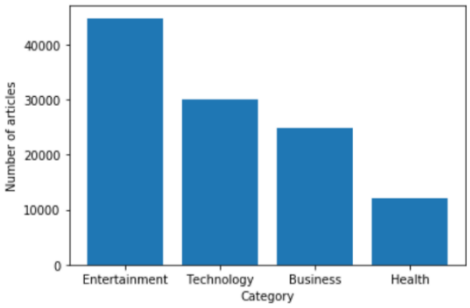
National and Kapodistrian University of Athens
Department of Informatics and Telecommunications
Big Data Mining Techniques (M118)
Winter Semester 2019 – 2020

Submitted by:
Ngomba Litombe SN1199003
Sarah Masaad SN1199008

Task 1: Text Classification

In this task, 111,795 labelled news articles were provided which belong to 4 different categories: business, entertainment, health, and technology. These articles constitute the training set, which was used to create a word cloud for the first part and then used to train a classifier for the second part of the task. To test the classifier, a set of 47,912 unlabelled articles was provided. Both these requirements are addressed in the following subsections.

The training dataset has the distribution shown in Figure 1.




```
Number of Entertainment articles: 44834
Number of Technology articles: 30107
Number of Business articles: 24834
Number of Health articles: 12020
```

Figure 1 Data distribution

Part a: Word Cloud

The training dataset was imported into a Pandas dataframe which is advantageous as functions can be applied to the whole dataframe instead of looping on every article. The title and content of every article were concatenated to create a single string of data per row.

Data pre-processing is an important step in large data analysis as it helps in reducing dimensionality. In natural language processing, there are several steps to ensure that only relevant words are analysed and that words stemming from the same root are termed as one. To this end, the following steps were done to pre-process the articles' strings:

- **Removing punctuation, symbols, and numbers:** although these form an important part in structuring sentences, they are not useful in the analysis and should be removed. The pandas.Series.str.replace method was used to remove all character types other than the English letters.
- **Lemmatization**: this is the process of returning words to their roots. Lemmatization is a complex process as it involves deciding the part of speech (POS) that a word belongs to. The WordNetLemmatizer was used from NLTK's library for this task. Since identifying the POS is a tedious job, a simplified approach was used. It was decided that the most significant parts of speech for this task were the verbs and nouns, which would ultimately identify the articles. Therefore, all words were passed through the lemmatizer twice, once with POS set to noun and once set to verb.
- **Stemming:** from our trials, it was noted that lemmatization was more accurate than stemming and, in most cases, can be substituted by it, making stemming a redundant step. Therefore, words were only lemmatized and not stemmed.
- **Removing stop words:** stop words are words that are frequently used in languages and include things like articles and helping verbs in the English language. The set of stop words used in this part included an extensive set of words much larger than those contained in NLTK's English stop words. The file containing the stop words was downloaded online and is contained within the submission file as "english2"

After pre-processing, the articles were split based on their category and a corpus was created containing all the words that belong to a category. A mask that represents each category was used for the clouds with the background set to white. Figures 2-5 show the generated clouds.

Figure 3 Entertainment word cloud

Figure 2 Technology word cloud

Figure 4 Business word cloud

Figure 5 Health word cloud

Part b: Classification

The classification problem involves a classifier that is tasked with deciding on a class for every incoming sample. The classifier is trained using labelled data, or the training set, and then tested using unlabelled data or the test set. Two different classifiers were used in this part, namely Random Forest and Linear SVM.

Random forest is an ensemble classifier, which relies on using several decision trees that individually decide on the class of a sample. The final decision is made based on a "vote", with the class having the highest votes returned. Every tree uses a random set of training samples and a random subset of the features. This classifier was imported from Python's sklearn library, and the only value changed from the default settings was the number of trees, which was set to 150.

Linear SVM involves creating several hyperplanes which divides the sample space and separates the data into classes. SVM tries to create the optimal hyperplanes by maximizing the margins between the hyperplane and the support vectors (i.e. the samples closest to the hyperplane). Again, this classifier was also imported from sklearn as LinearSVC which according to the documentation scales better to larger datasets compared to SVC with the kernel parameter set to linear.

Since the data at hand is not numeric, every article had to be transformed into a numeric feature vector. Feature extraction was done using Bag of Words and SVD.

In the bag of words method, sentences are broken down into individual words which form a dictionary. The frequency of each word in all the sentences is also stored, which would facilitate reducing the size of the dictionary to only contain the most frequent words. Next, every sentence is represented in terms of the words in the dictionary. This is done through a vector which holds in every position the number of times a word in the dictionary occurred in the sentence. The size of this array is equal to the number of words in the dictionary. To implement this, CountVectorizer was imported from the sklearn library, which returns a sparse matrix. The maximum number of features set to 10,000. In this feature vector, the least frequent word was used 210 times in all the articles. Trials with more features and did not yield any improvement but increased complexity, therefore 10,000 words were used.

For the second feature extraction method, truncated SVD was used. This method performs dimensionality reduction on the data. In python, this can be implemented using sklearn's TruncatedSVD, which takes as input sparse matrices and reduces their dimensionality to the specified value. The sparse matrices created using the count vectorizer were used and their dimensionality was reduced to 50 and 1000.

Using the two classifiers and the two feature extraction methods, four combinations were formed and tested. The same data pre-processing described in part A was performed for all methods. 5-fold cross validation method was used and the macro averages for the accuracy, precision, recall, and F-measure were calculated and are shown in Table 1.

Table 1 Results

Static Measure	RF (BOW)	SVM (BOW)	RF (SVD)	SVM (SVD)	My Method
Accuracy	94	95	93	95	97
Precision	94	95	93	94.8	97
Recall	93	94.8	89.6	94	97
F-measure	93.8	95	92	94	97

Initial results with SVD set to 50 components showed that it was not as good as BOW, scoring an accuracy of 89% for RF and 90.8% for SVM. Through dimension reduction, some of the features that helped in better distinguishing the classes were lost, thus resulting in lower scores overall. The methods were tried again with the number of features set to 1000 now, which is a tenth of the size of the original sparse matrices. This resulted in better accuracies, closer to those obtained with BOW.

The Random Classifier was also initially trained with 100 trees. Upon increasing the number of trees, the performance of the classifier improved to the results shown in the table. However, the more the number of trees the longer the model takes to train, this prevented further increase (increasing is also limited by possibility of overfitting, since more complex models are prone to this).

The table shows a fifth column containing the results of a fifth method which should improve on the results of the previous four. For this, TFIDF was selected for feature extraction, performed using the TfidfVectorizer from the sklearn library.

TFIDF is short for term frequency–inverse document frequency and is a method that improves upon the regular vectorization by giving weights to words and just by relying on their frequency within an article. Words that appear in many articles are discounted, with those appearing in all articles having a value of 0. This helps in removing words that are very frequent but do not contribute to the distinction of classes. TFIDF is achieved through finding two values for every word: the term frequency and the inverse document frequency.

The term frequency (TF) is a value specific for every word in every article, in other words, similar words in different articles have different term frequencies. It is equal to the number of times the word appears in an article divided by the total number of words in that particular article.

The inverse document frequency (IDF) on the other hand is the log of the total number of articles divided by the number of articles containing the word whose IDF is calculated. This term decreases as the number of articles containing a word increase.

The final sparse matrix is similar to that formed using the bag of words, however, the values in the sparse matrix are now the product of TF and IDF. Although stop words were removed from the articles, TFIDF attempts to remove words specifically common in our dataset. This will help remove noise from the data, which is words that do not contribute to correct classification but are excessively present.

This feature extraction method was combined with the better performing classifier, which was SVM. The results in Table 1 show that this combination outperformed the other 4, giving an accuracy of 97%. On the test set, the method performed a score of 97.453%.

Task 2: Nearest Neighbour Search and Duplicate Detection

Part A:

In this part, a large number of Quora questions is given as the training set (531990 questions). In the test set, another smaller group of question is provided. The task is to check how many of the questions in the test set exist in the training set. Duplicates are defined as questions with a similarity of more than 0.8. To evaluate similarity, cosine and Jaccard similarities are used.

Cosine similarity is defined as the cosine of the angle between to vectors. Thus, this metric requires sentences to be vectorized. The Jaccard similarity of two sets is defined as the size of their intersection divided by the size of their union, which does not require vectorization. Since Jaccard similarity only considers unique words, repetition in a sentence is not accounted for. This is contrary to cosine similarity where repetition of words alters the value. Therefore, cosine similarity may be preferred in cases where this feature is important.

The general idea for the duplicate search is to take a query at a time and iterate over the whole training set. Once a duplicate is found, the duplicate counter is incremented and the search stops. In other words, it does not matter how many duplicates are found within the training set, it is counted as one. Then, the next query comes in and so on until the test set is completed.

This is a very time-consuming task, and the time required to answer a single query increases linearly with the number of questions in the training set. An alternative approach which uses hashing is implemented. LSH, or Local Sensitivity Hashing, tries to maximize hashing collisions, contrary to normal hashing which attempts to prevent or minimize collisions. In LSH, similar inputs will be hashed into the same buckets. Thus, when looking for a duplicate, instead of looking through the whole dataset, only a subset is used, which is the subset of questions that fall within the same bucket as the question. This property of LSH introduces a number of false negatives, since there may be similar items that fall in different buckets and will not be checked. Nonetheless, LSH increases the probability that two similar items will be hashed together.

Two LSH techniques were used in this question. The first is implemented using MinHash LSH which approximated the Jaccard similarity. The second was done using Random Projection LSH and cosine similarity was used to compute the question similarities.

MinHash signature matrix can be used for the dataset and when a query comes, the jaccard similarities between the query MinHash and all the MinHashes of the original dataset are computed returning a set satisfying values above a set threshold. This is still O(N) algorithm, meaning the query cost increases linearly the number of sets. A popular alternative is using LSH index. LSH can be used with MinHash to achieve sub-linear query cost. More permutation functions improve the accuracy, but also increases query cost, since more processing is required as the MinHash gets bigger.

Random projection is a technique for representing high-dimensional data in low-dimensional feature space (dimensionality reduction). It gained traction for its ability to approximately preserve relations (pairwise distance or cosine similarity) in low-dimensional space while being computationally less expensive. The main idea behind random projection is that if points in a vector space have a sufficiently high dimension, then they can be projected into an appropriate low dimensional space such that the distances between the points are approximately preserved. For the hash function, a random vector v is generated, which determines a hash function h with two buckets.

- $h(x)$ = +1 if $v \cdot x > 0$;
- $h(x)$ = -1 if $v \cdot x < 0$

Two points which have an angle of separation less than 180 degrees have at least 0.5 similarity probability when separated by one hyperplane. As the number of separating hyperplanes increase (number of random vectors used for projection), the search space also reduces.

In the quest of finding the duplicates the following methods were implemented:

- Exact Cosine Similarity
- Exact Jaccard Similarity
- LSH Random projection
- LSH Minhash

For the implementation, the training and test datasets were pre-processed. Data pre-processing only involved removing special characters and numbers in addition to changing to lower case letters. Since the question are short and usually concise, removing stop words could alter the meaning of the question and impair the results, therefore this step was skipped. The count vectorizer was used and set to 1000 features for all methods except Minhash LSH which does not require vectorization. Table 2 shows a comparison of the results obtained from the different methods.

As the subsections will present, the cosine similarity methods (both exact and LSH) were relatively fast. This allowed further experimentation on these methods, in which TFIDF was used once with 1000 features and once with the full vector. These results are shown in Tables 3 and 4.

Exact Methods: Cosine and Jaccard Similarities

To calculate cosine similarity, every question from the test set was taken at a time and its cosine similarity evaluated against the training set. The cosine_similarity was used from the sklearn library and accepts sparse matrices as inputs. However, due to the size of the training set, it was not computationally feasible to use the whole bulk of the training set at once. Instead, the set was broken into chunks of 10,000 samples and the test was performed chunk by chunk. This also added the advantage of allowing the search to stop once a duplicate has been found in a chunk, which is when the cosine similarity value of 0.8 or more is found.

For Jaccard Similarity, the pairwise_distance method from sklearn was used. The Jaccard similarity was computed by subtracting the Jaccard Distance from 1. The main issue with this method was that it did not accept sparse matrices. Therefore, they had to be converted to dense matrices making the computation extremely long.

Random Projection LSH

A hash table was created for the training set using random projection vectors with lengths equal to the number of features in our samples. To create a hash for a sample, dot product is carried out between the sample and the generated random hashing vector. If the value is positive, the hash is set to 1, otherwise it is 0. The number of hashing vectors determines the number of bits (k) used to represent a hash. It is also the number of hyperplanes dividing the sample space into 2^k segments. This implies that the query time reduces as the value of k increases since the search problem now has complexity $N/2^k$. Figure 6 shows the number of training data samples that fall within the buckets 0-3 which were created using 2 projection vectors. Figure 7 shows the samples in buckets 0-7 where k was set to 3.

```
hash distribution
3:54824
2:38026
1:285272
0:153868
```

Figure 7 Hashes with K=2

```
hash distribution
3:62235
5:59308
0:104552
6:105490
4:95164
2:86102
7:72037
1:53328
```

Figure 6 Hashes with k=3

Note that due to the random nature of the projection vectors, the data will not be equally divided. There may be better (and worse) distributions achievable when the algorithm is run again. Before querying our data, it was attempted to obtain a relatively good distribution.

The build time was computed, which is the time taken to hash every training sample and store it under the appropriate key in the hash table. When querying, every vector in the test set was hashed and then its cosine similarity with the samples in its hash was calculated. The duplicate counter was incremented when a similarity of 0.8 was found.

The number of hyperplanes introduces a complex trade-off. On one hand, the larger the K value, the smaller the number of items within a bucket. This increases the chances of missing out on a duplicate because it was in a nearby bucket, therefore increasing the chances of false negatives. On the other hand, it reduces the unnecessary comparisons with samples that are definitely unsimilar. In lower k's there is a huge number of samples in every bucket, therefore taking a longer time to query but the number of duplicates found is closer to that in the baseline case.

Using a very high number of bits leads to a loss of control over the similarity threshold since only very similar vectors will be hashed in the same buckets. To obtain this flexibility with higher bits, more hashing tables can be used. The figure below illustrates the trade-off between

MinHash LSH

This section was carried using the Datasketch library. The questions in the test and training set were first tokenized, as this section does not require vectorization. Then Minhashing was performed the training set, creating a minhash table. The minhashes (signature matrix) have the property of retaining the same Jaccard similarity values as the original matrix. Lsh is then performed on the minhash table obtained from the training set. For the build-up time, the time taken for minhashing and LSH on the training set was considered.

In the query phase, a minhash table was computed for the test set and every minhash was hashed using LSH and compared (using Jaccard similarity) with the set of minhashes that were hashed into the same bucket during LSH, returning all neighbours with Jaccard similarities greater that the threshold (0.8). this process wass performed intrinsically by the library and all we had to do was pass in the minhashed query. If the returned list is not empty (duplicate found), the counter is incremented.

This procedure was done for three different number of permutations (16, 32 and 64). As mentioned earlier, the larger the number of permutations, the longer the time it takes to build up the signature minhash matrix. This leads to a possible increase in the number of LSH buckets (depending on the intrinsic definition of the relationship between LSH bands b and rows r), ultimately leading to an increase in query time. This is quite evident from the results obtained.

Results

Table 2 shows the results obtained from the different methods using the countvectorizer with 1000 features. The LSH Jaccard methods are in grey indicating that they were not involved in the vectorization. We note that the exact methods have longer querying times than their LSH models. However, the total time for the exact cosine is the shortest due to the small number of the test samples and the fragmentation of data which allowed early returns. This also produced the largest number of duplicates. The LSH cosines showed that increasing the number of hyperparameters decreased the querying time.

The exact Jaccard was by far the slowest in querying. The LSH Jaccard duplicates are relatively very low which may be partially attributed to the fact that the text was not lemmatized.

We also note that due to the false negatives in LSH, the number of duplicates is lower than their exact counterparts.

Table 2 CountVectorizer (1000 features)

Type	Build Time (s)	Query Time (s)	Total Time (s)	#Duplicates	Parameters
Exact Cosine	-	797	797	3382	-
Exact jaccard	-	20880	20880	2001	-
LSH jaccard	102.65	0.934	103.584	889	P=16
LSH jaccard	124.28	1.15	125.43	676	P=32
LSH jaccard	172.20	1.639	173.839	732	P=64
LSH cosine	1382.618	197.337	1579.995	3129	K=2
LSH cosine	737.93	76.25	814.18	2918	K=3

The tables below show the results obtained from the cosine similarity methods using different vectorization. In Table 3, the TFIDF maximum features were set to 1000, and in Table 4 the whole vector was used.

Note that when the longer feature vectors were used, the number of duplicates found more than halved. The build times for the models when using the full feature vectors were also longer.

Table 3 TFIDF (1000 features)

Type	Build Time (s)	Query Time (s)	Total Time (s)	#Duplicates	Parameters
Exact Cosine	-	706	706	3064	-
LSH cosine	944	106.65	2127	2747	K=2
LSH cosine	678.5	58.56	814.18	2638	K=3

Table 4 TFIDF (all features)

Type	Build Time (s)	Query Time (s)	Total Time (s)	#Duplicates	Parameters
Exact Cosine	-	1424	1424	1424	-
LSH cosine	2544	167	2711	1145	K=2
LSH cosine	1231	87.75	1318.75	992	K=3

Part B

This task takes the duplicate detection problem a step further. A set with pairs of questions is given and a label which indicates if the questions are asking the same thing or not. A model which can correctly predict this label for an unlabelled set of questions is required.

The classifier used for this task is called XGBoost. It is a decision-tree-based ensemble Machine Learning algorithm that uses a gradient boosting framework by introducing some optimization schemes in terms of speed and performance like tree pruning and hardware awareness (could use different threads in parallel for example). The algorithm was developed as a research project at the University of Washington.

Boosting in general involves aggregating weak learners to form a strong learner. A popular boosting algorithm is Adaboost. The AdaBoost Algorithm begins by training a decision tree where each observation is assigned an equal weight. After evaluating the first tree, the weights of observations that are difficult to classify are increased the weights for those easily classified are reduced. The second tree is therefore grown on this weighted data and so on. Here, the idea is to improve upon the predictions of the first tree. Predictions of the final ensemble model are evetually the weighted sum of the predictions made by the previous three models.

A similar rationale is held for gradient boosting except:

- Trees can have higher depths
- Successor trees are built based on the residuals (errors) of their predecessors and not on modified weights of observation.
- The predictions of every generated tree have equal weights (learning rate).

The XGBoost library in python provides a parallel tree boosting (also known as GBDT, GBM) that solve many data science problems in a fast and accurate way. Some hyperparameters of the xgboost classifier are listed below

- learning_rate: step size shrinkage used to prevent overfitting. Range is [0,1]
- max_depth: determines how deeply each tree is allowed to grow during any boosting round.
- subsample: percentage of samples used per tree. Low value can lead to underfitting.
- colsample_bytree: percentage of features used per tree. High value can lead to overfitting.
- n_estimators: number of trees you want to build.
- objective: determines the loss function to be used like reg:linear for regression problems, reg:logistic for classification problems with only decision, binary:logistic for classification problems with probability.
- Subsample: ratio of the training instances. Setting it to 0.5 means that XGBoost would randomly sample half of the training data prior to growing trees. and this will prevent overfitting. Subsampling will occur once in every boosting iteration.

XGBoost also supports regularization parameters to penalize models as they become more complex and reduce them to simple (parsimonious) models.

- gamma: controls whether a given node will split based on the expected reduction in loss after the split. A higher value leads to fewer splits. Supported only for tree-based learners.
- alpha: regularization on leaf weights. A large value leads to more regularization.

For feature extraction, two different methods were used. In the first method, the Jaccard similarity was computed between the two questions and the cosine similarity was obtained between the vectorized pair. The two features were combined and fitted to the XGBoost classifier without tuning.

In the second method, TFIDF was used as the feature extraction method. The following hyperparameters were used:

- token_pattern=r'[a-zA-Z_] {1,}' to specify the characters that would be used for n-grams (alphabets and underscore)
- Ngram_range = (2,3): this range specifies that 2 or 3 grams can be formed because since having features consisting of more than one word would help with retaining some meaning.
- Max_features : is simply stating the size of the resulting vector.

Next, the resulting vectors of the question pairs were concatenated and fed into an xgboost classifier with the following hyperparameters:

- n_estimators = 80; as the number of trees is increased, the classifier is able to learn more robust features but at the cost of overfitting.
- Max_depth = 50; this also increases model complexity since each tree is now allowed to go deeper but also increases learning features.
- Alpha = 4; this is a regularization parameter to the avoid the high possibility of overfitting.
- Subsample = 0.8; to reduce overfitting as well since not all of observations of the training set are used at a training instance.
- Colsample_bytree = 0.7: limiting the proportion of attributes used for the creation of trees. This also helps with the reduction of overfitting.

Due to the fact that the xgboost model had a high complexity, it took some time to train the model. Therefore, the sparse matrix size was limited to 600 for cross validation. However, the larger the vector size (more words represented), the better the accuracy. Therefore, when building the model to predict the test set labels, no limitations were imposed, and the vector size was left at 5000. A kaggle score of 0.82737 was obtained.

Table 2 shows the 5-fold cross validation results obtained from methods 1 and 2.

Method	Accuracy	Precision	Recall	F-measure
Method_1	74	73	75	73
Methosd_2	81	81	77	78

Before finally arriving at the models described above several other models which failed to perform as well were experimented with. These include:

- Word2Vec embedding + LSTM neural network: this achieved a 5-fold CV accuracy of 67%. The unbalanced dataset contributed to this poor result, as the network was more biased to one class. A few remedies were attempted like data stratification where entries of the '1' class were duplicated to create a balance as well as doubling the weight of the minority class (1), but both schemes did not improve the accuracy.
- Pretrained word vectors from Glove embedding were used with LSTM and trained models were created using the question pairs. The XGBOOST classifier was used at the end, and an accuracy of 68% was obtained.

Task 3: Sentiment Analysis

In this task, two models are to be build using classical machine learning models as well as neural networks to correctly predict the sentiment of IMDB reviews. A balanced and labelled training set is provided which labels sentiment as 0 for negative sentiment or 1 for positive sentiment. The figure below shows the class distribution of the set. The test set is an unlabelled set of reviews equal in size to the training set.

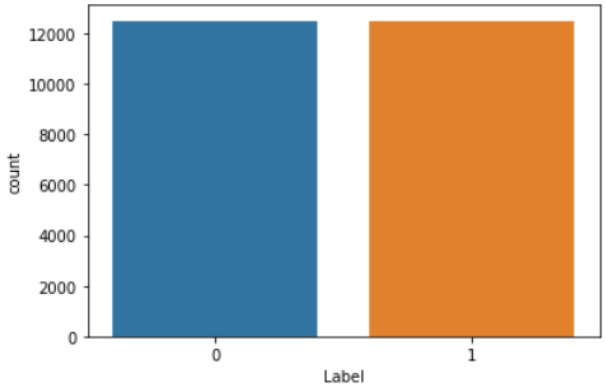


Table 5 Training data distribution

In sentiment analysis in particular, it is debatable whether or not stop words should be removed. Although some of the words unarguably do not help in distinguishing classes, some of the words in common stop word lists actually contribute to the meaning. Therefore, a list of stop words was used from which words that carried sentimental value were excluded. Take as an example the phrases wouldn't like and didn't like, with the stop words didn't and wouldn't the phrases turn into positive ones. This pre-processing approach was decided for the classical machine learning model. Lemmatization, special character and number removals, as well as changing the letters to lower cases were also performed as part of the pre-processing. Using this pre-processing, the reviews had an average of 102 words with a standard deviation of 77.

The training and testing sets were transformed using a TFIDF vectorizer and were the inputs to a Linear SVM classifier. The results on the test set scored 86.68%.

For the neural network model, the keras vectorizer was used to turn every article into a sequence of integers. This is done by mapping every word to an integer. The maximum number of words can be limited. In the training set, there were almost 24,000 unique words. Therefore, the tokenizer was tried once with 10,000 words and once with 20,000 words.

When using deep learning models, it is arguable that pre-processing texts reduces the accuracy. Since and embedding layer is used, the data's semantic meaning is explored, which advocates leaving the text in a more structured form. This was also explored as trials with and without pre-processing were performed (using both 10000 and 20000 words vectorizers). It was found that when using pre-processing the overall test accuracy ranged between 85% and 86%. This was inferior to the results obtained when omitting pre-processing.

Therefore, the only text pre-processing done was removing special characters, breaks, numbers, and changing to lower case letters. After this step, the tokenizer was used. Since sentences consist of different lengths, the feature vectors have different sizes. To work around this, the maximum sentence length was set to 500, which should accommodate the lengths of most sentences (on

average the sentences had 227 words with 168 standard deviation). For vectors shorter than 500, the vector was padded with zeros.

For the choice of classifier, the popular deep learning model for natural language processing is LSTM. However, this model was very complex and time consuming to build and train. Therefore, another popular model, less so for NLP though, was used which was CONV1D from Keras library. This model allows learning the features of the input data without paying specific attention to the position of the features

Embedding layers allow exploring sentence similarity through the similarity of the sentence vectors in the high dimensional space. An embedding layer is created with the input dimension set as the maximum number of words to be used. The output dimension is specified as well. In this model, the input was specified as 10,000 or 20,000 depending on the tokenizer limit used. The output dimension is not a strict parameter; however, some resources advise to leave it as a power of 32. This was chosen as the output layer.

The second layer is the convolutional layer, whose inputs were set as the outputs of the previous layer. The second parameter set in this layer is the kernel size which was set to 3. The activation function was set to 'relu'. A pooling layer was used to reduce the size of the input matrix by a factor decided in the parameters. This was set to 2, which means the output matrix has a size half the input matrix. This is used to avoid overfitting, however results using this parameter set to 1 also performed well.

The next layer flattens the input so that it becomes a single dimensional vector. The last two layers are fully connected layers with sigmoid activation functions. The first dense layer contains 10 nodes, while the second one has one node since we have 2 classes only.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (None, 500, 32)           640000
_________________________________________________________________
conv1d_1 (Conv1D)            (None, 500, 32)           3104
_________________________________________________________________
max_pooling1d_1 (MaxPooling1 (None, 250, 32)           0
_________________________________________________________________
flatten_1 (Flatten)          (None, 8000)              0
_________________________________________________________________
dense_1 (Dense)              (None, 10)                80010
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 11
=================================================================
Total params: 723,125
Trainable params: 723,125
Non-trainable params: 0
```

Figure 8 Model description and parameters

Due to the high number of parameters, most of the time, training with one epoch was enough. We noted that when the training accuracy was as low as 75%, the actual accuracy on the test set achieved results above 85%. We also noted that the performance using 20,000 words was better than that using 10,000 words.

There are multiple parameters involved in deep learning models, which makes finding the optimum parameters to work with very hard. Multiple trial and errors were used for this. Another problem we faced was that using the same model to fit the data multiple times would result in overfitting, most likely due to the same initialization of weights as the previous round. This was reflected on the accuracy values seen in Table 5, which reports the 5-fold cross-validation measures for both the classical and deep learning models. Although the values reported show 95.6%, the actual best achieved result was 88.68%, which is no where close to the values seen here. However, it is close to the value of the first fold accuracy which was 88%.

After cross-validation the model was recreated (to avoid using the initialized weights) and trained and tested for the test set.

Table 6 Results

Method	Accuracy	Precision	Recall	F-measure
Classical	88.6	88.6	88.6	88.6
Deep Learning	95.6	95.6	95.6	95.6