Sarcasm Detection on News Headlines Dataset

Author: *Luca de Martino*

Email: lucademartino.dev@gmail.com

Abstract

The project aims to detect sarcasm in news headlines provided by the "News Headlines Dataset For Sarcasm Detection" [9]. Some data understanding techniques are applied for cleaning the dataset consequently. Later, different types of vectorization are analyzed, both semantics-based and non-semantics-based. Finally, both Neural Network-based models and non Neural Network-based models are tested on them.

1 Introduction

The goal is to understand which headlines related to newspapers and photo titles are sarcastic and which are not, using the *News Headlines Dataset For Sarcasm Detection* [9].
I proceed analyzing the dataset and understand which can be transformations necessary for the preparation of the successive phases.
To achieve this, plots are produced on the distribution of headline elements, from these I will perform cleaning operations aimed at improving a classifier's understanding of the text.
After the cleaning phase, methods of vectorization will be tested both based on the semantics of the words and not based on semantics. In particular, the type of non-semantic vectorization used will be the TF-IDF while semantic vectorization will be both static, based on Word2Vec, GloVe [4], Fasttext [2], Google News [5], and dynamic, based on BERT.
The models used will be both Neural Networks-based and non Neural Networks-based and will use different types of embeddings to try to achieve the best performance on this dataset. I apply the *hold-out* technique and I use the *k-fold cross-validation* methods to validate the models.

2 Dataset

Past studies in Sarcasm Detection mostly make use of tweets datasets collected using hashtag-based supervision but such datasets are noisy in terms of labels and language. Furthermore, many tweets are replies to other tweets, and detecting sarcasm in these requires the availability of contextual tweets.
To overcome the limitations related to noise in Twitter datasets, this "News Headlines dataset for Sarcasm Detection" is collected from two news websites. TheOnion [10], aimed at producing sarcastic versions of current events collecting all the headlines from *News in Brief* and *News in Photos* categories (which are sarcastic), and HuffPost [6], from which real (and non-sarcastic) news headlines are collected.
This new dataset has the following advantages over the existing tweet datasets:

- Since news headlines are written by professionals formally, there are no spelling mistakes and informal usage.
- Furthermore, since the sole purpose of TheOnion is to publish sarcastic news, they get high-quality labels with much less noise as compared to Twitter datasets.
- Unlike tweets, which are replies to other tweets, the news headlines they obtained are self-contained. This should help us in getting apart the real sarcastic elements.

The dataset consists of 26709 rows and each record consists of three attributes:

- *is_sarcastic*: 1 if the record is sarcastic otherwise 0.
- *headline*: the headline of the news article.
- *article_link*: link to the original news article. Useful in collecting supplementary data.

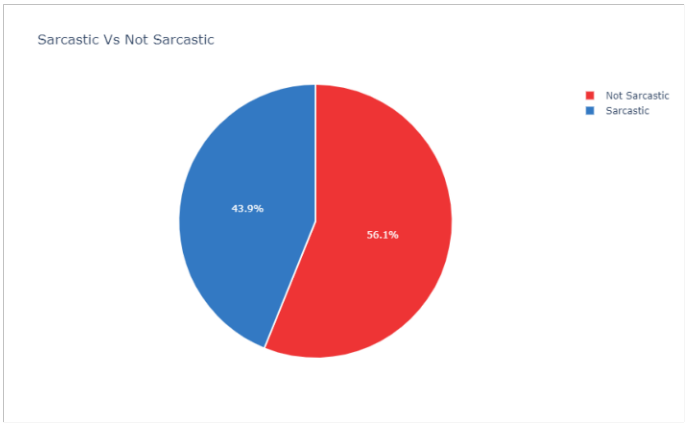

Figure 1: Pie Plot Distribution - Sarcastic vs Not Sarcastic

2.1 Data Understanding

First, I verified there were no null values in any feature and analyzed the distribution concerning sarcastic and non-sarcastic headlines as can be seen from the pie plot in Figure 1. As we can see the dataset is slightly unbalanced since we have about 56% of non-sarcastic headlines. However, since the difference is very small, it should not be necessary to apply any oversampling or undersampling technique that would alter the dataset. To be more precise, we are dealing with 14985 non-sarcastic headlines and 11724 sarcastic ones.
Then, having available the column *article_link* with the link to the article, and since the authors claimed that the sarcastic headlines were taken exclusively from TheOnion website while the non-sarcastic ones only from HuffPost website, I decided to verify that there were no errors in the creation of the dataset, specifically in the assignment of labels. Consequently, I applied a regex on *article_link* to derive the source of each headline. From this, I verified the number of headlines for each type of source.
As we can see from the bar plot in Figure 2 the sarcastic ones are all coming from TheOnion and the non sarcastic ones from HuffPost.

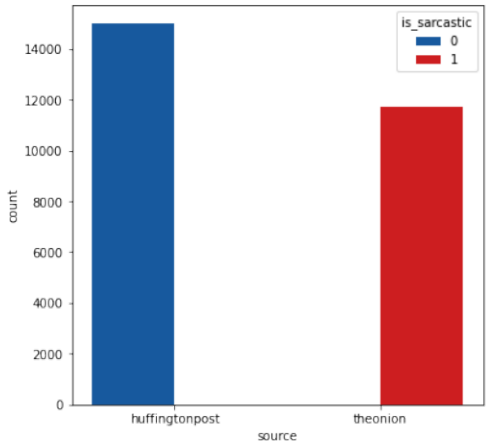

Figure 2: Bar Plot Distribution - Source

Proceeding with the analysis, the average length of the headlines was analyzed, both in total and about the is_sarcastic attribute (bar plots in Figure 3). From these, we can see that on average sarcastic headlines are longer, one might think that this stems from the fact that sarcasm needs more words to be catchy than a real news headline. While in Figure 4 we can see how non-sarcastic words are on average longer than sarcastic ones.
Later, I analyze the frequency of headlines with numerical values to understand whether or not these can be kept, in Figure 5.

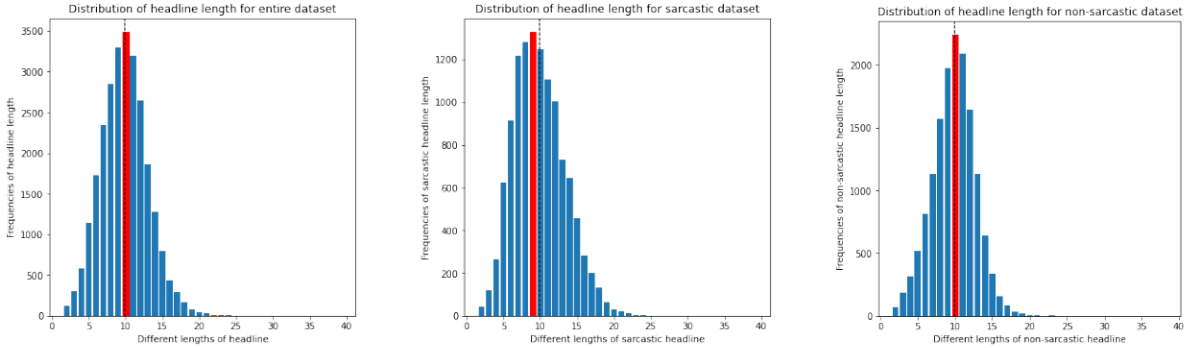

Figure 3: Bar Plots - Different Headlines Lengths

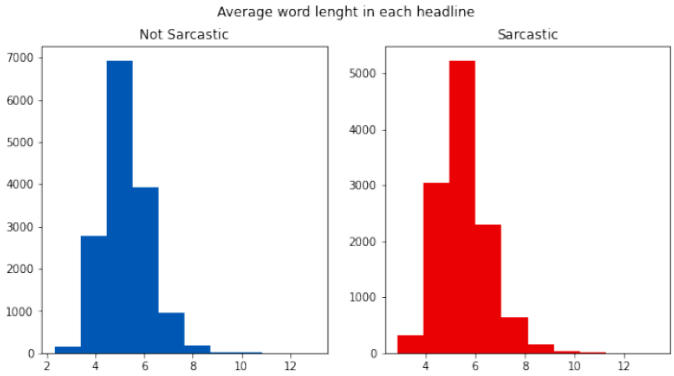

Figure 4: Bar Plot Distribution - Average Word Length

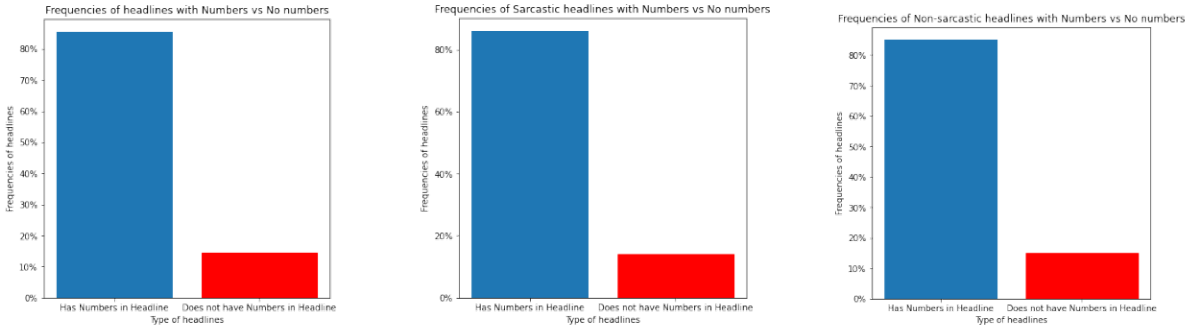

Figure 5: Bar Plots - Headline with number frequencies

The plot in Figure 6 shows which are the most frequent words both in total and by type. From these, it is clear that many stopwords should be removed.

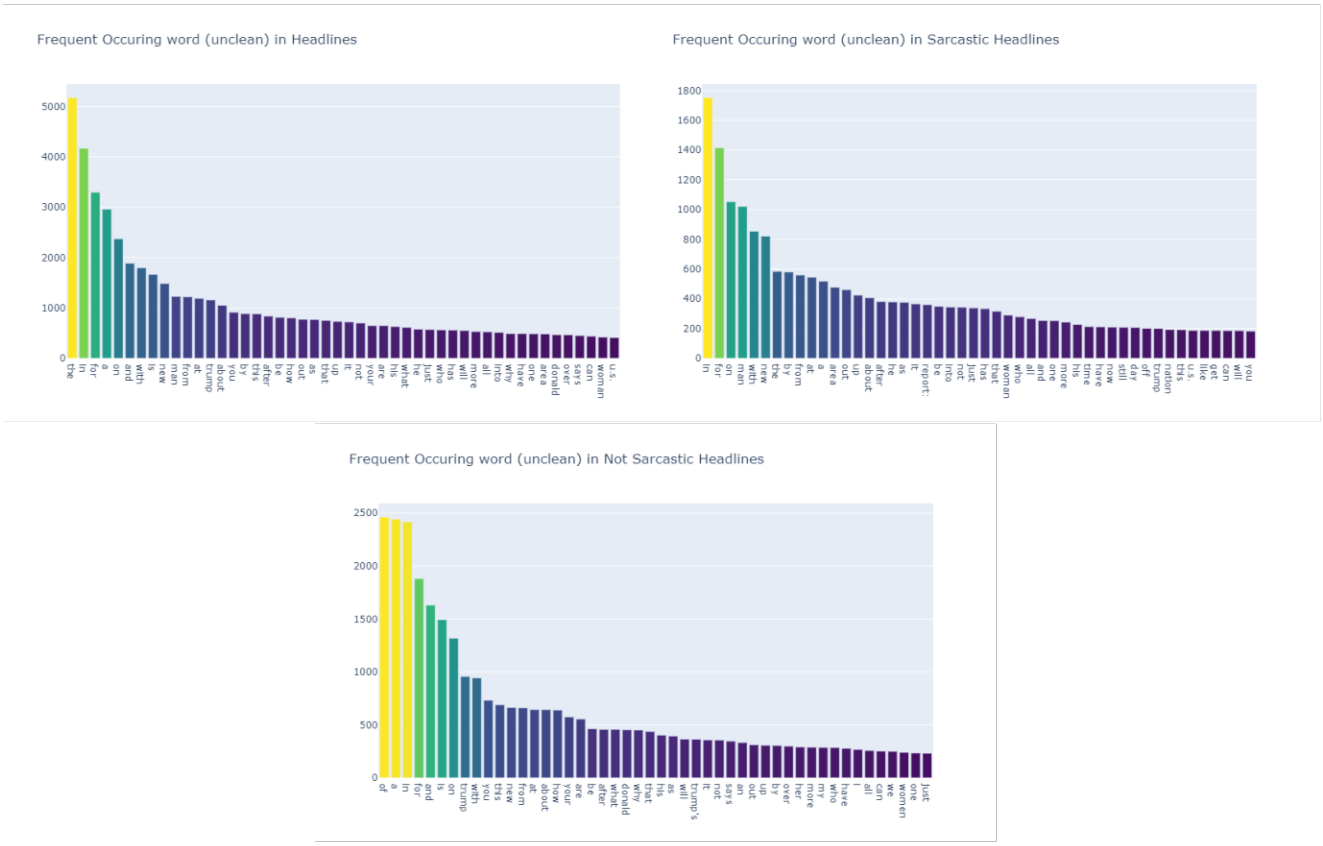

Figure 6: Bar Plots - Most frequent words

2.2 Data Cleaning

I proceed with the data cleaning phase which is one of the most important phases. If the text is not properly cleaned or processed, incorrect words or punctuation and associated redundancies get added to the data. This impacts the performance when we will have created static or dynamic embeddings [1].
As we have seen, the data does not present particular problems, so the choice of taking data from headlines compared to tweets is a winning strategy, as suggested by the authors. At this stage, I just remove stopwords, and punctuation and make all words in small cases after which all words are lemmatized.
As we know, *N-grams* are contiguous sequences of n-items in a sentence. N can be any positive integer, although usually, we do not consider very large N because those n-grams rarely appear in many different places. Through the gram analysis, I present what are the most frequent bigrams according to the class they belong to, to see which are the most frequent minimal *phrases* and try to understand which topic we are dealing with. Figure 7 shows the bigrams analysis for both the headline classes.

Finally, Figure 8 presents the word clouds that show the most frequent words after removing the stopwords in our dataset and lemmatizing the other words.

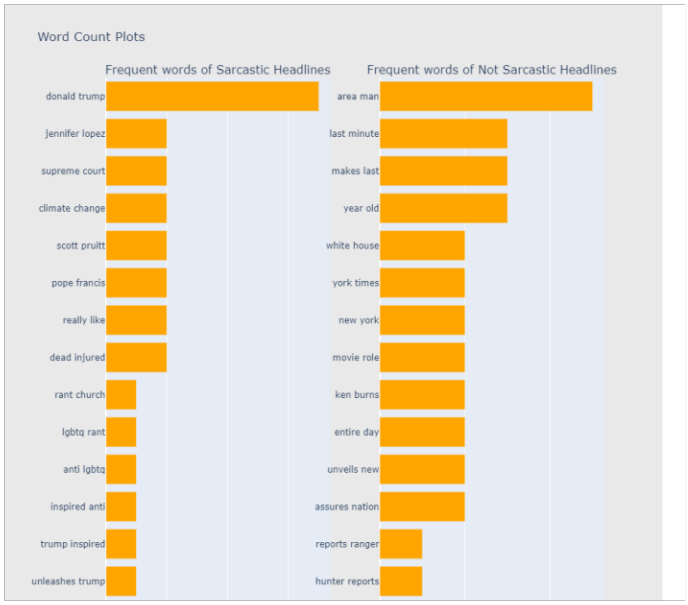

Figure 7: Bigram Analysis

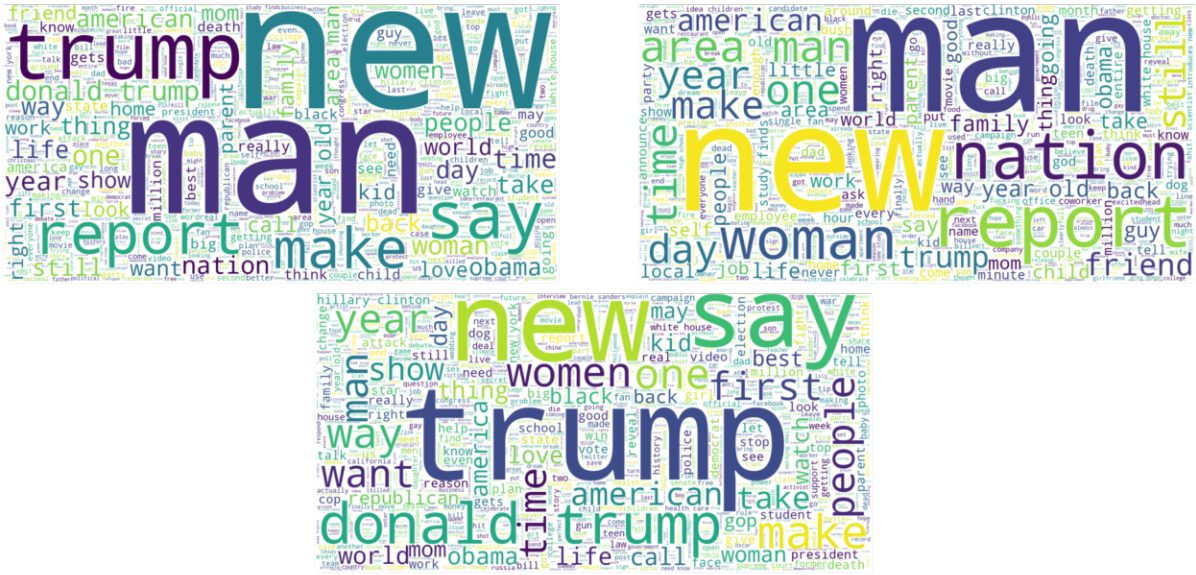

Figure 8: Word Clouds

2.3 Vectorization

In this section, I will vectorize the dataset. This should allow us to convert our data to higher dimensional containers (matrices). These vectorization strategies allow the word corpus to be properly suitable for advanced semantic analysis. We have two ways of transforming the textual corpus to a numerical vector:

- Vectorize without taking into account the semantic
- Vectorize taking into account the semantic

In the first case, the vectorization strategy is used to provide a co-occurrence probabilistic distribution for vectorization. Methods like TF-IDF and One-hot vectorization falls under this criteria. These methods leverage statistical co-occurrence probabilities and log-likelihoods for determining the frequently occurring sentences or groups of words in a corpus. The second case relies on applying vectors concerning semantic importance. Word embeddings fall under this category.
These are largely of two kinds:

- **Static Embeddings**: Word2Vec, GloVe, Fasttext, Paragram
- **Dynamic Embeddings**: ELMO, BERT and its variants, XLNet/Transformer-XL

All of these embeddings rely on pre-trained word vectors where a probabilistic score is attributed to each word in the corpus. These probabilities are plotted in a low dimensional plane and the "meaning" of the words is inferred from these vectors. Generally speaking cosine distance is taken as the major metric of similarity measurement between word and sentence vectors to infer similarity. In this brief sub-section, I will graphically explore what are the different types of vectorization that will then be considered for the next step.

2.3.1 TF-IDF

The TF-IDF, short for term frequency–inverse document frequency, is a numerical statistic that is intended to reflect how important a word is to a document in a collection or corpus [8]. In my case, after transforming the dataset I visualized it via PCA and TSVD as shown in Figure 9.

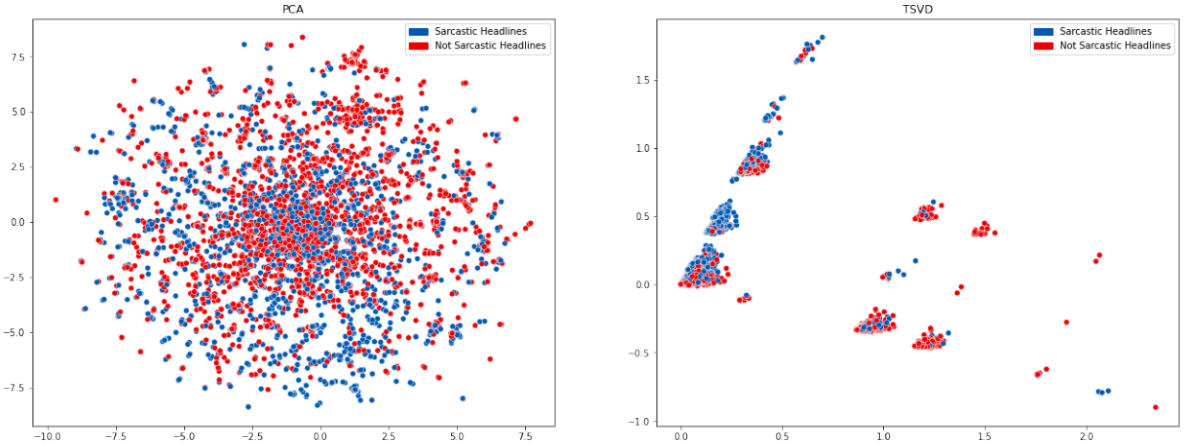

Figure 9: PCA and TSVD on TF-IDF

2.3.2 Semantic Embedding

In this context, I include embeddings that can be both static and dynamic. These are vector space transformations of the words present in the corpus. When converted to vectors, several metrics can be applied like finding similarity, and distance measurement between the vectors. With word vectors, we can specify semantic similarity between different words or between a collection of words. In particular, among the static embeddings, Word2Vec, GloVe, Google News and Fasttext were analyzed.
I present the PCA representation in Figure 10 and Figure 11 and an example of cosine similarity applied to the words *"former"* and *"store"* given the pre-trained embeddings of GloVe, Google News and Fasttext (Figure 12).

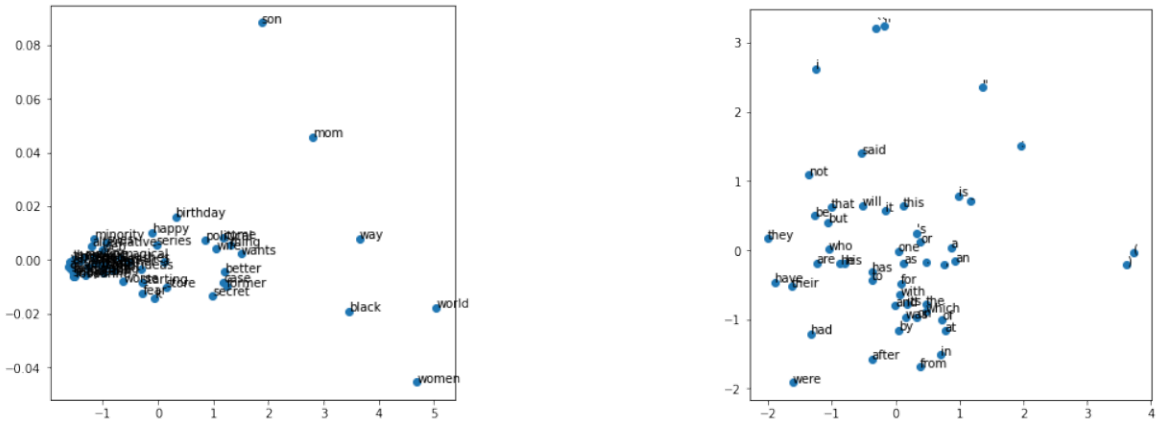

Figure 10: PCA on W2V and GloVe

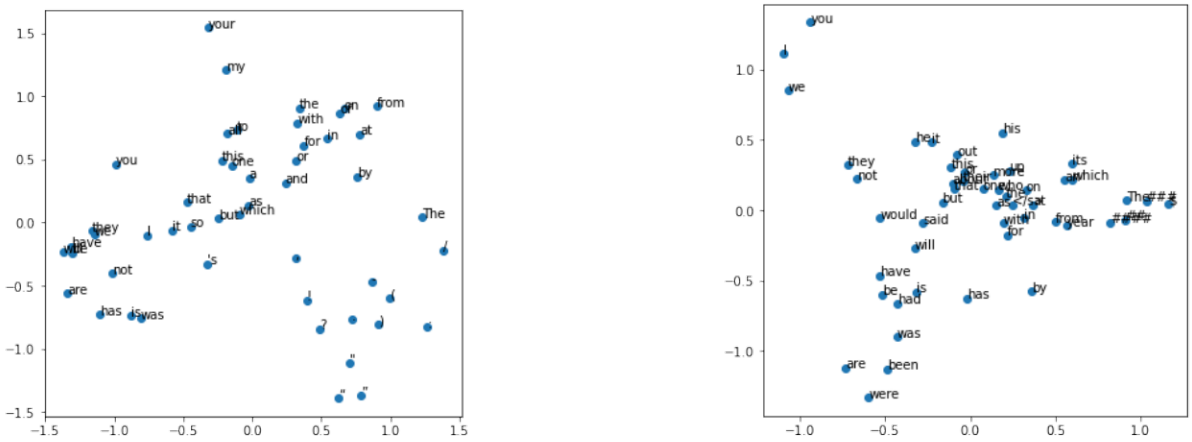

Figure 11: PCA on Fasttext and Google News

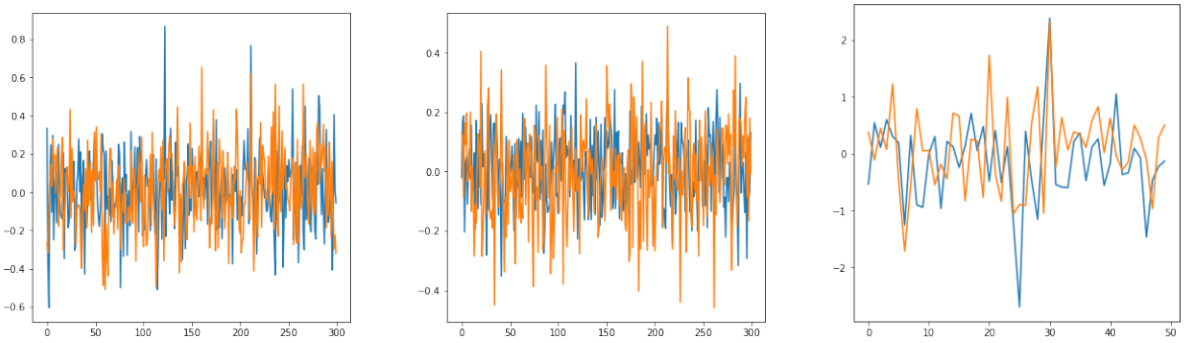

Figure 12: Cosine similarity between "former" and "store" on GloVe, Fasttext, and Google News

As dynamic embeddings, instead, I used BERT which allowed me to transform the dataset and reuse it for the next phases.
In my case, I have used the HuggingFace Transformer method [7] for extracting sentence embeddings, using only the output of the last layer of BERT.
The model I used is the *bert-base-uncased* (12-layer, 768-hidden, 12-heads, 110M parameters, trained on lower-cased English text).

3 Classifiers

In this part, several models will be presented, constructed, and compared. In particular, this section will be divided according to the vectorization method used, on which, Neural Network-based and non-Neural Network-based models have been applied.

3.1 Validation Schema

I decided to split the original training dataset into 85% for training purposes and 15% as the test set. To search for the best hyperparameters, I used the cross-validation technique with 3-fold, as a good trade-off between the computation time and the estimation of the error, for all the models computed in the next sections.

3.2 Models based on TF-IDF

3.2.1 Logistic Regression

As an optimization method, *l-bfgs* was used and grid search was performed with the hyperparameters in Table 1. Here, *penalty* is the type of regularization technique and C is a hyperparameter that specifies the strength of regularization.

Hyperparameters	**Values range**
Penalty	l2, None
C	0.001,0.01,0.1,1,10,100,1000

Table 1: Range of Hyperparameters for Logistic Regression

I get the following hyperparameters as a result: C: 100, Penalty: l2, which produce the following accuracy on the test set: 0.809, as we can check on the Confusion Matrix Figure 13.

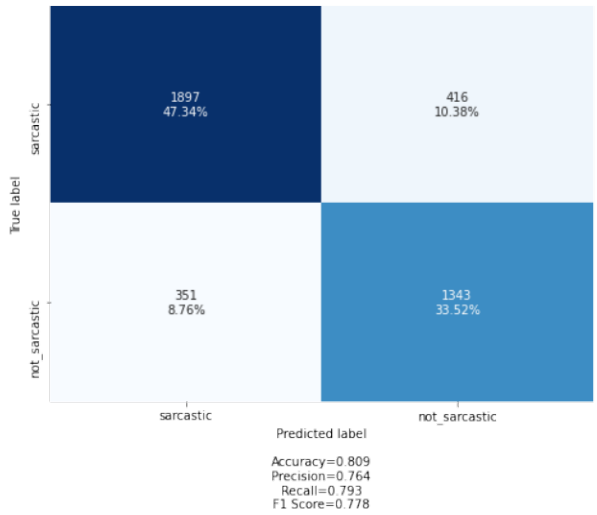

Figure 13: Logistic Regression - Confusion Matrix on TF-IDF

3.2.2 KNN

The grid search was performed with the hyperparameters in Table 2. Here, *Number of neighbors* is the number of neighbors to use by default for k-neighbors queries and *Weight* is an hyperparameter that specifies the weight function used in the prediction, it has the value *uniform* if all points in each neighborhood are weighted equally and the value *distance* if weight points by the inverse of their distance. In this case, closer neighbors of a query point will have a greater influence than neighbors which are further away.

Hyperparameters	**Values range**
Number of neighbors	1, 4, 7, 10, 13, 16, 19, 22, 25, 28
Weight	uniform, distance

Table 2: Range of Hyperparameters for KNN

I get the following hyperparameters as a result: Number of neighbors: 25, Weight: distance, which produces the following accuracy on the test set: 0.711, as we can check on the Confusion Matrix Figure 14.

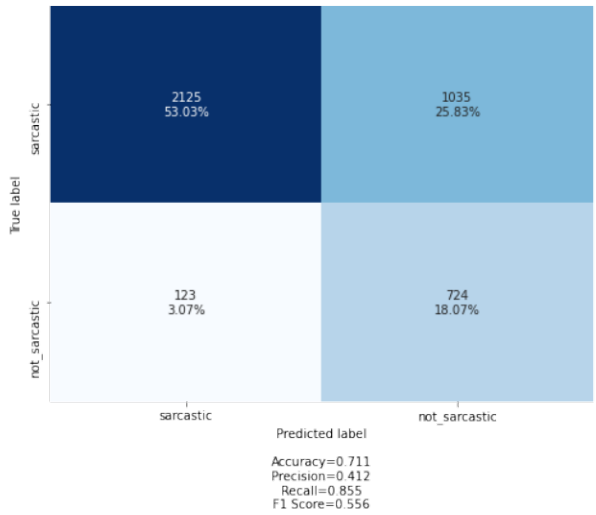

Figure 14: KNN - Confusion Matrix on TF-IDF

3.2.3 SVC

The grid search was performed with the hyperparameters in Table 3. Here, C is a regularization parameter and the strength of the regularization is inversely proportional to C. *Kernel* specifies the kernel type to be used in the algorithm. *Gamma* is a hyperparameter that specifies the kernel coefficient.

Hyperparameters	**Values range**
C	0.1, 1, 10, 100, 1000
Gamma	0.0001, 0.001, 0.01, 0.1, 1, 3, 5
Kernel	rbf, linear, poly, sigmoid

Table 3: Range of Hyperparameters for SVC

I get the following hyperparameters as a result: C: 10, Gamma: 0.1, kernel: rbf, which produce the following accuracy on the test set: 0.808, as we can check on the Confusion Matrix Figure 15.

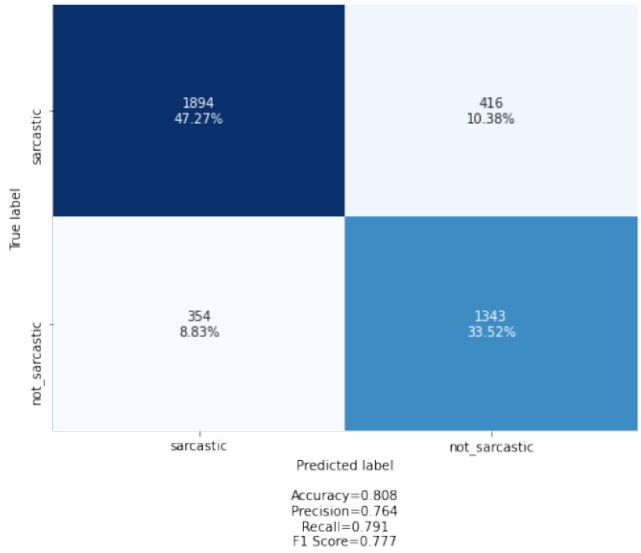

Figure 15: SVC - Confusion Matrix on TF-IDF

3.2.4 Multinomial Naive Bayes

The grid search was performed with the hyperparameters in Table 4. Here, *alpha* is the additive (Laplace/Lidstone) smoothing parameter.

Hyperparameters	**Values range**
alpha	1, 0.1, 0.01, 0.001, 0.0001, 0.00001

Table 4: Range of Hyperparameters for Multinomial Naive Bayes

I get the following hyperparameters as a result: alpha: 0.1, which produces the following accuracy on the test set: 0.792, as we can check on the Confusion Matrix Figure 16.

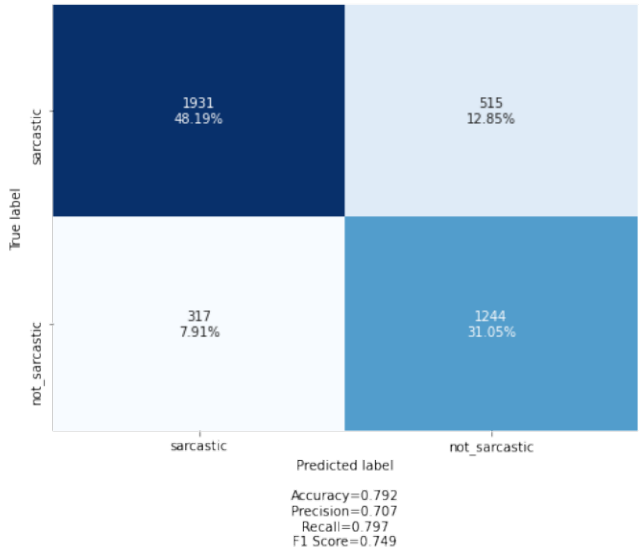

Figure 16: Multinomial Naive Bayes - Confusion Matrix on TF-IDF

3.2.5 Decision Tree

The grid search was performed with the hyperparameters in Table 5. Here, *Criterion* is the function to measure the quality of a split. *Max Depth* is the maximum depth of the tree. *Min Samples Split* is the minimum number of samples required to split an internal node. *Min Samples Leaf* is a hyperparameter that specifies the minimum number of samples required to be at a leaf node.

Hyperparameters	**Values range**
Criterion	gini, entropy
Max Depth	2, 5, 10, 15, None
Min Samples Split	2, 5, 10, 20
Min Samples Leaf	1, 5, 10, 20

Table 5: Range of Hyperparameters for Decision Tree

I get the following hyperparameters as a result: Criterion: entropy, Max Depth: None, Min Samples Split: 1, and Min Samples Leaf: 20, which produce the following accuracy on the test set: 0.714, as we can check on the Confusion Matrix Figure 17.

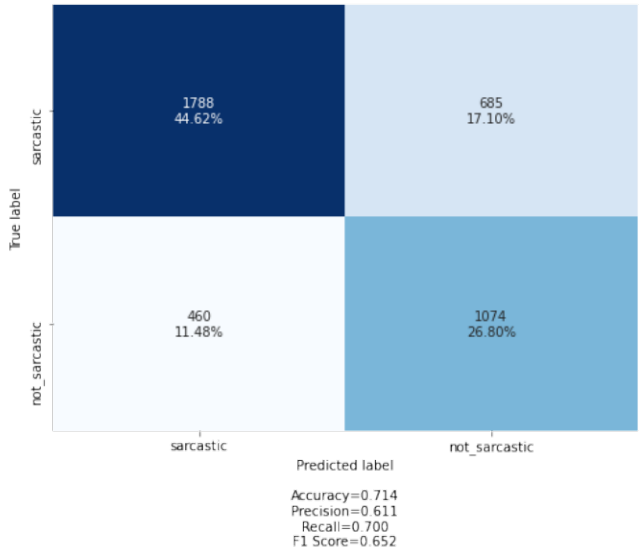

Figure 17: Decision Tree - Confusion Matrix on TF-IDF

3.2.6 Random Forest

The grid search was performed with the hyperparameters in Table 6. Here, *Criterion* is the function to measure the quality of a split. *Max Depth* is the maximum depth of the tree. *Number of Estimators* is the number of trees in the forest. *Bootstrap* a hyperparameter that checks whether to use the whole dataset for building the tree or not

Hyperparameters	**Values range**
Criterion	gini, entropy
Max Depth	2,3,5,10, None
Number of Estimators	25, 50, 100
Bootstrap	True, False

Table 6: Range of Hyperparameters for Random Forest

I get the following hyperparameters as a result: Criterion: entropy, Max Depth: None, Number of Estimators: 100, and Bootstrap: False, which produce the following accuracy on the test set: 0.755, as we can check on the Confusion Matrix Figure 18.

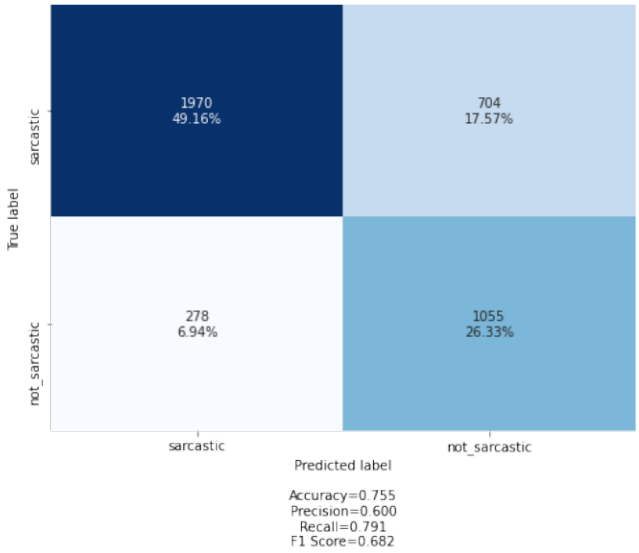

Figure 18: Random Forest - Confusion Matrix on TF-IDF

3.2.7 Conclusions

From the summary Table 7, we can see that the models seem to perform quite well. KNN, probably, suffers from the slight unbalance of the dataset and the trees are not a model (with the hyperparameters found) that seems to perform well.
We can be satisfied with the Multinomial Naive Bayes which achieves good performances that are exceeded only by SVC and Logistic Regression.
In conclusion, the performance of the Logistic Regression is comparable according to all metrics reported in the respective confusion matrices of the SVC but the former turns out to be much faster (concerning time in the training phase) than the latter.

Model	Accuracy
Logistic Regression	0.809
KNN	0.711
SVC	0.808
Multinomial Naive Bayes	0.792
Decision Tree	0.714
Random Forest	0.755

Table 7: Best Accuracy on Models based on TF-IDF

3.3 Models based on Static Embedding

In this subsection, I am going to use different static embeddings with different models, Neural Networks-based and non Neural Network-based.
In particular, Word2Vec embedding will be trained with the following parameters: *min_count*=0, *size*=500, *sample*=6e-5, *alpha*=0.03, *min_alpha*=0.0007, *negative*=20, using the *gensim library* [3] and I also use pre-trained embedding i.e. GloVe, Fasttext and Google News.
In all cases after applying the respective embedding on the words, a mean pooling is applied to create the sentence vector.

3.3.1 Logistic Regression

In this case, I apply the same previous Grid Search Table 1 on Word2Vec, GloVe, Google News, and Fasttext, and the best results can be found in Table 8.

Penalty	C	Type of Embeddings	Test set Accuracy
l2	0.001	W2V	0.561
l2	0.001	GloVe	0.561
l2	0.001	Google News	0.561
l2	0.001	Fasttext	0.561

Table 8: Best results with different Static Embeddings for Logistic Regression

3.3.2 KNN

In this case, I apply the same previous Grid Search Table 2 on Word2Vec, GloVe, Google News, and Fasttext, and the best results can be found in Table 9.

Number of Neighbours	Weights	Embedding Type	Test set Accuracy
28	uniform	W2V	0.534
28	uniform	GloVe	0.556
22	uniform	Google News	0.555
28	uniform	Fasttext	0.553

Table 9: Best results with different Static Embeddings for KNN

3.3.3 CNN

In this subsection, I decided to use an approach based on Neural Networks, in particular with CNNs both using the pre-trained GloVe embeddings and using the Embedding Layer. After a preliminary phase of tuning and architecture selection, I decided to fix the following architecture (Figure 19) and to launch a grid search, presented in Table 10, to search for the best hyperparameters. I used Adam as the optimizer, the ReLu as the activation function among the hidden layers and I set the epochs value to 20.

Hyperparameters	Values range
Batch Size	128, 256
Learning Rate	0.0001, 0.001, 0.01
Dropout 1	0.5
Dropout 2	0.5, 0.3
Dropout 3	0.3, 0.1
Kernel Size	26 (maxlen), 5, 3

Table 10: Range of Hyperparameters for CNN

Batch Size	Dropouts	Kernel Size	Learning Rate	Embedding Type	Accuracy
256	[0.5, 0.5, 0.3]	3	0.0001	Embedding Layer	0.688
256	[0.5, 0.5, 0.1]	3	0.0001	GloVe	0.695

Table 11: Best results with different Static Embedding for CNN

After getting the best hyperparameters on the respective embedding, I retrain everything with all the data (training set and validation set) obtaining the accuracy (on the test set) shown in Table 11.

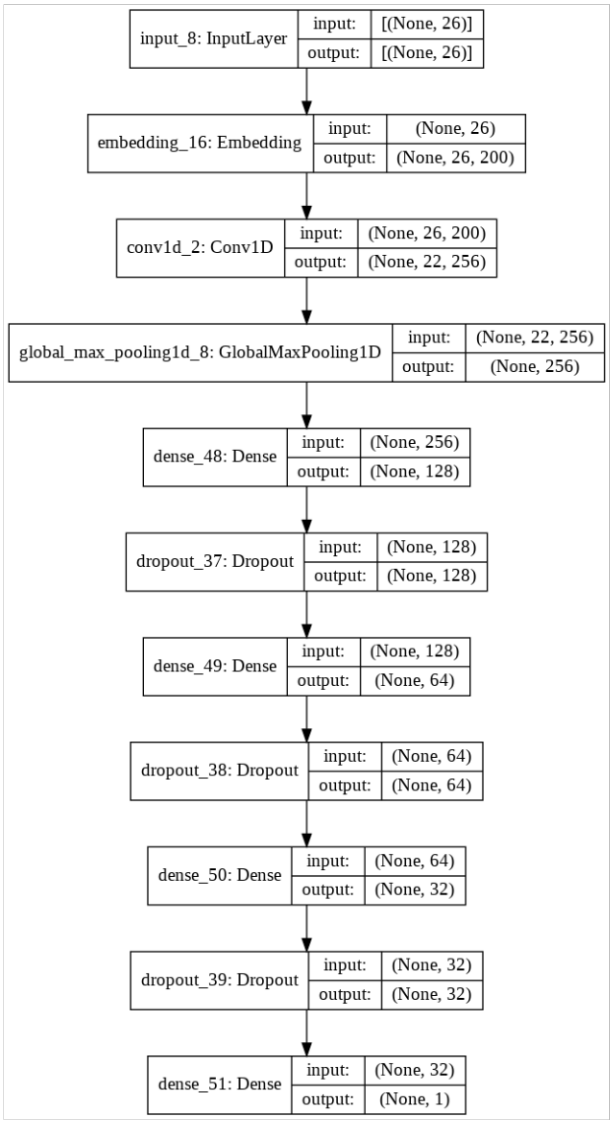

Figure 19: CNN Architecture

3.3.4 Bidirectional LSTM

In this subsection, I decided to use another approach based on Neural Networks, the Bidirectional LSTM. Also, in this case, I use the pre-trained GloVe embeddings and the Embedding Layer. After a preliminary phase of tuning and architecture selection, I decided to fix the following architecture (Figure 20) and to launch a grid search, presented in Table 12, to search for the best hyperparameters. I used Adam as the optimizer, the ReLu as the activation function among the hidden layers and I set the epochs value to 20.

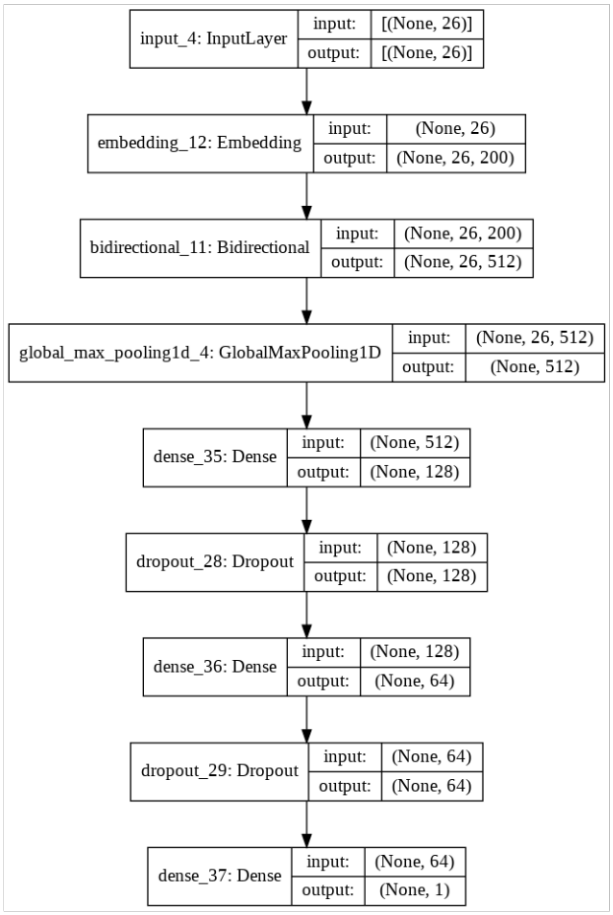

Figure 20: Bi-LSTM Architecture

Hyperparameters	**Values range**
Batch Size	128, 256
Learning Rate	0.001, 0.01, 0.1, 0.3
Dropout 1	0.5, 0.3
Dropout 1	0.3, 0.1
Kernel Size	26 (maxlen), 5, 3

Table 12: Range of Hyperparameters for Bi-LSTM

Batch Size	**Dropouts**	**Learning Rate**	**Embedding Type**	**Test set Accuracy**
256	[0.5,0.1]	0.001	Embedding Layer	0.681
256	[0.5, 0.1]	0.001	GloVe	0.675

Table 13: Best results with different Static Embedding for Bi-LSTM

After getting the best hyperparameters on the respective embedding I retrain everything with all the data (training set and validation set) obtaining the accuracy (on the test set) shown in Table 13.

3.3.5 Conclusions

As we can see, in Table 14, the models not based on Neural Networks perform very poorly with embedding, while those based on Neural Networks are certainly better but without having truly remarkable performance, as we could expect.
I could justify it by the fact that, in the case of the pre-trained embeddings, I am using the same lemmatized dataset and the pre-trained embeddings have not been trained on lemmatized datasets but the purpose of this paper is to understand the differences of different models on the same dataset, without trying to advantage one class of models over another, therefore I have decided never to use the non lemmatized dataset.

Model	Embedding Type	Accuracy
Logistic Regression	GloVe	0.561
KNN	GloVe	0.556
CNN	GloVe	0.695
Bi-LSTM	Embedding Layer	0.681

Table 14: Best Accuracy on Models based on Static Embedding

3.4 Models based on Dynamic Embedding

In this subsection, I have decided to use the generated BERT embedding only for the non Neural Network-based models, since the Neural Networks-based has proven to be particularly slow to train and since, given the previous results, it seems to be particularly difficult to tune the right set of hyperparameters.

3.4.1 BERT

In the case of BERT, for the sake of time, I simply retrain the best models found with the approach in subsection 3.2, therefore I obtain the following Table 15 which reports the results on the test set.

Model	Test set Accuracy
Logistic Regression	0.788
KNN	0.754
Naive Bayes	0.687
SVC	0.769
Decision Tree	0.629
Random Forest	0.740

Table 15: Accuracy on BERT

Again we can see, that as in subsubsection 3.2.7 the models that continue to perform best are Logistic Regression and SVC. Probably, performing a grid search or a deeper tuning of the hyperparameters could have exceeded the performance of the previous section.

4 Conclusions

In conclusion, several models have been tried to test as many vectorization methods as possible, to understand their relative influence.
Initially, I thought that I would have obtained better performance with the classical approaches based on Neural Networks since these are widely used.
In this specific case, I had to change my mind and after numerous efforts in tuning and launching grid searches, I had to accept that models based on a simple vectorization such as TF-IDF perform better.
Probably, with further efforts, it would be possible to overcome these performances but for now, it is noted that the best model found is the Logistic Regression based on TF-IDF.

References

[1] Xiaoou Ding et al. "Improve3c: Data cleaning on consistency and completeness with currency". In: *arXiv preprint arXiv:1808.00024* (2018).

[2] Fasttext. *Fasttext Library*. URL: https://fasttext.cc/.

[3] Gensim. *Gensim Library*. URL: https://radimrehurek.com/gensim/.

[4] GloVe. *GloVe Library*. URL: https://nlp.stanford.edu/projects/glove/.

[5] Google. *Google News Vectors*. URL: https://code.google.com/archive/p/word2vec/.

[6] Huffpost. *Huffpost Website*. URL: https://www.huffpost.com/.

[7] huggingface. *Hugging Face*. URL: https://huggingface.co/.

[8] TF-IDF. *TF-IDF Wikipedia*. URL: https://en.wikipedia.org/wiki/Tf%5C%E2%5C%80%5C%93idf.

[9] Rishabh Misra and Prahal Arora. "Sarcasm Detection using Hybrid Neural Network". In: *arXiv preprint arXiv:1908.07414* (2019).

[10] TheOnion. *TheOnion Website*. URL: https://www.theonion.com/.