Text Mining and Natural Language Processing

2022-2023

[Tito Nicola Drugman 502252, Angelica Pagni 505887, Antonio Simoncini 502215]

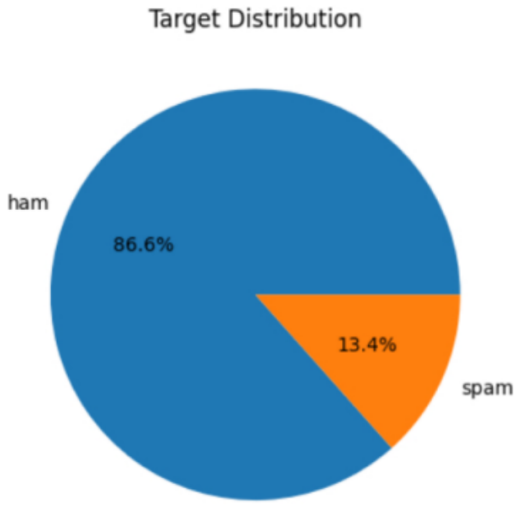
ADSM - (Almost) Detecting Spam Messages

Introduction

The task of our project is to create a machine learning classifier model able to binary predict whether a certain **sms message is spam or ham** (that is not-spam). Since it's a text classification with binary output, our efforts will be tailored towards elaborating the data in such a way that the two classes can be fairly distinguishable in the multi-dimensional vector space where the representations of each phrase will end up in. This implies finding a **suitable tokenization technique** and an **appropriate embedding strategy** for each resulting sequence of tokens. Every strategy we decide to use will be described in the sections below.

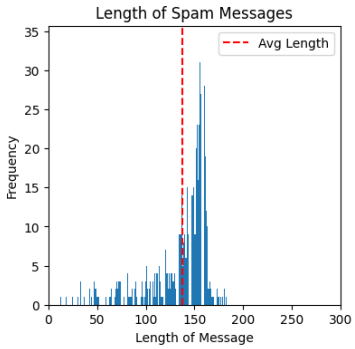
Data

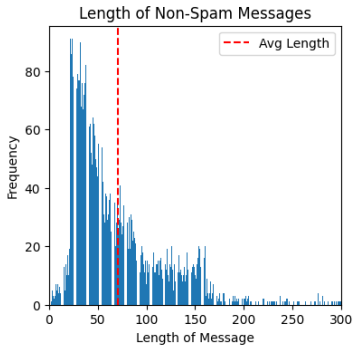
The dataset comes in .csv format and describes, in the second column, a set of **5572 sentences** (which are the text messages to be classified) along with their corresponding correct classification (spam or ham) in the first column.



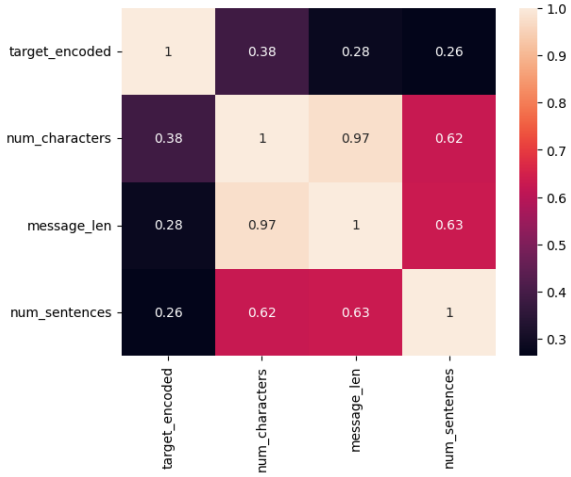
Regarding the distribution of the target labels, it's worth noting that the two are highly **unbalanced**. The total number of spam messages represents around 13% of the whole dataset, such a small percentage will make it **harder for the models to learn** the differences with efficient generalization of the distinction pattern. This is because a machine learning model tries to find an algorithm that allows to predict the target class given the features. If the data are unbalanced, the risk we encounter is that **the model will not be able** to achieve a good predicting score, since it will **quickly learn that predicting the most frequent class will lead to greater results**, at the price of a lower precision in the classification. To overcome this issue we decided to apply ***RandomOverSampler***, which duplicates some random elements from the minority class to make it less outnumbered. Even if it is well known that this technique sometimes may lead to overfitting in our case it turned out to be quite useful since it has slightly improved our results.

The first thing to notice regarding the **form and structure of the texts** is that since we are talking about text messages, things like grammar, special characters used and the structure itself of the phrases are different from the sentences that we could find in a book or anywhere else. We will handle this with a ***cleaning function*** and other steps where all of the unnecessary characters and elements of the sentences will be removed from the dataset.





The **mean length** of all the messages (both spam and non spam) turns out to be around 80 characters, while when we decided to study the length of the spam and non spam messages separately we found something quite interesting: the mean length of the spam messages was around 140 characters, while the mean length of the ham messages was around 70 characters. Because of this unexpected discovery, the use of ***pad_sequences*** function when vectorizing the sentences will be even more useful, because a smaller length of the message will be represented by a larger number of zeros at the end of the vector, helping with the recognition of this useful characteristic.



With the use of a heat map we can see the dependencies between the target and other variables such as target_encoded, num_characters, message_len and num_sentences. As we discussed above, the number of characters in a message is a useful information to predict whether a message is spam or not. Obviously, there are very high dependencies between the number of characters in a message and the length of it.

Methodology

Checking duplicates and cleaning

We checked if there were some duplicate messages and we found out there were some identical ones and so we decided to drop them. The cleaning function that we used transformed a text with:

- convert uppercase to lowercase, to avoid issues related to case sensitivity
- remove non-ASCII characters from the text. If we encounter non-ASCII character they will be replaced with an empty string
- remove the punctuation from the text

We tried to implement additional cleaning strategies but they did not improve the performance.

Removing short messages

Furthermore, we searched for messages that are made up of less than 3 characters (like 'ok', 'no', …), and therefore are not helpful to predict if a message is spam or not. Luckily, only few messages are contained in this category, suggesting that our dataset is well structured and filtered to the most useful data.

Stop words

An additional step we computed was to remove all the English stop words by using nltk. Since these stop words are only the one obtained from different corpus, we decided to remove also the most common ones used in the messages such as *k* which generally stands for *ok*, *u* that is *you* and others.

Stemming

Stemming is a process that allows us to take a word and reduce it to its word stem (base word), by doing these words like *connection, connections, connective, connected, and connecting* are all mapped to *connect*. To do so we used ***nltk.SnowballStemmer*** built by Martin Porter.

Splitting and RandomOverSampler

After finishing the preprocessing of our dataset we split it in the training and test set. Once completed the split we implemented *RandomOverSampler* to duplicate a portion of spam messages, contained in the training set, until they reach a 80% ratio with the ham messages, by doing this we were able to drastically decrease the gap in the target.

CountVectorizer

After splitting and applying *RandomOverSampler* we used ***CountVectorized*** on the x_train, which **converts a collection of text documents to a matrix of tokens counts**.

- The text was already transformed in lowercase, but as an additional safety measure, we decided to transform it again with CountVectorizer.
- As stop words we used the list we created (called *stop_words*) by merging it with the nltk list.
- As the lower and upper boundary of the range of n-values we used unigrams and bigrams.
- Furthermore, by using min_df and max_df we chose to ignore words which occur less frequently than 10% and more frequently than 70% of the text. These words could be like "*the*" that occurs in every document and does not provide any valuable information to our text classification and also "*ambidextrous*" which might be very rare.
- Finally, CountVectorizer will select the words/features/terms which occur the most, it will select the 100 most common words in the data.

TF-IDF

We decided to use ***Term-Frequency-Inverse-Document-Frequency*** (even though the small length of each "document" and moreover the total number of them actually make it computationally inefficient) such that we could capture the **relative importance of each word in spam documents** and retain the information on the most distinctive ones for the class to be

predicted.

Word Embedding: BPE + torch /GloVE

First of all we want to point out that we use two different dictionaries to realize the vector embeddings of the words: the first one is resulted from a ***BPE personalized tokenization*** (coupled with random PyTorch vector representation for each token) and the second one is the ***standard GloVe reference txt file***.

Regarding the personalized **BPE**: on the dataset website we found out that the dataset we are working with was actually a subset of a bigger dataset containing more than 55 thousands messages kite1988/nus-sms-corpus and so we decided to use this massive dataset to train a Byte Pair Encoding tokenizer (with the use of a code found online on Github.com) which would be adapted to the specific kind of text we're analyzing and strictly personalized to our goals. This allowed us to model into our tokenization the very own grammatical structure, typos and abbreviations found in the messages themselves. After training the BPE we then created the tokenization of a cleaned version of spam.csv. In our opinion it seems logical to use the **same cleaning strategy** for both the *nus sms corpus* (used to train the BPE) and the spam dataset.

Once cleaned, we trained the BPE to effectively extract the best tokens based on the frequency that single characters appear together, and memorized the results in a .txt file. Initially we were using a dictionary to store all the different tokens and their representations, but then we realized that the most efficient embedding representation would result in implementing the tokenization similar to the one of the GloVe model. So we stored the tokens in a .txt file that has the identical structure of the GloVe file with a random representation of each token obtained with PyTorch.

GloVe actually comes with its own .txt encoding scheme file, so we compared it with the BPE file to see if our idea was actually making any difference. We used a dataset that contains English word vectors pre-trained on the combined Wikipedia 2014 + Gigaword 5th Edition corpora (6B tokens, 400K vocab).

Models

Our “competition” involved three candidates. We decided to use XGBoost, LSTM and BERT.

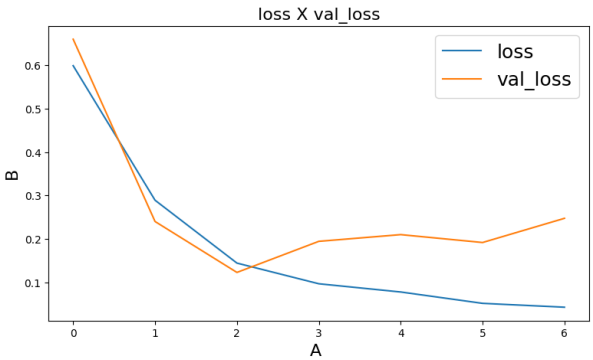
XGBoost

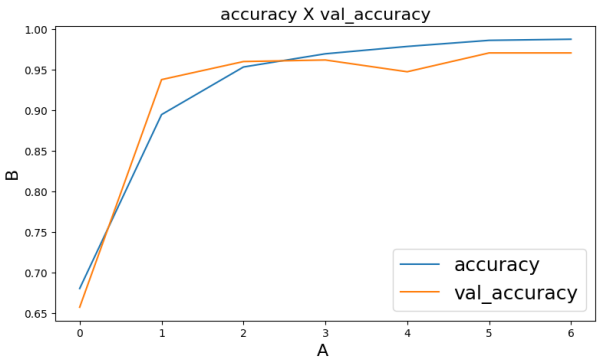
Regarding the choices of the classifiers, for the first model we used XGBoost, which is a version of gradient boosted decision tree classifier. We thought it was a good choice since it has the ability to handle high-dimensional, sparse data. XGBoost also provides feature importance analysis, regularization techniques to prevent overfitting, and scalability for processing large volumes of SMS messages. It’s very effective in handling imbalanced datasets.

The pipeline of XGBoost contains CountVectorizer and TF-IDF. The results that we get are discussed later.

LSTM

The second strategy we used is LSTM implemented in a recurrent neural network. We tried to use this approach since LSTM is designed to handle sequential data such as sentences. If the dataset is unbalanced, meaning there is a significant class imbalance between spam and ham messages, the LSTM may have a bias towards the majority class. To overcome this issue we applied RandomOverSampler and it was interesting to notice that the confusion matrix actually was worse for LSTM with respect to XGBoost.



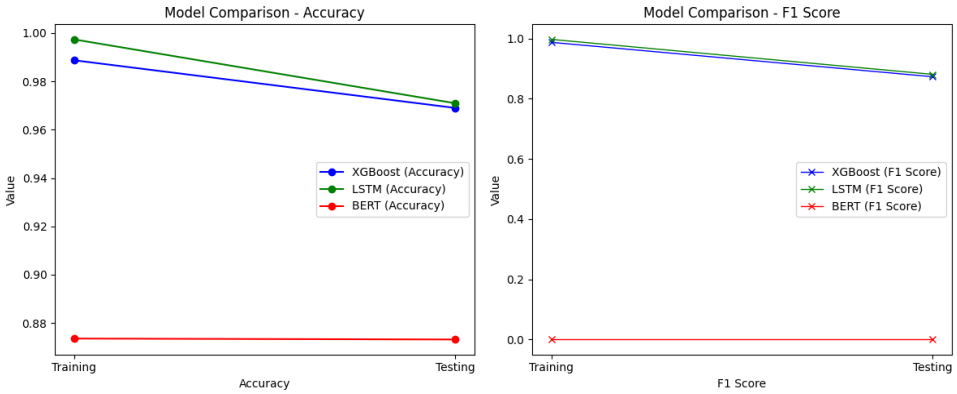


BERT

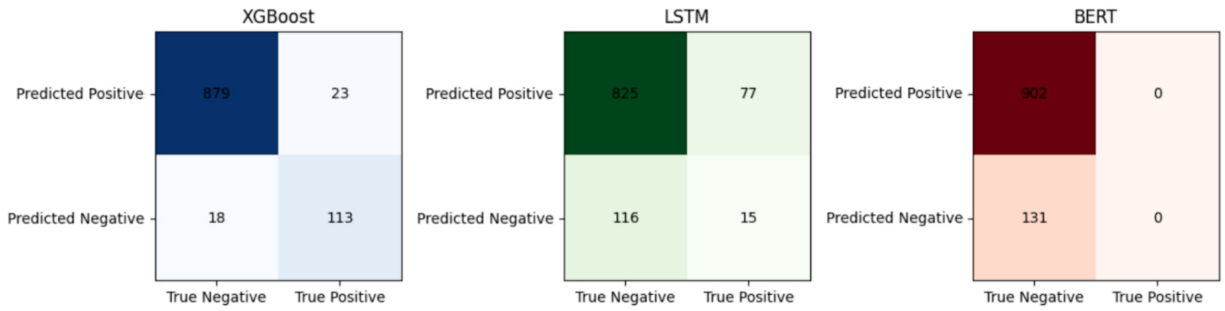
The last, but maybe also least, model we decided to implement was BERT. BERT, known as Bidirectional Encoder Representations from Transformers, is a highly advanced language model used in SMS spam classification. BERT can effectively capture patterns and improve classification accuracy. We had very high expectations for this model, but sadly it was not very efficient.

Result

With BPE we have the following results:



With the BPE, XGboost and LSTM performed quite well on the training accuracy and, as we could expect, we had a lower accuracy on the test set. On the other hand, BERT seems to perform almost the same on the train and test set. We can see that when we compared the different models with respect to the F1 score XGBoost and LSTM are almost identical, while, sadly for BERT, his performances measured on accuracy and F1 score are very low. The comparison between these two graphs may seem misleading: if we observe the second chart even if BERT looks the same, it is not anymore around 0.88, but rather at 0.



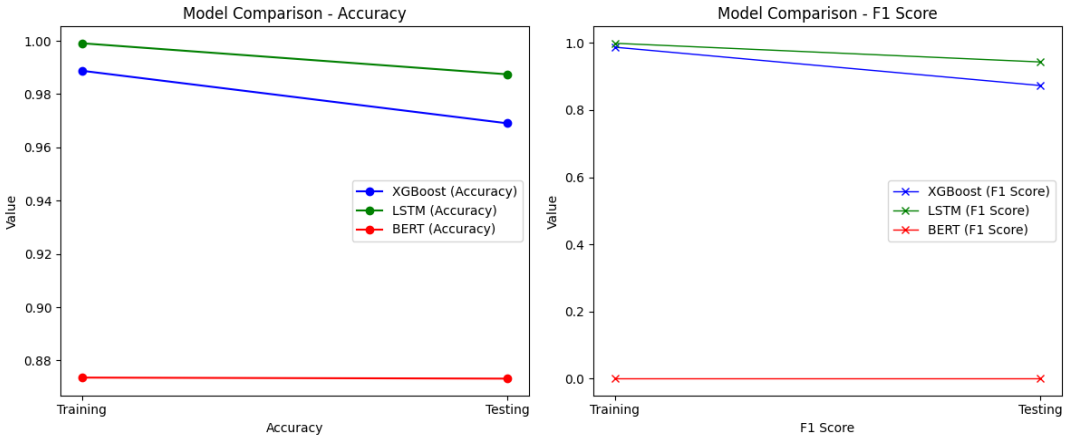
(Confusion matrix with BPE)

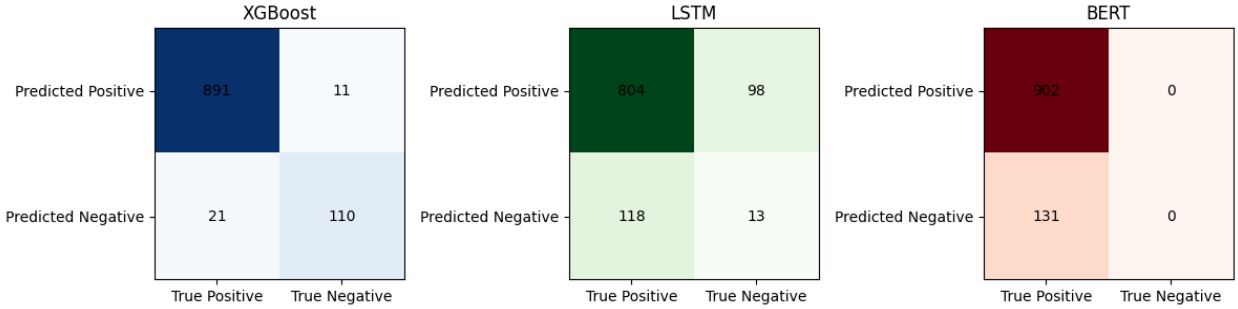
If we analyze the confusion matrix we can see that XGboost seems by far the best model. It correctly classified almost the total number of messages. On the other hand, we can see an unbalanced situation in LSTM and BERT: in both confusion matrices there are more elements on the first column rather than the second one, this might suggest that the model is more biased towards predicting the positive class.

Model	buy a new toy and get one for free	only today	win a free vacation	limited time offer	free free free get free	my best friend is marco	should I visit venice?
XGBoost	ham	ham	ham	ham	ham	ham	ham
LSTM	ham	ham	ham	ham	spam	ham	ham
BERT	ham	ham	ham	ham	ham	ham	ham

We also decided to test our models. To do so we added the possibility to take user generated sentences and to see the results obtained from our models. The results, as we could expect by looking at the confusion matrix, are not so great. All the sentences are considered by all the models as ham. This result is even worse if we remember that *free* and *get* are respectively the second and thirteenth most used words in all the spam messages available in our dataset.

On the other hand with GloVe the results are the following:





(Confusion matrix with Glove)

Model	buy a new toy and get one for free	only today	win a free vacation	limited time offer	free free free get free	my best friend is marco	should I visit venice?
XGBoost	ham	ham	ham	ham	ham	ham	ham
LSTM	ham	ham	ham	ham	ham	ham	ham
BERT	ham	ham	ham	ham	ham	ham	ham

Even with GloVe, which was pre-trained on the combined Wikipedia 2014 + Gigaword 5th Edition corpora (6B tokens, 400K vocab) the results are not particularly exciting. The accuracy and F1 score are even worse than the one obtained by BPE, when we compare the models by using F1 score we can see a deeper drop. Also the confusion matrices with the two tokenization are not that good, XGBoost, LSTM and BERT almost stayed the same between the two embedding methods. We can see that even if GloVe was trained on a more massive corpus it was probably more general and less specific to our task, so the BPE we trained seemed to be more efficient.

Conclusion

Issues encountered

One of the first issues we encountered was related to the dataset used for training the BPE. The file was in SQL format, and none of us knew how to properly manage it. We tried different strategies, and only at the end we were able to convert it into a more usable text format.

Another issue we faced was the performance of Google Colab. After running it multiple times a day, we discovered that there is a limited amount of GPU resources available to each user. This posed a significant challenge as it slowed down our project. However, we eventually found a simple workaround. We shared the notebook with an additional personal Gmail account and added a shortcut to the drive folder containing the "spam.csv" file. By doing this, we were able to access almost unlimited GPU power. Additionally, we utilized the developer tools of the Google Colab webpage to create a small JavaScript program that automated clicks on the page, preventing the notebook from shutting down.

At first we tried to implement the SMOTE to decrease the gap between ham and spam messages but then we decided that the best option was to use RandomOverSampler. An issue that we encountered was that the output was a panda dataframe while it should be a panda series. Luckily we find out that we can use .squeeze() to complete the comparison.

Project recap

We could reassume our project as follows:

Part 0

Transforming the kite1988/nus-sms-corpus in a file suitable for training the BPE. Save the result in a txt.

Part 1

Importing, loading the dataset.

Part 2

Preprocessing, check the distribution of spam and ham, compute the number of characters, length of the message and number of sentences in each message. Then we made the text lowercase, removing punctuation and numbers. We also removed duplicate sentences and short ones. Then we checked the most frequent words in the spam and ham messages.

Part 3

We splitted between train and test and we used RandomOverSampler and created a document-term matrix from the train thanks to CountVectorizer. We then compute the TF-IDF.

Part 4

Use the BPE and embedding obtained from another notebook to create the embedding of our dataset. If we would like to use GloVe instead, we should just comment the first part and change the embedding size to 100.

Part 5

Create the models XGBoost, LSTM and BERT and compute the accuracy score, F1 score and confusion matrix of each model.

Part 6

We compare the obtained results of the three different models with the help of some graphs.

Part 7

We let the user create new sentences and check if they are spam or not.

Final conclusion

To conclude this report we would like to point out that this was the most challenging project among all of our courses. We also relied on some suggestions that we received from our classmates and online. In the end it was very useful to try to implement what we learned during this course, even if the results are not so great. We had great expectations, but we believe that maybe we should have lowered what we could get from our first NLP project.

We would have liked to add more comments on the code since we are aware that they are very helpful to understand more clearly the different steps of the process, but we decided to do the whole code together from the beginning by using the same pc and so we thought it was needless because all of us knew what we were already doing.

Sadly it is not possible to define who did what, since we started and finished this project all together. Most of the time we developed the code from the same computer and we shared our ideas and put them together in order to complete this project as a team work.

Sources

- Spam dataset, available on Kaggle.com.
- The nus dataset that sadly is not available from the main website but luckily we found it on Github.com.
- The BPE created by Shahad Mahmud available on Github.com.
- We were inspired by Marie and her amazing notebook available on Kaggle.com.