Big Data Analytics Project

George Mandilaras, DS1.190012
Theodore Mandilaras. CS2.190018

February 2020

1 Text Classification

1.1 Wordclouds

A good way to visualize the content of the documents of each category, is Wordclouds. So for each category, we unified the texts of its documents and constructed wordclouds based on the frequencies of each word. Moreover, before calculating the frequencies, we cleaned the documents from the unwanted characters using *gensim*'s *simple_preprocess*, we removed stopwords and applied lemmatization. Since *wordsnet*'s Lemmatizer default mode is to lemmatize only nouns, we used position tags in order to lemmatize all parts of speech. Furthermore, after producing the first wordclouds, we noticed some further unwanted words. These words, were not necessarily stopwords, but were words that appeared in documents of multiple categories, and hence we thought that might incommode the classification experiments. As a result, we decided to add these words to our stopwords set and repeat the whole procedure. We would also like to mention, that we tried to create the wordclouds using the TF-IDF weights instead of just words' frequencies, but the time required for computing these weights prevented us from doing it.

The produced wordclouds are depicted in Figure 5.

1.2 Classification Task

In this part, the task was to experiment with some well known techniques for text classification, such as *Bag of Words (BoW)* and *SVD* which is a dimensionality reduction method, and then to perform classification using *SVM* and *Random Forest* classifiers. In more details we wanted to examine the performance of *SVM* and *Random Forest*, firstly using only *BoW* and then using also *SVD*. We also examined the performance of *Stochastic Gradient Descent Classifier (SGD)* and propose our method which uses *Bagging Classifiers*.

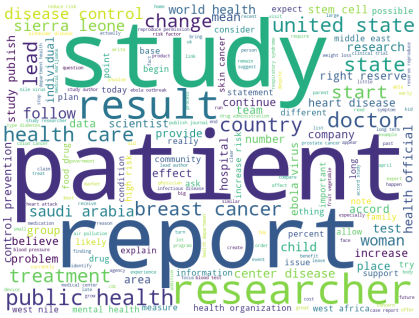

Figure 1: Health

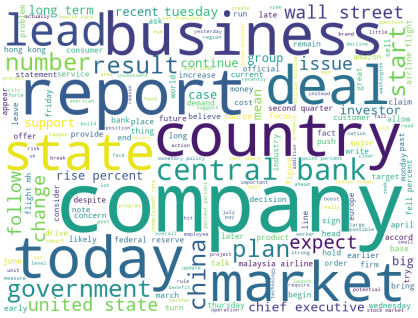

Figure 2: Business

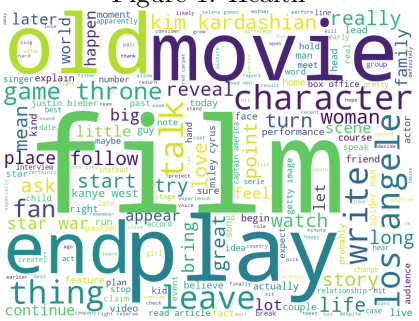

Figure 3: Entertainment

Figure 4: Technology

Figure 5: Wordclouds

Pre-Process, Vectorization and SVD

Before vectorizing the training set, we removed stopwords and non-alpha characters using regular expressions, and we also applied stemming. Regarding stopwords, we used the extending set that we used in the creation of the wordclouds.

After cleaning our documents, we vectorised them. However, as the training set is relatively big, we could not simply use the *TfidfVectorizer* of *sklearn* as it requires an extreme amount of time and leads to memory errors. As a result, we managed to surpass this difficulty by applying the hashing trick using the *HashingVectorizer*, which returns the frequencies of the words, just like the *CountVectorizer*. Afterwards, in order to convert the frequencies into TF-IDF values, we used the TfidfTransformer to the produced vectors. This way we accomplished to significantly reduce the vectorization time.

Regarding *SVD*, we reduced the dimensions of vectors into 200 components. Unfortunately, due to the big training dataset and the extreme amount of time required for evaluation, we were not able to further experiment in order to find

the optimal number of components.

Hyper-Parameter detection and evaluation

As mentioned before, the classifiers used in the first experiments are *Random Forest*, *SVM* and *Stochastic Gradient Descent*. For each experiment (e.g. BoW+RF, BoW+SVM, BoW+SGD, SVD+RF, SVD+SVM and SVD+SGD) we had to find the optimal hyper-parameters of each classifier. So, we used *Grid Search* which allows to give a range of arguments for the hyper-parameters and then exhaustively tests all the combinations of them and finds the optimal one for the problem. The results of grid search are displayed in Table 1.2. However, once again it is important to mention that this procedure requires an immense amount of time, thus we were not able to provide it with a wide range of arguments for testing and tuning.

Methods/Classifiers	SVM	Random Forest	SGD
BoW	C: 10 gamma: 0.1 kernel: 'rbf	n_estimators: 140	loss: hinge max_iter: 3000
SVD	C: 10 gamma: 1 kernel: 'rbf'	n_estimators: 200	loss: modified_huber max_iter: 3000

Table 1: Hyper-parameter detection using Grid Search

Then, after acquiring the optimal combination of hyper-parameters, we evaluated our models using *5-Fold Cross Validation*. The results are depicted in Table 2. From the results we can see that BoW+SVM produced the best performance. However, it was also by far the most time consuming one, as in *sklearn*'s implementation there is not a multiprocessing option. The execution times of the evaluations are displayed in Table 3. In the cases of Random Forest and SGD we used 12 concurrent jobs.

Statistic Measure	SVM (Bow)	Random Forest (Bow)	SGD (Bow)	SVM (SVD)	Random Forest (SVD)	SGD (SVD)	Our Method
Accuracy	0.9754	0.9356	0.9641	0.9590	0.9546	0.9291	0.9648
Precision	0.9754	0.9356	0.9641	0.9590	0.9546	0.9291	0.9648
Recall	0.9754	0.9356	0.9641	0.9590	0.9546	0.9291	0.9648
F1-Measure	0.9754	0.9356	0.9641	0.9590	0.9546	0.9291	0.9648

Table 2: Evaluation

	BoW+SVM	Bow+RF	BoW+SGD	SVD+SVM	SVD+RF	SVD+SGD
Evaluation Time (sec)	14560	621	4	1595	255	3

Table 3: Evaluation Time

Our Method

After observing the good performance of SVM, our idea was to construct an improved method that uses SVM. So, in our pre-process instead of stemming we use lemmatization with position tags, punctuations removal and TF-IDF vectorization. Furthermore, instead of the regular SVM we used *BaggingsClassifiers* with SVMs in order to enable the execution in multiple cores and therefore to reduce the total execution time. Even thought our method underperforms in comparison to the plain SVM during the evaluation, it produces better results for the testing set, achieving **Accuracy 0.96521**.

We also implemented an other method, using Deep Learning approaches. In this method we transform the input texts into *Word Embedding*[1] vectors and then feed them into an one dimensional *Convolution Neural Network (CNN).* In more details, the architecture of this model consists of a *keras' Embedding Layer*, responsible for transforming the input text into word embedding vectors, then by two one dimensional *keras' Convolution Layers* followed by *Max Pooling layers*. In the end, there are two hidden *keras' Dense Layers* followed by the output layer. This is a very complex architecture containing more than five millions parameters, and hence its execution was very slow. Unfortunately, we was not able to perform the evaluation procedure, and in our prediction experiment, the model over-fit to the training set (even though we used early stopping callbacks), leading to a significantly bad performance in the testing set. Due to the excessive amount of time required for just the training procedure (over 9h on Google's Colab using GPU), we were not able to experiment even further and decided to abandon this method.

1.3 Software Hardware

All the programs are developed in python 3.7 as Jupyter Notebooks.

Project Structure: Files related to this task are the following:

- **text_classification/wordcloud_producer**: notebook that produces the wordclouds.
- **text_classification/classification_experiments**: notebook that contains all the experiments that are mentioned in this chapter.
- **text_classification/Classification_ML**: notebook that contains the execution and the performance of Baggings SVMs
- **text_classification/Classification_DL**: notebook that contains the execution and the performance of E+CNN model (Embedding + CNN)

All the experiments were performed in a Laptop containing an Intel i7-9750H with 12 cores, 8GB RAM, and a GTX 1050 GPU and in Google's Colab.

[1]For more information about Word Embedding, see section 3, DL Approach

2 Nearest Neighbor Search and Duplicate Detection

2.1 DeDuplication with Locality Sensitive Hashing

In this task, we were provided with a training set containing Quora's questions and and a testing set of the same format. The goal was to find how many of the documents in the testing set exist in the training set, using the methods below:

- Exact Cosine
- Exact Jaccard
- Random projection (LSH Family)
- MinHash (LSH family)

Duplicates are defined as the documents with pair similarity more than a threshold t = 0.8

Preprocess: For each method we cleaned the given datasets (train and test) with the same way as mentioned in our method of **Text Classfication**[2].

Exact Cosine: After cleaning the questions, we convert them into vectors of words frequencies using *CountVectorizer*. In this step, the vectorizer is fit only to the training set and therefore words of testing set that are not included in the training set, are omitted. Then for each vector of the testing set, we calculate its cosine similarities with all the vectors of training set and we assume that a pair is a duplicate if its similarity exceeds the threshold of 0.8. The cosine similarity was calculated using the *cosine_similarity* of *sklearn.metrics.pairwise*. It's worth noticing, that this implementation allows the user to give the whole datasets (instead of vectors) as its input arguments and it returns an array of their similarities. However, in our case this would result to a huge array of 2858914260 elements (= train * test— 531990*5374), which our machines were incapable of storing in their memories, leading to MemoryErrors. As a result, we decided to calculate the similarities using the nested loops option. Furthermore, we managed to accelerate the procedure by using sparse arrays instead of regular ones.

Exact Jaccard: Similarly, for each question of the testing set, we calculate the *Jaccard similarity* with every question of the training set. In this task, we don't vectorize the input questions, but instead we calculate their intersection and union, and then divide them ($J(Q_1, Q_2) = |Q_1 \cap Q_2|/|Q_1 \cup Q_2|$). An

[2]Pre-process: Removal of non-alpha characters, punctuations, stopwords and then Lemmatization using position tags

alternative way that we could have implemented this, is by converting the texts into one hot vectors (i.e. vectors that have 1 in the positions of the words they contain or else 0), and then to compute the unions and the intersections using binary operations. We strongly believe that this would have been a faster implementation, but unfortunately we thought it too late to implement it.

MinHash - LSH (or LSH-Jaccard): Regarding MinHash, we use the implementation of datasketch[3]. After cleaning the questions, we create a minHash table using the cleaned training set, the threshold and the requested number of permutations. Then, for each question of the clean testing set, we make a query to the minHash table and then we consider that all the returned elements are duplicates of the examined question of the testing set.

Random Projection - LSH (or LSH-Cossine): Similarly with the *Exact Cosine*, we transform the questions into word's frequencies vectors using the *CountVectorizer*. Then, we build the LSH table for the current K. To build the LSH table, at first K random vectors are generated (with same dimensions as our vectors). Then for each vector of training set we compute its signature, in which each digit represents the position of the vector according to the position of the current random vectors. Then, according to their signatures, the vectors are inserted into a hash table. The values of the hash table are the vectors separated according to the signatures they got, vertically concatenated in a *sparse matrix*. After the construction of the hash table, we loop over the vectors of the testing set and compute their signatures. If a testing vector's signature exist in the hash table we calculate the cosine similarities with all the vectors that are pointed by the signature in the hash table. If the cosine similarity exceeds the declared threshold, then the pair is considered as a duplicate.

The results of the duplicate detection procedure are displayed in Table 2.1.

Type	**BuildTime**	**QueuryTime**	**TotalTime**	**#Dupilcates**	**Parameters**
Exact-Cosine	-	376.48	376.48	27087	-
Exact-Jaccard	-	5348.67	5348.6	5644	-
LSH-Cosine	595.22	176.96	772.18	23694	K=1
LSH-Cosine	520.27	92.45	612.72	19877	K=2
LSH-Cosine	377.663	46.48	424.14	16758	K=3
LSH-Cosine	335.7	24.61	360.31	15598	K=4
LSH-Cosine	482.62	16.9	499.52	13077	K=5
LSH-Cosine	346.20	16.22	362.42	12211	K=6
LSH-Cosine	360.614	9.08	369.69	10709	K=7
LSH-Cosine	701.06	11.72	712.78	9577	K=8
LSH-Cosine	715.8	9.11	724.91	8733	K=9
LSH-Cosine	513.5	9.9	523.4	8025	K=10
LSH-Jaccard	61.35	0.651	62.001	13984	num_perm=16
LSH-Jaccard	86.38	0.887	87.267	12529	num_perm=32
LSH-Jaccard	131.37	1.32	132.69	12771	num_perm=64

Table 4: Results of Duplicate Detection

[3]http://ekzhu.com/datasketch/lsh.html

Notes:

- **Text Preprocess Time**: Full train-set: 237.1(s) Full test-set: 2.4(s)
- **Vectorization Time**: 2.6(s)
- In the above results, the build times of the random projections' hash tables, are affected by other activities of the computer.
- Regarding the choice of the vectorizer in the Exact Cosine and in the Random Projection methods, we believe that the TF-IDF weights don't have a significant importance in this kind of problem, and therefore we decided to use the *CountVectorizer*.
- We also implemented our own Cosine Function named *Angular Similarity*, however, we did not use it as it required too much time to complete a query.

2.2 Same Question Detection

In this task, we get to create a model which should be able to predict if a pair of Questions is duplicate or not, given a dataset with questions of Quora. To deal with that problem, first of all we had to find and extract some features from the data which would assist us to create the desired model. Then, we searched to find which classifier performs the best. In the end, we concluded with two different approaches regarding the features. In both cases, we trained a Random Forest, Stochastic Gradient Descent, and an XGBoost model. Then, we evaluated them using 5-Fold Cross Validation and picked the one with the best performance for our final predictions of testing set.

Regarding the hyper-parameters of the classifiers[4], we used grid search to detect the best hyper-parameters, just like in **Text Classification**.

Approach 1:

In the first approach, we concatenate the pairs of questions and then clean them from the unwanted characters, punctuations and stopwords, and vectorize them using their TF-IDF weights. Then, in order to reduce the dimensions of these vectors we apply *SVD* with 180 components. In the end, we have vectors of 180 length, which we use for training. The results are appeared in Table 5.

Approach 2:

In the second approach, we proceed with some feature engineering.
We start by creating some **basic features** which include length-based and

[4]Grid search was applied only to Random Forest and SGD

Method	**Precision**	**Recall**	**F-Measure**	**Accuracy**
Stohastic Gradient Descent	0.7042	0.7042	0.7042	0.7042
Random Forest Classifier	0.7814	0.7814	0.7814	0.7814
XGBoost	0.7040	0.7040	0.7040	0.7040

Table 5: Same Question Detection using TF-IDF as features

string-based features. They can be easily extracted using basic string functions. We call these features **feats_1** and they are displayed bellow.

1. Length of question1
2. Length of question2
3. Difference between the two lengths
4. Character length of question1 without spaces
5. Character length of question2 without spaces
6. Number of words in question1
7. Number of words in question2
8. Number of common words in question1 and question2

The next set of features are based on **Fuzzy** string matching. Fuzzy string matching is also known as approximate string matching and is the process of finding strings that approximately match a given certain pattern. The closeness of a match is defined by the number of primitive operations necessary to convert the string into an exact match. They are part of the larger family of edit distances, distances based on the idea that a string can be transformed into another one. These features were created using the *fuzzywuzzy* package. We assumed that these features might add value to our models. We call these features **feats_2** and they are displayed bellow.

1. QRatio
2. WRatio
3. Partial ratio
4. Partial token set ratio
5. Partial token sort ratio
6. Token set ratio
7. Token sort ratio

The last set of features are based on **Word2vec** embeddings. Word2vec models are two-layer neural networks that take a text corpus as input and output a vector for every word in that corpus. After fitting, words with similar meaning

Method	**Precision**	**Recall**	**F-Measure**	**Accuracy**
Stohastic Gradient Descent	0.6302	0.6302	0.6302	0.6302
Random Forest Classifier	0.746	0.746	0.746	0.746
XGBoost	0.7186	0.7186	0.7186	0.7186

Table 6: Evaluation using the Extracted Features

have their vectors close to each other, so the distance between them is small compared to the distance between vectors for words that have very different meanings. We use *gensim* to load Word2vec features. In this case we decided to use a pre-trained Word2vec model, so we downloaded the *GoogleNews-vectors-negative300.bin.gz* which is a binary pre-trained model trained with Google News corpus. In our case, we need vectors for all of questions1 and questions2 in order to come up with some kind of comparison. In order to do that, we get the vectors of all the words of a sentence, which are included in the word2vec model, and sum and normalize them producing a single vector which represents the semantic of the sentence. With these vectors now we can calculate similarities between the questions. In the end, we export as features a variety of distances used from word2vec embeddings, to get the similarities between two text questions. We call these features **feats_3** and they are listed bellow.

1. Cosine distance between vectors of question1 and question2
2. Manhattan distance between vectors of question1 and question2
3. Jaccard similarity between vectors of question1 and question2
4. Canberra distance between vectors of question1 and question2
5. Euclidean distance between vectors of question1 and question2
6. Minkowski distance between vectors of question1 and question2
7. Braycurtis distance between vectors of question1 and question2

To sum up, we export 22 different features for each pair of questions. We feed them to the classifiers and we search for the best hyper parameter. The performance of the classifiers are displayed in Table 6.

Conclusion: Our final predictions of the test set is implemented using the features of **Approach 2**, and the Random Forest Classifier with *460 estimators*. This method achieves 0.746 accuracy during evaluation and reaches **Accuracy** up to **0.75512** on the predictions.

2.3 Software Hardware

All the programs are developed in python 3.7 as Jupyter Notebooks.
The second task is located in the duplicate_detection folder.
2a DeDuplication with Locality Sensitive Hashing: Files are located at deduplication folder.

- Exact Cosine: cosine.ipynb
- Exact Jaccard: jaccard.ipynb
- Random projection (LSH Family): minhash_lsh.ipynb
- MinHash (LSH family): random_projection_lsh.ipynb

2b Same Question Detection: Files are located at duplicate_prediction folder.

- TfIdf Based Predictions: tfidf_predictions.ipynb
- Feature Based Predictions: feature_prediction.ipynb

All the experiments were performed in a Desktop PC containing an Intel i5-6500 with 4 cores and 16GB RAM and in Google's Colab.

3 Sentiment Analysis

In this task, we were provided with IMDB movie reviews data and our goal was to apply sentiment analysis, i.e. whether a review is positive or negative. For this purpose, we used two approaches, a machine learning one, and a deep learning one.

3.1 Machine Learning Approach

In this approach, we used the method used in the classification part of **Text Classification**, which means that we used lemmatization, TF-IDF weights as vectors and then bagging SVM classifiers, except of removing stopwords. In sentiments analysis, removing stopwords is not considered a good technique as it may affect the meaning of the sentences, and therefore we chose to not remove them. For the evaluation we used *5 Fold Cross Validation* and the results are displayed in Table 7. Furthermore, we used it to predict the testing set and it accomplished to achieve **Accuracy** up to **0.8620**, as it was computed by Kaggle.

3.2 Deep Learning Approach

In the deep learning approach, we decided to not uses the regular text pre-process routine, but to do something else that fits more to deep learning algorithms. Therefore, we used *Word Embedding*, which are projections of words into vector space that allows words with similar meaning to have a similar representation.

Before computing the word embedding vectors, we applied basic text pre-processing techniques to data in order to clean the texts, such as tekenization, removal of the non-alpha characters and punctuations etc. Then we constructed our own *Word2Vec* dictionary using *gensim*. We decided to create our vectors instead of using pre-trained ones in this case because we wanted to be more subject oriented and to contain all the possible words of the dataset.

Regarding the architecture of our deep learning model, we constructed a simple *Recurrent Neural Network (RNN)*, using an Embedding layer as input, a GRU layer followed by a hidden Dense layer. The first layer is a *keras' Embedding layer*, in order to be able to read and to expect word embedding vectors as input. Needless to say, that since we have already constructed our embeddings using *Word2Vec* we set the trainable argument of the layer to false. Then follows an *Gated Recurrent Unit (GRU)* layer consisting of 16 *GRU* cells, which enhances the model with short term memory, a property essential for sentiment analysis. After that follows a dense layer with 16 units and then the output layer with a *sigmoid* activation function.

The evaluation of this model was performed in Google's Colab which offers several hours of GPU processing power. For the evaluation we used *5-Fold Cross Validation*, in which, in each iteration the model is constructed from

Method	Precision	Recall	F-Measure	Accuracy	Evaluation Time
Classic Machine Learning	0.8691	0.8691	0.8691	0.8691	81s
Deep Learning (5 epochs)	0.8438	0.8438	0.8438	0.8438	2.6h

Table 7: Model's Evaluation using 5-Fold Cross Validation

scratch. Furthermore, we used 5 epochs and 256 batch size. The results of the evaluation are displayed in Table 7. We would like to mention, that also in this case, we were not able to experiment as much we wanted, since the evaluation time was exceeding **2h**. Moreover, even though the in the evaluation the machine learning model produced better accuracy, in the prediction the deep learning model trained using 10 epochs, over-passed the machine learning model, achieving accuracy up to **0.8713**.

3.3 Software Hardware

All the programs are developed in python 3.7 as Jupyter Notebooks.

Project Structure: Files related to this task are the following:

- **sentiment_analysis/ML_approach**: notebook that contains the Machine Learning approach.
- **sentiment_analysis/DL_approach**: notebook that contains the Deep Learning approach.

All the experiments were performed in a Laptop containing an Intel i7-9750H with 12 cores, 8GB RAM, and a GTX 1050 GPU and in Google's Colab.

Our Kaggle submissions were submited using our emails which are gmandi@di.uoa.gr (George Mandilaras) and cs2190018@di.uoa.gr (Theodore Mandilaras)