Big Data Analytics
Data Science and Information Technologies
Fall 2019-20

Final Assignment

HELLENIC REPUBLIC
National and Kapodistrian
University of Athens

Thanasis Polydoros (ID: DS1.19.0016)
thanasispolydoros@gmail.com

Dimitrios Roussis (ID: DS1.19.0017)
droussis@outlook.com.gr

Supervisor: Dr. Dimitrios Gunopulos

Table of Contents:

Requirement 1: Text classification

In this part of the project, we are given a large dataset which is comprised of 111,795 news articles and we train different classifiers in order to classify those articles in 4 categories: **Business**, **Entertainment**, **Health** and **Technology**.

The classification accuracy will be evaluated on unlabeled test data (47,912) and submitted to the Kaggle competition "BigData2020 Classification" which can be found in the following link: https://www.kaggle.com/c/bigdata2020classification

The 1st requirement is divided into two questions. In the first one, we will generate a WordCloud for each article category and in the second one, we will evaluate the performance of 2 classifiers using 2 different ways of extracting the features.

Before anything else, however, we load the dataset and drop the 'Id' column, as it is unnecessary at the current stage.

Question 1a: WordCloud

In the first question of requirement 1, we use the 'wordcloud' library of Python to generate word clouds, i.e. figures which depict the most common words in a given category of news articles. More specifically, for each category of news articles, we depict 2 word clouds, one which is **based on the titles** of the articles and one which is **based on their contents**. Before displaying the word clouds, we make sure that we have removed the stopwords.

Business (24,834 articles)

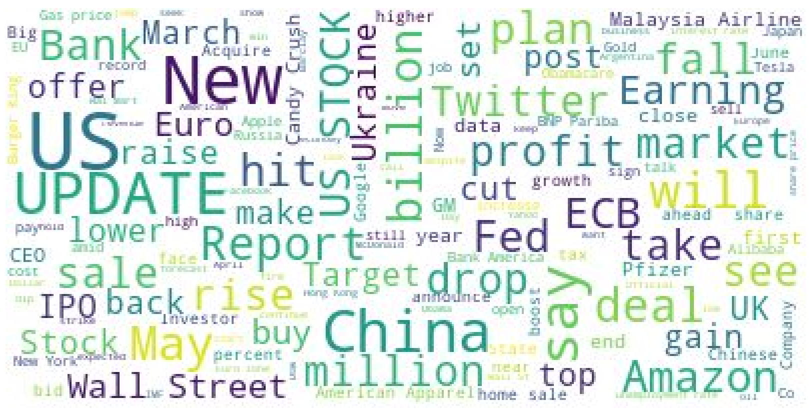
Fig 1: Word Cloud of Business category based on title

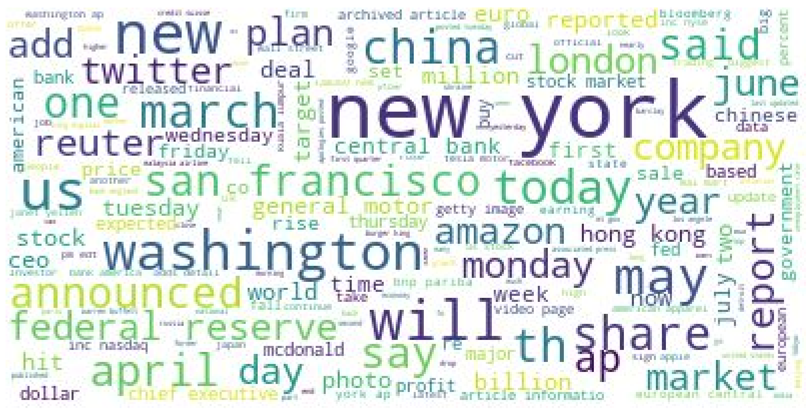
Fig 2: Word Cloud of Business category based on content

We can see that some of the most common words in *both* the titles and the contents of the news articles categorized as 'Business', have to do with specific countries, such as "US" and "China", or specific corporate entities and business terms such as "Amazon", "Twitter", "market", etc. In the word cloud based on the *titles*, we can see abstract terms such as "profit", "million", "billion", "stock", "Wall Street", "sale", "bank" etc. as more prevalent, whereas in the word cloud based on the *contents*, more specific terms appear which have to do with cities (e.g. "Washington", "San Francisco") or time (e.g. "March", "April", "June"). However, words that have to do with months as well as words such as "will", "say", "hit", "one", which are -of course- quite expected for business news articles, are probably also commonly used in other article categories as well -in different contexts- and would not be very helpful for our classifiers.

<u>Entertainment (44,834 articles)</u>

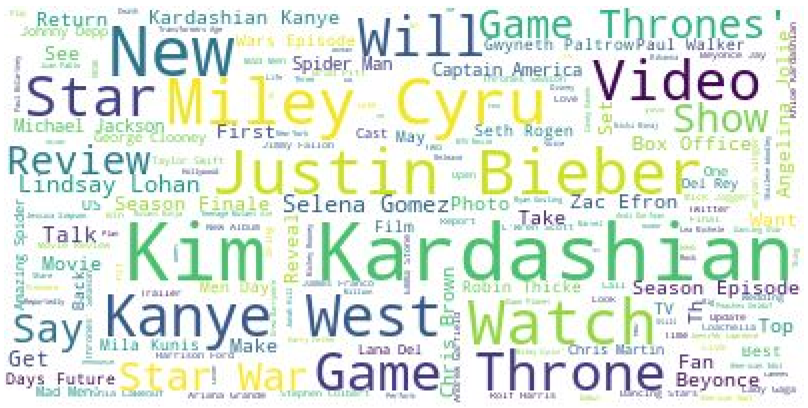

Fig. 3: Word Cloud of Entertainment category based on title

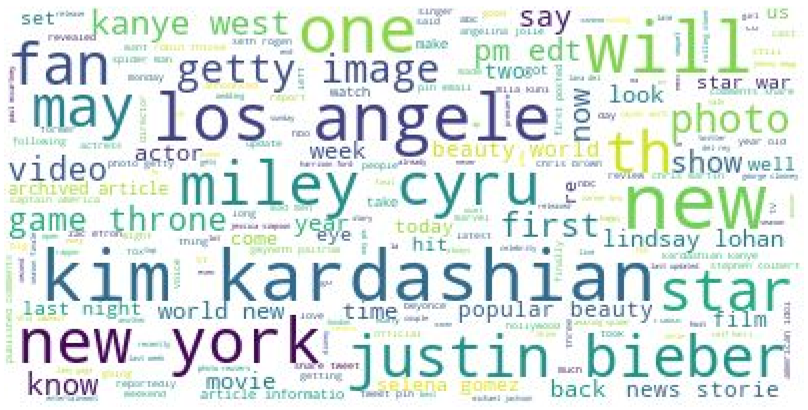

Fig. 4: Word Cloud of Entertainment category based on content

Regarding the news article with the 'Entertainment' label, we can easily observe that specific celebrities appear in both, such as "Kim Kardashian", "Miley Cyrus", "Justin Bieber", etc. Furthermore, we can see that -at least in the word cloud based on contents- the city of "New York" appears quite a lot and thus, would not be helpful in distinguishing between entertainment and business news articles; on the other hand "Los Angeles" seems unique in entertainment news articles.

Health (12,020 articles)

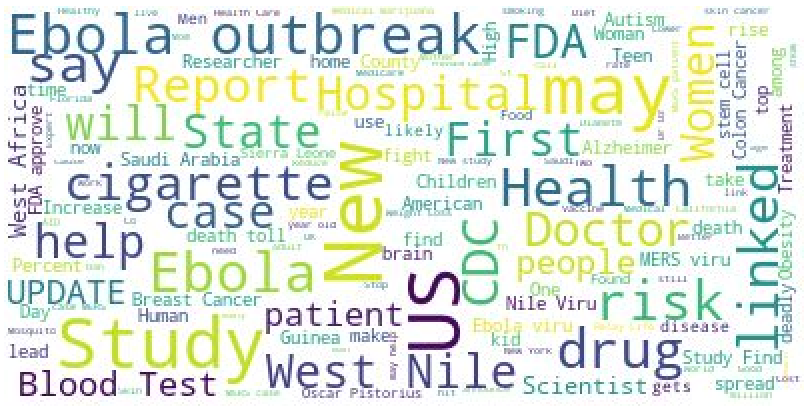

Fig. 5: Word Cloud of Health category based on title

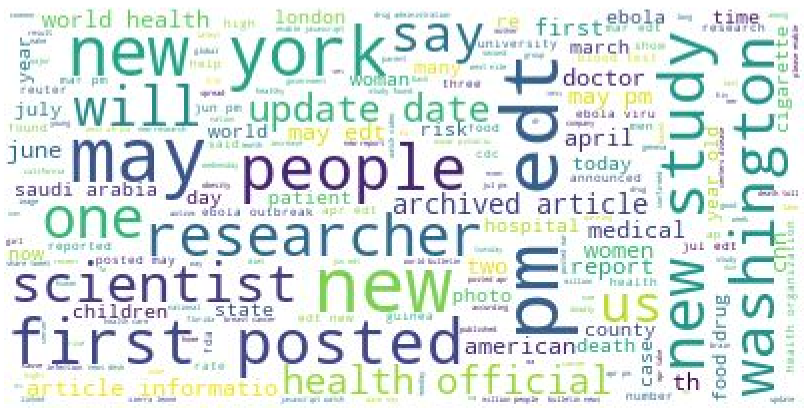

Fig. 6: Word Cloud of Health category based on content

As expected, the word clouds of the 'Health' news articles reveal frequent occurrences of words such as "study", "doctor", "hospital", "health", "medical", "Ebola", "scientist", etc. Yet again, we can see that the cities of "New York" and "Washington" are very frequent and therefore, would not constitute good features for our classifiers. At this point, it has become quite obvious that the news articles have been collected from websites and publishers in the USA, which are probably based on New York and/or Washington.

<u>Technology (30,107 articles)</u>

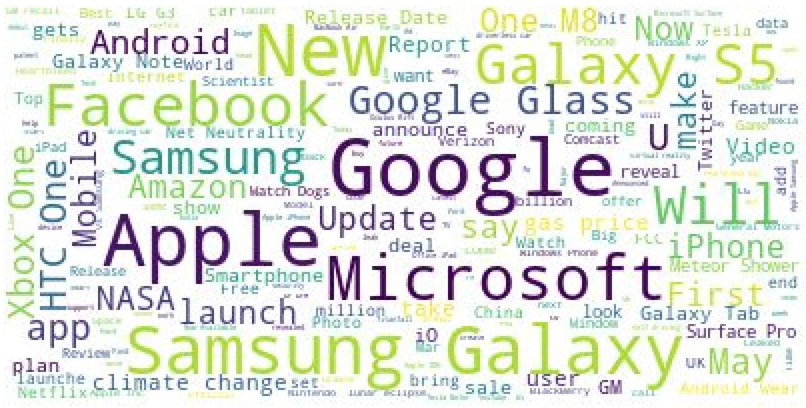

Fig. 7: Word Cloud of Technology category based on title

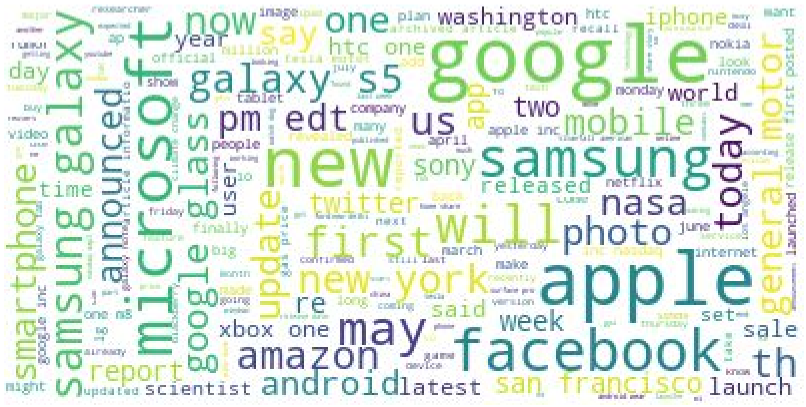

Fig. 8: Word Cloud of Technology category based on content

News articles in the ‘Technology’ category are clearly dominated by brands, products and large technological corporations, such as “Apple”, “Facebook”, “Microsoft”, “Google”, “Samsung Galaxy”, “Android”, etc. Note, however, that the word “Amazon” also appeared frequently in the business category and that means that it cannot serve as a good feature either.

Question 1b: Classification Task

In the second question of requirement 1, we implement different classifiers with different feature extraction methods from the raw texts, evaluate their performance using 5-fold cross-validation and report their accuracy, precision, recall and F-measure. In particular, we implement the following configurations:

- Linear Support Vector Machines classifier with Bag of Words feature extraction
- Random Forest classifier with Bag of Words feature extraction
- Linear Support Vector Machines classifier with Singular Value Decomposition feature generation
- Random Forest classifier with Singular Value Decomposition feature generation
- Custom proposed model to beat the benchmark

Preprocessing and implementation details

First of all, we **remove the duplicate articles** in the train dataset, i.e. those which have the exact same content. 1811 duplicates were removed in this way. We also depict the **class imbalance problem** of the dataset, which could be observed in question 1a.

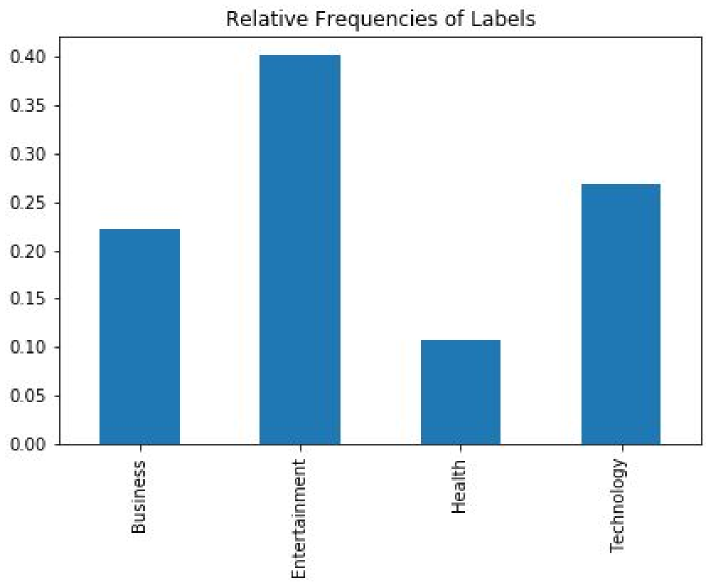

Fig. 9: Relative frequencies of the different categories

In order to extract numerical features from the raw texts, we convert them to a Bag of Words representation using the **HashingVectorizer** function of scikit-learn library. This particular vectorizer is used in order to deal with the large number of words in our dataset, as it implements the "hashing trick", i.e. it determines the column index of features in sample matrices directly via a hash function. It also uses white spaces and punctuation to separate words into tokens (tokenization) and counts the occurrences of said tokens in each document, while simultaneously removing the stop words.

However, the hashing vectorizer ends up giving us 2^{18} features and it would probably be a good idea to reduce the dimensionality of the feature space. Thus, for each classifier, we also apply **truncated singular value decomposition** (a matrix factorization technique which is also known as latent semantic analysis when used on term count matrices, such as in our case), by using the TruncatedSVD function of scikit-learn library and returning 150 components.

The two classifiers that we experiment with are the **Linear Support Vector Machines** classifier and the **Random Forest** classifier. In particular, we use the standard implementation parameters of the LinearSVC function (C=1, Squared Hinge loss) and we slightly modify the parameters of the RandomForestClassifier (50 trees in the forest, without bootstrapping).

Proposed method

In order to beat the benchmark scores, we followed a quite different approach, concerning both the way of extracting features from the texts and the way of classifying new documents. An important preprocessing step that we did, is that *we used the titles of each article as well*, by concatenating the content to the title of each news article.

In particular, we extract features using the **TF-IDF vectorizer** (TfidfVectorizer function of scikit-learn library), which although does not use the "hashing trick", it also re-weights the features by returning their term-frequency multiplied by the inverse document-frequency, thus taking into account how often they appear in all the news articles. Again, we remove the stop words, but also set *upper* (max_df = 0.5) and *lower* (min_df = 3) *thresholds of document frequency* for a given term to be actually taken into account in the vocabulary. We do this because a given term will not improve the accuracy of our classifier if it only occurs in very few samples (note that our samples are now combinations of the title and the content of each article) nor if it occurs in more than 50% of the articles.

Furthermore, instead of treating only individual words as tokens, we modified it so as to extract **bigrams** (2-grams) of words as well, i.e. sequences of two words (ngram_range = (1, 2)) and apply all the other aforementioned methods; this is because we saw the prevalence of terms such as "Kim Kardashian", "Samsung Galaxy", etc.

Our choice of classifier was an ensemble and in particular, a **voting classifier** (VotingClassifier), which combines 3 classifiers and gives as an output the predicted label which has the majority among the 3 individual classifiers. After testing many classifiers in the scikit-learn library and fine-tuning their parameters, we decided to combine the following classifiers:

- Linear Support Vector Machines classifier
- Ridge classifier ($\alpha = 0.8$)
- K-Nearest Neighbors classifier (5 neighbors)

Furthermore, we decided to tune the hyper-parameters of the individual classifiers (not only those that we actually used) and of the TF-IDF vectorizer using 5-fold cross-validation on 80% of the training data, leaving the rest **20% of the data as a** -not previously seen- **test set**. This way, we can get a better estimate of the actual performance of our voting classifier on unknown data. Another reason is that the hyper-parameters of the classifiers are "fitted" on the training and cross-validation dataset and thus, we cannot be confident that the ensemble classifier will generalize well on new data necessarily. It is ***important to note***, however, that in the table below (see Table 1), we report the performance metrics of our proposed method based on 5-fold cross-validation on 80% of the data and not based on the aforementioned test set, on which it resulted in an **F-Measure of 97.80 %**.

Statistic Measure	**SVM (BoW)**	**Random Forest (BoW)**	**SVM (SVD)**	**Random Forest (SVD)**	**Our Proposed Method**
Accuracy	97.04 %	93.37 %	93.03 %	92.55 %	97.80 %
Precision	96.89 %	93.75 %	92.55 %	92.76 %	97.65 %
Recall	96.65 %	91.86 %	91.98 %	90.56 %	97.47 %
F-Measure	96.77 %	92.74 %	92.26 %	91.56 %	97.56 %

Table 1: Performance metrics on the text classification task

In general, we can observe that the SVD feature generation method does not improve the results and in fact gives a lot F-Measure on both the SVM and the Random Forest classifier. It is quite obvious as well that the Linear SVM classifier outperforms the Random Forest classifier with the Bag of Words feature representation, scoring an F-measure of 96.77 % compared to the 92.74 % of the Random Forest. However, it cannot outperform our proposed method, which does not improve so much on the accuracy, as it improves on the F-measure (i.e. both on the precision and recall) and manages to score an impressive F-measure of 97.56 %.

Requirement 2: Nearest Neighbor Search and Duplicate Detection

This part of the exercise requires experimenting with text similarity. In the first part, we are experimenting on large datasets and we compare the time and accuracy of probabilistic and exact methods. In the second part, we try to solve the problem of deciding whether or not two questions are duplicate.

Question 2a: De-Duplication with Locality Sensitive Hashing

In this part of the exercise, we are given two files containing Quora questions. We are asked to create an index for the first file (train file) and then check how many of the questions from the second file (test file) have a similarity greater than 80%.

Unfortunately, the datasets are too large to run on my workstation so only 10% of the data will be used from the training file.

Before parsing the text in the models we apply to the questions some basic pre-processing (the goal of this part is not to find the best pre-processing method, but instead to compare methods of calculating text similarity), so a simple TF-IDF vectorizer is used.

For the exact methods, the time has been calculated by approximation over the whole dataset. Build time is 0 sec because no index is required and there are no parameters to tune. The total time has been calculated by multiplying the query time with the number of data.

For the MinHash LSH-Jaccard method, we have used the datasketch library and for Cosine Lsh we have used NearPy library. Using libraries makes it hard to optimize code in order to utilize the most out of the underlying data structures (e.g. NumPy sparse arrays are not handled in the best way possible).

As it is mentioned above, the numbers in the table are calculated only by using 10% of the data. (train data 53199, test data 537).

Type	**Build Time** *(sec)*	**Query Time** *(sec)*	**Total Time** *(sec)*	**# Duplicates**	**Parameters**
Exact Cosine	0	7.51	7.51	28	-
Exact Jaccard	0	11.38	11.38	57	-
LSH-Jaccard	27.76	0.29	28.16	38	perm = 16
LSH-Jaccard	38.04	0.39	38.43	27	perm = 32
LSH-Jaccard	61.97	0.63	62.60	25	perm = 64
LSH-Cosine	119.77	85	213	11	K = 10
LSH-Cosine	107.59	140	262	11	K = 9
LSH-Cosine	102.16	255	381	11	K = 8
LSH-Cosine	91.82	505	632	13	K = 7
LSH-Cosine	89.21	930	1092	14	K = 6
LSH-Cosine	78.50	1865	2084	15	K =5
LSH-Cosine	70.34	3795	4149	18	K = 4

Table 2a: Performance of different similarity methods
(All the times are computed in seconds)

We can see for the exact distances the required time are very large (as expected to be) due to the fact that for each question we need to calculate our distance metric (Jaccard or Cosine).

More precisely let's examine the **LSH-Cosine** with the use of the **Random projection** LSH algorithm. The time and the number of results of this method are highly dependent on the K parameter. When the value of K is small building the index is faster but this has as a tradeoff the increase of query time due to the fact that most of the vectors are hashed in the same bucket. This also means that we have to compute more distances between vectors which is the reason why the number of similar increases while the K values drop (higher probability for similar items being in the same bucket).

The **Min Hash with Jaccard** method has a build time that depends on the number of permutations. Adding more permutations increases build time and query time but it improves the number of duplicates that are found see *table 2b*. Min hash is a fast method but it has a higher number of missed similar objects.

Overall

Checking for the similarity large lists of strings requires preprocessing and the use of an index is boosting the time performance significantly. However, if losing some entries is not acceptable then LSH techniques can not be used. LSH methods, if tuned correctly, can have an accuracy of 95%.

Question 2b: Same Question Detection

This part of the exercise requires us to predict if two Quora questions are duplicate. Two datasets have been given, a train set that contains 283,004 labeled data (37% are duplicate) and a test set that contains 121,287 unlabeled data.

There are two main challenges in order to create a great model for this problem. The one is the **preprocessing** of the data and the second one is to **pick** and **tune** the correct model.

Preprocessing - Feature engineering

For each pair of questions, the following features have been generated. In order to predict vector-based features (such as earth movers distance, cosine similarity), we have used a word2vec model pre-trained by google. Here is the complete list of features on which we will use for our model.

The features mentioned below have been used after searching papers and blogs of people dealing with similar problems.

1. Length of words
    a. len_q1
    b. len_q2
2. Diff_len (difference between the lengths)
3. Total number of characters without spaces
    a. Len_char_q1
    b. Len_char_q2
4. Total number of words
    a. Len_word_q1
    b. len_word_q2
5. Common_words
6. Fuzzy string matching metrics
    a. fuzz_qratio
    b. fuzz_WRatio
    c. fuzz_partial_ratio
    d. fuzz_partial_token_set_ratio
    e. fuzz_partial_token_sort_ratio
    f. fuzz_token_set_ratio
    g. Fuzz_token_sort_ratio
7. Earth mover's distance(emd)
8. cosine_distance
9. cityblock_distance
10. jaccard_distance
11. canberra_distance
12. euclidean_distance
13. minkowski_distance
14. Braycurtis_distance
15. Skewness
    a. Skew_q1vec
    b. skew_q2vec
16. Kurtosis

a. Kur_q1vec
b. kur_q2vec

Machine learning algorithm

The machine learning algorithm that we used is XGBoost. We run our XGBoost algorithm for different values of eta and depth and we found out that the best values for our data are 4 for the depth and 0.2 for eta (the rest of parameters are not worth tuning).

In order to train the XGBoost model, we have split the given set in train, test and validation sets. More precisely 80% of the data have been used for training and the rest 20% has been equally split in validation and test sets.

Bigger values of depth overfit on our train data and do not improve the accuracy on the test set at all. Also, using bigger values for eta results in high log-loss values on the training set (log-loss was our performance metric while training).

Removing features that have low f-score does not improve the accuracy of our model. XGBoost can itself ignore some of the features if needed.

This is a table describing our performance for various models that we experimented on.

Method	**Precision**	**Recall**	**F-Measure**	**Accuracy**
XGBoost (4 layers)	75.5 %	76.1 %	75.7 %	76.9%
XGBoost (5 layers)	74.6 %	75.3 %	74.9 %	76%
XGBoost (6 layers)	74.4 %	75 %	74.7 %	76.1%

Table 2b: Performance metrics on the duplicate detection task

This seems to be the importance of each feature after training the XGBoost model:

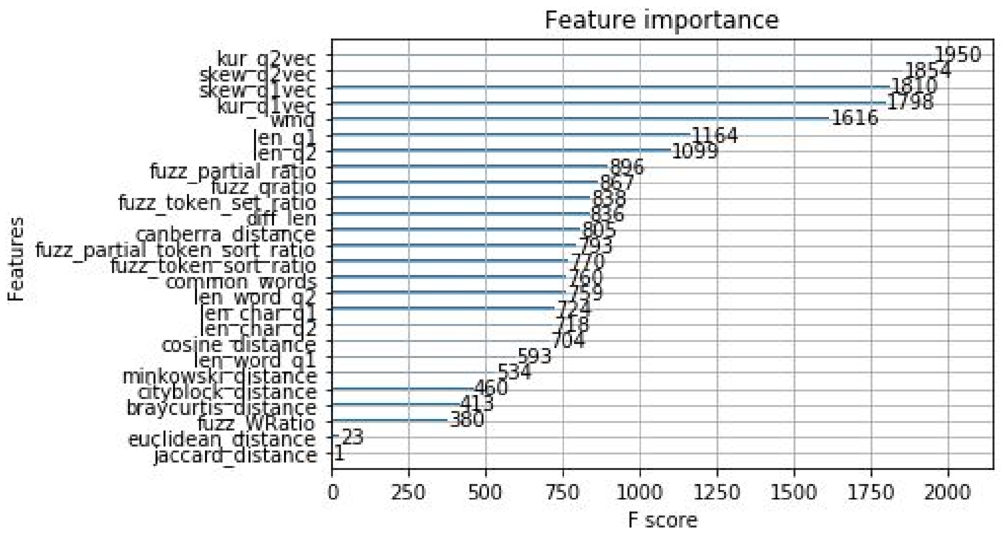

Fig. 10: F-score of each feature

Overall

Picking the correct features to feed a model is the most important part when dealing with NLP-related problems. The similarity between strings is something that is not easy to answer even for a human. This part of the exercise helped us in understanding the importance of creating features that contribute the most in machine learning models.

Working with "big data" (half a million strings) is something that is not straightforward and requires different techniques even for simple tasks. For example, the preprocessing step needs to be run beforehand and stored somewhere for use in the later stages (training, experimenting); even different preprocessing steps need to be stored in order to make it easier to create new features or debug part of your program.

Also, the first part gave us the chance to experiment with probabilistic methods for similarity prediction and observe the trade-offs that occur when Locality-sensitive hashing is used.

Unfortunately, the amount of data is way too much for being handled by a laptop, so we chose to use only a part of them in order to make the data comparable, even though we could run the program for all data at once.

Requirement 3: Sentiment analysis

In this part of the project, we are given a dataset which is comprised of 25,000 IMDB movie reviews and we train different classifiers in order to classify those articles into 2 different sentiment categories: **1** (Positive) and **0** (Negative).

The classification accuracy will be evaluated on unlabeled test data (25,000) and submitted to the Kaggle competition "BigData2020 Sentiment" which can be found in the following link: https://www.kaggle.com/c/bigdata2020-sentiment/

The 3rd requirement is essentially divided into two parts, as we are required to create and evaluate a classic machine learning model with the Bag of Words representation and a deep neural network using the Keras library.

We load the dataset and drop the 'Id' column, as it is unnecessary and we **remove the duplicate articles** in the train dataset, i.e. those which have the exact same content. However, only 96 duplicates were removed in this way. We can also observe that, unlike the dataset in requirement 1, the classes are pretty well-balanced in this case.

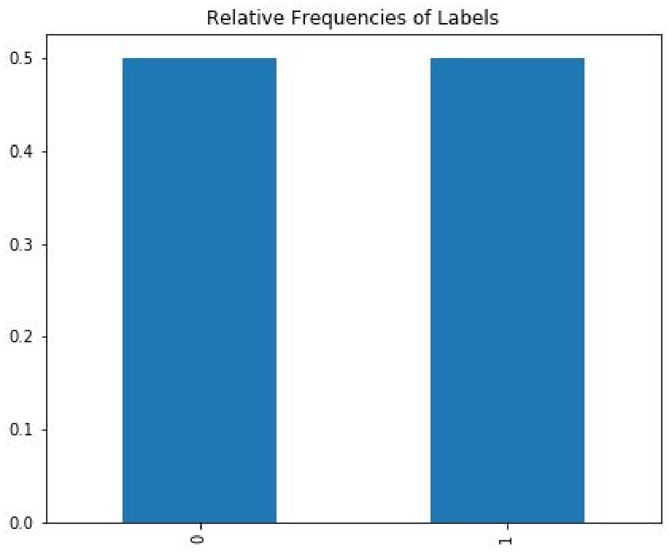

Fig. 11: Relative frequencies of the two categories

Classic machine learning method

We decided to extract features from the movie reviews in a very similar manner with question 1b and more specifically, using the **TF-IDF vectorizer** which *removes stop words* and extracts the *bigrams* as features (ngram_range = (1, 2)). However, we set *different upper and lower thresholds for the document frequency* and in particular, a maximum data frequency of 0.45 (max_df = 0.45) and a minimum data frequency of 2 (min_df = 2). We did so, because in this case, the classes are just two, they are almost perfectly balanced and the dataset is smaller.

We also decided to implement a Linear Support Vector Machines classifier, without tampering with the standard configurations of the scikit-learn function.

Deep neural network method

After many experiments with LSTM and bidirectional LSTM networks, we found out that a simpler, **Multilayer Perceptron** (MLP) combined with a Bag of Words representation gave us better results. In order to implement the model, we used the same feature extraction technique (TF-IDF vectorizer) as with the classical machine learning method and we defined its architecture using the **Keras** library, running on top of **TensorFlow 2**, so as to take advantage of GPU accelerated training in our notebooks.

In order to evaluate our MLP with 5-fold cross validation, we defined a function which iteratively fits the TF-IDF vectorizer on the training dataset and provides the shape of the feature matrix as the input shape for the network. Because of this, each of our 5 models which are trained, has a different input shape and thus, a different architecture. Each of the models is then trained on 90% of the training set and uses the other 10% for validation, i.e. it is trained on 90% of the 80% of the 24,904 movie reviews (17,930 samples) which are left after the removal of the duplicates.

We also used two Keras callbacks. In particular, we used '**Early Stopping: 3**', meaning that the training of the network automatically stops if the validation loss does not improve after 3 consecutive epochs and '**Model Checkpoint**' which saves the latest best model, i.e. the one with the smallest validation loss. Each model was trained with the AMSGrad variant of the Adam optimizer (lr = 0.005) for a maximum of 80 epochs with a batch size of 256. Below, we can see the architecture of a sample MLP (note that the number of dimensions in the input layer is not necessarily constant):

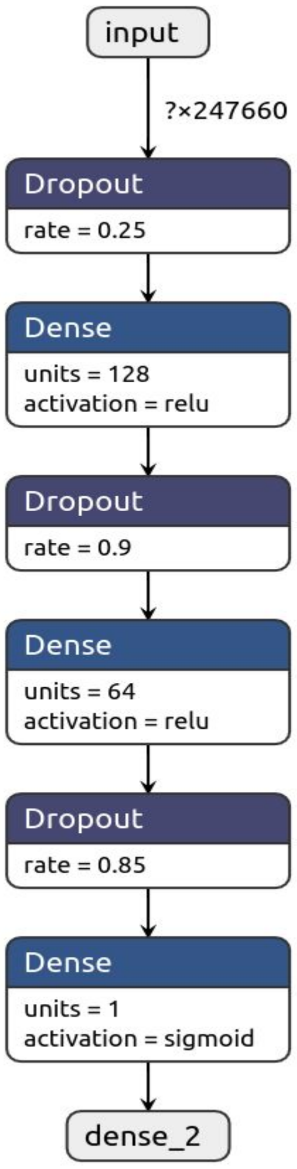

Fig. 12: The architecture of the Multilayer Perceptron

In Fig. 12, we can see that the input is immediately passed to a *dropout layer* with a dropout rate equal to **0.25**, so that there is a chance that some of the features are instantly forgotten by the network. This is because the features have been extracted by the TF-IDF vectorizer which has been fitted on the training data and there is always the chance that they will not appear in the validation or test data as well. Afterwards, there are *two fully connected* (dense) *layers* with **96** and **64** neurons (with ReLU activation functions), which are followed by *dropout layers* with **0.9** and **0.85** dropout rates respectively. Finally, the last layer is a fully-connected layer with a **single neuron** and the Sigmoid activation function so as to give the probability of a movie review belonging to one of the two categories; positive sentiment (1) or negative sentiment (0). Thus, we use the Binary Cross-entropy loss (between true and predicted labels) in the models. Below, we can see a typical training process of one of the models:

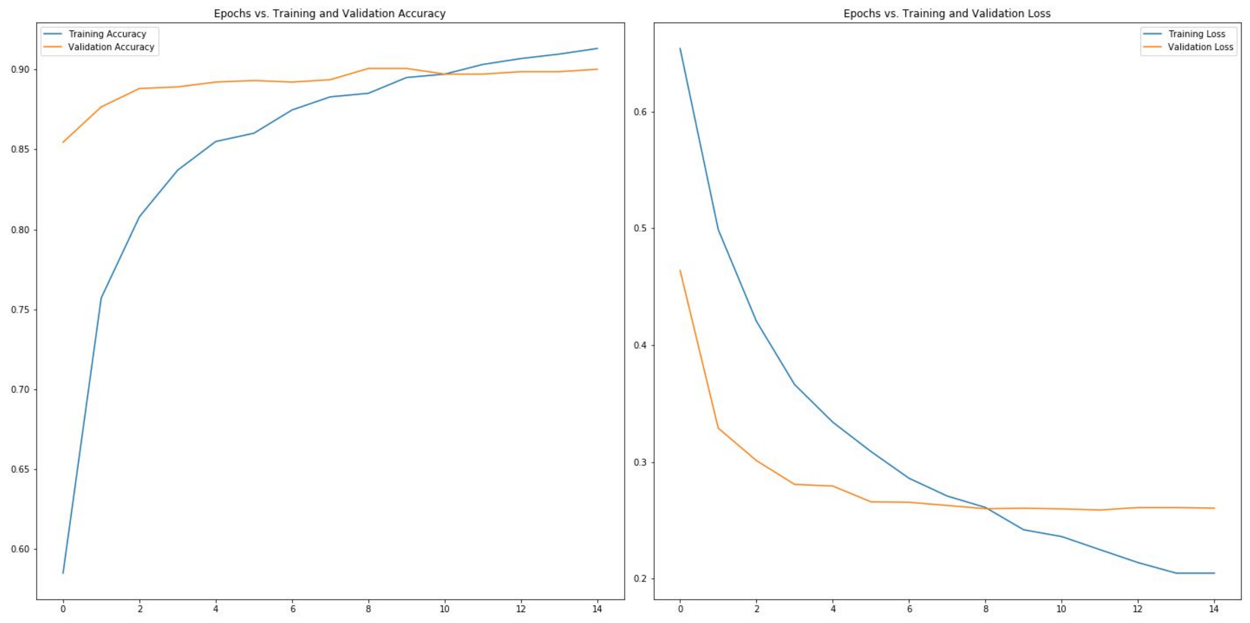

Fig. 13: Epochs vs. training and validation accuracy and loss

In Fig. 13 above, we can see that the training of a single model converges in ~10 epochs and afterwards, the Early Stopping callback automatically stops training as the training accuracy keeps getting higher while the validation accuracy does not improve, a situation known as overfitting.

Proposed method

Although not strictly a part of the project, we decided to train a classifier which can beat the benchmark.

We use the same TF-IDF vectorizer that we used for the SVM and MLP classifiers above and follow the general procedure that we had used in question 1b. More specifically, we split our data into two sets, one that will be used for training and cross validation and which comprises 80% of the samples and one that will be used as a test set to measure the generalization potential of our method and which comprises 20% of the samples. We do this because, as we mentioned in question 1b, we evaluate the performance of many classifiers (contained in the scikit-learn library), choose the best ones and fine-tune their hyper-parameters on the training and validation set (using 5-fold cross validation) and thus, the test set can provide us with a better estimate about the generalization performance of the classifier.

Our proposed method is once more the **voting classifier** (VotingClassifier), which combines 3 classifiers and predicts the output label based on the majority vote. In requirement 3, however, we decided to combine the following classifiers:

- Linear Support Vector Machines classifier (C = 0.85)
- Ridge classifier (α = 1.15)
- Passive Aggressive classifier (C = 0.75, Early Stopping)

We should note, that, yet again, we report the performance metrics of our method based on 5-fold cross-validation on 80% of the data and not based on the test set which comprises 20% of the data and thus, ***its actual performance is bound to be a lot higher***, as it resulted in **89.96 % accuracy** on our test set.

Method	**Precision**	**Recall**	**F-Measure**	**Accuracy**
SVM	89.73 %	89.71 %	89.70 %	89.71 %
MLP	90.35 %	90.31 %	90.31 %	90.31 %
Proposed Method	89.49 %	89.46 %	89.46 %	89.46 %

Table 3: Performance metrics on the sentiment analysis task

There are few things that we can observe based on the results of Table 3. First of all, even though the MLP is a more complex model to implement, train and evaluate compared to the Linear SVM classifier, the difference in their accuracy (as well as in the other performance measures) is not very large, ~0.6 %. Our proposed method seems to underperform compared to both the SVM and the MLP, but we have explained the reasons why above. Nevertheless, the test accuracy of the proposed ensemble was 89.96%, which is between that of the SVM classifier and the MLP, although it is more robust, as the ensemble can generalize to new data better. Therefore, it is probable that these are the levels of the peak performance of any classifier, given the preprocessing methods that we have implemented. If we used thorough preprocessing of the texts, for example through lemmatization, stemming, part-of-speech tagging, etc., we would possibly be able to create the basis for a good word embedding and thus, get a better performance with a bidirectional LSTM network. Alternatively, if wanted to get top-tier results, we would use transfer learning (i.e. get the pretrained model, add an output layer and train on our task) with a highly optimized model, such as BERT.

Conclusion

In this project, we used a variety of methods in order to be able to handle large datasets and address different important tasks related to Natural Language Processing.

In **requirement 1**, we displayed the word clouds of the 4 categories of news articles that we want to classify and we compared two classifiers (Linear Support Vector Machines and Random Forest) and two feature extraction/generation techniques (Bag of Words and Singular Value Decomposition). We also created an ensemble model (Voting classifier) which was able to give pretty good results on the task.

In **requirement 2**, we benchmarked some text similarity methods which can be generalized to a vector similarity problem in order to see the tradeoffs of using approximate vs. exact methods and how the approximate methods can be tuned. In addition, we faced the challenge of solving the real problem of predicting whether two questions are similar. This required further experimentations on string preprocessing, feature engineering over word vectors and picking/tuning a machine learning XGBoost model.

In **requirement 3**, we handled a task which is close to that of requirement 1. In particular, we classified IMDB movie reviews into positive and negative sentiments, comparing the performance of a Linear Support Vector Machines classifier and a Multilayer Perceptron, an artificial neural network. We also followed the same method that we used in requirement 1 and created an ensemble model that combined three classifiers and was able to give good results, as well.