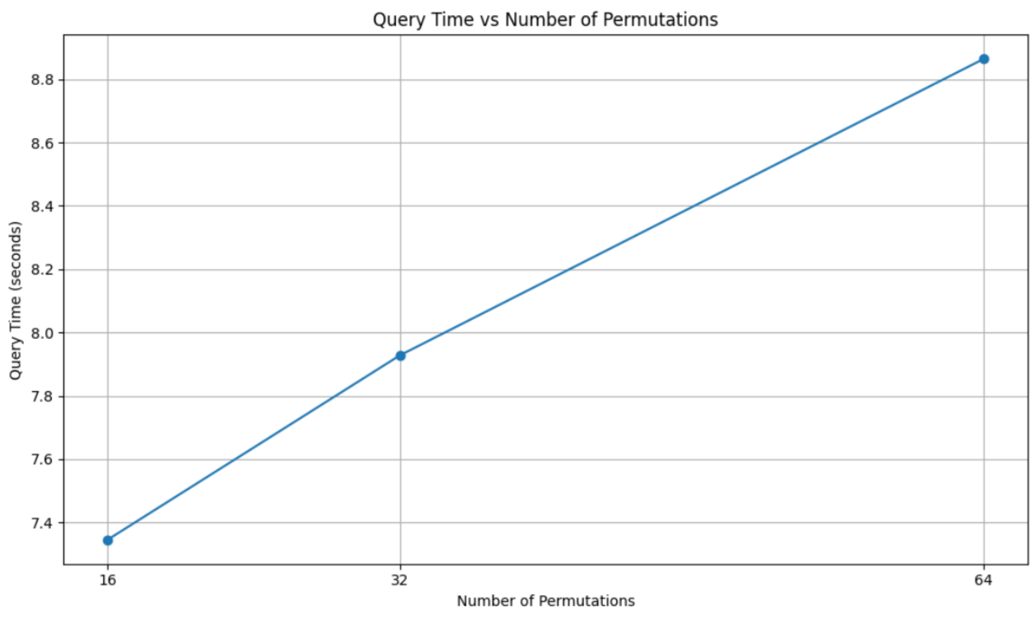

Figure 14: Query Time - Number of Permutations

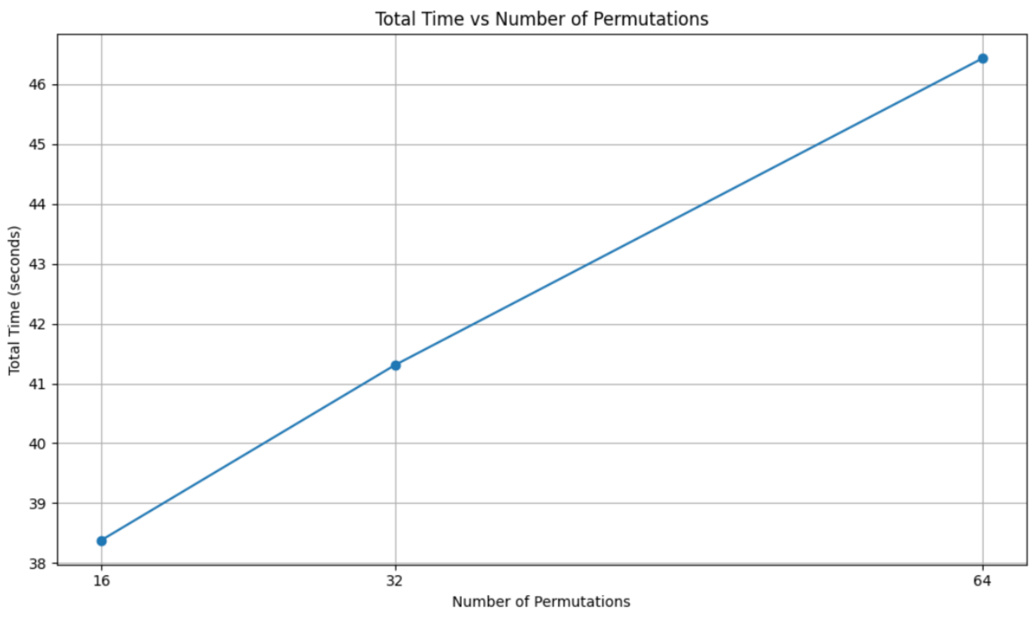

Figure 15: Total Time - Number of Permutations

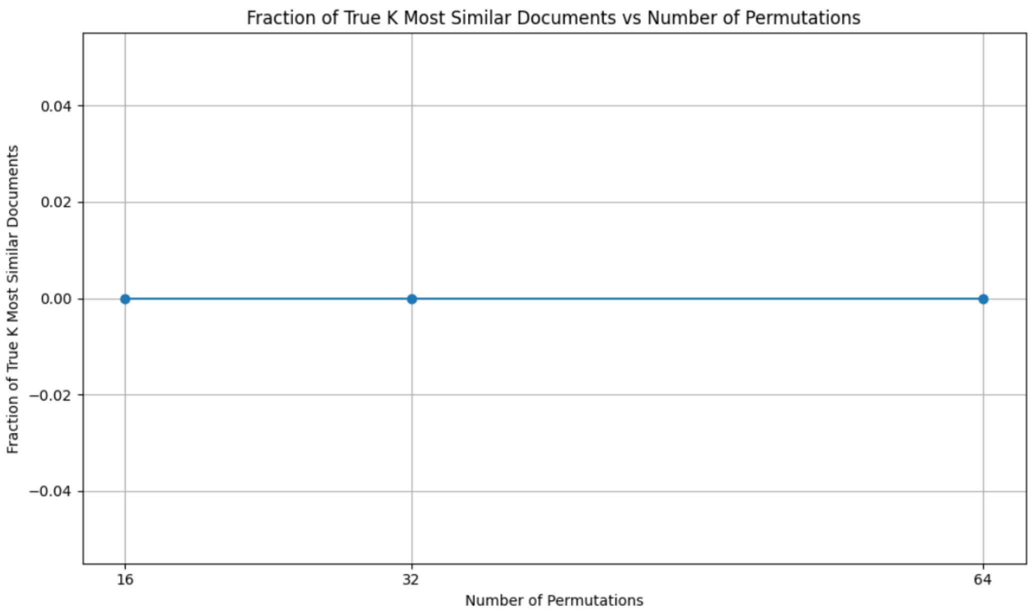

Figure 16: Fraction of True K Most Similar Documents - Number of Permutations

Results:

LSH Results: [[1539], [5260, 3348, 527], [], [], []] **True Similarities:** [2592, 6946, 6726, 1735, 7913, 3531, 5773, 2542, 607, 3309, 5746, 6291, 3383, 5752, 2815, 5502, 5506, 2340, 2374, 2342, 2344, 7305, 5454, 558, 5424, 5494, 5434, 2332, 574, 575, 640, 641, 1473, 1568, 3433, 5737, 6380, 943, 5552, 2865, 112, 2578, 1592, 2619, 4799, 3618, 7913, 5035, 652, 4589, 1710, 3627, 3440, 4307, 5045, 6998, 3420, 3512, 667, 6044, 4353, 4610, 450, 3489, 1473, 4073, 6891, 6380, 5037, 1902, 5387, 2096, 7895, 6652, 6111]

LSH Results: [[], [], [], [], []] **rue Similarities:** [2592, 6946, 6726, 1735, 7913, 3531, 5773, 2542, 607, 3309, 5746, 6291, 3383, 5752, 2815, 5502, 5506, 2340, 2374, 2342, 2344, 7305, 5454, 558, 5424, 5494, 5434, 2332, 574, 575, 640, 641, 1473, 1568, 3433, 5737, 6380, 943, 5552, 2865, 112, 2578, 1592, 2619, 4799, 3618, 7913, 5035, 652, 4589, 1710, 3627, 3440, 4307, 5045, 6998, 3420, 3512, 667, 6044, 4353, 4610, 450, 3489, 1473, 4073, 6891, 6380, 5037, 1902, 5387, 2096, 7895, 6652, 6111]

LSH Results: [[], [], [], [], []] **True Similarities:** [2592, 6946, 6726, 1735, 7913, 3531, 5773, 2542, 607, 3309, 5746, 6291, 3383, 5752, 2815, 5502, 5506, 2340, 2374, 2342, 2344, 7305, 5454, 558, 5424, 5494, 5434, 2332, 574, 575, 640, 641, 1473, 1568, 3433, 5737, 6380, 943, 5552, 2865, 112, 2578, 1592, 2619, 4799, 3618, 7913, 5035, 652, 4589, 1710, 3627, 3440, 4307, 5045, 6998, 3420, 3512, 667, 6044, 4353, 4610, 450, 3489, 1473, 4073, 6891, 6380, 5037, 1902, 5387, 2096, 7895, 6652, 6111]

Type	BuldTime	QueryTime	TotalTime	Fraction of the true K most similar documents that are reported by LSH method as well	Parameters (different row for different K or for different number of permutations, etc)
Brute-Force-Jaccard		5096.22	5096.22	100%	-
LSH-Jaccard	20.57	4.68	25.26	0%	Perm=16
LSH-Jaccard	21.69	4.97	26.67	0%	Perm=32
LSH-Jaccard	24.20	5.86	30.06	0%	Perm=64

In this case, for the LSH-Jaccard method with Perm=16, although it managed to find some neighbors, none of them corresponded to the true similars identified by the brute-force method.

The failure to match true similars in this instance may be attributed to the fact that we executed the code on only a subset of the dataset due to its large size, which significantly reduced the runtime but may have compromised the accuracy of the LSH method's results.