AirBnB Unlisting Classification Problem

Text Mining - Group 15

Master's in Data Science and Advanced Analytics

Authors:

André Filipe Silva	20230972
Guilherme Moreira	20230538
João Gonçalves	20230560
João Pedro Mota	20230454

NOVA IMS
2023/2024

Contents

1 Data Exploration - before preprocessing

We will analyze our ”train” and ”train_reviews” files before preprocessing. ”test” and ”test_reviews” follow the same structure, except test does not contain the labels for listing status.

Analyzing the shape, it seems like we have 6248 different properties on the train set. These come with descriptions about the property and the host, but they are not just in English. They come in several languages.

The reviews/comments follow suit; we have a very large dataset of reviews, with 361 281 comments about the properties from visitors, and only 2 of these were identified as being empty observations. A lot of them are in English, but a lot of them are in other languages. The comment word count goes all the way from 6232 words to just 1 word comments. Even 1-worded comments seem valid, apart from non-alphabetical ones.

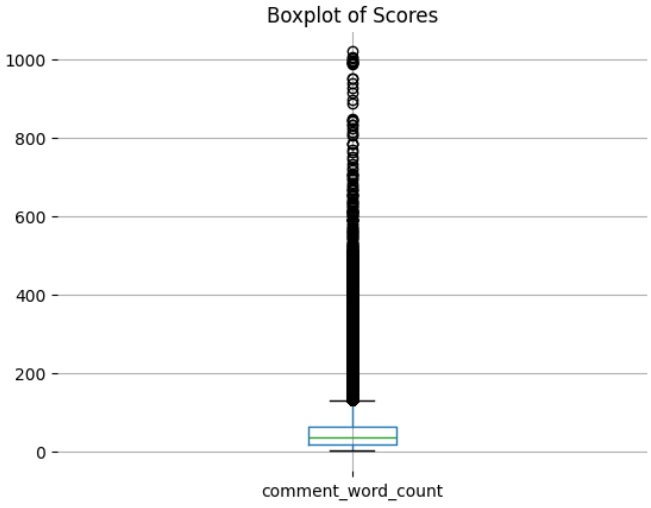

Figure 1: Box plot of word count in comments

We were able to identify **43 different languages** in the comments, with 918 comments not being properly matched to any particular language with our detector. Still, this is a very good result, as 918 out of 361 281 represents just 0.25% of the dataset. The most common language was, of course, English, and the least common was Urdu - with 1 comment identified.

The description of the property also varies in word count immensely. It goes from a maximum of 210 words to a minimum of just 3 words. The average is around 133 words.

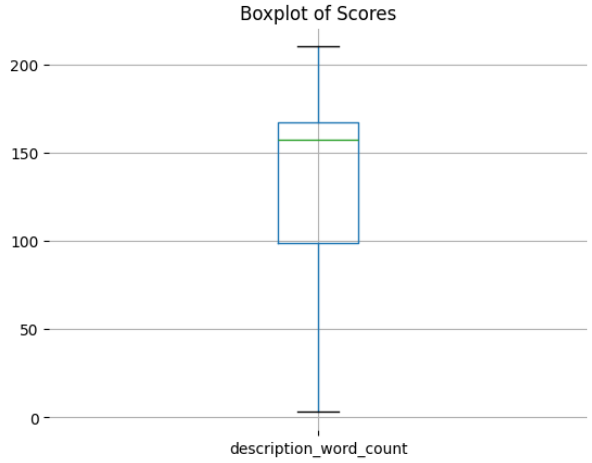

Figure 2: Box plot of word count for Description of properties

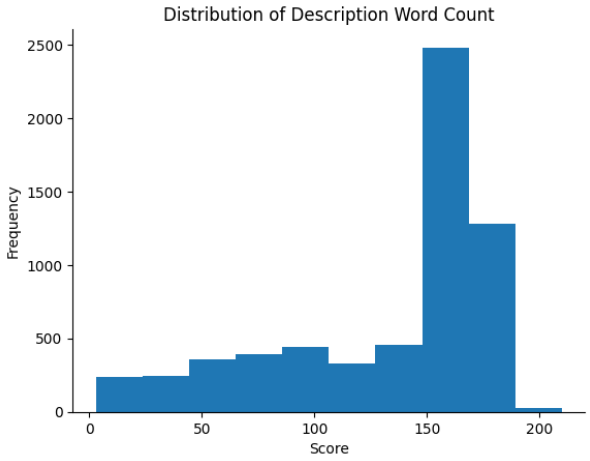

Figure 3: Distribution of word count for Description of properties

If we analyze the most common words in the description of properties, we find results that are not very surprising - especially before preprocessing. The most common word is ”the” - the most common word in the English language, which is also the most represented language in the dataset. Other than that, we find some odd HTML tags that will be removed with our preprocessing, so the results should be fairly different later.

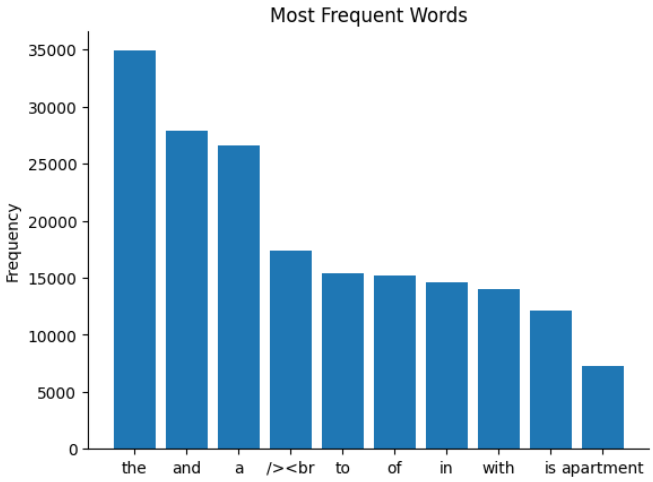

Figure 4: Most frequent words in property description

Going back to the comments, the character count (not to be confused with word count) has a huge range. It goes from a max of 1334 characters to a minimum of 1 character. Reviews with 1 character are not very useful, but from 3 characters and onwards they seem to be helpful: you see comments like ”Top”, ”Bom”, ”:-)” pop up, which are helpful, even if short.

For the exploration, we focused more on character count than word count, because there are comments

with one word that might be useful or useless, but comments with one character can be pretty safely assumed to be useless. Notice that some comments consist simply of an emoji, which makes our task harder in plotting and analyzing, as they don't appear in plots. You can see in the plot below that the missing squares represent emojis, with the first one from the left representing a "thumbs up sign", and the second one representing a "OK hand sign". But, as it is possible to see, many times one word comments do not provide any real information.

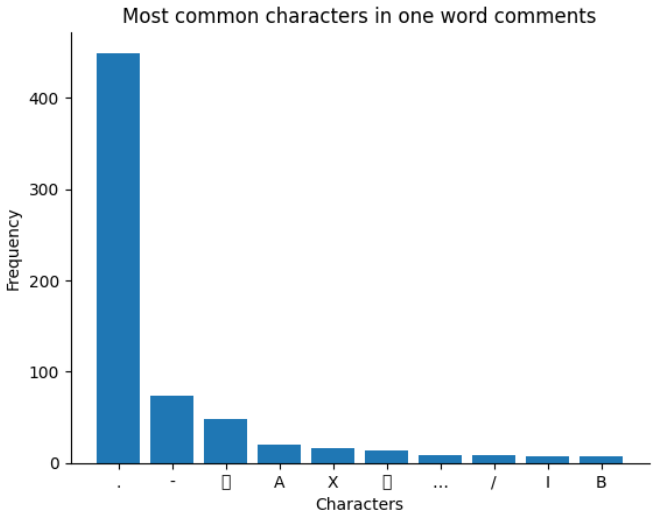

Figure 5: Most frequent characters in one word comments

The property with more comments is the one with "listing_id" 265. It has 891. This can be either good or bad, but we assume it to be good: if a property had these many comments and most of them were negative, it would probably be unlisted. This property is still listed, according to our labeled data.

Instead of looking at one word comments, let us now look at all comments, and see the most common words.

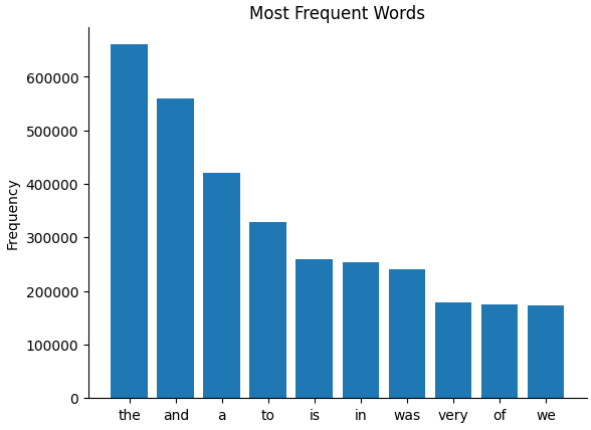

Figure 6: Most frequent words in all comments

This shows us more interesting data than focusing on one word comments (even though that exploration is also important for the reasons mentioned above). It highlights how all the most common words are English words, which again makes sense in light of the over representation of English in our dataset compared to other languages.

We also checked for duplicated entries in our 'train_reviews'. There are quite a few, and that is important to know as we need to handle it in our preprocessing stage.

It is also interesting to check the listing status and its proportion. In the plot below you can see the frequency, with a lot more properties staying listed (those are the zeros) than unlisted. The proportions are 73% still listed and 27% unlisted.

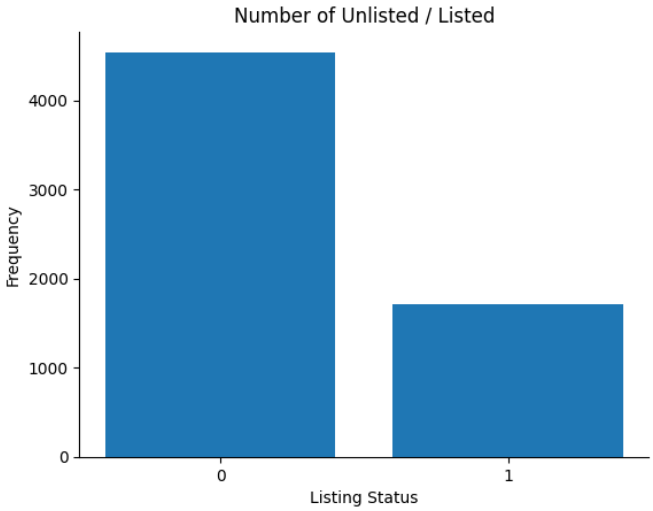

Figure 7: Listing Status

Finally, we created a visualization with the most common words (or HTML tags / artifacts, as we have not removed them at this point), joining all the comments in the dataset together. It show us 'Lisbon' and 'apartment' are very popular, but it also shows there is a lot of data cleaning to be done.

Figure 8: Word cloud of most frequent words/strings

2 Data Preprocessing

As was made clear by our exploration, a lot of preprocessing is needed in order to render our dataset useful.

It is important to note that in this section we will describe the preprocessing applied to the 'train' and 'train_reviews' datasets, but the same preprocessing was done for the 'test' and 'test_reviews' datasets. The only exceptions have to do with the 'unlisted' column, which does not exist in the 'test' dataset, and with the fact that we do not drop any observations on our test set. Other than that, exactly the same preprocessing was applied to both datasets: train and test. If any other exceptions/differences to the preprocessing arise, they will be described in the next paragraphs.

We started by translating our entire dataset into the English language. Please note that this counts as one **extra method** of data preprocessing, according to the project guidelines.

All files and all of their string fields were translated to English utilizing the package 'deep-translator', which allows you to connect to Google Translator's API. This took some time but the results were very good.

Focusing on the comments both 'train_reviews', the number of comments that could not be translated was in the range of 0.0058% our entire dataset. These errors happened because the text length would exceed 5000 characters, which is the limit for the API. We decided to drop the rows with error in translation since they are such a minuscule part of our dataset, and will not affect our overall results later on. For 'test_reviews', there were no comments that could not be translated.

When our translations were ready, we made some minor adjustments to the dataset, such as renaming 'index' to 'listing_id' for ease of understanding what it means. We dropped the original version of the columns, and kept only the translated ones.

Next, we performed an outer merge between the 'train' and 'train_reviews' datasets. This decision comes because it is useful to have them both together, as we can see the 'host_about' and 'description' fields in the same row as each comment. The most important thing is that we can see, for each comment, if it corresponds to a property that was unlisted or not (through the column 'unlisted'). Performing an outer join requires some caution: it might generate duplicate entries. We checked for that, and removed the duplicates. Finally we just reordered the columns for ease of visualization, in the following order: ['listing_id', 'host_about', 'description', 'comments', 'unlisted'].

Following this, we applied a custom built function that checks for word contractions (things like "you're", "aren't", "it's") row by row, and creates new columns for 'comments', 'host_about', and 'description' with a boolean value in case it finds contractions. We then have defined another function that expands these contractions (i.e. 'you're' becomes 'you are') using regular expressions.

At this point in our approach, we decided to check if we had any NaN values in our dataset. We found 4513 'host_about' empty observations, and 2732 'comments' empty observations. Note that for the 'host_about', this is likely the same few hosts but repeated over and over again, as we merged our datasets. We opted not to drop anything, because we were concerned with losing information on whole listings if so. Instead, we replaced the empty values with a default string, the same for all fields where there is an empty value: "Empty string".

At this point, we created a very extensive function ('clean_df(df)') to perform cleaning of our data. The cleaning was applied to all string columns: 'host_about', 'description', and 'comments'. Several steps were applied in the cleaning, and they are described below:

1. Utilized regular expressions (regex) to remove HTML tags, monetary symbols, time references, and punctuation.
2. Utilized regular expressions (regex) to transform date formats.
3. **Extra Preprocessing method**: Utilized a function, 'emoji_to_text' to convert emojis into regular text using the 'demojize' method of the 'emoji' library. This counts as an **extra preprocessing method** according to the project guidelines.
4. Lower-cased everything.
5. Expanded the word contractions based on the previously defined contraction detector.
6. Removed the 'x000d' tag from the text, as it appeared quite often and it is just noise (this can be verified in the Word cloud plot provided in the Data Exploration before preprocessing part).
7. Tokenized all observations in 'comments', 'host_about' and 'description'.
8. Removed all stop words for the same three columns.
9. Joined the comments back into their string format (de-tokenized) after removing the stop words.
10. Finally, dropped the columns with boolean values for containing contractions as we have no use for them anymore - it has already been taken care of.

Next, we merged all the comments for the same property (through 'listing_id') on the same line. This means that now we have only one row per 'listing_id', and not several as before. Each row contains, in the 'comments' column, every single comment for that property, saved in a list. We performed a sanity check on the proportion of listed/unlisted properties at this point and gladly, the proportions were maintained (73% still listed and 27% unlisted).

Next up, we took our dataset and applied **Lemmatization** and **Stemming**. One at a time, of course. We essentially wanted to have both done and saved to files so that we could load them and use later, when approaching our modelling.

For Lemmatization we used the WordNetLemmatizer(), and for Stemming we used the SnowballStemmer, with the parameter 'english' passed inside it.

To conclude the Preprocessing part of our work, we applied these different methods: Lemmatization, Stemming, stop word removal, regular expressions to transform text, punctuation removal, regular expressions to expand word contractions. Our **extra Data Preprocessing methods** were translating to english and converting emojis into text.

3 Data Exploration - after preprocessing

After our preprocessing stage, it is useful to perform data exploration again to see if everything is correct and if something still escaped our methods. We do this exploration based on our cleaned but unmerged

(i.e. the one with all comments for the same property in the same row) dataset.

Starting by checking empty values, there are none in any of the columns.

Analyzing the box plot for the word count of descriptions, there is a clear reduction in the median number of words compared to the raw data. This is likely explained by the removal of stop words, HTML tags (which were being counted as words) and other artifacts in the text.

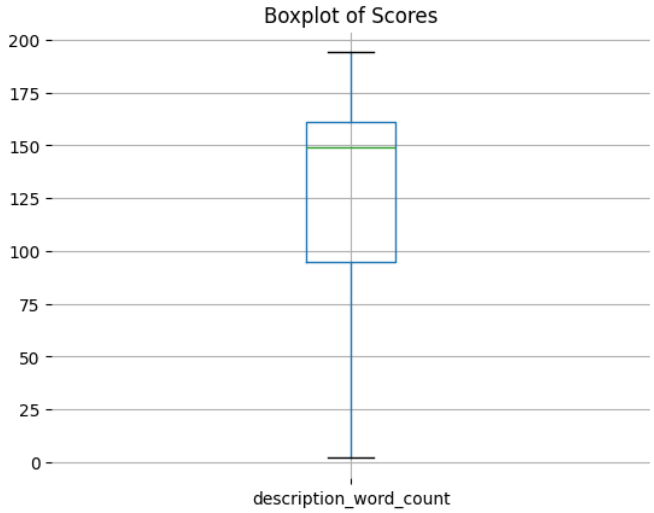

Figure 9: Box plot of word count for Description of properties, after preprocessing

Analyzing the distribution of the word counts, however, there are not very significant differences. This is good to see, as it means we were able to preserve the integrity of the distribution even with the application of our preprocessing methods.

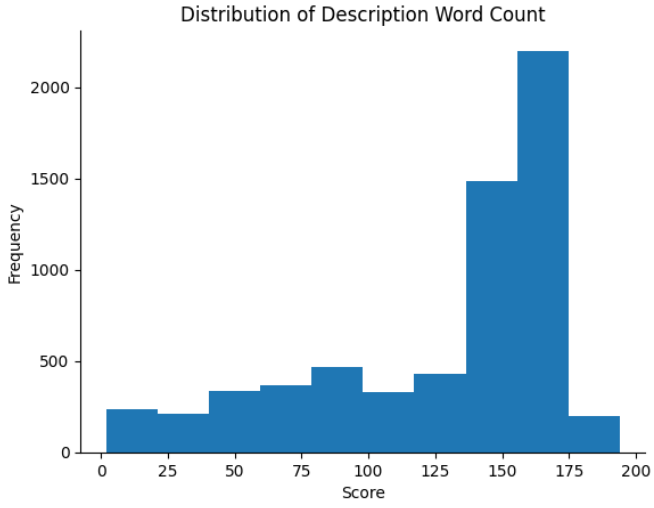

Figure 10: Distribution of word count for Description of properties, after preprocessing

Analyzing the comment character count, there is a significant change, as the values now range from a max of 832 to a minimum of 1 (compared to the range of 1334 to 1 before preprocessing). This is likely due to the removal of artifacts (HTML tags for example) in the text.

The box plot for the word count in the 'comments' column is very similar to the unprocessed one. This indicates that we did not do anything that would alter the distribution.

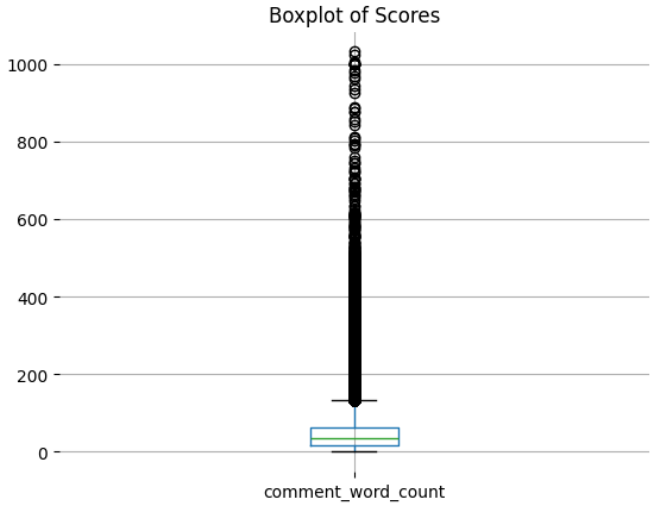

Figure 11: Box plot of word count in comments, after preprocessing

This chart shows significant differences to our unprocessed data. This can be explained essentially by the removal of stop words ('the', 'and', ...). In fact, most of the words in the preprocessed chart are stop words. Since we removed those, now we get only the meaningful words that appear the most.

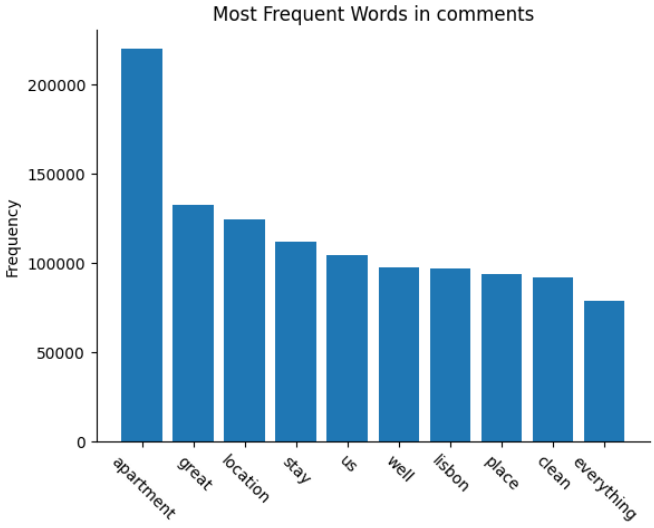

Figure 12: Most frequent words in all comments, after preprocessing

Checking for duplicates, we find there are none, constrasting with the raw data. This was an explicit step in our preprocessing stage, so it is not surprising.

As has already been mentioned in the previous section, the proportion of Listing Status remains the same after our preprocessing. For that reason, a plot will not be inserted here as it would just be a repetition of the one for the raw data.

The only plot left to visualize is the preprocessed Word Cloud.

Figure 13: Word cloud of most frequent words, after preprocessing

In here we see vast differences. The weird '_x000d_' tags are gone, such as other text artifacts, leaving only the relevant words.

4 Feature Engineering

For feature engineering, a variety of methods were employed.

First and foremost, we split our dataset into train and test data (you may call it train and validation data, but we did not do the train, val, test split - only the train, test split). We performed this with a split of 70% - 30%, stratifying on the target variable, and did it for both our Lemmatized and Stemmed datasets. Taking care of this right away means we can use either of them for what comes next.

Our main concern here was making sure that the listing status proportion held, after splitting. If it did not, it was a sign that we were experiencing data leakage somewhere down our pipeline. Fortunately, the proportions were maintained, and we continued our work.

The methods employed were: Word2Vec, TF-IDF, and GloVe. The **extra methods** implemented were the Transformer-based embeddings, utilizing ROBERTa and BERT. These account for the 2 extra methods required by the guidelines.

4.1 Word2Vec

Word2Vec is a group of related models that are used to produce word embeddings. These models are shallow, two-layer neural networks that are trained to reconstruct linguistic contexts of words. In our case, we implemented the Skip-gram word2vec model, that predicts surrounding words given a current word.

Taking the lemmatized split dataset, we implemented wWrd2Vec, converting our mass of text into vectors (one per word), and learning relationships between words automatically.

Using feature reduction techniques, it is possible to plot the results, although they are not very telling. But we leave the static plot here to get a visual idea on how word2vec results in.

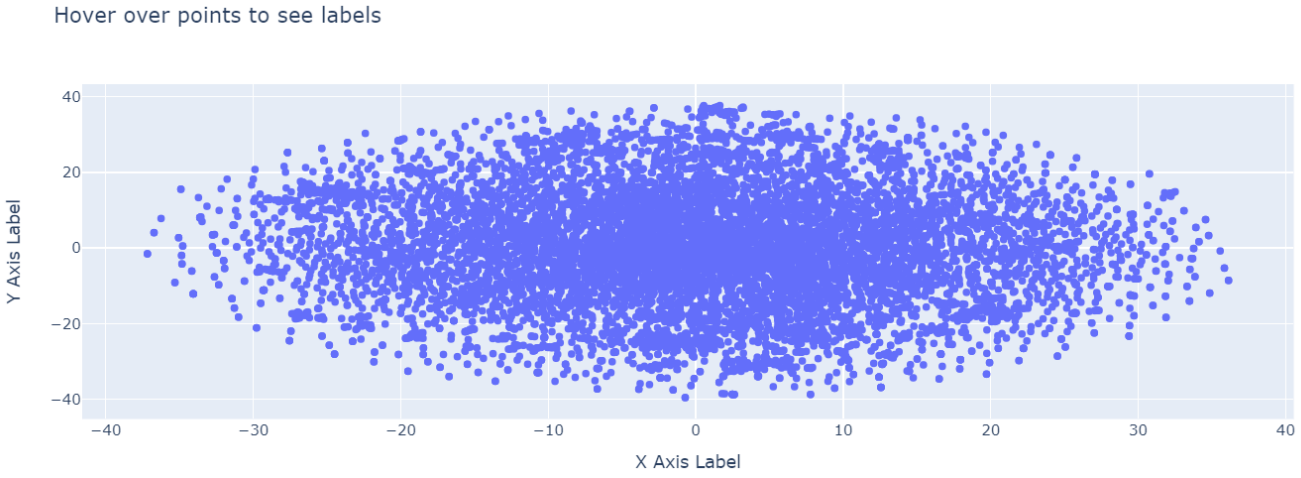

Figure 14: word2vec plot

4.2 TF-IDF

The TF-IDF algorithm was also implemented, in order to then apply later to our classification models and do the predictions.

TF-IDF is a statistical measure used to evaluate the importance of a word to a document in a collection or corpus. The importance increases proportionally to the number of times a word appears in the document but is offset by the frequency of the word in the corpus. This helps in distinguishing the significance of a word in individual documents.

The Term Frequency (TF), measures how frequently a term occurs in a document. TF is often normalized by dividing by the total number of terms in the document to prevent bias towards longer documents.

The Inverse Document Frequency (IDF) measures how important a term is. It is calculated by taking the logarithm of the number of the documents in the corpus divided by the number of documents where the specific term appears.

The only relevant parameter to mention here, is that we performed the vectorization with a maximum number of features equal to 15 000 ('max_features = 15000').

4.3 GloVe

GloVe is an unsupervised learning algorithm for obtaining vector representations for words. It is designed to aggregate global word-word co-occurrence statistics from a corpus, and then these statistics are used

to infer word vectors. The model essentially captures the probabilities that two words appear together, helping it capture both syntactic and semantic meanings of words.

For our GloVe algorithm implementation, we resorted to the 840B embeddings. GloVe is not as straightforward to implement as TF-IDF, meaning we had to define our own function 'sentence_to_vec_840B' in order to apply it. We also had to split our big list of lists of comments into a flattened list, and then pass it into that function, in order for it to work properly. Finally, we transformed the results into PyTorch tensors, and also transformed our target variables into tensors so the model would work properly.

4.4 Transformer - BERT

Developed by Google, BERT makes use of Transformer, an attention mechanism that learns contextual relations between words (or sub-words) in a text. Unlike previous models which read the text input sequentially (left-to-right or right-to-left), the Transformer in BERT reads the entire sequence of words at once. This characteristic allows the model to capture the context of a word based on all of its surroundings (left and right of the word).

We constructed our own custom function to get the embeddings. We performed every necessary operation inside that function, like tokenization and other necessary transformations to our comments list. Outside the function, we needed to do some convertion to numpy arrays and reshaping them in order to get the proper format to feed to classification models.

You can find more information about the specific model used by clicking here.

4.5 Transformer - ROBERTa

RoBERTa iterates on BERT's methodology. Developed by Facebook, it modifies key hyperparameters in BERT, including removing BERT's next-sentence pretraining objective and training with much larger mini-batches and learning rates. This results in improved performance over BERT on several benchmark NLP tasks.

Again using the lemmatized dataset, we applied Roberta to our data to get embeddings. You can find more information on the specific model we used by clicking here.

The details for applying Roberta are essentially the same as for applying Roberta. We even used the same custom function - with very small adaptations. As such, there is no need to repeat ourselves.

5 Classification Models

In order to get the predictions for the test set, which is the whole point of this work, we need classification models that can learn from our training data and then predict labels for the test data. As such, we implemented KNN, Logistic Regression, Stochastic Gradient Descent, and Multi-Layer Perceptron. These are our 4 'basic' models. We also implemented two advanced models, for the **extra methods** part of the guidelines. These models were Random Forest and HistGradientBoosting.

5.1 Logistic Regression

Logistic Regression is used to estimate discrete values (usually binary values like 0/1) based on given set of independent variable(s). It does so by predicting the probability of the output variable using a logistic function (usually the sigmoid), which is an S-shaped curve.

For the implementation, we set the parameter 'class_weight' to balanced, to adjust the outcome given our target variable has such an unbalanced distribution. The parameter 'n_jobs' was set to -1. This allows it to use all of the computational capacity available.

5.2 KNN

KNN is a simple, instance-based learning algorithm. It classifies a new data point based on the majority vote of its 'k' nearest neighbors. The algorithm follows these steps:

1. Distance Calculation: It calculates the distance (usually Euclidean) between the new data point and all the other points in the dataset.
2. Find Nearest Neighbors: It identifies the 'k' closest points (neighbors) to the new data point.
3. Majority Voting: The new data point is assigned the most common class among its 'k' nearest neighbors. The number of nearest neighbors should always be an odd number so as not to have ties.

   In our application, we used 5 as the number of nearest neighbors - a common choice in the literature. We also set the parameter 'n_jobs' to -1.

5.3 Stochastic Gradient Descent (SGD)

SGD is a simple yet very efficient approach to fitting linear classifiers and regressors under convex loss functions such as (linear) Support Vector Machines and Logistic Regression. Unlike traditional gradient descent which uses the entire data set to compute gradients at each step, SGD randomly picks one data point from the dataset at each step to compute the gradients. This makes SGD faster and able to be used on large datasets. We thought that this was a good choice given the very big dataset we have.

For our implementation, we left many parameters as default but changed the following ones:

1. 'fit_intercept' = False, as there is no reason to assume our data should have to be centered at all.
2. 'verbose' = 1, so we could get some output.
3. 'n_jobs' = -1, as for all the other models.
4. 'random_state' = 42, to get consistent results across runs.
5. 'learning_rate' = "adaptive", to optimize the descent.
6. 'eta0' = 0.000001, this is used in conjunction with the previous parameter.
7. 'early_stopping' = True, by doing this the algorithm sets aside part of the data for validation and terminates when it is not improving significantly.

8. 'class_weight' = "balanced", for the same reason described in Logistic Regression.

5.4 Multi-Layer Perceptron (MLP)

MLP is a type of neural network consisting of at least three layers: an input layer, hidden layer (one or more), and an output layer. Each neuron in a layer connects to all neurons in the next layer, forming a densely connected structure. MLP uses backpropagation for training, which involves:

1. Forward Pass: Compute the output for a given input through layers.
2. Loss Calculation: Calculate the error at the output.
3. Backpropagation: Propagate the error back through the network, updating weights to minimize the error.

In our implementation, we used the as activation function the logistic function, given the binary classification nature of our problem. We set the random_state to 42 once again, to ensure consistency across runs. Finally, we set our 'hidden_layer_sizes" parameter to (2,2), a simple enough method to balance the of the network with the computational requirements.

5.5 Random Forest

Random Forest is an ensemble learning method based on decision tree algorithms. It builds multiple decision trees and merges them together to get more accurate and stable predictions. The key steps include:

1. Bootstrap Aggregating (Bagging): Multiple subsets of the original dataset are created with replacement, and a decision tree is built for each subset.
2. Random Feature Selection: At each split in the learning process, a random subset of the features is selected to decide the best split among those subsets.
3. Majority Voting: For classification, the final prediction is the majority vote across all the decision trees.

Our implementation merely maintained a consistent 'random_state' = 42, 'verbose' = 1, and 'class_weight' = "balanced". All the rationale for these decisions has already been explained in the description of previous models.

5.6 HistGradientBoosting

Histogram-based Gradient Boosting is a scalable machine learning algorithm for gradient boosting that uses histograms to approximate the gradients and improve speed. It functions similarly to other GBM frameworks but with some optimizations:

1. Bin Continuous Variables: It places continuous feature values into discrete bins, which speeds up the computation of gradients and splits.
2. Iterative Refinement: The model is built iteratively, improving the model with each step by minimizing a loss function.

Once again, the implementation sets a 'random_state' = 42 and 'verbose' = 1.

6 Approaches to Classification

Given that we have implemented 8 different methods for Data Preprocessing, 5 different methods for Feature Engineering, and 6 Classification models, experimenting will all of these would lead to a total of 240 combinations possible. This is not feasible, and as such, we are going to try 3 different approaches, combining different methods and seeing what the best results are. All approaches will include all preprocessing methods, only differing between Lemmatization or Stemming.

We will focus on discussing the best model of each approach, in terms of F1 score and, if tied, the accuracy. But we will present the results for F1 score, Accuracy, Recall and Precision for all models.

1. Approach 1: Lemmatization + TF-IDF + Classification (All Models)
2. Approach 2: Lemmatization + GloVe + Classification (All Models)
3. Approach 3: Stemming + BERT + Classification (All Models)

6.1 Approach 1

For our first approach, consisting of **Lemmatization + TF-IDF + Classification (All Models)**, our best result comes from the MLP classifier.

Our results are so close to each other though, that the criteria to break the tie had to be the F1 weighted score. First, we looked at the F1 Score: tied across all models except KNN (behaves particularly bad on this kind of problem). Then we went through accuracy, recall, and precision, and everything was very much tied. The only difference, and even then it was small, was on the F1 weighted score.

Our best model performed very good, however, with a F1 Score of 0.77 for label 1, and 0.90 for label 0. It also had an accuracy of 0.86, a precision of 0.94 for label 1 and 0.70 for label 1, and a recall of 0.87 for label 0 and 0.85 for label 1.

6.2 Approach 2

For our second approach, consisting of **Lemmatization + GloVe + Classification (All Models)**, very surprisingly we have a complete three-way tie. The SGD, Random Forest, and HistGradientBoosting all give the same results in all metrics: f1-score of 0.90 for label 0 and 0.77 for label 1, accuracy of 0.86, precision of 0.94 for label 0 and 0.70 for label 1, recall of 0.87 for label 0 and 0.85 for label 1. All other metrics are equal.

Given this, our only option is to decide based on the model that usually generalizes better. The literature suggests that among these three, it would be HistGradientBoosting, particularly in scenarios where models face large and complex datasets. As such that is our champion model from Approach 2.

6.3 Approach 3

Our third and final approach, consisting of **Stemming + BERT + Classification (All Models)** did not finish running in time for delivery, due to the generation of the embeddings, and as such we could not see its results. This means we will decide the best of all combinations based on the other two Approaches.

7 Discussion and Results

The contenders for best model are the Lemmatized TF-IDF MLP Classifier, and the Lemmatized GloVe HistGradientBoosting Classifier. They are extremely close in terms of metrics:

	precision	recall	f1-score	support
0	0.94	0.86	0.90	1362
1	0.70	0.86	0.77	513
accuracy			0.86	1875
macro avg	0.82	0.86	0.84	1875
weighted avg	0.88	0.86	0.87	1875

Table 1: Lemmatized GloVe HistGradientBoosting Classifier

	precision	recall	f1-score	support
0	0.94	0.87	0.90	1362
1	0.70	0.85	0.77	513
accuracy			0.86	1875
macro avg	0.82	0.86	0.83	1875
weighted avg	0.87	0.86	0.86	1875

Table 2: Lemmatized GloVe HistGradientBoosting Classifier

Given that the F1 macro average and weighted average are slightly better for the MLP classifier, and also taking into consideration that in NLP tasks, MLPs usually generalize better than HistGradientBoosting due to their ability to learn from the high-dimensional and abstract nature of language data, our **final model** will be the **Lemmatized TF-IDF MLP Classifier**. We use this for our predictions.

8 Limitations and Future Improvements

We had some limitations with computational resources and could not run as many models as we wanted to. Perhaps we would have obtained better results using other Approaches.

For future improvements, other combinations of engineering and models can also be tried, grid-search can be employed to tune hyperparameters, and a heavier preprocessing might be a good bet to guarantee even better results.

9 Conclusion

We were given the task of predicting the de-listing of AirBnB properties based on a set of characteristics, mainly comments. By utilizing NLP techniques, we went all the way from raw text data to relatively good prediction scores. Our best model was found by applying Lemmatization, TF-IDF and an MLP Classifier.

Below you can find the results of all our approaches:

```
APPROACH 1

Lemmatized TF-IDF Logistic Regression Classifier


              precision    recall  f1-score   support

           0       0.94      0.87      0.90      1362
           1       0.70      0.85      0.77       513

    accuracy                           0.86      1875
   macro avg       0.82      0.86      0.83      1875
weighted avg       0.87      0.86      0.86      1875


Lemmatized TF-IDF KNN Classifier


              precision    recall  f1-score   support

           0       0.73      1.00      0.84      1362
           1       0.60      0.01      0.01       513

    accuracy                           0.73      1875
   macro avg       0.66      0.50      0.43      1875
weighted avg       0.69      0.73      0.61      1875
```

Lemmatized TF-IDF SGD Classifier

	precision	recall	f1-score	support
0	0.94	0.87	0.90	1362
1	0.70	0.85	0.77	513
accuracy			0.86	1875
macro avg	0.82	0.86	0.83	1875
weighted avg	0.87	0.86	0.86	1875

Lemmatized TF-IDF MLP Classifier

	precision	recall	f1-score	support
0	0.94	0.86	0.90	1362
1	0.70	0.86	0.77	513
accuracy			0.86	1875
macro avg	0.82	0.86	0.84	1875
weighted avg	0.88	0.86	0.87	1875

Lemmatized TF-IDF Random Forest Classifier

	precision	recall	f1-score	support
0	0.94	0.87	0.90	1362
1	0.70	0.85	0.77	513
accuracy			0.86	1875
macro avg	0.82	0.86	0.83	1875
weighted avg	0.87	0.86	0.86	1875

Lemmatized TF-IDF HistGradientBoosting Classifier

	precision	recall	f1-score	support
0	0.94	0.86	0.90	1362
1	0.70	0.85	0.77	513
accuracy			0.86	1875
macro avg	0.82	0.86	0.83	1875
weighted avg	0.87	0.86	0.86	1875

APPROACH 2

Lemmatized GloVe Logistic Regression Classifier

	precision	recall	f1-score	support
0	0.94	0.86	0.90	1362
1	0.70	0.85	0.77	513
accuracy			0.86	1875
macro avg	0.82	0.86	0.83	1875
weighted avg	0.87	0.86	0.86	1875

Lemmatized GloVe KNN Classifier

	precision	recall	f1-score	support
0	0.94	0.86	0.90	1362
1	0.70	0.85	0.77	513
accuracy			0.86	1875
macro avg	0.82	0.86	0.83	1875
weighted avg	0.87	0.86	0.86	1875

Lemmatized GloVe MLP Classifier

	precision	recall	f1-score	support
0	0.94	0.86	0.90	1362
1	0.70	0.85	0.77	513
accuracy			0.86	1875
macro avg	0.82	0.86	0.83	1875
weighted avg	0.87	0.86	0.86	1875

Lemmatized GloVe SGD Classifier

	precision	recall	f1-score	support
0	0.94	0.87	0.90	1362
1	0.70	0.85	0.77	513
accuracy			0.86	1875
macro avg	0.82	0.86	0.83	1875
weighted avg	0.87	0.86	0.86	1875

Lemmatized GloVe Random Forest Classifier

	precision	recall	f1-score	support
0	0.94	0.87	0.90	1362
1	0.70	0.85	0.77	513
accuracy			0.86	1875
macro avg	0.82	0.86	0.83	1875
weighted avg	0.87	0.86	0.86	1875

Lemmatized GloVe HistGradientBoosting Classifier

	precision	recall	f1-score	support
0	0.94	0.87	0.90	1362
1	0.70	0.85	0.77	513
accuracy			0.86	1875
macro avg	0.82	0.86	0.83	1875
weighted avg	0.87	0.86	0.86	1875