UNIVERSITY OF ATHENS
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

Big Data Mining Techniques

Student name: *Konstantinos Plas & Konstantinos Nikoletos*
71151122200025 & 7115112200022

Course: *M161*
Semester: *Fall 2023*

Part 1: Text classification

Data

For this task we had to read two files. We used pandas python package.

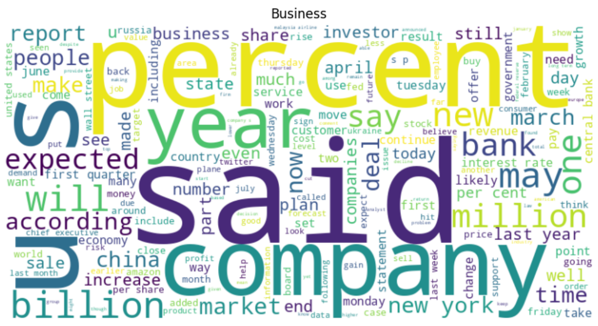
- train.csv: Contains the columns and number of records same as the description. [Shape: (111795, 4)]
- test_without_labels.csv: Contains the columns and number of records same as the description. [Shape: (47912, 3)]

Pre-processing. The given dataset was raw and had not been cleaned. So, a preprocess technique was used in order to clean and format the dataset to get the best scores. Our pre-processing method:

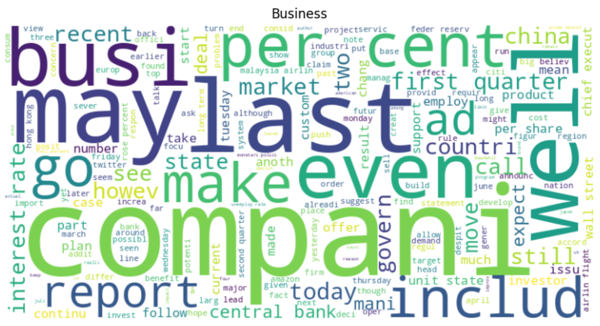
- Converts the entire text to lowercase.
- Removes non-alphanumeric characters and replaces them with a space.
- Removes single-letter words surrounded by spaces.
- Removes single-letter words at the beginning of the text.
- Replaces consecutive spaces with a single space.
- Removes the prefix 'b' at the beginning of the text.
- Lemmatizes each word using a lemmatizer.
- Filters out words that are in a predefined set of stop words. The set of stopwords created by the set provided by nltk and pypi package stop_words in combination with a hardcoded set that we added after viewing the wordclouds.

Question 1.1: WordClouds

In this section we will see the wordclouds for each category {Business, Entertainment, Health, Technology} before and after the pre-processing.

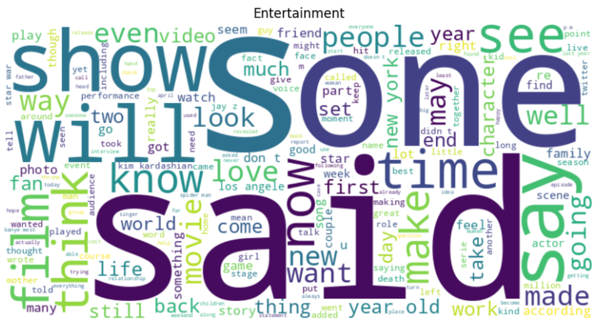


(a) **Without** pre-processing

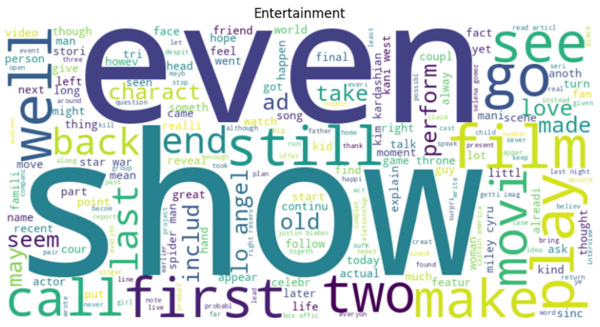


(b) **With** pre-processing

Figure 1: Business Category

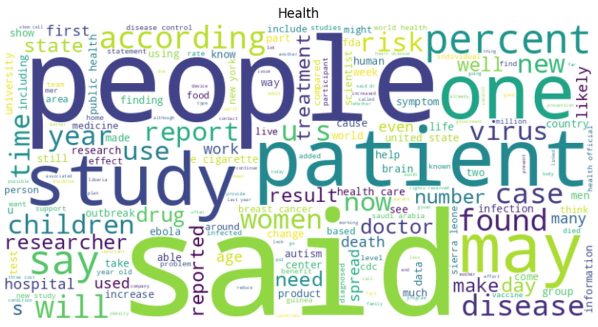


(a) **Without** pre-processing

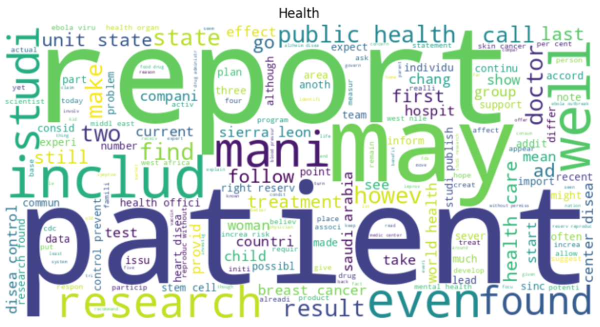


(b) **With** pre-processing

Figure 2: Entertainment Category



(a) **Without** pre-processing



(b) **With** pre-processing

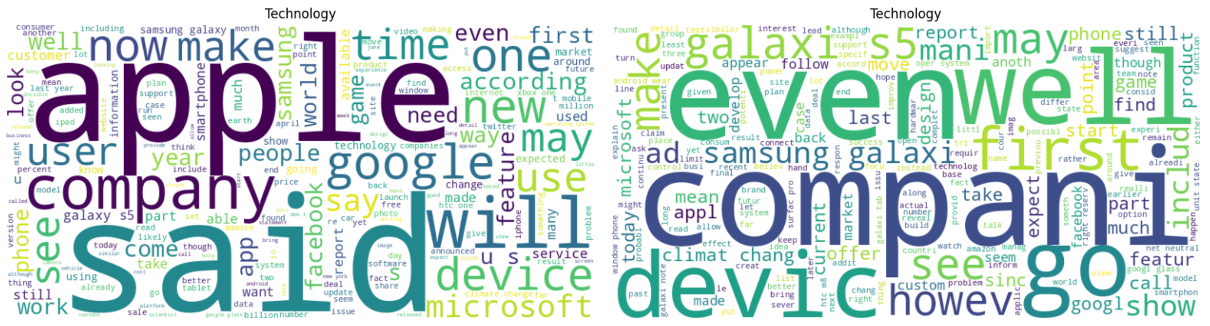
Figure 3: Health Category

Remarks

We can see multiple words removed, and the words after pre-processing describe and are relevant to the category. Words like: said, year and other verbs have been removed. This will be really useful, when we will use TfIdf vectorizer that needs the text to "best describe" the cluster.

Also another point that our pre-processing is successful, is that unigrams and many contextually same words have been reduced. For example Health wordcloud, has words ["one", "said", "will"] as the most most frequent. After pre-processing, most frequent words are ["report", "patient"] that are indeed relevant with the category.

Of course, many other words, could be also have been removed, as for example ["may", "even"] but we decided not to go on an extensive pre-processing, so we have some misses, that



(a) **Without** pre-processing (b) **With** pre-processing

Figure 4: Technology Category

still do not affect our results.

Question 1.2: Classification Task

For the classification task, we experimented with:

- Support Vector Machines (SVM)
- Random Forests

and:

- Bag of Words (BoW)
- Singular Value Decomposition (SVD)

Results

Table 1 contains the performance off all methods tested. We can see that **Bag-Of-Words** yields better results that SVD technique. This is expected as, **SVD** is dimensionality reduction technique that aims to reduce the memory and time complexity of the problem without losing much information and hence decrease significantly the scores. In this experiment SVD lowers the scores at approximately 1% which is a nice trade-off for the difference it occurred in runtime.

SVM and **RandomForest** are two of the most popular classifiers. SVM seeks to find the optimal hyperplane that maximizes the margin between classes in the feature space, making it effective for binary and multi-class classification. Random Forest is an ensemble learning method that constructs multiple decision trees during training and aggregates their predictions through voting. Their performance is relevant to the data characteristics and imbalance. In this experiment we can see that RandomForest is better than SVM in all cases and combinations. We experimented with both of these algorithms with their hyper-parameters to achieve these scores.

Statistic Measure	**SVM (BoW)**	**Random Forest (BoW)**	**SVM (SVD)**	**Random Forest (SVD)**	**My method**
Accuracy	0.9191	0.9265	0.9053	0.9140	**0.9724**
Precision	0.9120	0.9234	0.9008	0.9124	**0.9709**
Recall	0.9077	0.9136	0.8887	0.8946	**0.9693**

Table 1: Q1 Results

My Model

We experimented with 2 classifiers, XGBoost and with LinearSVC.

LinearSVC (Linear Support Vector Classifier) is a variant of the Support Vector Machine (SVM) algorithm specifically designed for linear classification tasks. LinearSVC aims to find the optimal hyperplane that separates classes in the feature space while maximizing the margin between the classes. Unlike the standard SVM (SVC), which uses a nonlinear kernel to map the input features into a higher-dimensional space, LinearSVC relies on linear kernels, making it computationally efficient, particularly for large datasets.

XGBoost (Extreme Gradient Boosting) is an advanced implementation of the gradient boosting algorithm. It constructs an ensemble of weak learners, typically decision trees, iteratively improving upon the residuals of the previous models to minimize a user-defined loss function. XGBoost incorporates regularization techniques to prevent overfitting, handles missing values in the input data, and supports parallel and distributed computing for scalability.

However, after our experimentation, we found LinearSVC to work better, under this configuration:

- Input: Cleaned and pre-processed text
- Vectorization: TfIdf
- Hyperparams: random_state=42 & tol=1e-5 & penalty = 'l2'

Code & Implementation

Part1.py file contains all the the code needed for this task. Input data was the cleaned and pre-processed text. For the baseline methods we vectorized the dataset using CountVectorizer. Cross validation was implemented with StratifiedKFold and number of folds equal to 5. This way we ensure that each fold of the cross-validation procedure maintains the same class distribution as the original dataset.

To execute Part1.py, run *python Part1.py*, after configuring the dataset path.

Kaggle contest

For the Kaggle contest

- Team name: **Nikoletos & Plas**
- Score: **0.97105**

Part 2: Nearest Neighbor Search with Locality Sensitive Hashing

Dataset

The dataset is the same with the previous question and was pre-processed in the same manner.

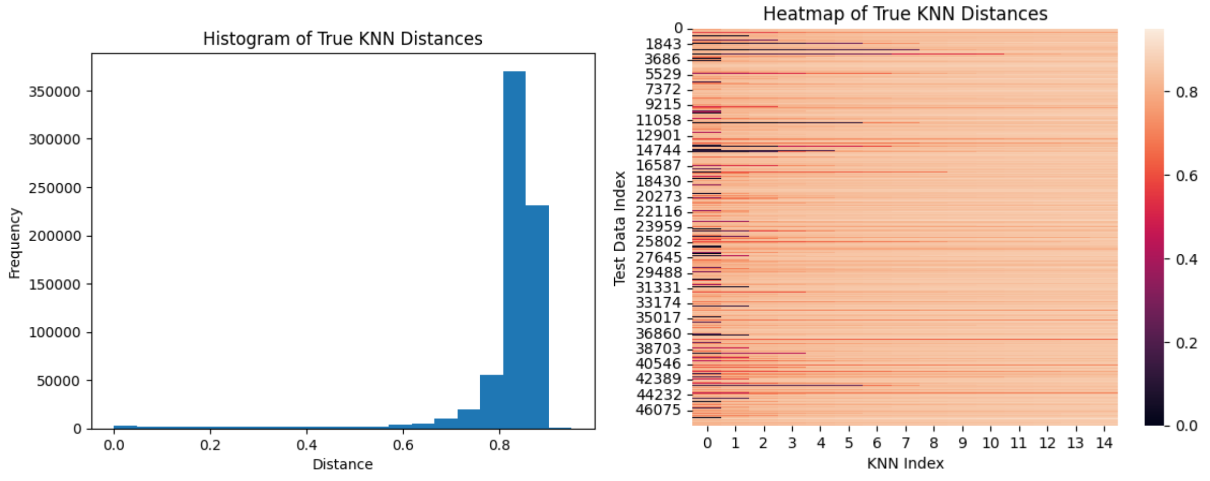
Implementation

For this task we tried two main techniques and their variations. Firstly, comparing every question with every question and generally duplicate/similarity detection over big datasets is almost impossible. That's due to the great cost both in memory and mostly in computation time.

KNN. As proposed in the tutorial, we used scikit-learn **KNN** method. This method takes as input the vectorized dataset. For the vectorization we tried **Bag-Of-Words** and **Tf-Idf** techniques, with the first resulting in better results. Because KNN is extremely expensive and each of our experiments, needed from 8 to 18 hours, we used a lower dimensional vectors of *size 512*. Also, **k was set to 15**, as requested. However we tried also increasing the dimensions of vectors to 1024 and k to 100, but it exploded the runtime and we dropped this experiments. For the KNN we also employed from 1 to 4 processes, as we noticed that using more that 4 processes didn't scale well in the parallelization.

We used the jaccard method, provided by scipy, that is an implementation of the Jaccard Distance.

Data Scaling. KNN is many times used with scaled data and produces even better results. For this reason we tried scaling our vectors after Bag-Of-Words technique. This experiment was due to the fact that Jaccard distance resulted into very few similar documents and we couldn't compare it with LSH with high threshold in a productive way. For this reason we created some heatmaps of the distances returned by the KNN and Jaccard. Figures 5 depict the frequencies of Jaccard distances produced by KNN.



(a) Histogram (b) Heatmap

Figure 5: Jaccard Distance Variance

MinHash-LSH

MinHash is a probabilistic data structure used for estimating the similarity between sets. The basic idea is to represent each set as a signature, which is a small "sketch" of the set's contents. LSH is a method for hashing data points in a way that similar items are mapped to the same "buckets" with high probability.

For this task, we used **datasketch MinHash LSH** implementation, as requested. After our literature review and understanding the concept of LSH we came across with two alternative techniques. The first one is the hashing using datasketch implementation on the vectors produced and fed into the KNN. And the other was to hash the original given text. For the document retrieval and duplicate detection tasks, the second technique is widely used. However, in our task, because we miss a ground-truth file and the KNN is the "true labels", we decided to test both of these cases. As, our goal is to speed-up the KNN method. Meaning that we want to reduce the search space as LSH effectively does.

Basic concept. The core of our implementation relies on building an Index with the train data by saving each token/value and its signature and then Querying the test set with it. Such techniques, work fast as, for each document, we get a set of candidates to compare from the train dataset. This reduces extremely the search space, as instead of $O(n^2)$ complexity we go to $O(nk)$.

LSH using vectors as signatures

In this section we will study and remark the vector LSH. In this technique, every vector is given to the Index and Queries are made with vectors.

Type	**BuildTime**	**QueryTime**	**TotalTime**	**Fraction of true K**	**Parameters (LSH Permutation)**	**LSH Threshold**
Brute-Force-Jaccard	0	28243.58	28243.58	100%	-	-
LSH-Jaccard	459.95	298.07	758.03	47.07%	Perm=16	0.8
LSH-Jaccard	482.15	271.72	753.87	43.12%	Perm=32	0.8
LSH-Jaccard	493.39	282.11	775.51	47.70%	Perm=64	0.8
LSH-Jaccard	566.50	352.67	919.17	59.50%	Perm=128	0.8
LSH-Jaccard	452.07	598.76	1050.84	89.37%	Perm=16	0.6
LSH-Jaccard	467.92	639.16	1107.09	89.17%	Perm=32	0.6
LSH-Jaccard	495.16	747.27	1242.44	91.21%	Perm=64	0.6
LSH-Jaccard	565.09	1049.72	1614.82	94.60%	Perm=128	0.6
LSH-Jaccard	454.68	1193.11	1647.80	99.45%	Perm=16	0.4
LSH-Jaccard	468.22	1314.95	1783.17	99.54%	Perm=32	0.4
LSH-Jaccard	498.42	1417.92	1916.35	99.27%	Perm=64	0.4
LSH-Jaccard	572.55	2252.13	2824.68	99.89%	Perm=128	0.4

Table 2: Results with Vector hashing

Remarks on the Vector hashing. In table 2 we see the results of the vector based approach. Here are some remarks:

- We get the candidates (for example at threshold = 0.6) at 28243.58/1107.09 approximately 25 times faster and we lose only 10%. This is a good trade-off between decreasing the score and making the application faster.
- As we increase number of permutations, we get a higher fraction but also the build and query time increases. That's the expected behaviour as more signatures lead to more buckets and more candidate documents
- As we lower the threshold, more candidates fall in the same bucket, and this is why we get better results. It's an analogous behavior with the increase in number of permutations.

LSH using word-tokens as signatures

In this technique, each document is spitted in tokens, i.e the words that is consisted. Each token becomes a signature and this way we build the Index. However this technique, yields poor top-K fraction results. That's due to fact that we take KNN as the "true labels". Many documents have really low similarity, however they appear in the KNN. So when we set a threshold of 0.8, all these documents are not taken into account from the LSH Index and are being pruned. At the same time, another reason that yields poor results, is that KNN was calculated upon the Jaccard of the vectors, whereas the MinHash LSH method, over the set of tokens. This is a different technique and comparing between them may not be effective. It would be more accurate to calculate KNN with the token-Jaccard distance between the sets of words and then apply LSH. But this would be extremely costly to calculate. In the table 3 we see the results of the token based approach.

Type	**BuildTime**	**QueryTime**	**TotalTime**	**Fraction of true K**	**Parameters (LSH Permutation)**	**LSH Threshold**
Brute-Force-Jaccard	0	28243.58	28243.58	100%	-	-
LSH-Jaccard	154.32	66.55	220.88	12.75%	Perm=16	0.8
LSH-Jaccard	163.23	70.76	233.99	12.73%	Perm=32	0.8
LSH-Jaccard	179.75	77.06	256.81	12.96%	Perm=64	0.8
LSH-Jaccard	219.34	92.48	311.83	13.10%	Perm=128	0.8
LSH-Jaccard	156.80	67.39	224.20	07.56%	Perm=16	0.6
LSH-Jaccard	164.04	70.17	234.22	11.68%	Perm=32	0.6
LSH-Jaccard	181.31	77.07	258.39	13.95%	Perm=64	0.6
LSH-Jaccard	221.25	92.66	313.91	14.65%	Perm=128	0.6

Table 3: Results with Token hashing

Here are some remarks:

- LSH has again a similar behavior with the vector-based approach.
- The time complexity is also even better than the previous approach.
- Increasing the number of permutations, increases both the candidates and hence the number of documents found (Figure 6).

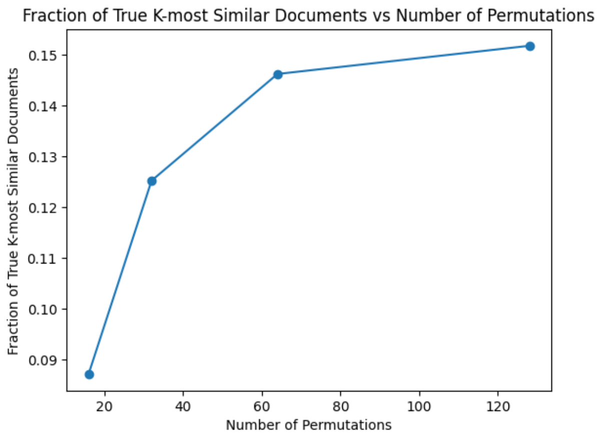
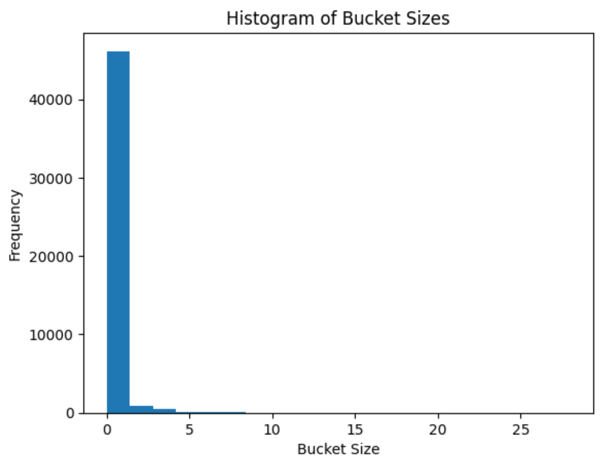


Figure 6: Fraction per increasing permutation number

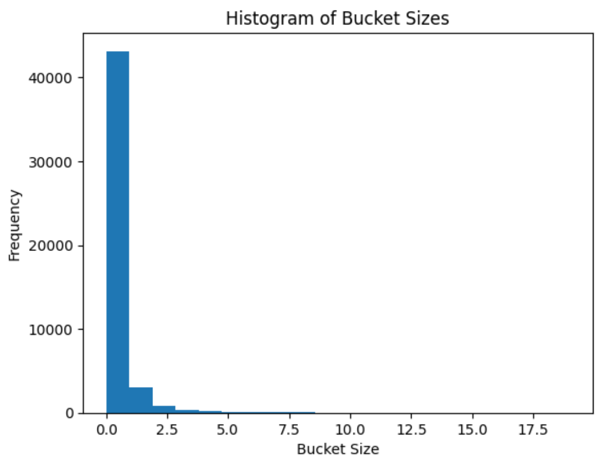
Also, for the token-blocking method we printed and evaluated the candidate sets. A nice example can be viewed from Figures 7 and 8. We see that most of the buckets for threshold=0.8 have sizes mostly 0 and among 1 and 20 candidates. Also, as we increase the number of permutations, the bucket size is also slightly increasing.

Code

Find the relevant code in the Part2.py (contains vector-based implementation) and Part2_tokens.py (contains token-based implementation) files. To execute run *python Part2.py* .

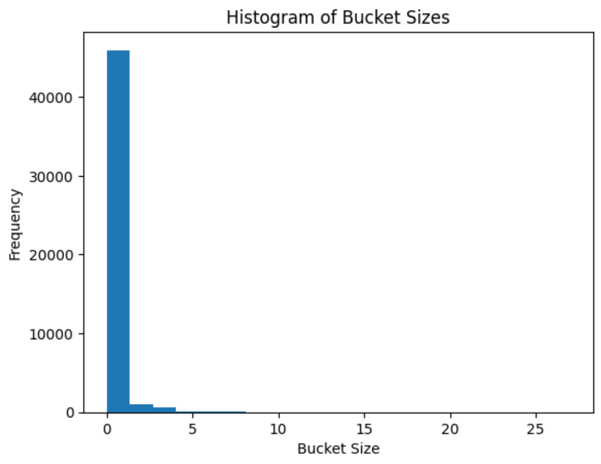


(a) Permutation=16

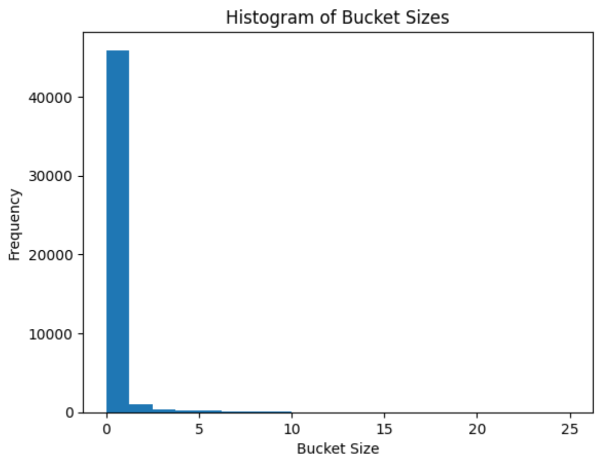


(b) Permutation=32

Figure 7: Bucket sizes for threshold 0.8 and permutations=(16,32)



(a) Permutation=16



(b) Permutation=32

Figure 8: Bucket sizes for threshold 0.8 and permutations=(64,128)

Part 3: Nearest Neighbor Search and Duplicate Detection

Data

Dataset was parsed and read with pandas package. Train dataset contains 3000 questions and test dataset 1000 questions.

Cosine Similarity: Random projection LSH family

We firstly, calculated the exact cosine similarity using cosine_similarity metric from **sklearn**. We applied cosine in the vectors created from **CountVectorizer**. We have implemented two main methods for this task:

- **exact_cosine**: Calculates exact cosine similarity between the vectors of test and train dataset.
- **random_projection_hashing**: Performs **GaussianRandomProjection** to the vectors given. The technique is similar with the one proposed on the homework tutorial. It first trans-

fromrs the vectors to binary and afterwords produces a hash for each vector. This hash is the identifier of each bucket. This way, when we traversed the test set questions, we only had to calculate the similarity between the test question and the bucket formed.

Jaccard Similarity: Min-Hash LSH family

For the exact Jaccard similarity calculation, we created a Jaccard Similarity that takes each question as a set of tokens. Over these sets we calculated all the similarities between all the questions.

Following datasketch implementation and guides, we created a MinHash Index from the train dataset and the we queried all test documents with this index.

Type	**BuildTime**	**QueryTime**	**TotalTime**	**#Duplicates**	**Parameters**
Exact-Cosine	-	0.024	0.024	89	-
Exact-Jaccard	-	18.10	18.10	26	-
LSH-Cosine	0.04	5.39	5.44	76	K=1
LSH-Cosine	0.04	4.81	4.92	79	K=2
LSH-Cosine	0.08	5.24	5.32	67	K=3
LSH-Cosine	0.11	4.81	4.92	54	K=4
LSH-Cosine	0.10	4.37	4.48	50	K=5
LSH-Cosine	0.12	3.70	3.82	48	K=6
LSH-Cosine	0.14	4.52	4.67	48	K=7
LSH-Cosine	0.25	3.96	4.21	43	K=8
LSH-Cosine	0.19	4.42	4.61	46	K=9
LSH-Cosine	0.17	3.72	3.89	42	K=10
LSH-Jaccard	2.70	0.85	3.55	26	Permutations = 16
LSH-Jaccard	3.19	1.13	4.32	26	Permutations = 32
LSH-Jaccard	6.90	1.91	8.81	26	Permutations = 64

Table 4: Question 3.1 Results

Analysis and Remarks

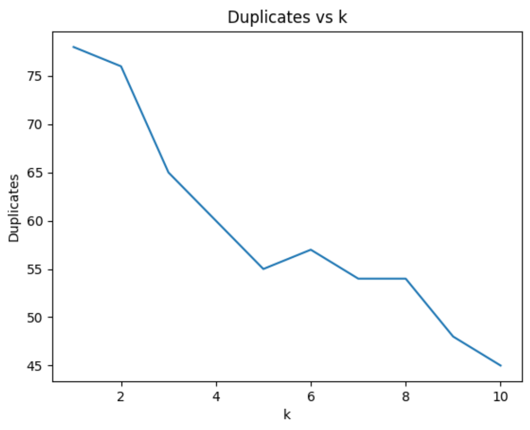
The purpose of this homework, is to see the time and performance of each method requested. However due to the packages used and numpy optimizations some calculations went faster that they should.

Calculating cosine similarity between all the vectors (3000x1000) is faster than LSH technique. But this is happening because of the numpy and sklearn vector operations that are optimally implemented. So, in our report we will use another metric, the number of comparisons (cardinality) of each technique to produce better insights.

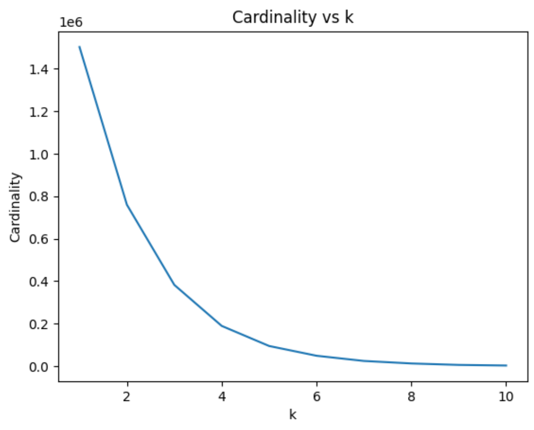
For the exact cosine implementation we needed only 0.024 seconds. But the numper of comparisons in this brute-force method is 3000x1000=3000000 comparisons. This method due to the fact that we use a small dataset is really fast. But if the dataset was bigger, it wouldn't scale. And that's because we compare everything to everything.

For solving and optimizing the performance we implement a hashing scheme, as described. The hashing scheme is a trade-off between performance and time. The Cosine LSH method, creates smaller buckets of training data, and thus reduces significantly the number of comparisons. A major impact on this, has the k parameter.

As we increase the k, more buckets are being built, resulting to better query time. That's because the number of train data questions which fall in the buckets, decreases and we get more buckets with smaller sizes. However, more buckets mean that questions that are duplicate have less probability to belong in the same bucket. This results to less duplicates found. As we can see it starts with 76 and 79 duplicates, very close to the result we got with the brute-force approach. But, as the k increases, less duplicates found (figures 9). The BuildTime, is slightly increasing, as more buckets are being formed.



(a) Duplicates found per k

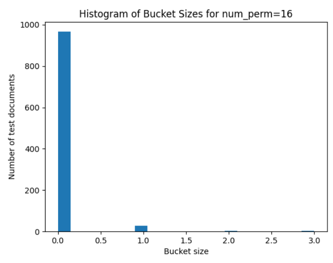


(b) Number of comparisons per k

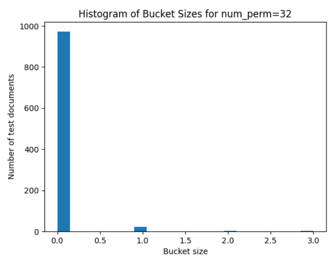
Figure 9: Cosine LSH

An analogous performance, we obtain with Jaccard and MinHash. However, due to the nature of the dataset and MinHash performance, in each trial it managed to get all the duplicates. But this is not the case with hashing schemes and we will again comment the number of comparisons. For the brute-force approach, since we hard-code implemented it, it needed almost 18 seconds. The total time needed for MinHash to find all duplicates is 3,26 with permutations set to 16. That is almost four times faster. In this case hashing shows it's power, as it has found all duplicates really fast. But this is not the case, when we for exampe have really big datasets.

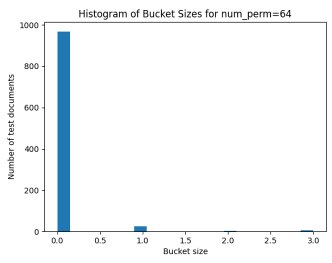
To better show-case the performance, we have produced some diagrams based on the number of comparisons. Starting with the Cosine LSH technique. For k=1 and k=2, we have approximately 1.4 to 1.2 million comparisons. Let's remind here that the brute-force method, needs 3 million comparisons. That's 50% less comparisons. But as we increase k, we get even less number of comparisons. That is visualized in Figure 10c. As for example for k=10 we have less than 200.000 comparisons, which are 15% of the brute-force method. But, as we mentioned, its a trade-off between performance and time. In Figure 10b we see the reduction of duplicates found as we increase the k. And for example, for k=10 we get 42 out of 89 duplicates that is a great loss. For this experiment, setting k=2 or k=3 may seem one of the best scenarios as we have the best trade-off.



(a) #Permutations=16



(b) #Permutations=32



(c) #Permutations=64

Figure 10: Jaccard MinHash with threshold=0.8

For the Jaccard MinHash method, we have a slightly different scenario. That's because, we have setted a threshold of 0.8 so every question less than this threshold is rejected. This leads to a very good performance. This can be shown by the figure 11. For number of permutations equal to 16, MinHash LSH with Jaccard needed only 44 comparisons, for 32 needed 38 and for permutations equal to 64 needed 46 comparisons. From the brute-force approach that's significantly lower, and that's due to the performance of datasketch MinHash algorithm. Another important point for the MinHash technique is the sizes of the buckets created. For most documents that did not have any duplicates, it returns empty buckets (that's due to the threshold criterion). And only for those that have duplicates we get non-empty sets of small sizes like containing max three candidates. For a more realistic analysis, we removed the threshold criterion and did some experiments. Figures 12a, 12b, 12c, 13 show the case where we don't apply a threshold. In this experiment we have bigger buckets and the number of comparisons is increasing as we are performing more permutations. That's the expected behavior, as increasing the permuattaions, increases the number of buckets and candidates for every test data question.

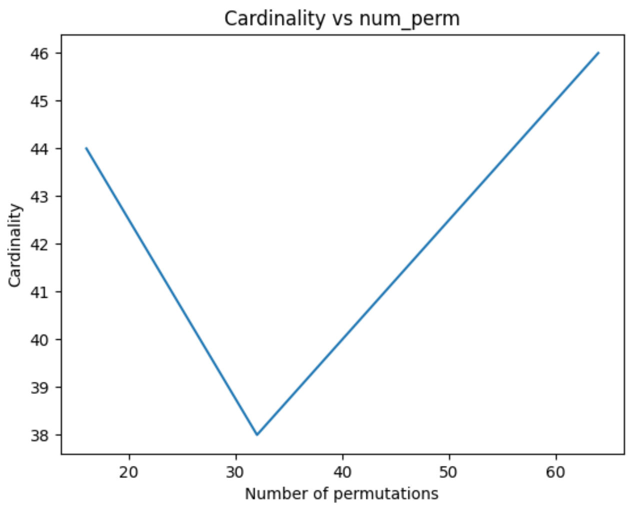
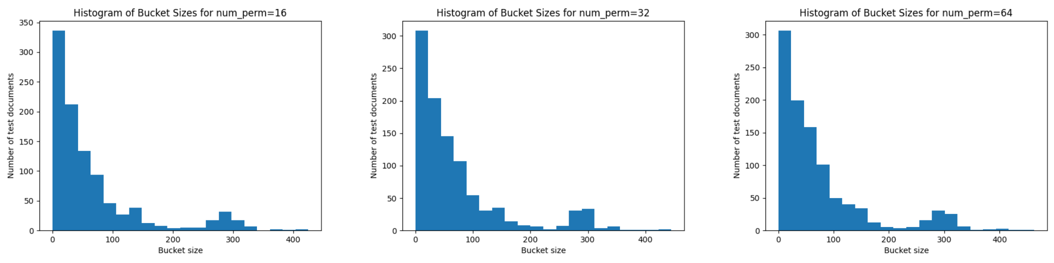


Figure 11: Number of comparisons per #Permutations for threshold=0.8



(a) #Permutations=16 (b) #Permutations=32 (c) #Permutations=64

Figure 12: Jaccard MinHash with threshold=0.0

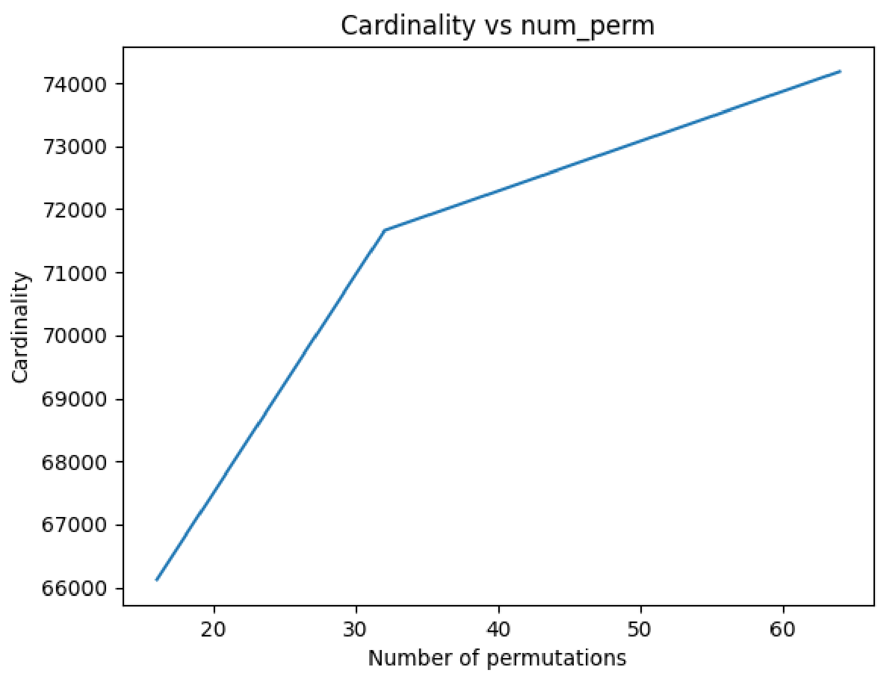


Figure 13: Number of comparisons per #Permutations for threshold=0.0

Deiverables and Project Split

In the deliverable you will find:

1. Part1.py: Contains all code for Part 1.
2. Part2.py and Part2_tokens.py: Contains all code for Part 2.
3. Part2.py: Contains all code for Part 3.

Project split:

- **Konstantinos Plas**: Part 1 and Part 3 - Cosine Similarity: Random projection LSH family
- **Konstantinos Nikoletos**: Part 2 and Part 3 - MinHash LSH