TEXT MINING

PREDICTING AIRBNB UNLISTING

André Cunha 20191224
Gonçalo Aires 20220645
Pedro Ferreira 20220589
Quintino Fernandes 20220634

1. INTRODUCTION

The goal of this project was to use a dataset with airbnb listings - their description, host information and reviews - to predict whether or not a specific listing was going to be unlisted or not in the next quarter. So, we wanted to use text data to perform binary classification: 1 if the airbnb is unlisted, 0 if not.

2. DATA EXPLORATION

For exploration, we started by lightly exploring the two datasets (listings and reviews) separately and then joining them into one single dataset.

2.1. Labeled Listings

Upon initial analysis, we observed a significant class imbalance between the two target values. The "0" class, representing active listings, comprised 9,033 instances, while the "1" class, indicating unlisted listings, contained only 3,463 instances. This evident class imbalance indicated that there were considerably fewer listings being unlisted, representing less than half (approximately a third) of the total compared to the number of listings that maintained their presence on the website.
We conducted an examination for missing values within the dataset. Fortunately, it revealed no instances of missing values, indicating that all required fields were populated for each listing in the dataset - but we were aware that an empty string would not be counted as a null.
In the initial analysis of the **description** column, we performed a word count description. Our findings revealed that the average number of words per description was approximately 133, with no description exceeding 210 words. We generated a histogram and a boxplot, which are presented in Figure 1 and 2. Finally, we checked the top 10 most used words which can be seen in Figure 3.

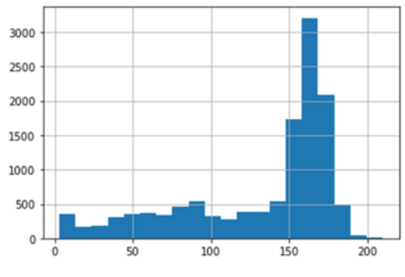


Figure 1

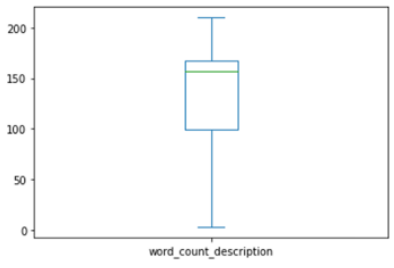


Figure 2

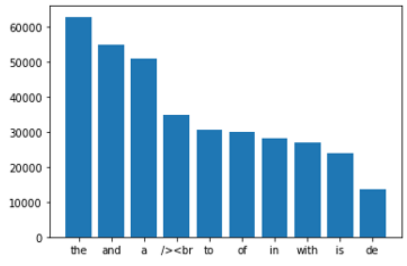


Figure 3

Then in the column **host_about**, we also conducted a word count assessment. The results indicated that the average number of words per host_about entry was approximately 73. Notably, we encountered a particular challenge with a subset of rows, approximately 67 instances, where the word count exceeded 512 words, which would not be good for some word embeddings later on this project (Figure 4 and 5). Finally, we also checked the top 10 most used words which can be seen in Figure 6.

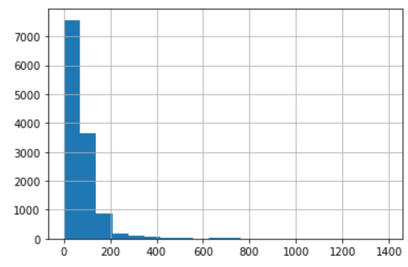


Figure 4

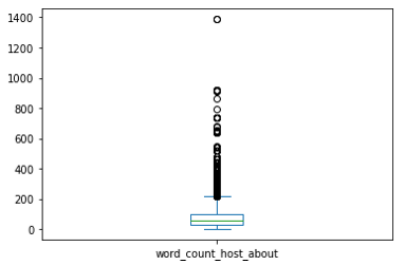


Figure 5

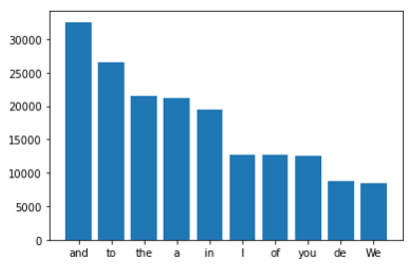


Figure 6

2.2. Reviews

In the reviews dataset, our primary focus was on the **comments** column. To ensure data completeness and quality, we examined the presence of null values, and fortunately, no null values were found in this column as well. Subsequently, we conducted a word count analysis on the comments. The results revealed that the average comment consisted of approximately 48 words. However, we found approximately 236 rows where the word count exceeded the 512-word limit (see Figure 7 and 8). Finally, we also checked the top 10 most used words which can be seen in Figure 9.

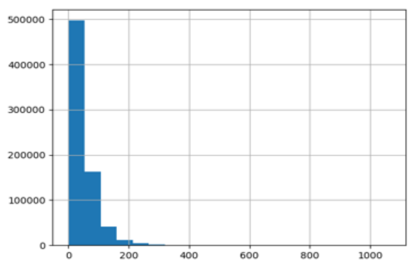


Figure 7

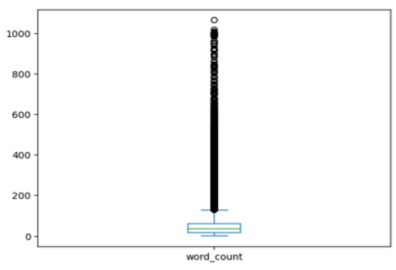


Figure 8

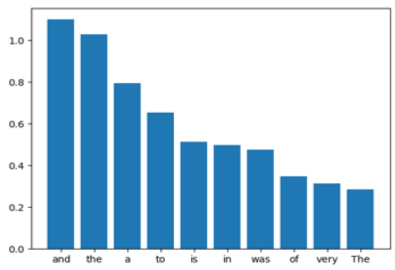


Figure 9

3. DATA PREPROCESSING

3.1. Train/Val/Test Split

The corpus had 12496 rows after joining all the different datasets, so we could safely do a train, validation and test split. Each one of these rows contained three columns: **description** with the listing's description, **host_about** with the host's description and **comments**, which had a list of every review of that listing, that was done like this since we did not want rows with repeated **description** and **host_about** columns (which we would need to have if we wanted to have a review per row), as we thought that could bias our model towards not caring as much about reviews or increase overfitting because of that repetition. As the corpus wasn't very big, we chose the split percentages of **80%/10%/10%**. We used the training and validation data to train and improve the models and the last 10% to assess the quality of the model in unseen data. In the end we used the provided unlabeled test set to do the final predictions we were asked to deliver.

3.2. Language Detection

Our dataset had a multitude of different languages, specially in the comments column, so for the preprocessing to be meaningful we needed to know what language was in each text so we could

preprocess each language individually, particularly in the steps of removing stopwords and doing lemmatization.
For this we detected individually the language of each column (**description**, **host_about** and **comments**) and the lists contained in them. We used the *langdetect* library and this gave a country/language code like “en” for english, that we would append to each row as a new column for each of the other 3 columns, for example: for the **description** column we added a **description_lang** column for the language labels (analogous for the other two features).
The language detected in the **description** and **host_about** columns was mostly either english or portuguese, but as for the **comments** columns we had multiple reviews per listing and consequently multiple languages, so this process assigned a language label to each individual review and stored them in the new column as a list for the same review, in the same order as the reviews themselves, which helped us later access these languages.

3.3. Text Cleaning

For text cleaning we applied the techniques we learned in class as well as two more we thought that made sense in our case:

- **Lowercasing:** The first step was to convert all text to lowercase. This standardized the text and ensured consistent processing.
- **Removing HTML Tags:** Any HTML tags presented in the text were removed using regular expressions (re.sub('<.*?>', "", text)). This step eliminated any HTML elements, including comments and other tags.
- **Removing Numerical and Punctuation Data:** All numerical and punctuation characters were removed from the text using regular expressions (re.sub("[^A-zÀ-ú", " ", text)). This step ensured that only alphabetic characters remained. Furthermore, it kept every letter with an accent, as that would prove to be important later, because we tried with a normal alphabetical regular expression and it did mess up text in non-english languages.
- **Removing Emojis** (<u>extra method</u>)**:** Emojis were removed from the text using a regex pattern (emoji_pattern.sub(r'', text)). This ensured that emojis do not interfere with subsequent processing or analysis. The pattern itself involves emoticons, emojis and system flags.
- **Removing Stopwords:** Stopwords, which are commonly used words that do not carry significant meaning (e.g., "the," "is," "and"), were removed from the text. If the language was not recognized, English stopwords were used as a fallback.
- **Lemmatization:** Lemmatization was performed to reduce words to their base or dictionary form (we applied lemmatization to the text data in 14 different languages, which covers a substantial portion of our dataset). This helps consolidate words with the same meaning.
- **Stemming:** Stemming was an alternative to lemmatization that reduced words to their root or stem form. This would help in further normalizing the text. Stemming is performed based on language availability; if the language was not supported, English stemming would be used.*
- **Removing Rare Words** (<u>extra method</u>)**:** Words that appeared only once in the entire dataset were considered rare. These words are unlikely to contribute meaningful information and may add noise to the analysis, so we decided to remove them.

By following these steps, the code achieves a comprehensive cleaning and preprocessing of the text data, making it more suitable for subsequent tasks such as natural language processing and machine learning models.

***Note:** We figured we would use lemmatization, since stemming alters the original form of words (potentially cutting them) and, particularly for the GloVe embedding (which is based on word co-occurrence statistics and capture semantic relationships between words), consequently disrupts the semantic relationships captured by the embeddings. Furthermore, as we used pre-trained embeddings, we needed to make sure the words in our text would match the ones in the embedding and stemmed words did not, most of the time.

Extra work: Removing Emojis, Removing Rare Words.

4. FEATURE ENGINEERING

For feature engineering, we decided to try four different ways of word embedding. Our initial goal was to use them all inside models to compare which ones worked best.

4.1. Get the dataset ready for word embedding

To optimize our data preprocessing workflow, we decided to save the processed dataset as a CSV file. This allowed us to avoid repeatedly running the preprocessing steps. However, since the data was stored as lists in the CSV file, we needed to convert the columns back to strings. Additionally, we handled missing values by replacing them with empty strings. After this, to simplify further analysis, the individual comments within the comments column were combined into a single text by joining them with spaces.
Considering that our word embedding models could have a maximum input limit, typically around 512 tokens, we aimed to divide each string into substrings containing approximately 510 words. This ensured that each substring remained within the acceptable size limit for word embedding purposes.

4.2. Word Embeddings

4.2.1. TF-IDF

We started by trying TF-IDF, as it would be the simplest one - this does not mean it is the most efficient, though - while knowing about its limitations. Since it uses word counts to build the vectors, we decided to join each row's features into one big string, as it would not cover word order anyways and at maximum would only get the local context of each word, to feed the tf-idf vectorizer. Contrary to what would be expected, the fact that it does not care about context or keep the semantic relationship between words that are very far apart in a sentence - especially as we did not use n-grams - was not our biggest problem. Because this embedding is based on bag of words, each one word is transformed into a feature, whose value will be weighted according to its importance to the text, so we got around 80000 features. Our thought process was that we could simply apply dimensionality reduction techniques, like PCA, SVD or t-SNE, to make this dataset less sparse and more usable in the models, but our computational resources did not allow us so, because even in Google Colab the RAM limitations hindered this process. So, we would have to use the non-reduced dataset in our models, which we ended up not doing, as there are way too many features to get a good model without falling to the curse of dimensionality. Because of all this, and because we knew we would use more advanced techniques for word embedding, in spite of implementing tf-idf on our dataset, we did not use it in our models after this analysis.

4.2.2. GloVe Embeddings 6B 300

Then we went ahead to try glove embeddings. As was mentioned earlier, we had a dataset with multiple languages, which meant that our word embedders would have to be multilingual, so that is exactly what we used. This specific glove embedding model is pre-trained in multiple languages. This is good not only because we have an embedding usable for our dataset, but also one that is already trained with hundreds of thousands of words, which helped our process be more efficient. The fact that we chose lemmatization also helped, as we managed to get matches for the words on our text, therefore could embed our dataset. So, we got embeddings with size 50 for each feature by matching our data to the loaded embedding. Then, we decided to concatenate the three obtained vectors for each row to get a final embedding with 150 dimensions.

4.2.3. STSB-XLM-R-Multilingual (extra method)

We jumped to transformer-based word embeddings. This one is actually based on RoBERTa and was fine tuned to work with multiple languages and to capture semantic relationships within the text across languages. Just like glove, the one we used was already pre-trained, as this proved to be much more efficient. This one embeds the text into a 768 dimensions vector. To do so, we provided the model with the previously divided strings, getting some embeddings per feature and averaging their coordinates. Because in the end we wanted to get one vector per row, we also averaged the three vectors. In the end, we got a dataset with 768 columns, one for each vector coordinate.

4.2.4. DistilUSE base multilingual-cased v2 (extra method)

The procedure to use this word embedding was very similar to the previous one. The differences between them are all under the hood, as the purposes for which they were built are different. While the previous one - let’s call it XLM - is concerned with semantic similarity, this one - distilUSE - is more general purpose and supposedly more versatile, while it being a distilled version of USE (Universal Sentence Encoder) makes it smaller and more efficient than XLM. The main difference we can see between them, however, is the final number of features. This one, doing the same averaging and concatenation of vectors we did in the last section, provided us a dataset with 512 features.

Extra work: STSB-XLM-R-Multilingual, DistilUSE base multilingual-cased v2.

5. CLASSIFICATION MODELS AND EVALUATION

As we mentioned earlier, the data was split very early into train, validation and test split, in order to keep data leakage away from us as much as we could. Because of this, all preprocessing and feature engineering were done using the training set only. So, we created auxiliary functions to implement the preprocessing and feature engineering in any given dataset, allowing us to prepare the validation and test datasets for Modelling. For tuning, we used small random searches as we needed a time efficient approach to hyperparameter tuning, while being aware that meant we would probably not extract the best possible model. To test the and compare the models, we used precision, recall, accuracy and f1 score and our goal was to get the best embedding for each basic model first, to then tune it to get the best model we could. This last model would go into the next phase, where we compared the best models we got to choose the final one.

5.1. K-Nearest Neighbors

5.1.1. Embedding Testing and Hyperparameter Tuning

We decided to do both the embedding selection and the hyperparameter tuning at the same time, as KNN is a very simple model so it would not be so time consuming. For this we did a Random Search for each of the embedding methods. For the Random Search we decided to run it for 10 iterations with 3 folds each, testing the following parameters of the classifier: n_neighbors, weights, algorithm and metric; for the scoring method we chose "F1-Score", as it is a good alternative for imbalance datasets. At the end we had 3 best models, one for each of the embedding methods.

5.1.2. Best Models

The best embedding method was **Glove**, it achieved the best training score of 0.99 and the best validation score of 0.79. **XLM** had a good training score of 0.98, but overfitted massively achieving only a score of 0.29 on the validation score. **DistilUSE** had a poor performance on both the training data and the validation data.

5.1.3. Best Model Testing

So, as we did both the selection of the embedding and the hyperparameter tuning we already had the best parameters for the **Glove** embedding, which were: n_neighbors = 12, weights = distance, metric = manhattan and algorithm = brute. We decided to predict on unseen test data, where it achieved a training score of 1, but a test score of 0.89, meaning the model was overfitting.

5.2. Logistic Regression

5.2.1. Model Results

For the Logistic Regression (LR), we started by testing the standard model performance with our 3 different embeddings, on both the training and validation datasets. Glove embeddings seem to have performed the best in all metrics (precision, recall, accuracy and f1 score), as the other two overfitted a lot and got around 0.6 to 0.7 when it comes to validation metrics. So, we proceeded to use glove embeddings with our model.

5.2.2. Hyperparameter Tuning of Best Model

We used Random Search with 5 folds, running 30 different combinations of parameters. We achieved a best score of 0.79 and the best f1 score with the following parameters: C = 0.4881529393791153, penalty = l1, solver = saga.

5.2.3. Best Model Testing

Finally, we tested this last model with our testing set, achieving results of 0.88 in precision and f1 score and 0.87 of accuracy and recall.

5.3. Multi-Layer Perceptron

5.3.1. Models' Results

For the Multi-Layer Perceptron (MLP), we followed a different approach compared to KNN, similar to what we did with the Logistic Regression. Here we started by doing an evaluation of the MLP classifier's performance (with hidden layer sizes of (50, 50, 50) and a maximum of 1000 iterations)

with our 3 different embeddings on both the training and validation datasets without doing the hyperparameter tuning simultaneously.
The MLP model with glove embedding showed the best overall performance, with high accuracy and balanced precision and recall scores on both the training and validation datasets. XLM embedding had a significant drop in performance on the validation set, suggesting overfitting. Distiluse embedding yielded decent results, but its performance was slightly inferior to the Glove embedding.

5.3.2. Hyperparameter Tuning of Best Model

After evaluating our results, we moved on to finding the best combination of hyperparameters for the classifier, optimizing the F1 score as the evaluation metric. For our best model, we conducted a Random Search with 10 iterations and 3-fold cross-validation.
The best combination of hyperparameters achieved a lower F1-score compared to the initial model and a best score of 0.79. This suggests that the initial model performed better in terms of F1-score, indicating that the hyperparameter tuning did not improve the model's performance in this particular evaluation metric. This can happen as we use a random search so there is not an extensive search in the parameter space, allowing for the results to be worse than the standard model.

5.3.3. Best Model Testing

Finally, we compared the results with our test data. Our best MLP classifier with tuned hyperparameters showed promising results in classifying whether or not Airbnb listings will stay active in the next year. On the training set it achieved an F1-score of 0.92 for the active listing class and an F1-score of 0.80 for the inactive listing class. On the test set, the model achieved an F1-score of 0.91 for the active listing class (0) and an F1-score of 0.78 for the inactive listing class (1), this discrepancy being likely caused by the imbalance of the dataset. These F1-scores demonstrated a good balance between precision and recall for both classes, indicating the model's ability to accurately do the classification.

5.4. Additional Models

We also tried to implement two different pre-trained Transformer models, the DistilBERT and the XLM. These models are already pre-trained in huge Datasets so we only need to fine-tune them to our Dataset. Also these models contain a Tokenization method and a word embedding layer so only the Preprocessed Dataset is needed. We trained the models using pytorch methods on only 3 epochs each (due to lack of runtime in colab), evaluated the training process on the validation data set using the metrics described before and evaluated the model on the test Dataset. While trying to implement it we faced several problems with lack of RAM in the GPU, lack of runtime in colab and some packages errors. The DistilBERT method ended up only predicting 0's and the XLM method always stopped training due to lack of memory. Due to all these drawbacks and errors we decided to not consider them in the final model choice and to not insert the code in the final notebook. Maybe with some more time, more research and some more compute resources we could implement these huge pre-trained models.

6. MODEL CHOICE

After trying all our models with the different embeddings, we realized that glove embeddings seemed to perform better in all three. So, we ran the models on a glove embedded train and test dataset and got their precision, recall, accuracy and f1-score (see figures 1 to 6 in the annexes) - as we did before. Out of all of them, KNN worked best, getting 0.89 scores all around, in spite of overfitting to the data, as it got 1 in all metrics for the training set. The other ones, MLP and Logistic Regression, did not overfit as much, but also got worse metrics overall. LR got 0.87 for recall and accuracy and 0.88 for precision and f1-score, while MLP achieved 0.87 for all metrics. The difference is not that big between models, but we chose KNN as it was objectively the one which got the better scores.

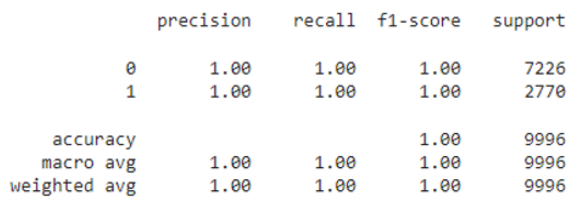
So, our final model was K-Nearest Neighbors using a glove embedded dataset, with the following hyperparameters: n_neighbors=12, weights="distance", metric="manhattan", algorithm="brute". Furthermore, we did a section with all the required steps to process the train and unlabeled data, so we could run everything a last time to train the model with all the available data and do the predictions, which were exported to a csv file. Because we did the evaluation step, we expect the results of the predictions to be satisfactory for the final unlabeled dataset.

7. CONCLUSION AND FUTURE WORK

We ended up with a pretty good performance, considering our model is one of the simplest ones out there. Trusting our test scores, it should predict around 89% of data correctly, but it may vary a little as we re-trained it with all available training data to do the final predictions, which could increase overfitting a little but also provide insightful information to our model - but it is standard practice to do so. But all of this means there is a lot of room for improvement. There could certainly be more steps inside the preprocessing of the model, as adding more and good options could help the model's performance (one could do spell checking, for example). Adding to that, there are probably better word embeddings than the ones used. Surprisingly, we ended up using glove and not the ones that are supposedly more sophisticated. This could be due to glove returning less features than the other ones, suffering from the curse of dimensionality and overfitting. Maybe using dimensionality reduction techniques on top of all of them would be a good idea, but just as we saw with tf-idf earlier, the computational requirements would be greater. We could also use the pre-trained models but unfreeze them to adapt it to our specific dataset, specifically when we use transformer-based embeddings. Then, we also wanted to use transformers to do the classification, as we thought that it may have been a lot better than the basic ones we used, but we faced the implementation issues and time constraints we talked about, which did not allow us to do that. The usage of those kinds of models would probably improve performance when compared to our simple KNN. Nonetheless, the hyperparameter tuning could also have been more extensive, but as the data had big volume this would also increase computational effort and the time to run random searches would as well. Techniques to help with imbalance data would help as well, as one of our labels has 3 times more rows than the other. It is not a huge difference, but oversampling and undersampling or class weighting techniques could help increase model performance.

In spite of all the room to improve, we managed to try different preprocessing, feature engineering and modeling techniques applied to text, ending up with a satisfactory solution to predict if an airbnb listing will be up or not in the following quarter by only using text descriptions and reviews from it.

8. ANNEXES

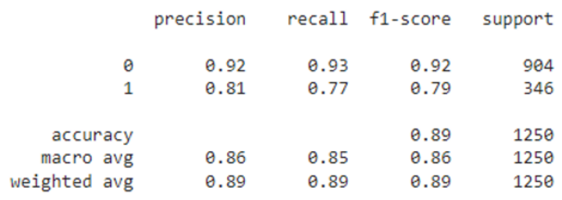


```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00      7226
           1       1.00      1.00      1.00      2770

    accuracy                           1.00      9996
   macro avg       1.00      1.00      1.00      9996
weighted avg       1.00      1.00      1.00      9996
```

Annex 1 - Best KNN Model Train

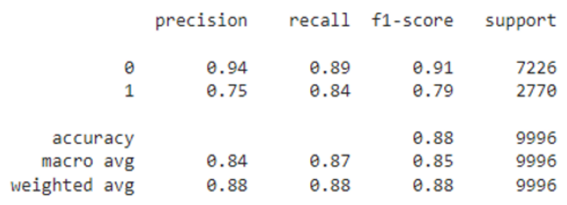


```
              precision    recall  f1-score   support

           0       0.92      0.93      0.92       904
           1       0.81      0.77      0.79       346

    accuracy                           0.89      1250
   macro avg       0.86      0.85      0.86      1250
weighted avg       0.89      0.89      0.89      1250
```

Annex 2 - KNN Test

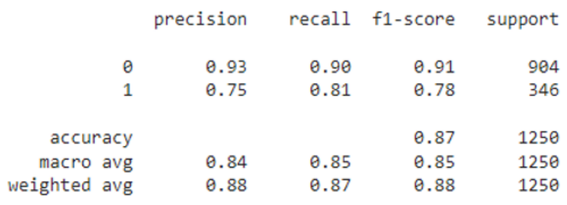


```
              precision    recall  f1-score   support

           0       0.94      0.89      0.91      7226
           1       0.75      0.84      0.79      2770

    accuracy                           0.88      9996
   macro avg       0.84      0.87      0.85      9996
weighted avg       0.88      0.88      0.88      9996
```

Annex 3 - Best LR Model Train

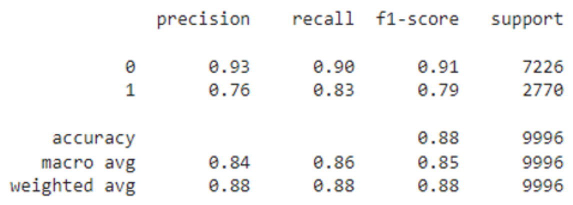


```
              precision    recall  f1-score   support

           0       0.93      0.90      0.91       904
           1       0.75      0.81      0.78       346

    accuracy                           0.87      1250
   macro avg       0.84      0.85      0.85      1250
weighted avg       0.88      0.87      0.88      1250
```

Annex 4 - LR Test

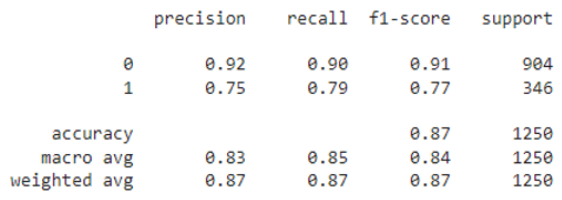


```
              precision    recall  f1-score   support

           0       0.93      0.90      0.91      7226
           1       0.76      0.83      0.79      2770

    accuracy                           0.88      9996
   macro avg       0.84      0.86      0.85      9996
weighted avg       0.88      0.88      0.88      9996
```

Annex 5 - Best MLP Model Train



```
              precision    recall  f1-score   support

           0       0.92      0.90      0.91       904
           1       0.75      0.79      0.77       346

    accuracy                           0.87      1250
   macro avg       0.83      0.85      0.84      1250
weighted avg       0.87      0.87      0.87      1250
```

Annex 6 - MLP Test

9. REFERENCES

- GloVe: GloVe: Global Vectors for Word Representation (stanford.edu)
- Hugging face's STSB-XLM-R-Multilingual: sentence-transformers/stsb-xlm-r-multilingual · Hugging Face
- Hugging face's DistilUSE base multilingual cased v2: sentence-transformers/distiluse-base-multilingual-cased-v2 · Hugging Face
- Huggin face´s models https://huggingface.co/models?pipeline_tag=text-classification&sort=downloads
- Fine-tuning guidelines https://huggingface.co/docs/transformers/tasks/sequence_classification#text-classification