ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ + ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΕΛΛΗΝΙΚΗ ΔΗΜΟΚΡΑΤΙΑ
Εθνικόν και Καποδιστριακόν
Πανεπιστήμιον Αθηνών
ΙΔΡΥΘΕΝ ΤΟ 1837

Big Data Mining Techniques assignment report

Fall semester 2023-2024

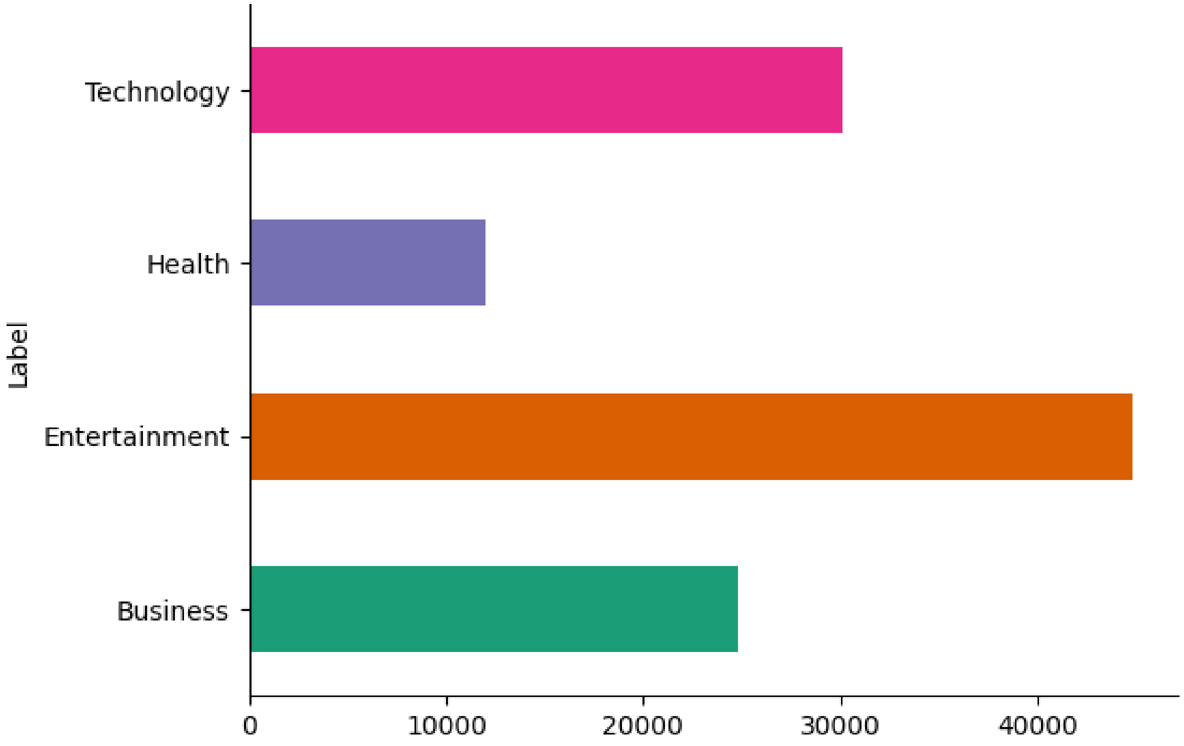
Dimitrios Tsiompikas
AM: 7115112300036

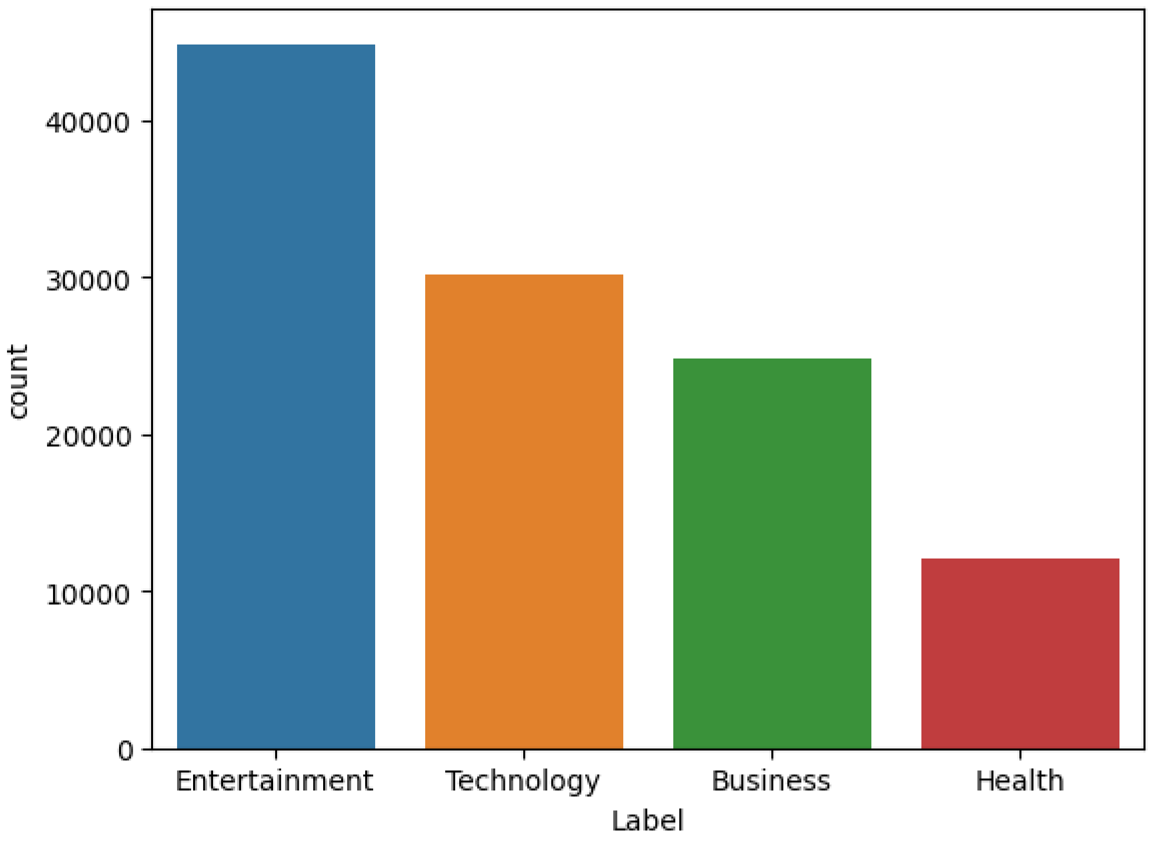
Part 1: Text classification

Initially, I loaded the data by using the Pandas library and I began cleaning the dataset from any NaN values. The dataset did not contain any , and in general it was a very easy dataset in terms of interpretation from the known python libraries that are used for NLP purposes. Afterwards, I started the data pre-processing part by doing the following: I used several libraries including NLTK and seaborn for plotting. I also used the following techniques: I lowered all text in the articles, removed all special characters, removed all stopwords (using NLTK's stopword set and some of my own that I noticed were very much used in the dataset without providing any meaningful context) and finally I lemmatized the Content column in order to maximize results during model training. For all techniques mentioned above I used the library Pandarallel to use Kaggle's extra cores and increase computation speed.

Question 1.1

Below you will find some diagrams regarding the dataset and the wordclouds:

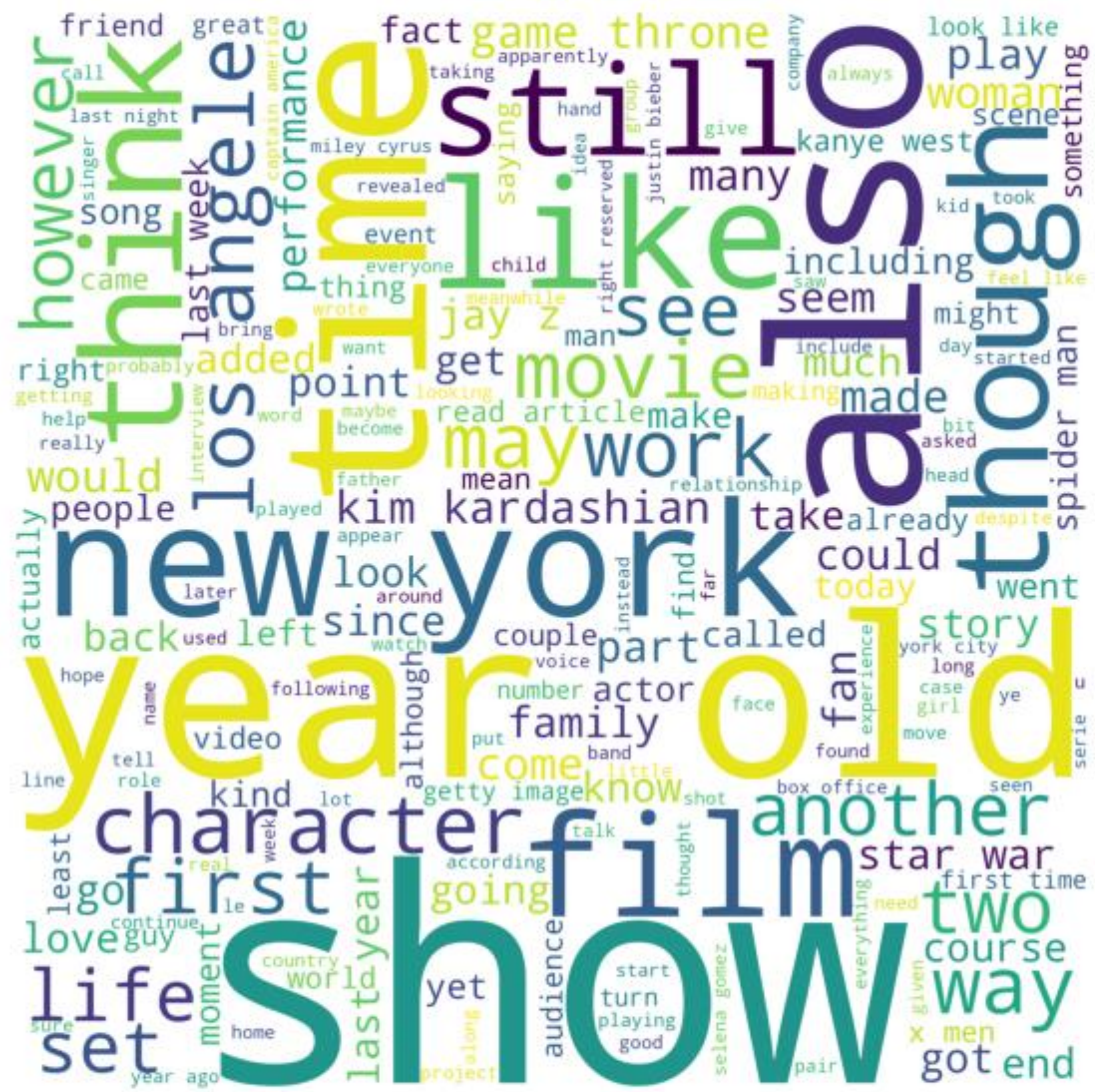




Here , in the distribution diagram and countplot we can see the distribution of data and how many articles each label has. Entertainment has the most articles as a label and health has the least, which could seem like a problem due to the uneven distribution of articles but later on the results will show that this is false.

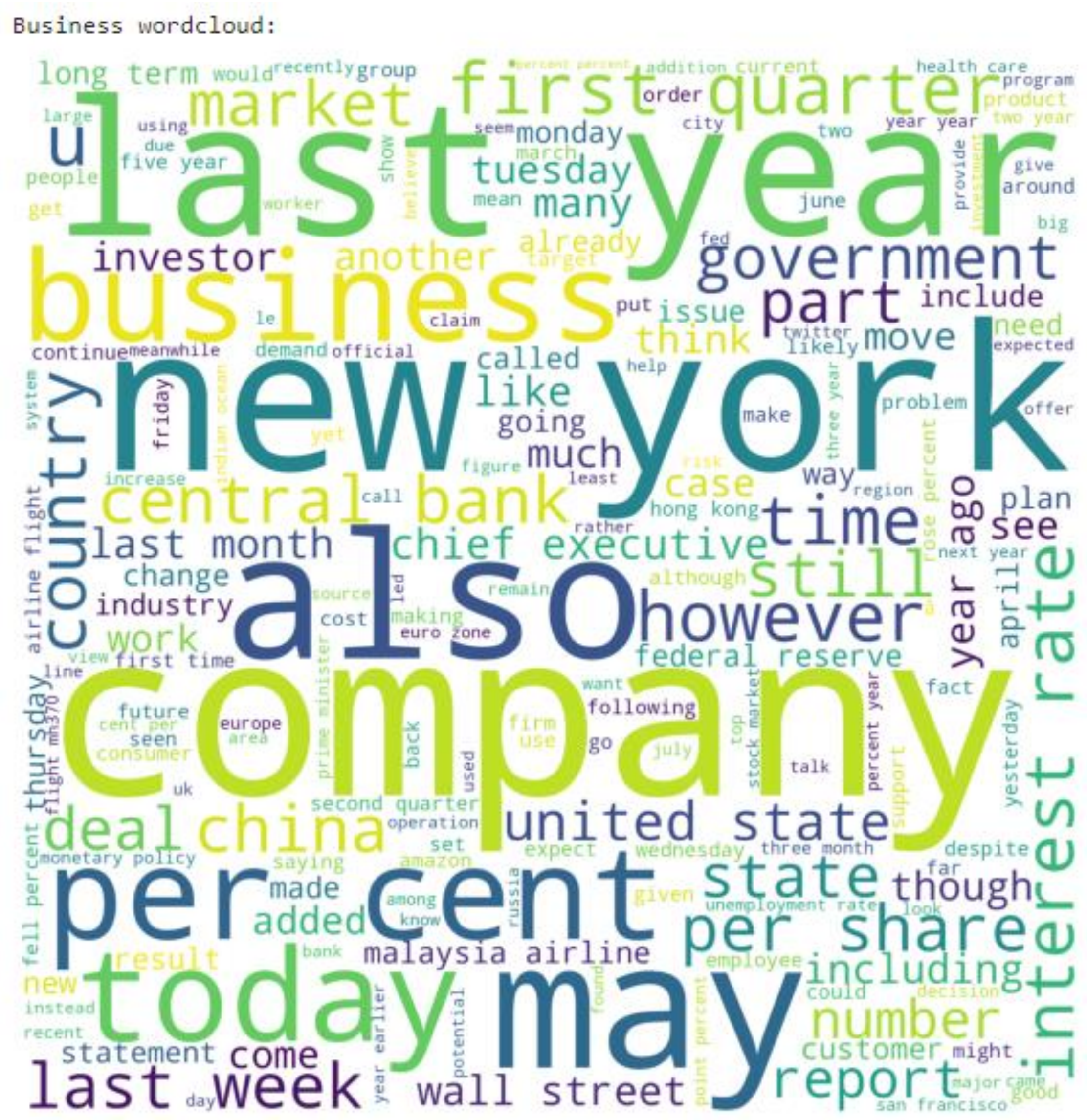
Entertainment word cloud:

```
Entertainment wordcloud:
```



Here we can see many words regarding tv shows, films , characters of films etc and some music related words which contain names like Miley Cyrus and Jay-Z. Also Los Angeles is mentioned a lot since it’s the city Hollywood is based in.

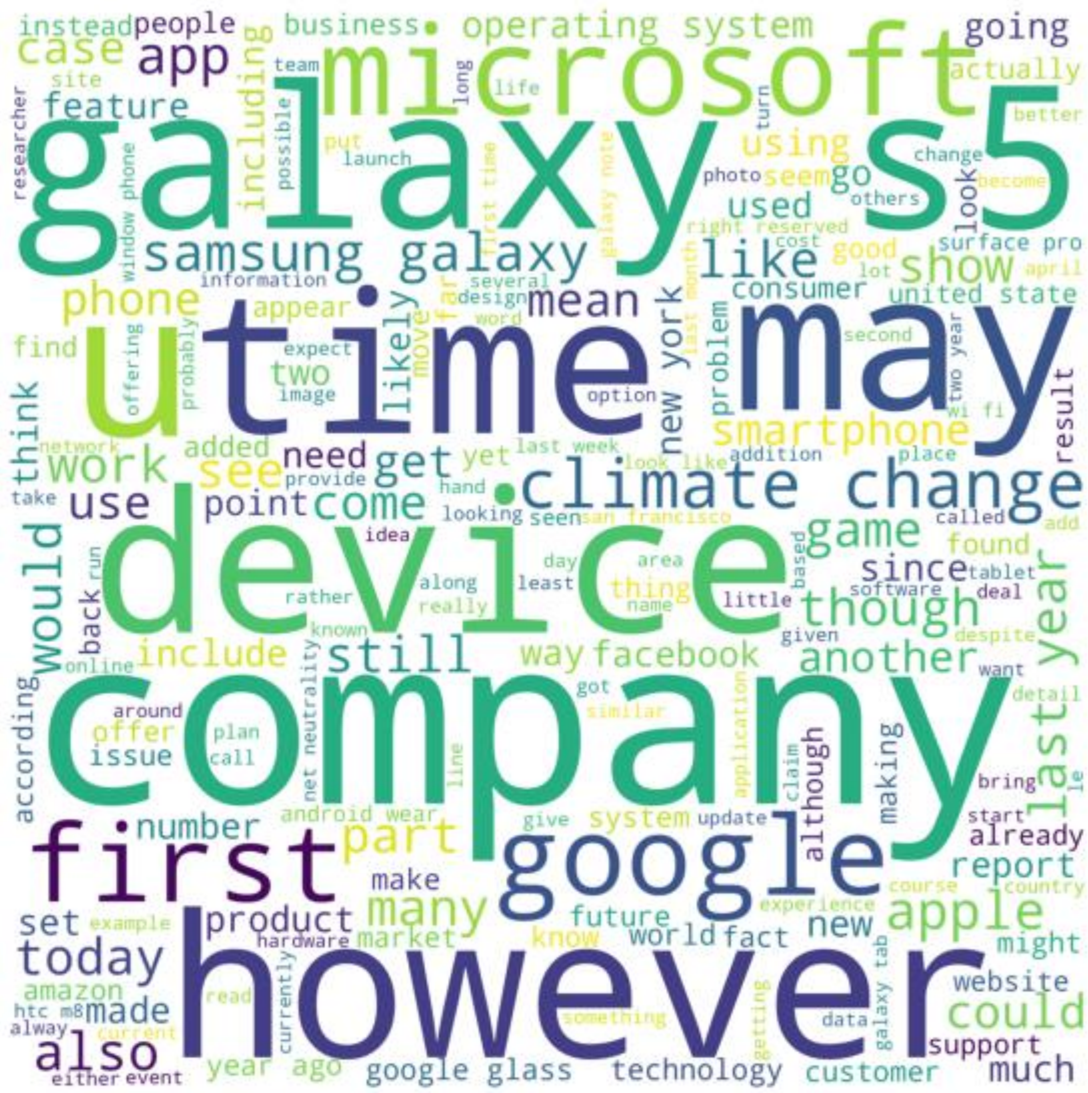
Business word cloud:


Business wordcloud:

Here we can see business-related words, like company, percent , central bank, market share, per share etc. which help with business articles being labeled correctly. We also see time related words since a large number of business articles talks about shares and their prices back in time and try to predict share prices.

Technology word cloud:

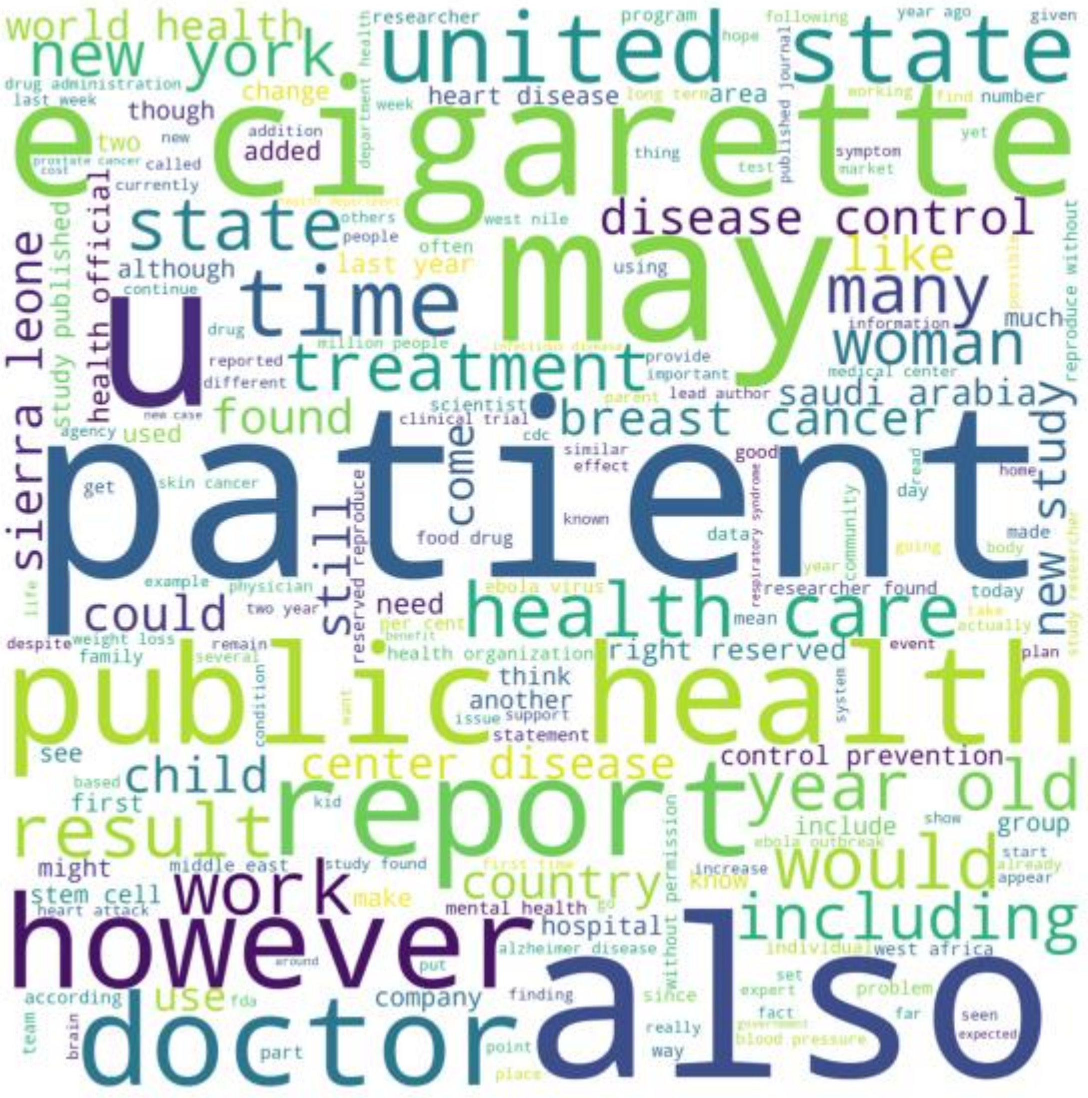
Technology wordcloud:



Here we see names of Technological “Giants”, companies that have made major advancements in the field of Computer Science and Computer Engineering like Google, Apple, Microsoft , Facebook and Samsung. Also some other words like operating system , device, hardware etc that relate to the Technology label.

Health word cloud:

Health wordcloud:



Here we see many words regarding health like patient, disease control, doctor, public health, treatment, hospital, mental health etc, which give a pretty good understanding of which article relates to health. Also some words about health research , mostly directed to cancer and stem cells, are mentioned since a lot of articles cover stem cell research.

Question 1.2

In this part I made several tests with scikit-learn algorithms such as XGBoost, SVM algorithms, Random Forest and Logistic Regression to predict the labels of the test set. Firstly, I used the count vectorizer on the Content column of the train set by scikit-learn to create a Bag of Words in order to run SVM and Random Forest with BOW and also used the TruncatedSVD algorithm to create the SVD technique for SVM and Random Forest with SVD. BoW had better performance than SVD.

For SVM, I used the LinearSVC method because it was the fastest implementation of the model taking only 30 seconds to run for 1000 iterations (both for SVD and BOW). For Random Forest , I used 20 estimators (trees) because it was really slow , taking an average of 10 minutes per run , quite costly in our case where we want to run multiple experiments to optimize the algorithm (for BOW) and 100 estimators for SVD since it was faster there taking only 5 minutes to complete per run (more estimators did not improve performance of the model however, hence why I stayed with 100).

In order to beat the benchmark, I used the hashing vectorizer and combined the title and content columns to create the combined column which was used to bring the best predictions possible. The model that was used was the LinearSVC once again since it provided the best results out of all tests. I also used XGBoost and Logistic Regression for this but LinearSVC was better than both.

Below you will find the table with the results of my experiments:

Statistic Measure	**SVM (BOW)**	**Random Forest (BOW)**	**SVM (SVD)**	**Random Forest (SVD)**	**XGBoost (hash vec)**	**Logistic Regression (hash vec)**	**My Method (LinearSVC with hash vec)**
Accuracy	96%	92%	90%	91%	81%	96%	97%
Precision	96%	92%	90%	91%	81%	96%	97%
Recall	96%	92%	90%	91%	81%	96%	97%

Some final comments are that XGBoost was pretty slow (just like RF , they're ensemble algorithms after all) hence the need for low number of estimators (20) and low depth (3). Logistic Regression was really good for this task (hit a pretty good 96%, however LinearSVC surpassed it). For evaluation I used the cross_val_predict method from sklearn which returns predictions for all folds (I used 5-fold cross validation as asked) and got the metrics from the classification report using the labels from the train set as y_train and the predictions from cross_val_predict as y_pred.

Part 2: Nearest Neighbor Search with Locality Sensitive Hashing

In this part I ran the brute force NN algorithm with jaccard metric, I used the datasketch library to implement minhash with LSH and ran experiments for thresholds of >= 0.8 and >= 0.5 and permutations of number 16,32 and 64 for both experiments. In order to calculate the jaccard metric for the brute force NN algorithm I had to perform dimensionality reduction using SVD and binarizing the data in order to produce proper and fast results.

For Min-hash LSH, I tokenized the train and test set combined columns, I created minhash objects for both the train set and test set, I created an LSH index for the train set and queried each article from the test set.

The way I find the true K most similar documents fraction is the following: I flatten the indices array from the brute-force NN and I also flatten the result array from the lsh.query method in order to get all indices found by minhash LSH.

Lastly I take the intersection of their lengths and divide it by K (in our case 15). The results table is on the next page.

Below you will find the table of the results:

Type	BuildTime	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	0	734	734	100%	-
LSH-Jaccard	206	100	306	44.7%	Perm = 16, threshold = 0.8
LSH-Jaccard	225	97	322	43.2%	Perm = 32, threshold = 0.8
LSH-Jaccard	248	105	353	41.6%	Perm = 64, threshold = 0.8
LSH-Jaccard	216	90	306	53.2%	Perm = 16, threshold = 0.5

LSH-Jaccard	228	95	323	51.5%	Perm = 32, threshold = 0.5
LSH-Jaccard	252	106	358	49.7%	Perm = 64, threshold = 0.5

From the results we can see that the more permutations I add the less percentage of similar documents I get comparing with brute-force. This happens because I get more accurate results with the cost of getting less actual documents. Of course ,time-wise, the more permutations I add the more time the build and query times require to finish. Reducing the threshold from 0.8 to 0.5 also helps find more articles but we lose precision on this , as many articles deemed as duplicates might not be duplicates after all.

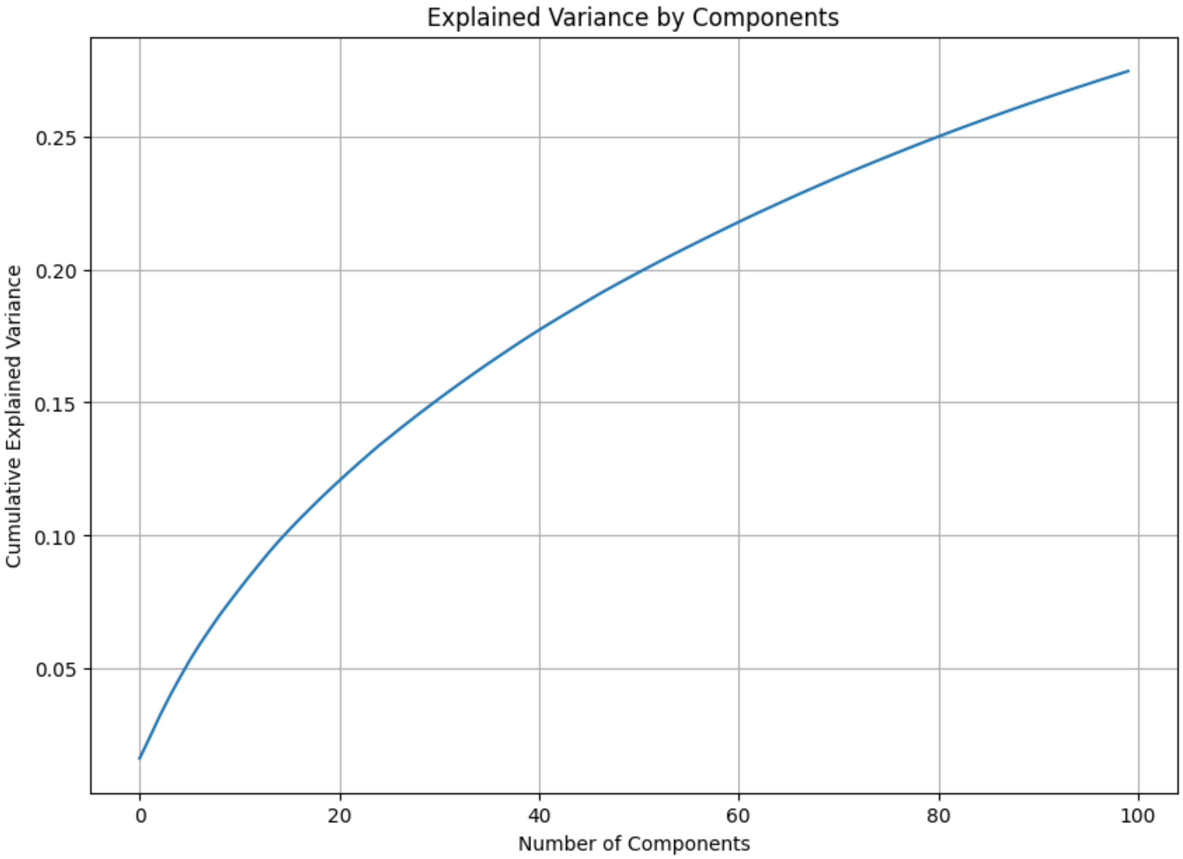
Part 3: Nearest Neighbor Search and Duplicate Detection

Question 3.1

In this part I began with pre-processing the data using Pandas. I checked for NaN values , some existed in both datasets so I filled them with an empty string. Afterwards, I combined the question columns into one which I named Both_Questions. I performed the following pre-processing techniques as in Question 1 but for the columns Both_Questions, Question1 and Question2: I lowered all text in the questions, removed all special characters, removed all stopwords (using NLTK's stopword set) and finally I lemmatized the text in these columns in order to maximize results during model training.

Furthermore, I again used the Hashing vectorizer provided by sklearn to make Both_Questions column text into vectors. I then ran brute force NN with cosine similarity as the metric , I flatten the distances , filter all similarities that are >= 0.8 and print this list’s length. This is the number of duplicates for cosine NN.

Similarly, for Jaccard-NN I use the same technique as question 2, I use SVD on the data, and then binarize them. However, I faced some issues with this dataset and had to use down-sampling in order to get Jaccard NN to finish. I used 20% of the train set and 20% of the test set to perform my experiments. I also created an explained variance ratio diagram for this (which you can see below) to see the optimal number of components that I can use with SVD. I found out (through the diagram and experimentation) that 100 components are enough and adding more components will not change the result.



I then use the same method as Cosine-NN to find the duplicates.

For the random projections part, I created a transformer with GaussianRandomProjection from sklearn, I transformed the vectorized train and test set Both_Questions columns and I created a hash for every question. Afterwards, for each hash in the train set I created a bucket, a nearest neighbors model that was fitted with the bucket and I saved that to a dictionary as a value with the hash as the key. Build time was calculated here.

Once that was done, for every hash in the test set I created a bucket as well and matched that with the respective hash from the train set in order to query the buckets for that hash using the trained NN model from the dictionary. For each hash I appended the distances to a list and then followed the same procedure as with the brute force algorithms. I flattened the list , I kept all distances that were >= 0.8 and the length of that filtered list was the duplicates. I performed the experiment for K = 1 to K = 10.

Lastly, for Minhash I tokenized the Both_Questions columns for train and test set, I created minhash objects for both sets, I added all train set minhash objects to the LSH index and finally I queried each test set minhash object and appended its results to a list. I flattened that list and its length was the number of duplicates.

I performed the above experiment for permutations of number 16,32 and 64.

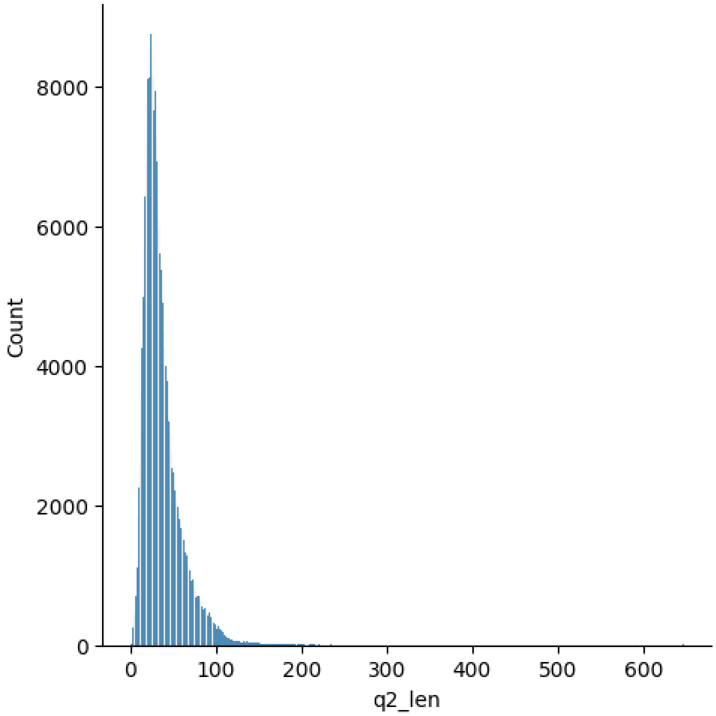
Below you will find the results table:

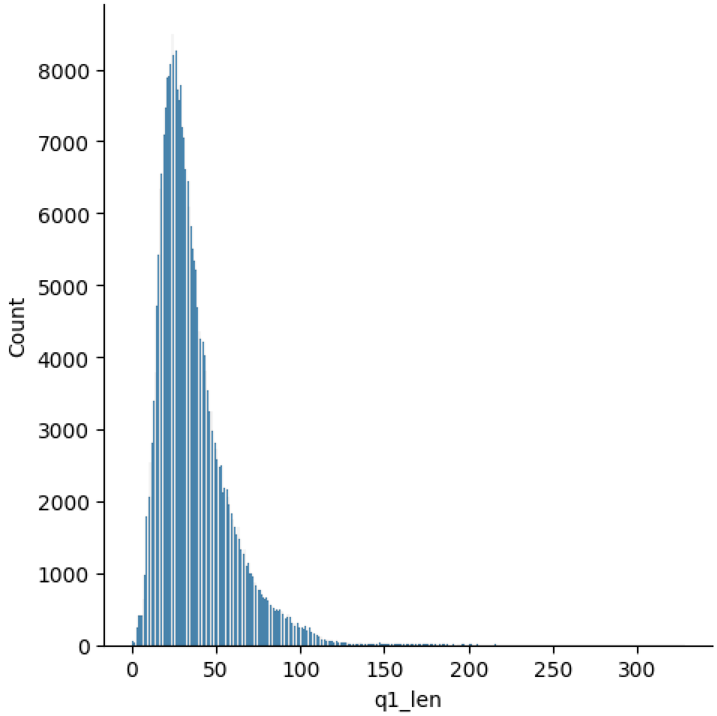
Type	BuildTime	QueryTime	# of Duplicates	Parameters
Exact-Cosine	0	959	1417	-
Exact-Jaccard	0	230	1500	-
LSH-Cosine	3.2	205	614	K = 1
LSH-Cosine	3.5	260	497	K = 2
LSH-Cosine	3.2	287	905	K = 3
LSH-Cosine	4	305	459	K = 4
LSH-Cosine	4.3	340	478	K = 5
LSH-Cosine	4.45	375	625	K = 6
LSH-Cosine	4.5	500	782	K = 7
LSH-Cosine	4.8	545	844	K = 8
LSH-Cosine	5	654	769	K = 9
LSH-Cosine	5.2	798	940	K = 10
LSH-Jaccard	214	90	1104	Perms = 16
LSH-Jaccard	279	118	893	Perms = 32
LSH-Jaccard	417	176	938	Perms = 64

Question 3.2 (Extra)

For this part , I tried the LinearSVC and Logistic Regression algorithms using the Hashing vectorizer on the Both_Questions column. I also tried Random Forest with this , but it was impossible to run due to computation cost (even with very low estimator number such as 5). For evaluation I used the cross_val_predict method from sklearn which returns predictions for all folds (I used 5-fold cross validation as asked) and got the metrics from the classification report using the labels (0 or 1 here, IsDuplicate column) from the train set as y_train and the predictions from cross_val_predict as y_pred. LinearSVC once again outperformed all other models.

I also tried using some feature engineering and created some new features which are used in similar NLP problems, more specifically: I got the length of each question (q1_len, q2_len columns), the number of words in each question (q1_num_words, q2_num_words columns), the common words in each row (word_common column), total words in each row (word_total), shared words of each row (word_share column). Below you will see two distribution plots for each question length in the train set:





From these, we can see that all questions are evenly distributed at around 35-40 characters (average is 36) and there are some outliers that are above 100 characters long questions.

I used these features with LinearSVC, Logistic Regression and Random Forest Classifier, all of which ran successfully, however their performance was subpar from that of the first feature (Both_Questions column). Thus, I used the first feature for my prediction of the test set duplicates and created the excel with them.

Below you will find the results table for my experiments:

Method	Precision	Recall	F-Measure	Accuracy
LinearSVC (both_questions)	75%	75%	75%	75%
Logistic Regression (both questions)	73%	73%	73%	73%
LinearSVC (Feature engineering)	57%	57%	57%	57%
Logistic Regression (Feature engineering)	66%	66%	66%	66%
Random Forest (Feature engineering)	69%	69%	69%	69%