



# **SCHULICH**

---

## School of Engineering

---

**ENSF 612 - Engineering Large Scale Data Analytics Systems**

**Final Report**

**Airbnb Price Prediction with Machine Learning and Sentiment Analysis  
University of Calgary**

**Schulich School of Engineering**

**Department of Electrical and Computer Engineering**

**Kunj Patel - 30030235**

**Michael Le - 10104883**

**Jean-Charl Pretorius - 10149133**

# Table of Contents

<b>Instructions</b>	<b>2</b>
<b>Introduction</b>	<b>2</b>
<b>Motivation</b>	<b>3</b>
<b>Part I: Sentiment Analysis on Reviews</b>	<b>4</b>
Data Collection	4
Data Inspection and Validation	4
Data Filtering	5
Exploratory Data Analysis	5
Manual Labeling	5
Data Transformations	7
Model Building and Results: Sentiment Analysis	10
Model 1: PretrainedPipeline (Notebook name: 1-Pretrained Pipeline (IMDB Reviews).ipynb)	10
Model 2: Custom-built Pipeline with SentimentDLApproach (Notebook name: 2-Custom Pipeline with SentimentDLApproach Model.ipynb)	12
Model 3A: Random Forest Classifier (Notebook name: 3A-Random Forest Classifier.ipynb)	13
Model 3B: Weighted Random Forest Classifier (Notebook name: 3B-Weighted Random Forest Classifier.ipynb)	14
Model 3C: Naive Bayes Classifier (Notebook name: 3C-Naive Bayes.ipynb)	15
Model 3D: Logistic Regression Classifier (Notebook name: 3D-Logistic Regression.ipynb)	16
<b>Part II: Listing Price Prediction</b>	<b>18</b>
Data Inspection and Validation	18
Data Filtering	19
Data Transformations	20
Exploratory Data Analysis	21
Model Building and Results: Listing Price Prediction	21
Linear Regression	21
Decision Trees	23
Gradient Boosting	24
<b>Conclusion/Reflection</b>	<b>26</b>
<b>References</b>	<b>27</b>
<b>List of Figures</b>	<b>27</b>
<b>List of Tables</b>	<b>29</b>

# Instructions

In our submitted project folder, there is a folder called “notebooks”, which contains all eight final notebooks for this project. Each notebook is numbered from 0 to 4 in the order they were intended to be read in, with the notebooks 3A, 3B, 3C, and 3D labeled as such because they were originally intended to be a part of the same notebook. Due to limitations of Google Colab, we thought it would be easier to separate each of the classification models into their own notebook. Please do not run any code cells, as the intended output should already be displayed. The project folder will also contain this report and the CSV files used in this project.

## Introduction

The problem we want to solve revolves around hosts facing challenges in competitively pricing their Airbnb properties. More specifically, the business question we want to answer is: “How does the sentiment of customer reviews affect the rental value of Airbnb locations in London?”. The output variable to be predicted is the price of an Airbnb listing after including review sentiment as a feature.

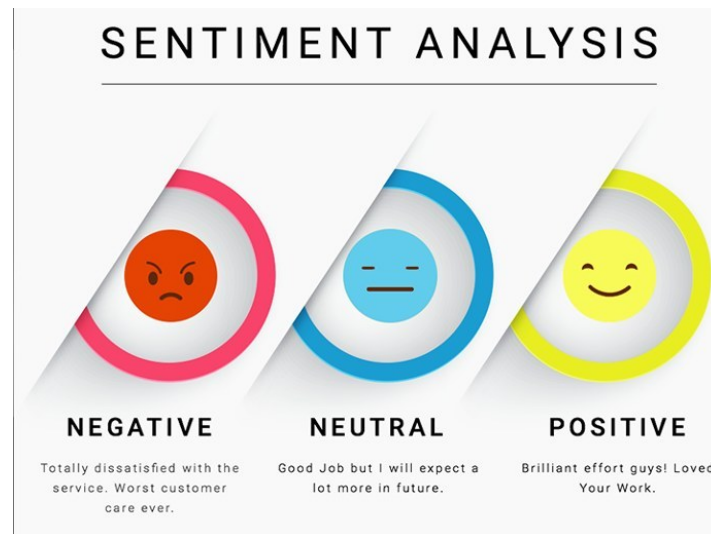
Given the high profitability of Airbnb rentals, the growing number of hosts entering the market intensifies competition. Looking to gain a competitive advantage, hosts seek a tool that can assist them in making informed decisions and reduce uncertainty when determining the rental pricing for their properties.

Our idea is that reviews submitted by previous guests play an important role in pricing the rental value of a property. These reviews capture information about the quality of the property, the communication and responsiveness of the host, and the overall guest experience. By analyzing the sentiment expressed in these reviews, we aim to extract insights into a guest’s perception of a property, creating an additional listing feature that can establish a direct link to the rental value of an Airbnb listing.

To address this challenge, we plan to leverage machine learning algorithms to analyze the large volume of data available. Through this analysis, we aim to provide hosts with accurate predictions of Airbnb prices by considering the sentiment from reviews and other relevant external factors. A predictive tool can allow hosts to set competitive prices in the rental market and assist guests in finding the best value for their money when booking through Airbnb.

# Motivation

Sentiment analysis is the process of classifying positive or negative sentiment in text. It is useful to businesses, as it assists in enabling them to understand the opinions and behaviors of customers. By analyzing the sentiment behind reviews or even social media conversations, a business will be able to make quicker and accurate decisions that are less uncertain. Sentiment analysis can be applied to any industry, including finance, retail, or hospitality.



**Figure 1:** Label Meanings in Sentiment Analysis

Airbnb is a competitive alternative to hotels for customers seeking short-term accommodation. Hosts on Airbnb face tremendous competition as they try to attract new customers and optimize their earnings, while customers are seeking the best deals. Our objective and motivation behind this project is to create a price prediction model that will provide beneficial pricing insights that will reduce the uncertainty of decision-making and pricing a rental property for Airbnb owners and guests. Our approach involves the examination of various factors, with a focus on analyzing the sentiment expressed by guests in the reviews. We believe that the sentiment expressed in a review for an Airbnb listing can be correlated to its price. We wish to explore the extent to which an overall sentiment for an Airbnb location can impact its rental pricing.

# Part I: Sentiment Analysis on Reviews

## Data Collection

The main source of data for this study was the publicly available Airbnb dataset for London, United Kingdom, obtained from Inside Airbnb, which is an independent project that provides data and advocacy about the impact of Airbnb on residential communities. As London is the most visited city in Europe, this dataset encompasses diverse information on Airbnb listings in the area, providing details about the listings, reviews, and pricing information. However, it's worth noting that this dataset lacked labeled sentiment data. Consequently, the sentiment of the reviews had to be manually labeled to integrate this essential information into our models.

## Data Inspection and Validation

The reviews CSV file is very large, containing over one million records. In order to save time and computational resources, a smaller subset of the data was selected for processing. First, a random sample containing approximately 10% of the full dataframe was extracted. This subset was inspected for missing values and duplicate rows. For the columns that are of interest to us as potential features for modeling (listing\_id, id (which refers to the reviewer ID), and comments), there were no missing values and therefore there was no need to perform imputation or removal of columns.

```
# take a subset of ~100,000 reviews for processing
sample_fraction = 0.1
reviews_df = entire_df.sample(fraction=sample_fraction, seed=42)
```

**Figure 2:** Ten Percent Subset Extraction

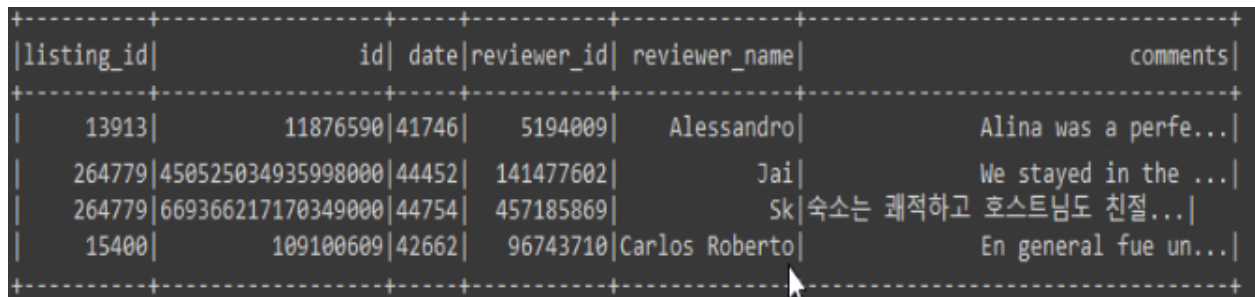
```
+-----+-----+-----+-----+-----+
|listing_id| id|date|reviewer_id|reviewer_name|comments|
+-----+-----+-----+-----+-----+
|          0| 0| 0|          0|          10|          0|
+-----+-----+-----+-----+-----+

Number of duplicate rows: 0
Number of rows: 105142
Number of columns: 6
```

**Figure 3:** Missing Values Data Inspection of Reviews

## Data Filtering

Next, any non-english reviews are filtered out using the langdetect library. Special characters and HTML tags are also removed so that the data is prepared for labeling. A 10,000 row subset is extracted from this sample for future use in our price prediction model. An additional 1000 rows were also extracted for manual labeling. The choice was made to take 20 reviews from each of the 500 listings. This would enable us to derive an average sentiment score for each listing as a feature in the price prediction model, after the sentiment analysis model has made the label predictions.



listing_id	id	date	reviewer_id	reviewer_name	comments
13913	11876590	41746	5194009	Alessandro	Alina was a perfe...
264779	450525034935998000	44452	141477602	Jai	We stayed in the ...
264779	669366217170349000	44754	457185869	Sk	숙소는 쾌적하고 호스트님도 친절...
15400	109100609	42662	96743710	Carlos Roberto	En general fue un...

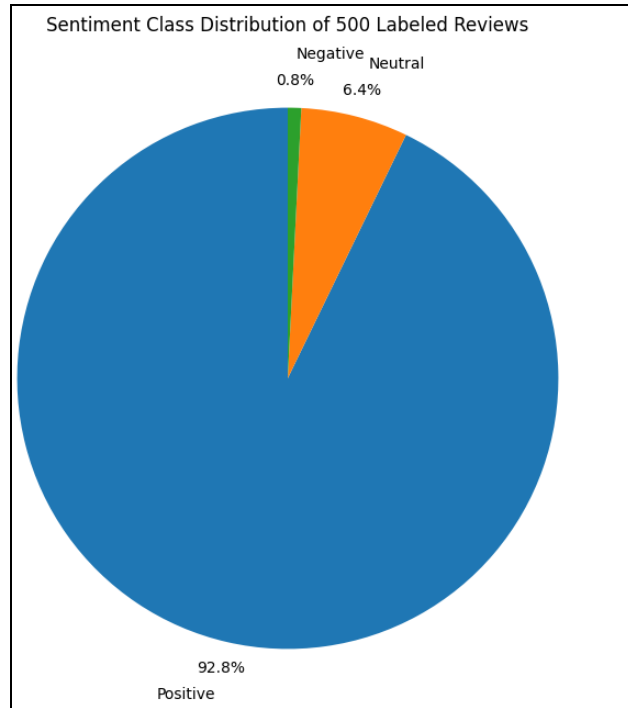
**Figure 4:** Example of Non-English Reviews

Following this we filtered out all of the html tags and some of the punctuation such as commas and newline characters so that the data could be successfully imported into Label Studio for manual labeling. We did not remove punctuation such as periods and exclamation marks because those are useful for separating sentences and maintaining the context of the text. Some sentence embedding transformers rely on these, including the Universal Sentence Encoder (USE) which takes raw text data.

## Exploratory Data Analysis

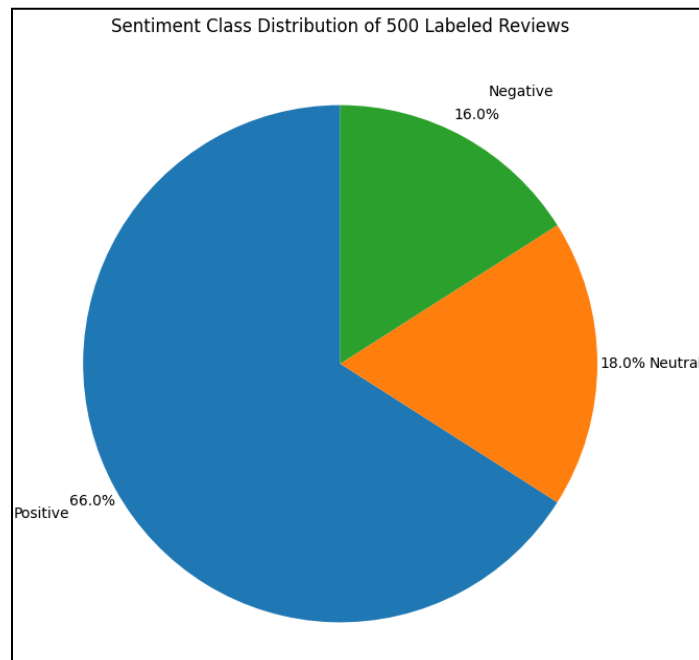
### Manual Labeling

Although a random subset of 1000 reviews was exported for manual labeling, the team decided to only label 500 of these reviews due to time constraints. Each of these 500 reviews were given a label for the overall sentiment out of three possible categories: positive, negative, or neutral. After labeling was completed, it was discovered that the London Airbnb dataset is heavily unbalanced, as there are very few negative reviews. Only 0.8% of the labeled reviews were negative and 6.4% were neutral.



**Figure 5:** Pie Chart - Sentiment Distribution of First Manually Labeled Set

Some preliminary modeling with this dataset yielded unsatisfactory results, so the team looked at the reviews dataset again to find more negative reviews. The second labeled set had 16% negative labels and 18% neutral labels, and it was this set that was eventually used to train and evaluate the sentiment analysis models.



**Figure 6:** Pie Chart - Sentiment Distribution of Second Manually Labeled Set

## Data Transformations

Throughout the sentiment analysis model building process, numerous data transformations were applied. Here, we will list and explain them.

### **Model 1: Pretrained Pipeline Trained with IMDB Reviews**

- The “comments” column in our labeled reviews DataFrame was renamed to “text”, to ensure it could be used in the Spark NLP PretrainedPipeline model.
- The manual labels and predicted labels of the reviews DataFrame after predictions were made were mapped into numeric values and separate columns, in order to evaluate model accuracy.

### **Model 2: Custom-built Pipeline with SentimentDLApproach**

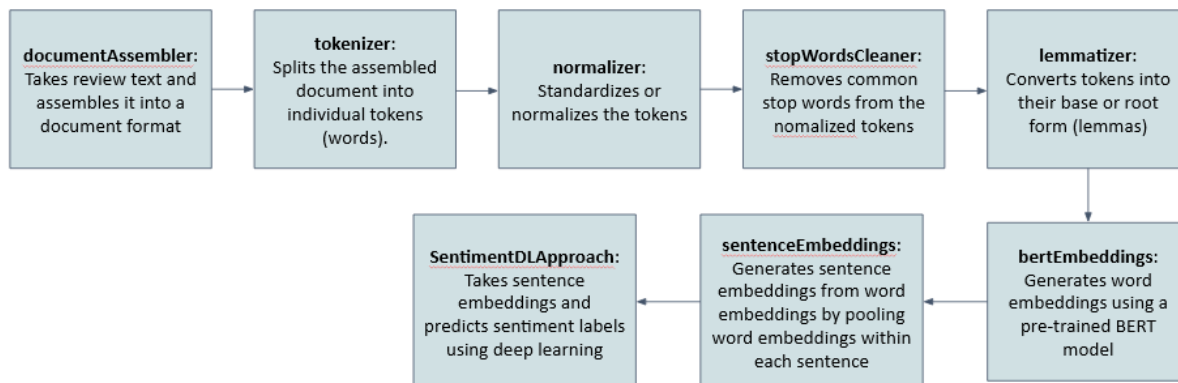
The second model we attempted during the model building step was the Spark NLP SentimentDLApproach, which was also used in our own pre-processing pipeline. SentimentDLApproach is an annotator that is used to train a deep learning model for multi-class sentiment analysis. Before training the model, the labeled dataset is loaded, converted into a DataFrame, and split into train and test sets.

```
# Split labeled data into training and testing samples
(train_dataset, test_dataset) = filtered_labeled_reviews.randomSplit([0.7, 0.3], seed=39)

print("Training data size: ", train_dataset.count())
print("Test data size: ", test_dataset.count())
```

**Figure 7: Train Test Split**

Next, a PySpark pipeline was created with several stages to preprocess and finally model the text data:



**Figure 8: Preprocessing and SentimentDLApproach Model Pipeline**



1. Document Assembly: converts raw text data into a format that is suitable for further processing by Spark NLP.
2. Tokenization: breaks down the document into individual tokens, which in this case is words.
3. Normalization: cleans the tokens from any dirty characters and converts them into a standardized form with transformations such as lowercasing.
4. Stop word cleaning: removes common stop words so that we only have the semantically relevant words.
5. Lemmatization: uses the pre-trained lemmatized “lemma\_antbnc”, which converts the tokens into their base form. (eg. “cars” -> “car”)
6. BERT word embeddings: uses the pre-trained BERT (Bidirectional Encoder Representations from Transformers) embeddings “bert\_base\_uncased” to generate contextual dense vector word representations.
7. Sentence embeddings: computes the sentence embeddings using an average pooling strategy on the BERT word embeddings in the sentence
8. SentimentDLApproach: the deep learning model for sentiment analysis that will be trained

Additionally, the neutral sentiment labels are filtered out from the dataset because SentimentDLApproach is primarily designed for binary sentiment classification (either positive or negative). An exception is generated otherwise.

In addition to these steps, this model will also use the transformations listed in model 1.

```

# Step that converts text data to a Spark NLP Format suitable for pre-processing
document_assembler = DocumentAssembler() \
    .setInputCol("text") \
    .setOutputCol("document")

# Tokenizes the documents into individual tokens (words)
tokenizer = Tokenizer() \
    .setInputCols(["document"]) \
    .setOutputCol("token")

# Converts tokens into a standardized form by applying transformations like lowercasing.
normalizer = Normalizer() \
    .setInputCols(["token"]) \
    .setOutputCol("normal")

# Removes common stop words from the tokens to reduce noise
stop_words_cleaner = StopWordsCleaner() \
    .setInputCols("normal") \
    .setOutputCol("cleanTokens") \
    .setCaseSensitive(False)

# Performs lemmatization on the clean tokens, converting them to their base form
lemmatizer = LemmatizerModel.pretrained("lemma_antbnc") \
    .setInputCols(["cleanTokens"]) \
    .setOutputCol("lemma")

# Loads pre-trained BERT word embeddings to generate contextual word representations
bert_embeddings = BertEmbeddings().pretrained('bert_base_uncased', 'en') \
    .setInputCols(["document", "lemma"]) \
    .setOutputCol("bert") \
    .setCaseSensitive(False)

# Computes sentence embeddings using the BERT embeddings
sentence_embeddings = SentenceEmbeddings() \
    .setInputCols(["document", "bert"]) \
    .setOutputCol("sentence_embeddings") \
    .setPoolingStrategy("AVERAGE")

# Sentiment analysis component using deep learning
sentiment_dl = SentimentDLApproach() \
    .setInputCols(["sentence_embeddings"]) \
    .setOutputCol("prediction") \
    .setLabelColumn("label") \
    .setMaxEpochs(5) \
    .setEnableOutputLogs(True)

# Combine all the specified stages in sequence to form a processing pipeline
sentiment_model = Pipeline().setStages([
    document_assembler,
    tokenizer,
    normalizer,
    stop_words_cleaner,
    lemmatizer,
    bert_embeddings,
    sentence_embeddings,
    sentiment_dl
])

```

**Figure 9:** SentimentDLApproach Pipeline

### **Model 3A/3B/3C/3D: Random Forest Classifier, Naive Bayes, and Logistic Regression Models**

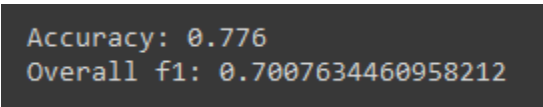
- In these four models, a function `custom preprocess_text` is created and used to preprocess text for a review by tokenizing it, filtering out stopwords, removing non-alphabetic words and punctuation marks, correct misspelled words, and stemming each word to create a new array of words
- HashingTF is used to convert the array of processed text words to a features array. The DataFrame is subsequently split into training and testing splits.
- Additionally, the transformations mentioned in Model 1 are also used.

## **Model Building and Results: Sentiment Analysis**

Before deciding on a model to predict sentiment labels on the subset of 10000 reviews, we evaluate the performance of six different models on a Google Colab environment (which has the PySpark and Spark NLP packages installed) and choose the best one. Because computational resources are limited in this environment, separate notebooks are created for each model and will be submitted as part of the final deliverable.

### **Model 1: PretrainedPipeline (Notebook name: 1-Pretrained Pipeline (IMDB Reviews).ipynb)**

The first model features the PretrainedPipeline from Spark NLP for sentiment analysis. The dataset containing 500 labeled Airbnb reviews is loaded and inspected for data quality, revealing insights into the distribution of sentiment classes. The sentiment analysis is performed using a pre-trained model that has been trained on IMDB reviews and uses the Universal Sentence Encoder embeddings. The model tends to predict reviews as either positive or negative, with minimal neutral predictions. The overall prediction accuracy on the dataset is 77.60%. Although a higher accuracy would have been preferable, it is still quite reasonable - especially considering that this is a pre-trained model designed for a different domain (IMDB movie reviews). The overall F1 Score of 0.70 also falls in line with what is generally considered a good F1 score.



```
Accuracy: 0.776  
Overall f1: 0.7007634460958212
```

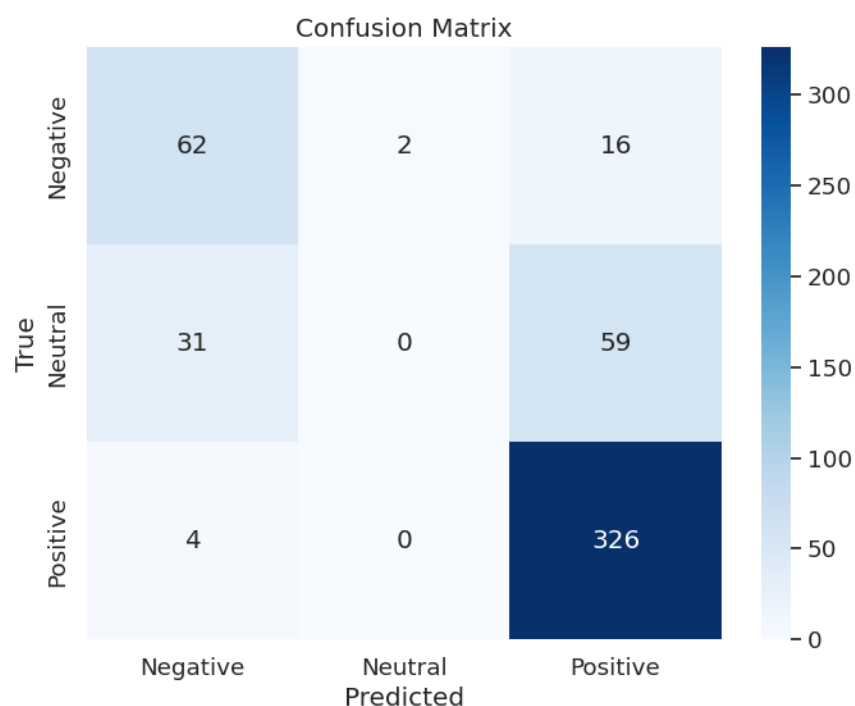
**Figure 10:** Pretrained Sentiment Pipeline Accuracy and F1 Score

The confusion matrix below demonstrates that this model performed quite well for predicting Negative and Positive classes, as shown with the high precision and recall scores. All predicted Negative reviews were actually negative, and all predicted positive reviews were actually

positive. Of all actual negative reviews, 77.5% were correctly predicted as negative and 98.8% of all actual positive reviews were correctly predicted as positive. However, the model performs poorly for the Neutral class as it fails to predict any Neutral reviews correctly, resulting in a precision, recall, and F1 score of 0.

```
Metrics for negative: Precision=1.0, Recall=0.775, F1 Score=0.873
Metrics for neutral: Precision=0.0, Recall=0.0, F1 Score=0.0
Metrics for positive: Precision=1.0, Recall=0.988, F1 Score=0.994
```

**Figure 11:** Pretrained Sentiment Pipeline Individual Metrics



**Figure 12:** Pretrained Sentiment Pipeline Confusion Matrix

The code segment below details what the PretrainedPipeline actually looks like underneath the hood. The embeddings specified below should be the same ones used when using this pre-trained model.

```

document = DocumentAssembler()\
    .setInputCol("text")\
    .setOutputCol("document")

use = UniversalSentenceEncoder.pretrained("tfhub_use", "en") \
    .setInputCols(["document"])\
    .setOutputCol("sentence_embeddings")

sentimentdl = SentimentDLModel().pretrained("sentimentdl_use_imdb") \
    .setInputCols(["sentence_embeddings"]) \
    .setThreshold(0.6F) # minimum threshold or result will be neutral
    .setOutputCol("prediction")

pipeline = Pipeline(
    stages = [
        document,
        use,
        sentimentdl
    ])

```

**Figure 13:** Equivalent Model to PretrainedPipeline('analyze\_sentimentdl\_use\_imdb')

## **Model 2: Custom-built Pipeline with SentimentDLApproach (Notebook name: 2-Custom Pipeline with SentimentDLApproach Model.ipynb)**

The second model demonstrates the construction and evaluation of a sentiment analysis model using a custom-built pipeline with the SentimentDLApproach annotator from Spark NLP. The dataset consisting of 500 labeled Airbnb reviews is loaded and preprocessed. The neutral sentiment labels are filtered out from the dataset because SentimentDLApproach is primarily designed for binary sentiment classification (either positive or negative). The sentiment analysis pipeline incorporates various NLP components, including tokenization, normalization, stop words removal, lemmatization, and the use of pre-trained BERT embeddings.

The model is trained using the filtered training dataset, and predictions are made on the test dataset, resulting in a test accuracy of 87.87%. Although this test accuracy is considerably higher than the test accuracy of 78.14% in the Pretrained Pipeline, the evaluation process reveals a significant concern regarding the model's behavior, as it predicted all samples, including negative ones, as positive.

```

Accuracy on the test set: 0.8787878787878788
Testing F1 Score: 0.8220918866080157

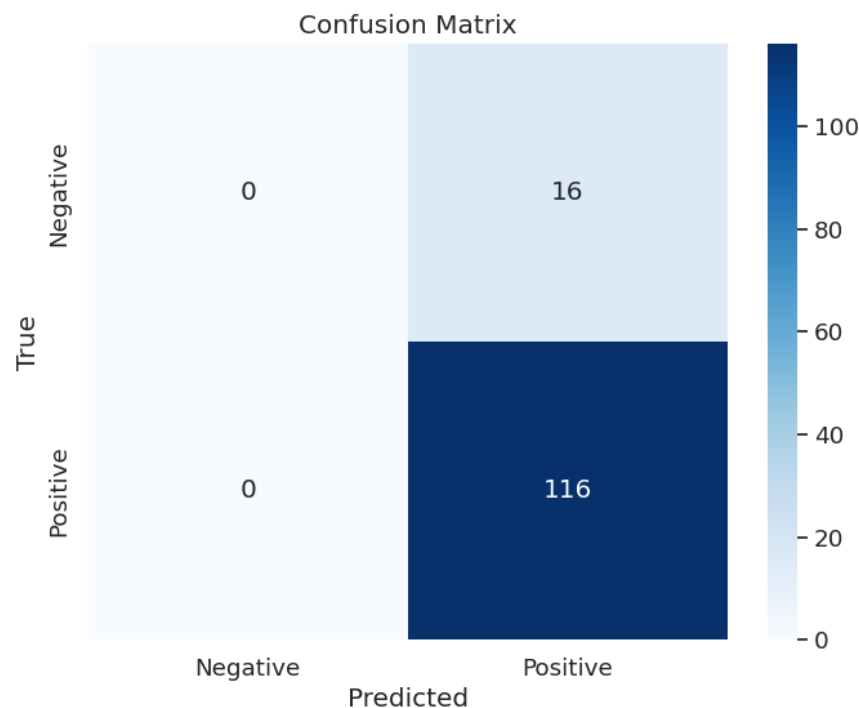
```

**Figure 14:** SentimentDLApproach Accuracy and F1 Score

The F1 score, a metric suitable for evaluating the performance of a model when there is an imbalanced dataset, is calculated to be 0.82. A general rule of thumb is that an F1 score of 0.7 or higher is generally an indicator of good overall model performance. However, it is still concerning that no negative labels were predicted at all in our dataset, and we must consider the consequences of misclassification in this case - a negative review being labeled as positive is not ideal if we want to use predicted sentiment as a feature for our price prediction model later on. Misclassifying negative reviews as positive underscores the importance of thorough model evaluation and highlights the need for further investigation, refinement, and exploration of more models to ensure robust performance across all sentiment classes.

```
Precision for negative: 0.000  
Recall for negative: 0.000  
Precision for positive: 0.879  
Recall for positive: 1.000
```

**Figure 15:** SentimentDLApproach Individual Metrics



**Figure 16:** SentimentDLApproach Confusion Matrix

This confusion matrix indicates that among all the predicted positive instances, 87.9% were actually positive. The model correctly identified all the actual positive instances, indicating that every positive instance was predicted correctly. No true negatives were predicted among the total or actual negatives.

Next, we will explore the effectiveness of classification models learned in 612.

### **Model 3A: Random Forest Classifier (Notebook name: 3A-Random Forest Classifier.ipynb)**

The model third building process involved using a regular Random Forest Classifier for sentiment analysis on a dataset of 500 labeled reviews. After loading the labeled review data as a DataFrame, the preprocessing steps included tokenization, removal of stopwords, spell checking, and stemming. The sentiment labels were converted into a numeric format for model training. The dataset was split into training and testing sets, and a HashingTF object was applied to transform the processed text into a numerical features array. The RandomForestClassifier was trained on the training dataset, and predictions were made on the test dataset.

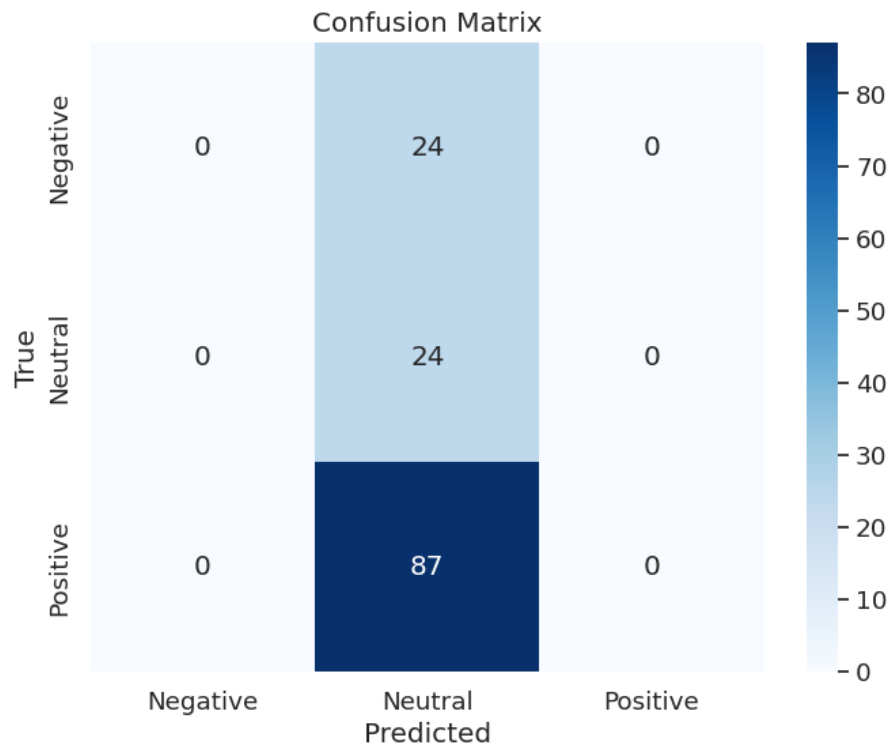
```
Accuracy: 0.177778  
F1 Score: 0.05366876310272537
```

**Figure 17:** Random Forest Classifier Accuracy and F1 Score

However, the results of the regular Random Forest Classifier were poor. The accuracy was 17.78% (far lower than models 1 and 2), and the F1 score, a metric that accounts for both precision and recall, was 0.05. The F1 score suggests the poor model performance potentially stems from imbalanced or insufficient data, as the dataset had a significantly higher number of positive labels compared to neutral or negative ones.

```
Precision for negative: 0.000  
Recall for negative: 0.000  
Precision for neutral: 0.178  
Recall for neutral: 1.000  
Precision for positive: 0.000  
Recall for positive: 0.000
```

**Figure 18:** Random Forest Classifier Individual Class Metrics



**Figure 19:** Random Forest Classifier Confusion Matrix

The confusion matrix revealed that the model failed to make correct predictions for both negative and positive classes, while having a high recall but low precision for the neutral class.

The results of this model brought up the following question: what if we used a Weighted Random Forest Classification to increase the weights of the less frequently appearing classes in our dataset?

### **Model 3B: Weighted Random Forest Classifier (Notebook name: 3B-Weighted Random Forest Classifier.ipynb)**

The fourth model uses a Weighted Random Forest Classifier for sentiment analysis on a dataset of 500 labeled reviews, to see if there is any improvement from the regular Random Forest model in model 3A. Like the regular Random Forest Classifier, preprocessing steps were applied to the review data, including tokenization, removal of stopwords, spell checking, and stemming. The sentiment labels were converted into numeric format for model training. The dataset was then split into training and testing sets, and a HashingTF object was utilized to transform the processed text into an array of numerical features. The Weighted Random Forest Classifier was employed to address the imbalanced nature of the dataset by assigning higher weights to less frequent classes, which would be negative and neutral sentiments for our dataset. This was done by dividing total number of reviews (500) by the product of the number of classes (3) and the



number of positive, negative, or neutral labels, giving us a positive weight, negative weight, neutral weight that can be added as a new feature to each row in the DataFrame depending on their manual label.

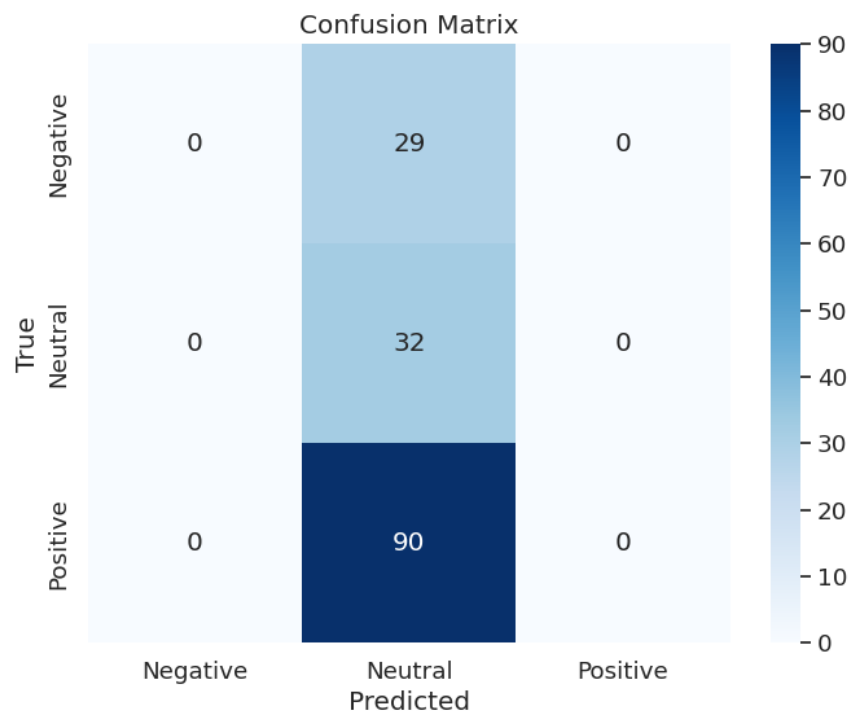
```
Accuracy: 0.211921  
F1 Score: 0.07411428364636485
```

**Figure 20:** Weighted Random Forest Classifier Accuracy and F1 Score

After training this new Random Forest model and predictions were made on the test dataset, the weighted random forest classification model showed a slightly improved performance at 21.19 % compared to the 17.78% accuracy from the regular random forest classification model, and a slightly improved F1 score of 0.07 compared to the 0.05 of the regular random forest model. However, the results are still quite poor overall.

```
Precision for negative: 0.000  
Recall for negative: 0.000  
Precision for neutral: 0.212  
Recall for neutral: 1.000  
Precision for positive: 0.000  
Recall for positive: 0.000
```

**Figure 21:** Weighted Random Forest Classifier Individual Class Metrics



**Figure 22:** Weighted Random Forest Classifier Confusion Matrix

Once again, the confusion matrix revealed that the model failed to make correct predictions for both negative and positive classes, while having a high recall but low precision for the neutral class. The neutral class having a low precision score of 0.212 is indicative of having more false positives compared to true positives. The high recall score of 1 for the neutral class indicates that when the actual class is neutral, the model correctly identifies most of them.

### **Model 3C: Naive Bayes Classifier (Notebook name: 3C-Naive Bayes.ipynb)**

The fifth implementation was of the Naive Bayes classification model, the process starts with data preprocessing and exploration, similar to the previous models. The labeled reviews dataset is loaded and the distribution of sentiment classes is visualized through bar and pie charts. Text preprocessing involves tokenization, removal of stopwords, punctuation, non-alphabetic words, and stemming. The PySpark ML Naive Bayes classifier is then trained on the processed data using a Multinomial model type, which is suitable for classification with discrete features. The model is evaluated on the test dataset, and performance metrics like accuracy and F1 score, are calculated. Finally, a confusion matrix is presented, detailing the precision and recall for each sentiment class.

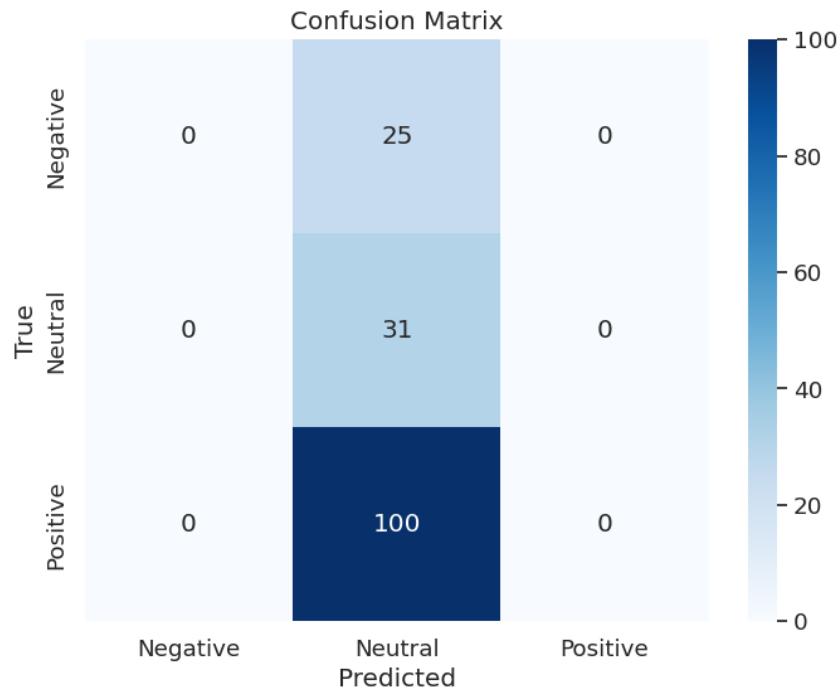
```
Accuracy: 0.198718  
F1 Score: 0.06588509529686
```

**Figure 23:** Naive Bayes Classifier Accuracy and F1 Score

The results indicate that the Naive Bayes model achieved an accuracy of 19.87% and an F1 score of 0.065. These metrics fall in between the performance of the weighted random forest model (21.19% accuracy, F1 score: 0.074) and the regular random forest model (17.78% accuracy, F1 score: 0.05).

```
Precision for negative: 0.000  
Recall for negative: 0.000  
Precision for neutral: 0.199  
Recall for neutral: 1.000  
Precision for positive: 0.000  
Recall for positive: 0.000
```

**Figure 24:** Naive Bayes Classifier Individual Class Metrics



**Figure 25: Naive Bayes Classifier Confusion Matrix**

The confusion matrix also speaks to the poor performance of the model, as the precision and recall scores of 0 for the negative and positive classes indicate that the model didn't make any correct predictions for both the negative and positive classes.. The neutral class having a low precision score of 0.199 is indicative of having more false positives compared to true positives. The high recall score of 1 for the neutral class indicates that when the actual class is neutral, the model correctly identifies most of them. Overall, the Naive Bayes model exhibits poor performance, aligning with the trends observed in the previous models.

### **Model 3D: Logistic Regression Classifier (Notebook name: 3D-Logistic Regression.ipynb)**

Finally, we will look at the logistic regression model, before deciding which model to use for predicting our sentiments before creating our price prediction model.

The sixth implementation was a Logistic Regression classification model for sentiment analysis. The sentiment labels are converted into a numerical format, and text preprocessing is performed to prepare the text for sentiment analysis. The dataset was then split into training and testing sets, and a HashingTF object was utilized to transform the processed text into an array of numerical features. The model is trained using Logistic Regression with the memory-efficient LBFGS (Limited-memory Broyden–Fletcher–Goldfarb–Shanno) optimization algorithm which is popular for parameter estimation in machine learning, and predictions are made on the test dataset.

The performance is then evaluated using metrics such as accuracy and F1 score, and a confusion matrix is generated to assess precision and recall for each sentiment class.

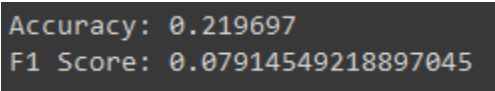


Figure 26: Logistic Regression Classifier Accuracy and F1 Score

The Logistic Regression model exhibits an accuracy of 21.97%, which is marginally better than the performance of the regular (17.78%) and weighted (21.19%) Random Forest and Naive Bayes (19.87%) models. The F1 score of 0.079 is also the best value compared to the Random Forest models and Naive Bayes model. However, these metrics for this model are only slightly better than those of the other models, and do not demonstrate any significant improvement.

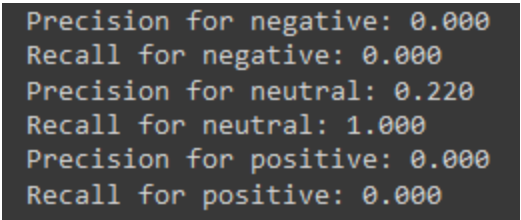


Figure 27: Logistic Regression Classifier Individual Class Metrics

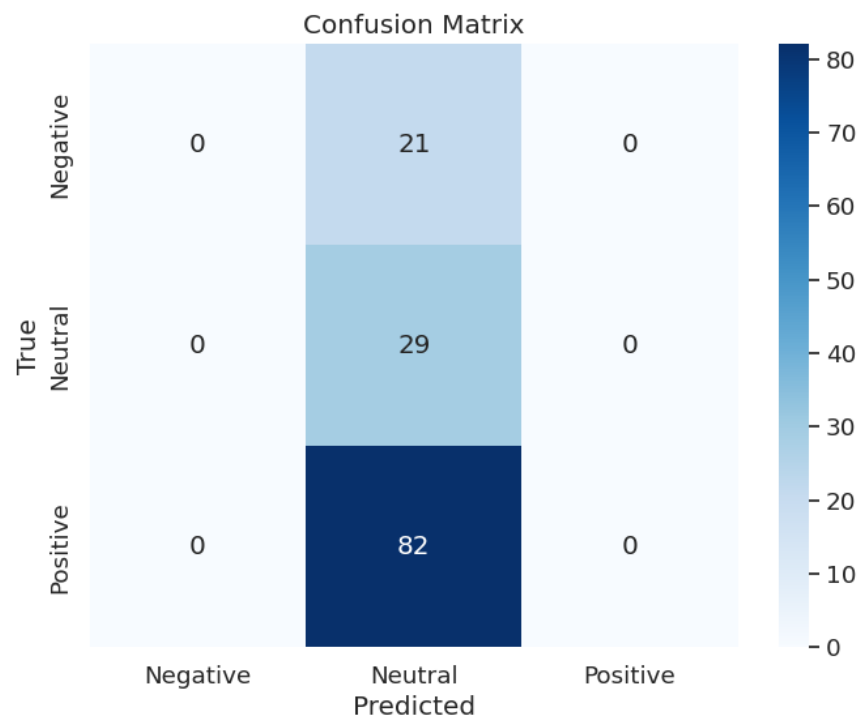


Figure 28: Logistic Regression Classifier Confusion Matrix

The confusion matrix further reveals challenges, with precision and recall scores of 0 for negative and positive classes, suggesting that the model struggles to make correct predictions for these sentiments. While the neutral class shows a higher precision, it still suffers from a large number of false positives compared to true positives. Although the results for Logistic Regression show slight improvement compared to the previous three models, the results are still quite poor and fall in line with their performance and remain unusable for sentiment classification.

**Table 1:** Summary of Sentiment Analysis Model Results

Model	Test Accuracy (%)	F1 Score
PreTrainedPipeline (IMDB Reviews)	77.60%	0.70
Custom Pipeline with SentimentDLApproach Annotator	87.87%	0.82
Regular Random Forest	17.78%	0.054
Weighted Random Forest	21.19%	0.074
Multinomial Naive Bayes	19.87%	0.066
Logistic Regression	21.97%	0.079

With the Random Forest, Naive Bayes, and Logistic Regression models, these are simpler models compared to deep learning models that leverage pre-trained language models when it comes to learning contexts in words and sentences. It is not surprising that the last four models performed poorly compared to the first two.

The decision is ultimately made to use a pre-trained Spark NLP pipeline from our first notebook, "1-Pretrained Pipeline (IMDB Reviews).ipynb," for sentiment prediction, as it demonstrates better accuracy and the least amount of issues. The choice is driven by the pretrained model's 77.60% accuracy and F1 score of 0.70, while also demonstrating that it is also capable of predicting negative and neutral labels.

We chose not to use the custom pipeline with the SentimentDLApproach model because although the accuracy and F1 scores are good, it does not predict any negative reviews.

## Part II: Listing Price Prediction

For the listings price prediction part of this project, we will introduce a second dataset from Inside Airbnb that has information about all of the Airbnb listings in London, where we want to add the average predicted sentiment as a feature after applying the pre-trained pipeline model to our dataset of 10000 reviews.

### Data Inspection and Validation

The listings data and the subset of ~10,000 selected reviews we extracted earlier are loaded and the two DataFrames are inspected for their number of records and missing values. There are 87,946 Airbnb listings in London and we selected 9,998 reviews. Out of the 75 columns in the listings dataset, many of them have a large number of missing values, and after careful examination and application of our domain knowledge, we selected what we considered the most important features for predicting prices. The features that were selected are the neighborhood name, room type, number of beds, days of availability in a year, and number of reviews. There were features which could potentially have been useful such as the number of bedrooms and the number of bathrooms, but the majority of the records have missing values for these features. The beds column had 1134 missing values, so to fix this we applied an imputer to replace them with the median value in that column.

```
number of listings: 87946
number of reviews: 9998
```

**Figure 29:** Total Number of Listings and Reviews

```
+-----+-----+-----+-----+-----+-----+-----+
|listing_id| id|date|reviewer_id|reviewer_name|review_num|comments|
+-----+-----+-----+-----+-----+-----+-----+
|          0| 0| 0|          0|          1|          0|          0|
+-----+-----+-----+-----+-----+-----+-----+
```

**Figure 30:** Missing Values in Selected Reviews

```
Number of columns: 75
Number of missing values in each column:
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id|listing_url|scrape_id|last_scraped|source|name|description|neighborhood_overview|picture_url|host_id|host_url|ho
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 0|          0|          0|          0| 0| 0|          1267|          40755|          3| 0|          0|
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
```

**Figure 31:** Missing Values in Listings Dataset (Columns Truncated)

id	price	neighbourhood_name	room_type	beds	availability_365	number_of_reviews
0	0	0	0	1134	0	0

**Figure 32:** Missing Values in Selected Features from the Listings Dataset

## Data Filtering

In order to use the predicted sentiment as a feature in our price prediction models, these labels need to be first converted into a numeric representation. The labels are mapped to values where -1 is negative, 0 is neutral, and 1 is positive. Next, by grouping the reviews by listing\_id and aggregating the numeric representation of sentiment, we can compute the mean sentiment for each listing. This DataFrame is then combined with the listings DataFrame using an inner join on the listing id. After renaming some columns to make the names more intuitive, removing the dollar signs from the price, and selecting the desired features, we now have the basic features that will be used for the price prediction model.

id	price	neighbourhood_name	room_type	number_of_reviews	availability_365	mean_prediction
92644	42	Wandsworth	Private room	216	217	1.0
102565	120	Lambeth	Private room	374	270	1.0
36660	82	Haringey	Private room	639	186	1.0
107051	221	Westminster	Entire home/apt	655	219	0.9
42010	65	Barnet	Private room	549	214	1.0
43129	60	Hammersmith and F...	Private room	234	347	1.0
46992	58	Hackney	Private room	225	0	1.0
47192	54	Camden	Private room	443	267	1.0
157714	69	Hammersmith and F...	Private room	179	62	0.8
182802	68	Richmond upon Thames	Private room	256	349	0.9
183128	92	Southwark	Private room	190	64	1.0
216350	84	Hammersmith and F...	Private room	308	354	1.0
84223	65	Southwark	Private room	401	331	1.0
219665	60	Southwark	Private room	149	21	1.0
85191	75	Lambeth	Private room	517	325	0.9
92399	94	Islington	Entire home/apt	281	319	1.0
236956	62	Lambeth	Private room	575	314	1.0
254996	40	Ealing	Private room	517	294	1.0
268398	55	Lambeth	Private room	529	311	1.0
306677	132	Southwark	Entire home/apt	218	0	1.0

**Figure 33:** Listings Features to be used for the Price Prediction Model

## Data Transformations

Before we can use this data in our models, we need to transform the categorical features into numeric features. The one-hot encoder is suitable for this purpose, as it converts the categories into a binary representation after the application of a string indexer.

Some models such as linear regression can benefit from scaling the numeric features, so that all features contribute equally to the model's learning process. The Standard scaler is applied to the numeric features, which transforms the data such that it has a mean of 0 and a standard deviation of 1. A vector assembler is used to combine all of the numeric features into a single vector column, and then a final vector assembler stage combines all of the numeric and encoded features into a single vector feature. All of these pre-processing steps are combined into a PySpark pipeline. This pipeline then gets fit on our training data, and then it is used to transform both the training and testing data sets.

```
# Randomly split the df_processed_with_hash into training (70%) and testing sets (30%)
train, test = combined_df.randomSplit([0.70, 0.30], seed=39)

# Fit the pipeline on the training set and use it to transform both training and test set
pipeline_model = preprocessing_pipeline.fit(train)

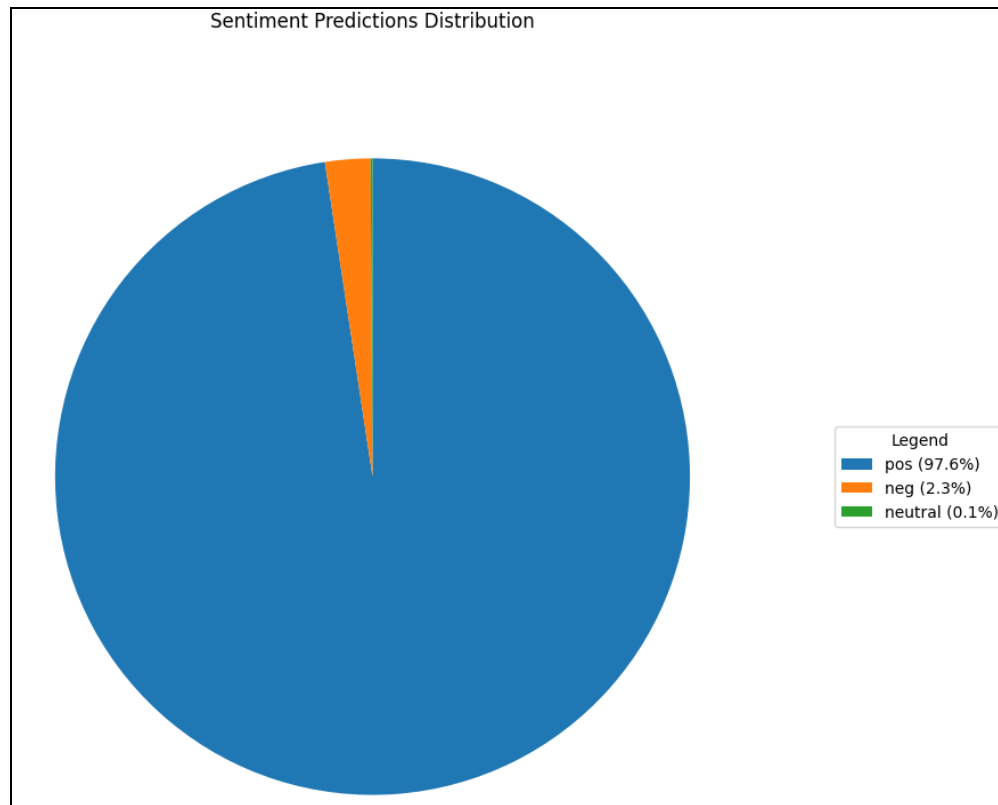
# Transform training and test sets
train_preprocessed = pipeline_model.transform(train)
test_preprocessed = pipeline_model.transform(test)
```

**Figure 34:** Split Data, Fit, and Transform on the Pre-Processing Pipeline

## Exploratory Data Analysis

Next, we load the sentiment analysis model we chose, which is the pre-trained sentiment analysis pipeline from Spark NLP that was trained on IMDB movie reviews. This model is used to transform the 9998 selected reviews to make predictions, and as expected, the vast majority of reviews are predicted as positive.





**Figure 35:** Selected Reviews Sentiment Prediction Distribution

## Model Building and Results: Listing Price Prediction

### Linear Regression

The first model used for price prediction was a Linear Regression model, which uses a straightforward approach of modeling the relationship between the target variable (the listing price) and the input features as a linear equation. This assumes a linear relationship between the features, which may be overly simplistic for our data.

The left side of Figure 28 below highlights the performance metrics of our Linear Regression model without any hyperparameter tuning. The testing RMSE is higher than the training RMSE, suggesting some degree of overfitting or a less accurate model on unseen data. It is encouraging that the two values are not drastically different though. The R-squared value also attests to the overfitting and variance, as the testing R-squared being lower than the training R-squared indicates that the model may not generalize as well to unseen data.

Root Mean Squared Error (RMSE): Training RMSE: 37.3470457918274 Testing RMSE: 43.676731992937874	Root Mean Squared Error (RMSE): Training RMSE: 37.906486518757596 Testing RMSE: 44.916637160136695
Mean Squared Error (MSE): Training MSE: 1394.8018293768528 Testing MSE: 1907.656917582923	Mean Squared Error (MSE): Training MSE: 1436.9017201967515 Testing MSE: 2017.5042937753728
Mean Absolute Error (MAE): Training MAE: 26.762636601488854 Testing MAE: 27.92936338889533	Mean Absolute Error (MAE): Training MAE: 26.895036897249447 Testing MAE: 28.365622067567045
R-squared ( $R^2$ ): Training $R^2$ : 0.5967494113049975 Testing $R^2$ : 0.5804907480337824	R-squared ( $R^2$ ): Training $R^2$ : 0.5845779290201598 Testing $R^2$ : 0.5563344177250107

**Figure 36:** Linear Regression Performance Metrics before (left) and after Hyperparameter Tuning (right)

In an attempt to get a better model, cross-validation was used to tune the hyperparameters. The regularization parameter is used to control the strength of the regularization between 0 and 1, where a larger value indicates a greater amount of regularization. The elastic net parameter is also a regularization parameter which combines the L1 (lasso) and L2 (ridge regression) regularization penalties. The maximum iterations is what sets the max number of iterations allowed to run the optimization algorithm, where a higher value will give the model a better chance of converging.

```
paramGrid = ParamGridBuilder() \
    .addGrid(lr.regParam, [ 0.0, 0.01, 0.1, 1.0]) \
    .addGrid(lr.elasticNetParam, [0.0, 0.3, 0.7, 1.0]) \
    .addGrid(lr.maxIter, [10, 50, 100]) \
    .build()
```

**Figure 37:** Linear Regression Hyperparameters Tuned

The best parameters for the linear regression model were found to be a regularization parameter of 0.1, an elastic net parameter of 0.7, and the maximum iteration set to 10.

Unfortunately, the hyperparameter tuning did not have the desired effect, as both the training and testing RMSE both slightly increased. It's not always guaranteed that tuning will always lead to better results, especially if the original model is already performing reasonably well or when the effects of the hyperparameters are limited. Given that the changes in RMSE are slight, it's

possible that the original configuration of the linear regression model was already optimal. The default parameters in the original model were included in the parameter grid during tuning, so it is unknown why the best tuned model performed worse.

## Decision Trees

Decision trees are models that recursively partition the features into segments creating a tree structure of decisions based on feature conditions. They can be prone to overfitting, so the hyperparameters are tuned to control the model complexity. The `maxDepth` sets the limit on how many layers deep the tree can be, where a deeper tree is a more complex model. The `minInstancesPerNode` sets the minimum number of data points required to create a node, and a higher value will reduce model complexity and overfitting. The `minInfoGain` specifies the minimum information gain required for a split in the tree to occur. A higher value will result in fewer splits and splits that are more informative for the model.

```
# Parameter Grid
paramGrid = (ParamGridBuilder()
             .addGrid(dt.maxDepth, [5, 10, 15])
             .addGrid(dt.minInstancesPerNode, [1, 5, 10])
             .addGrid(dt.minInfoGain, [0.0, 0.1, 0.2])
             .build())
```

**Figure 38:** Decision Tree Hyperparameters Tuned

Root Mean Squared Error (RMSE): Training RMSE: 30.900950867497436 Testing RMSE: 53.54717166402574	Root Mean Squared Error (RMSE): Training RMSE: 38.1098088645374 Testing RMSE: 51.71463245139917
Mean Squared Error (MSE): Training MSE: 954.8687645154905 Testing MSE: 2867.299593216641	Mean Squared Error (MSE): Training MSE: 1452.3575316915735 Testing MSE: 2674.403209583308
Mean Absolute Error (MAE): Training MAE: 22.519602616675886 Testing MAE: 33.09247410986642	Mean Absolute Error (MAE): Training MAE: 25.09710504434003 Testing MAE: 31.951784023091545
R-squared ( $R^2$ ): Training $R^2$ : 0.7239382804728838 Testing $R^2$ : 0.36945752853850533	R-squared ( $R^2$ ): Training $R^2$ : 0.5801095056550782 Testing $R^2$ : 0.4118770101859387

**Figure 39:** Decision Tree Performance Metrics before (left) and after Hyperparameter Tuning (right)

After hyperparameter tuning was performed, the gap between the training and testing RMSE for the tuned Decision Tree model has decreased, which resulted from the training RMSE increasing by nearly 8 and the testing RMSE decreasing by nearly 2. There is now less overfitting than the untuned model.

The best parameters for the decision tree model was a max depth of 5, the minimum instances per node of 10, and the minimum info gain of 0.0.

## Gradient Boosting

Gradient Boosting is an ensemble method that combines multiple, usually shallow, decision trees into a single model. The idea is that the weakness of one of the trees will be covered by other trees in the ensemble. The three main hyperparameters we tuned are the maximum depth, the number of trees (maxIter), and the learning rate (stepSize).

```
# Parameter Grid
paramGrid = (ParamGridBuilder()
             .addGrid(gb.maxDepth, [3, 5, 7])           # maximum depth of tree
             .addGrid(gb.maxIter, [10, 50, 100])       # number of trees
             .addGrid(gb.stepSize, [0.05, 0.1, 0.2])   # learning rate
             .build())
```

**Figure 40:** Gradient Boosting Hyperparameters Tuned

Root Mean Squared Error (RMSE): Training RMSE: 24.859822187788993 Testing RMSE: 52.27963251959272	Root Mean Squared Error (RMSE): Training RMSE: 36.24323526851462 Testing RMSE: 50.165641385314125
Mean Squared Error (MSE): Training MSE: 618.010759208486 Testing MSE: 2733.159976383656	Mean Squared Error (MSE): Training MSE: 1313.5721027289017 Testing MSE: 2516.5915755999413
Mean Absolute Error (MAE): Training MAE: 18.449832226113518 Testing MAE: 31.732369840722924	Mean Absolute Error (MAE): Training MAE: 25.724827506849888 Testing MAE: 30.012965914002027
R-squared ( $R^2$ ): Training $R^2$ : 0.8213271611623804 Testing $R^2$ : 0.39895591988863366	R-squared ( $R^2$ ): Training $R^2$ : 0.6202337044858823 Testing $R^2$ : 0.4465810703938984

**Figure 41:** Gradient Boosting Performance Metrics before (left) and after Hyperparameter Tuning (right)

After hyperparameter tuning was performed, the gap between the training and testing RMSE for the tuned Gradient Boosting model has decreased, which resulted from the training RMSE increasing by nearly 12 and the testing RMSE decreasing by nearly 2. There is now less overfitting than the untuned model.

The best parameters for the Gradient Boosting model was a max depth of 3, a max number of trees (maxIter) of 10, and a learning rate (stepSize) of 0.05.

**Table 2:** Summary of Price Prediction Model Results

Model	Testing RMSE (after hypertuning)
Linear Regression	44.917
Decision Trees	51.715
Gradient Boosting	50.166

Comparing the performance of the three regression models, it is clear that the tuned linear regression model has the best performance with a testing RMSE of 44.917. Although hyperparameter tuning actually made the performance of the linear regression model slightly worse, it was clear that this was the best model for price prediction because there was less overfitting compared to the other two models, even though the other two models saw improved performance after hypertuning.

## Conclusion/Reflection

The goal of this project was to predict the prices of Airbnb locations in London, after performing sentiment analysis on customer reviews and using the average sentiment for a listing as a new feature for price prediction. The motivation was to explore the extent to which an overall sentiment for an Airbnb location impacted its rental pricing. Deriving the average predicted sentiment for a listing and using it as a feature did have a significant effect on the pricing, as the untuned and tuned Linear Regression models produced testing RMSEs of around 43-44, and the untuned and tuned Decision Tree and Gradient Boosting models produced RMSEs of around 50-53. This represents a considerable difference between the actual prices used and the predicted prices, and if someone were to pursue this project further, there should be an emphasis on allocating more time and resources to perform the computations necessary to handle more data. Using 500 manually labeled reviews to train the failed sentiment analysis models did not feel adequate enough for creating a proper sentiment analysis model, but

unfortunately the time limitations and computational limitations of Google Colab made it necessary.

In terms of the models used, we enjoyed getting the opportunity to use the Random Forest, Naive Bayes, Logistic Regression models we learned in class to classify review sentiments. It was also interesting to collect feedback from our instructor and apply it to learn about and use pre-trained models from Spark NLP and incorporate them into our own data processing pipelines in a PySpark environment for big data problems. In hindsight, it was not surprising that the Random Forest, Naive Bayes, and Logistic Regression models produced such poor results, as they are simpler models compared to the deep learning used in the pre-trained model and the SentimentDLApproach annotator, when it comes to learning intricate contexts in words and sentences. For the regression models used in price prediction, it was interesting that the Linear Regression model performed the best, and demonstrated the least amount of overfitting compared to the Decision Tree and Gradient Boosting models. Although hypertuning the hyperparameters of the Linear Regression model resulted in slightly worse performance and improved the the performance of the other two models, Linear Regression still stood as the best model for price prediction.

Given more time, exploring the idea of using a Neural Network for price prediction would have been beneficial, as we feel the predicted prices would be more accurate. Overall, the process of collecting, inspecting and exploring the data and combining two different machine learning problems into one singular problem was a unique and enjoyable experience.

## References

- [1] InsideAirBnB. Get the Data. Retrieved from <http://insideairbnb.com/get-the-data/>
- [2] Kaggle. Airbnb Reviews Sentiment Analysis and Prediction. Retrieved from <https://www.kaggle.com/code/heeraldedhia/airbnb-reviews-sentiment-analysis-and-prediction>
- [3] Spot Intelligence. (2023, May 8). F1 Score: The Ultimate Guide in 2023. Retrieved from <https://spotintelligence.com/2023/05/08/f1-score/>

## List of Figures

Figure Number	Figure Title
Figure 1	Label Meanings in Sentiment Analysis
Figure 2	Ten Percent Subset Extraction
Figure 3	Missing Values Data Inspection of Reviews
Figure 4	Example of Non-English Reviews
Figure 5	Pie Chart - Sentiment Distribution of First Manually Labeled Set
Figure 6	Pie Chart - Sentiment Distribution of Second Manually Labeled Set
Figure 7	Train Test Split
Figure 8	Preprocessing and SentimentDLApproach Model Pipeline
Figure 9	SentimentDLApproach Pipeline
Figure 10	Pretrained Sentiment Pipeline Accuracy and F1 Score
Figure 11	Pretrained Sentiment Pipeline Individual Metrics
Figure 12	Pretrained Sentiment Pipeline Confusion Matrix
Figure 13	Equivalent Model to PretrainedPipeline('analyze_sentimentdl_use_imdb')
Figure 14	SentimentDLApproach Accuracy and F1 Score
Figure 15	SentimentDLApproach Individual Metrics
Figure 16	SentimentDLApproach Confusion Matrix

Figure 17	Random Forest Classifier Accuracy and F1 Score
Figure 18	Random Forest Classifier Individual Class Metrics
Figure 19	Random Forest Classifier Confusion Matrix
Figure 20	Weighted Random Forest Classifier Accuracy and F1 Score
Figure 21	Weighted Random Forest Classifier Individual Class Metrics
Figure 22	Weighted Random Forest Classifier Confusion Matrix
Figure 23	Naive Bayes Classifier Accuracy and F1 Score
Figure 24	Naive Bayes Classifier Individual Class Metrics
Figure 25	Naive Bayes Classifier Confusion Matrix
Figure 26	Logistic Regression Classifier Accuracy and F1 Score



Figure 27	Logistic Regression Classifier Individual Class Metrics
Figure 28	Logistic Regression Classifier Confusion Matrix
Figure 29	Total Number of Listings and Reviews
Figure 30	Missing Values in Selected Reviews
Figure 31	Missing Values in Listings Dataset (Columns Truncated)
Figure 32	Missing Values in Selected Features from the Listings Dataset
Figure 33	Listings Features to be used for the Price Prediction Model
Figure 34	Split Data, Fit, and Transform on the Pre-Processing Pipeline
Figure 35	Selected Reviews Sentiment Prediction Distribution
Figure 36	Linear Regression Performance Metrics before (left) and after Hyperparameter Tuning (right)
Figure 37	Linear Regression Hyperparameters Tuned
Figure 38	Decision Tree Hyperparameters Tuned
Figure 39	Decision Tree Performance Metrics before (left) and after Hyperparameter Tuning (right)
Figure 40	Gradient Boosting Hyperparameters Tuned

Figure 41	Gradient Boosting Performance Metrics before (left) and after Hyperparameter Tuning (right)
-----------	---

### List of Tables

Table Number	Table Title
Table 1	Summary of Sentiment Analysis Model Results
Table 2	Summary of Price Prediction Model Results