PROJECT REPORT

TEXT MINING

Predicting Airbnb Unlisting

GROUP 15

Adriana Monteiro 20220604
Mafalda Paço 20220619
Marta Dinis 20220611
Patrícia Morais 20220638

Introduction

The increasing popularity of the Airbnb platform has changed the tourism industry and the livelihoods of the people who are able to take part in the renting out market. Understanding the factors that influence property unlisting on Airbnb can provide valuable insights for the hosts, enabling them to take appropriate measures to avoid it.

Through the application of natural language processing (NLP) techniques, we're going to extract information from Airbnb listings and their reviews, and develop a classification model that can predict the listing status of properties.

We decided to create two notebooks, one solely dedicated to data exploration, as well as another one where all the features engineering, classification models and evaluation were performed.

Data Exploration and Preprocessing

Data Analysis

We started our data exploration by checking for null values, which our dataset didn't have. We then checked the distribution of our labels, finding some imbalance, as we only have 28% of unlisted properties, as seen in Figure 1:

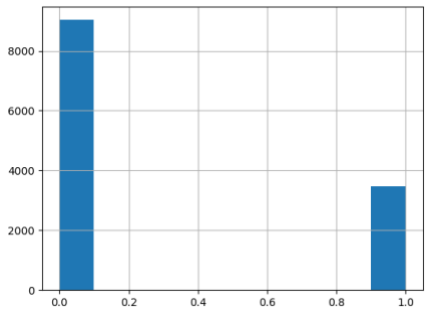

Figure 1: Distribution of the target variable

We also noticed that only 1% of the listings were reviewed. About the hosts, we found that 35% of the listings were made by unique hosts, meaning that the majority of hosts have multiple properties listed.

Up until this point, the analysis of the dataset was more structural, whereas, since we are dealing with text, we also need to analyze its content. Looking into our features we find that they all have multiple languages, which will imply more careful preprocessing. It was also uncovered that the text has html tags, repeated collections of characters that don't make up words and emojis. These will be dealt with and further explored in the preprocessing step.

Preprocessing

The first step in the preprocessing of the data is to remove html tags since they don't contain relevant information and add noise and dimensionality. Later we will remove punctuation for the same reasons.

Following this, we normalize our data by lowercasing it, which reduces our vocabulary by decreasing potential word variations and brings consistency to our data.

(extra) To further improve consistency, we expanded our contractions, enhancing semantic understanding by providing a clearer representation of the meaning of the text – for example, the word "*don't*" turns into "*do not*".

Next, we removed all punctuation from the text, so that these symbols wouldn't be considered in the model.

(extra) In our Airbnb listings dataset, we decided to remove emojis, as we don't intend to perform sentiment analysis with this dataset, and they tend to be used as graphical representations of emotions and ideas.

(extra) Our next step in the preprocessing was identifying the languages in our data, to preprocess each corpus accordingly. We found that our Airbnb dataset is less diverse than our reviews dataset, as seen on Figure 2.

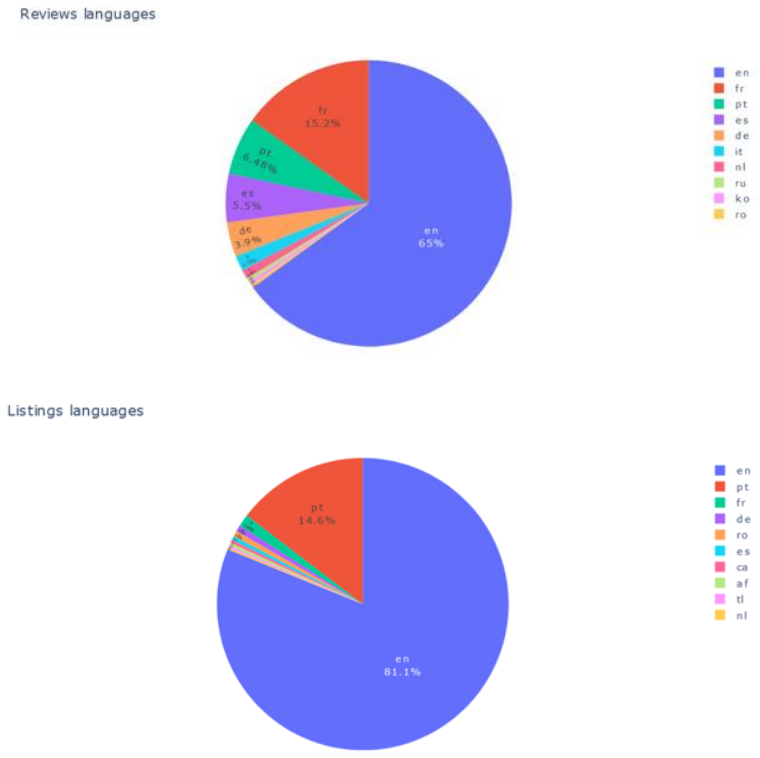

Figure 2: Language distribution in the datasets

Having the languages detected, we can now preprocess according to each language's specifications. We opted to only deal with the top 3 languages - English, Portuguese, and French - as they constitute around 90% of our data. We started by converting the emojis present in the *reviews* dataset into text, translating them to the language of the corpus, if the language was one of our top 3; otherwise, the emoji was deleted.

We noticed that all throughout our dataset, one string in particular that had no meaning appeared several times ("*x000d*"). Therefore, we removed all instances of this string. Furthermore, since specific numbers that express distances, costs, etc. aren't needed and overcomplicated our model, we also decided to delete all digits.

Subsequently, we removed the stop words, words that are frequently used but provide little to no value to our processing tasks.

To further standardize the text, we implemented lemmatization, reducing different inflections of a word to a common base. This ensures words with the same meaning have a consistent representation, avoiding ambiguous meanings, and helps reduce the vocabulary and complexity of the data.

(extra) In order to further explore our datasets, we decided to create word cloud visualizations, which show, through color and size, the words most frequently used. These representations can be seen on Figure 3:

Figure 3: Dataset Wordclouds

As will be explained in further detail in the next segment, we created a feature in the reviews dataset for the sentiment associated with each review. In Figure 4, we can see the distribution of each sentiment (positive, negative, and neutral) in our dataset, when we consider all languages, and when we filter for only English, Portuguese, and French reviews. From this figure, we can see that most Airbnb reviews were positive in both cases.

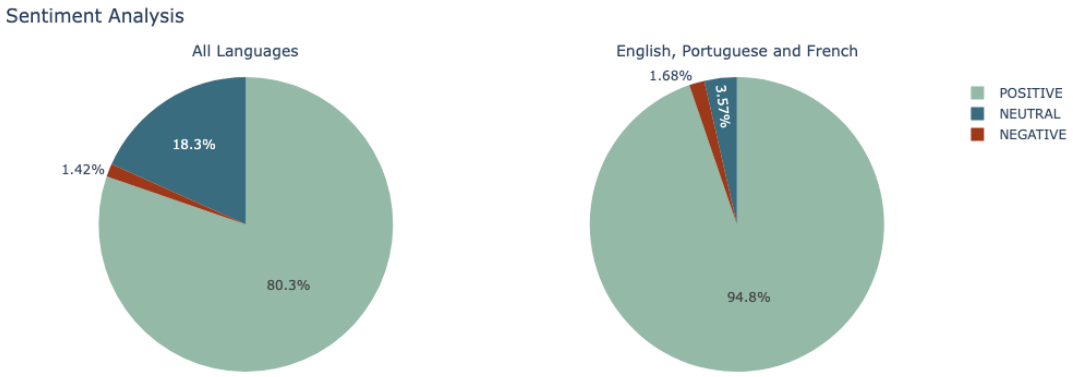

Figure 4: Sentiment distribution in the reviews dataset

Feature Engineering

TF-IDF

Term Frequency-Inverse Document Frequency (TF-IDF) allows to determine the importance of a word in a document within a collection of documents. This is done by assigning a higher weight to terms that have a high frequency within a specific document (TF) and a low frequency across all documents (IDF). Higher TF-IDF scores indicate that a term is more relevant to a particular document in comparison to other documents in the collection.

For each observation, we get a vector with the TF-IDF scores correspondent to the terms in that document and 0 for the ones that belong to the corpus but not to the document. Considering all observations, we get a matrix constituted by these vectors, and use it as an input feature in the classification models used.

We started by using the TF-IDF embedding technique without using the reviews dataset so we could have a baseline to compare in the future. The features were created through a FeatureUnion pipeline where the TF-IDF for each one of the columns is performed and then put together. With this method we tried several different models, such as KNN and Balanced RandomForest. Furthermore, we also tested this process with different sizes of n-grams using a range from 1 to 6 for KNN, Balanced RandomForest and an MLP with two hidden layers, 10 neurons in the first one and 5 in the second.

Regarding TF-IDF, we implemented it as well concatenating the reviews for each Airbnb, filling with an empty string the ones without reviews. As before, the TF-IDF was implemented but now with 3 columns (*host_about*, *description* and *comments*), creating again the features through the FeatureUnion pipeline.

GloVe Embeddings

Global Vectors for Word Representation (GloVe) is an unsupervised learning algorithm used to generate word embeddings. GloVe embeddings capture semantic relationships between words based on how frequently words co-occur within a fixed context window in the corpus. The vectors are based on global counts across the corpus. Words with similar meanings or contexts tend to have similar vector representations. In all the different approaches taken we used the '6B' GloVe vocabulary from the torchtext library.

In our project, we decided to create a word embedding for each term in each observation, and then calculate the average for each one.

Our first approach using the GloVe embedding method was, without using the reviews data, just concatenate the tokenized host information and description of the Airbnb together, creating a tokenized host_and_description column and then performing the embedding of the words with the GloVe vocabulary. This new embedding is the input feature for the models then used.

We also tried creating the embeddings for each column, host information and description of the Airbnb, and then as a second step we experimented with two approaches, concatenating the embedding, and averaging them. These methods were possible since we had previously defined our word embeddings' size as 300.

Having the reviews information in consideration as we performed the first method we described, we concatenated the three columns and then performed the embedding and their average. We decided to try this final one including the reviews since it was the best performing model.

FastText Embeddings (extra)

FastText embeddings are a type of word embeddings that capture semantic and syntactic information of words. They are an extension of traditional word embeddings but with additional features that enable handling out-of-vocabulary (OOV) words and capturing subword information.

FastText learns word representations through an unsupervised learning process. It constructs a vocabulary from the training corpus by splitting words into character n-grams. Then, it learns a vector representation for each word by training a shallow neural network using a technique called Skip-gram with Hierarchical Softmax. Then, the embeddings of the words of the text are combined, resulting in a vector that can capture the semantic information of the text and can be used for downstream tasks like text classification or sentiment analysis. This vector is then used as an input feature for our classification models.

We used a pre-trained FastText model, by the Gensim library, that was trained on Wikipedia and news data, with word embeddings of size 300. Pre-trained models offer advantages like being trained in large volumes of data, that have a rich vocabulary, resulting in better representations of words. In datasets like ours, since they revolve around one specific topic, the vocabulary will be limited and training data sparse.

Our baseline approach embedded the concatenated *description* and *host_about* columns and used the knn model. We were able to improve our f1 score by averaging the generated embeddings by column and using the balanced random forest model.

Sentiment Analysis (extra)

Sentiment Analysis is a process that classifies the polarity (positive, negative, or neutral) conveyed by a certain piece of text. This technique may have a great impact in our case, since our predictions might rely on customers' reviews - if most reviews of a certain listing seem to be negative, we expect it to be more likely to get unlisted.

With that in mind, we decided to create a new feature in our review's dataset, that would encompass the sentiment of the customer regarding a certain Airbnb. We replaced with 1 the reviews with a positive sentiment, with -1 the ones with a negative sentiment and with 0 all neutral reviews. When joining the Airbnb information with its reviews we averaged the sentiment and replaced also with 0 the observations without reviews. This would later be used as an input feature of the classification models used.

We implemented this feature on our TF-IDF model, concatenating it with our embedded columns and applying the KNN model. We did the same with the GloVe embedder, adding the sentiment feature to the embedded matrix.

Classification Models

Now, having the words embedded as features, we need a classification model to train and test. We applied, with the various embeddings created, several models described below.

K Nearest Neighbours (KNN)

KNN is a supervised machine learning algorithm that uses distance as a similarity measure and uses it to classify datapoints. In other words, data points with similar features tend to belong to the same class. We applied this model because of its simplicity and efficiency, comparing the distances between embeddings and clustering them according to their similarity. We opted for a model with the following parameters:

KNeighborsClassifier(n_neighbors = 5, metric = 'cosine', weights = 'distance')

Multi-Layer Perceptron (MLP)

Multi-Layer Perceptron is a type of artificial neural network that can be used to solve NLP tasks, as long as the text data is transformed into numerical representations, which can be achieved by applying word embeddings, for example. MLPs are good at generalizing to out-of-vocabulary words, and are able to capture non-linear relationships between features, making them a flexible model for this classification task. This model was instantiated with parameters listed as follows:

MLPClassifier(hidden_layer_sizes=(10,5), activation='relu', solver='sgd', learning_rate_init=0.01)

Logistic Regression (LR)

Logistic Regression is a supervised machine learning algorithm that models the relationship between a set of input features and a binary outcome by estimating the probabilities of the outcome belonging to each class. Although it assumes a linear relationship between the features it's robust to noise and a simple model, providing a good baseline model to establish a

benchmark performance. The parameters chosen for the Logistic Regression were the default parameters.

Gaussian Naïve Bayes (GNB)

Gaussian Naive Bayes is a probabilistic classifier based on Bayes' theorem. This model can be applied to NLP tasks such as ours by representing text data using numerical features (through TF-IDF, word embeddings, …) and assuming a Gaussian distribution for those features. Although we are assuming feature independence, which is not entirely accurate for text data due to the inherent dependencies between words, Gaussian Naive Bayes has been shown to still provide reasonable performance. As we did with the Logistic Regression, the GNB was executed with the default parameters.

Balanced Random Forest (BRF) (extra)

Balanced Random Forest is a variant of the Random Forest classification model that addresses imbalances in the distribution of the target variable, which is our case. In Balanced Random Forest, additional training samples are generated to balance the class distribution. This is achieved by employing two techniques: undersampling the majority class and oversampling the minority class. It benefits from assembling, making it robust to noise, and is able to capture non-linear relationships between features, making it an advantageous model for this classification task. Alike the two previous models, the parameters chosen were the default ones.

Evaluation

We created a function that, given the training and validation data, as well as the model, fitted the model to the training data and created the predictions. It also provided, if specified, the confusion matrix plot, the classification report and other desirable score metrics for training and validation. This method allowed us to keep track of each model's performance.

We will be basing our evaluation on the f1 score. F1 combines the precision and recall scores of a model, evaluating its predictive skill by elaborating on its class-wise performance, making it robust to class imbalance, which is our case.

In the process to narrow down the models we would use, we started by implementing a simple *TF-IDF* with all the previously mentioned models. Our best performances came from *BRF*, *KNN*, and *MLP*, making these the models we will apply to our embeddings.

Since MLP is a very hyperparameter dependent model we decided to implement a simple GridSearch with the following parameters.

```
parameter_space = {'hidden_layer_sizes': [(10,10), (5,5,5)],
                   'activation': ['relu'],
                   'solver': ['sgd', 'adam'],
                   'learning_rate_init': [0.001, 0.01]}

GridSearchCV(estimator =mlp ,param_grid = parameter_space,verbose=1, scoring='f1',cv=3)
```

However, the performance did not exceed the previously implemented MLP model. We chose not to execute the GridSearch with a bigger set of hyperparameters since it wasn't viable due to its computational demands.

Having this benchmark, we started experimenting with different embeddings. Across our *TF-IDF* experiments, whether it was different ngrams, including the reviews or its sentiment analysis, we hit a ceiling of 0.60 validation f1 score. This led us to try a different type of embedding altogether. Although we noticed a slight improvement in performance when using a maximum of 2-grams, the computational power available was not enough to support this when joining information about the reviews to our models.

With *GloVe* embedding our best f1 score was 0.58, when we generated the embeddings for each column and then averaged them by row.

FastText gave us similar results, also when averaging the embeddings by column, returning a 0.57 f1 score.

In the end, our model couldn't break out of this 0.60 ceiling on the validation score, even though it reached almost a score of 1 in the training data. This reflects its inability to generalize and, therefore, perform well on unseen data. One possible way of solving this problem would be to apply transformer models, which are known to perform very well on these types of tasks. However, after several attempts, it was clear we lacked the computational power to be able to implement this technique.

Overall, the best achieved performance was obtained using the default TF-IDF with the sentiment analysis performed. The model's results are represented in Table 1:

metric	train	val
balanced_accuracy	0.997238	0.726948
precision	0.997828	0.596073
recall	0.995307	0.613276
f1	0.996566	0.604552
roc_auc	0.997238	0.726948

Table 1: Best model performance

Finally, we applied our best model to the test dataset given in order to get the final predictions.

References

Radim Řehůřek. "Gensim FastText Documentation." Gensim. Available online: https://radimrehurek.com/gensim/models/fasttext.html#gensim.models.fasttext.FastText

TorchText Documentation, GloVe. Available online: https://torchtext.readthedocs.io/en/latest/vocab.html#glove