Airbnb Price Prediction

Ethan Ouellette

September 9, 2025

1 Exploratory Analysis

The initial step in my project involved thoroughly understanding the provided Airbnb listings dataset. I began by examining the accompanying Google Sheet, which detailed the descriptions and definitions of each feature.

Next, I constructed a simple histogram of these labels to visualize their distribution. This analysis confirmed that the classes were relatively balanced, mitigating concerns of model bias toward any particular price range.

Given the categorical nature of several key features, I performed value counts, which often help me gain a quick understanding of the data distributions. I did this for Property Type, Room Type, Neighborhood and Neighborhood Group. Then, I grouped the data based on room_type and neighborhood_group, and calculated the average price within each category. Significant discrepancies in average prices across different room types and neighborhood groups indicated that these features would be valuable predictors in my pricing models.

Recognizing that the dataset included numerous individual neighborhoods within broader neighborhood groups (equivalent to NYC boroughs), I delved deeper into neighborhood-level analysis. I grouped listings by each neighborhood and computed the average price for each group. Plotting a histogram of these neighborhood-wise average prices revealed pronounced tails on both ends of the spectrum. This pattern suggested that certain neighborhoods consistently commanded higher or lower prices, reinforcing the importance of incorporating neighborhood information into my predictive models.

1.1 Feature engineering

To enhance the model's predictive capabilities, I engaged in several feature engineering strategies:

1.1.1 Textual Feature Lengths:

In order to capture the level of detail and potential appeal of each listing, I calculated the number of characters in the listing name as well as the description, and counted the number of amenities in the amenities list.

1.1.2 Luxury Amenities Binary Features:

I compiled a complete list of amenities from the dataset and asked ChatGPT to identify amenities indicative of a more luxurious experience; I supplemented this list with additional amenities identified through my own review; and I created binary indicators for each luxury amenity to denote their presence or absence in each listing. I also added a column called "shared_bathrooms" as an indicator for if the "bathroom_text" column said the bathrooms were shared, and dropped the text column itself.

1.1.3 Data Correction:

I identified inconsistencies where listings had a bed value of 0 despite having more than 0 bedrooms and rectified these by setting the number of beds equal to the number of bedrooms in such cases. I also imputed missing numerical values using the median of each respective column to maintain data integrity without introducing significant bias.

1.1.4 Geospatial Analysis and Clustering

Understanding the spatial distribution of listings was pivotal, given that location heavily influences pricing, I undertook the following steps:

- **Scatter Plot Visualization:** I plotted listings based on their latitude and longitude coordinates and colored the points according to their neighborhood to visualize geographical clustering and price distribution. This visualization revealed that some neighborhoods spanned larger areas than others, potentially affecting price homogeneity within those regions.

- **KMeans Clustering:** With this in mind, I applied the KMeans clustering algorithm to the latitude and longitude data to identify distinct geographical clusters. I created a new feature, location_cluster, representing each listing's cluster assignment.
- I generated a scatter plot colored by the average price of each cluster, which highlighted regional price variations, such as higher prices in southern Manhattan clusters.

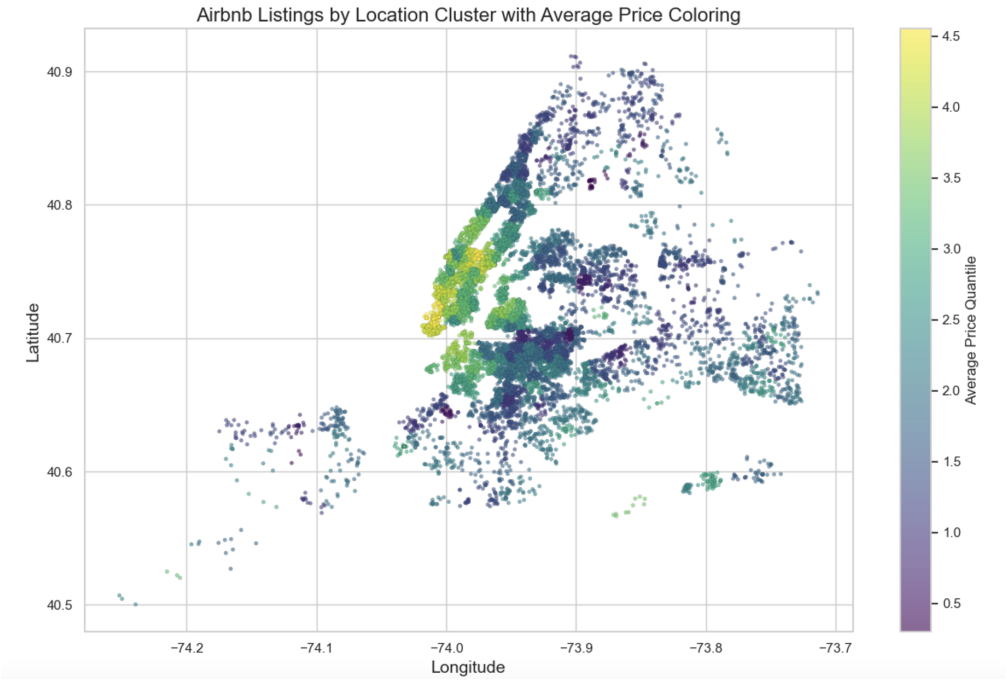

Figure 1: Location clusters colored by price

The successful differentiation of price levels across clusters underscored the utility of incorporating location-based clustering as a predictive feature.

1.1.5 Encoding Categorical Variables

To prepare the dataset for machine learning algorithms, I one-hot encoded all categorical features, including the newly created location_cluster. This transformation allowed tree-based models like XGBoost and Random Forest to effectively interpret categorical information.

1.1.6 Dimensionality Reduction

Although ultimately not utilized in my final models, I explored Principal Component Analysis (PCA) on location-related features (neighborhood and location_cluster) to assess

potential dimensionality reduction benefits. However, given that my chosen models are tree-based and handle feature interactions inherently, I opted to retain the original categorical encodings.

2 Models

For this project, I selected two ensemble-based regression algorithms: **XGBoost** (`XGBRegressor`) and **Random Forest** (`RandomForestRegressor`). The decision to use these models was driven by several key factors, including their proven performance in regression tasks, flexibility in handling diverse feature types, and the availability of robust, well-maintained libraries.

2.1 XGBoost

XGBoost (`XGBRegressor`) (1) was chosen as one of the primary models due to its exceptional performance in various machine learning competitions, including Kaggle. XGBoost is an optimized implementation of gradient boosting, which is known for their ability to handle complex nonlinear relationships and interactions between features effectively.

XGBoost consistently delivers high accuracy and employs regularization techniques to prevent overfitting (in an easy fashion for me, the user). The library is also optimized for speed and performance, leveraging parallel processing and efficient memory usage. This efficiency is crucial when working with large datasets, as it reduces training time significantly.

Another point is that XGBoost offers extensive hyperparameter tuning options, allowing fine-grained control over the model's complexity and learning process.

Finally, as an open-source library with comprehensive documentation and a large user community, XGBoost provides ample resources for troubleshooting and optimization. This support was useful during the model development and tuning phases.

The implementation of XGBoost was facilitated using the `xgboost` library, which is well-documented and widely adopted in the machine learning community.

2.2 Random Forest

Random Forest (`RandomForestRegressor`) (2) was selected as the second algorithm due to its robustness, ease of use, and ability to handle high-dimensional data effectively. Random Forest is an ensemble learning method that constructs multiple decision trees during training

and outputs the mean prediction of the individual trees.

By averaging the results of multiple trees, Random Forest reduces the risk of overfitting, especially when dealing with noisy or complex datasets. This characteristic ensures more generalizable predictions for Airbnb prices across different listings.

Random Forest provides insights into feature importance, enabling the identification of the most influential variables in price prediction. This interpretability is beneficial for understanding the underlying factors driving Airbnb pricing. This was also a plus for XGBoost as a model choice.

The `RandomForestRegressor` from the `scikit-learn` library is straightforward to implement, with minimal parameter tuning required to achieve reasonable performance. This simplicity allows for rapid model development and iteration.

Random Forest can naturally handle both numerical and categorical features (post-encoding), making it well-suited for the diverse feature set present in the Airbnb dataset.

The Random Forest model was implemented using the `RandomForestRegressor` from the `scikit-learn` library, which is I enjoy for its reliability and extensive documentation.

2.3 Comparison and Final Selection

Both XGBoost and Random Forest are powerful ensemble methods that excel in handling complex datasets with numerous features. XGBoost was primarily favored by me for its superior performance in making predictions.

3 Training

In this section, I detail the training algorithms employed for each of the selected models, **XGBoost** (`XGBRegressor`) and **Random Forest** (`RandomForestRegressor`), along with the associated runtime estimates for a single model fit.

3.1 XGBoost Regressor

The **XGBoost Regressor** utilizes an optimized implementation of gradient boosting, which sequentially builds decision trees to minimize the residual errors of previous trees (1). The training process involves iteratively adding trees that predict the gradient of the loss function with respect to the current model's predictions. XGBoost employs regularization

techniques, such as L1 and L2 penalties, to prevent overfitting and enhance model generalization. Additionally, it supports parallel processing, which significantly accelerates the training process by distributing computations across multiple cores. For a single fit, the training time for XGBoost ranged from approximately 3 to 60 seconds, with the majority of runs completing around 10 seconds. This efficiency makes XGBoost a highly suitable choice for handling large datasets and complex feature interactions in a competitive timeframe.

3.2 Random Forest Regressor

The **Random Forest Regressor** is an ensemble learning method that constructs multiple decision trees during training and aggregates their predictions to improve accuracy and control overfitting (2). Each tree in the forest is trained on a bootstrap sample of the data and considers a random subset of features when making splits, which enhances diversity among the trees and contributes to the robustness of the model. The training algorithm leverages parallel processing by building trees independently, allowing for efficient utilization of computational resources. For a single model fit, the training time for Random Forest varied between 3 seconds and 7 minutes, with most fits completing in approximately 3 minutes. This variability is influenced by factors such as the number of trees, the depth of each tree, and the size of the feature set.

4 Hyperparameter Selection

4.1 XGBoost Regressor Hyperparameter Tuning

For the **XGBoost Regressor** (`XGBRegressor`), I defined a comprehensive parameter grid to explore a wide range of hyperparameter values. The grid was specified as follows:

```
xgb_param_grid = {
    'n_estimators': [100, 200, 300, 400, 500, 600, 700, 800, 900, 1000],
    'max_depth': [3, 4, 5, 7, 8, 9, 10],
    'grow_policy': ['depthwise', 'lossguide'],
    'learning_rate': [0.08, 0.05, 0.03, 0.01],
    'reg_alpha': [3, 2.5, 2, 1, 0.05, 0.03, 0.01]
}
```

To efficiently navigate this extensive search space, I utilized Scikit-learn's `RandomizedSearchCV`

with 100 iterations and 5 folds. This approach balances thorough exploration with reasonable computation time, allowing me to identify promising hyperparameter combinations without exhaustive searching. During the tuning process, I also experimented with both `reg_alpha` (L1 regularization) and `reg_lambda` (L2 regularization) to assess their impact on model performance. I found L1 to be better for lower RMSE scores (post-rounding).

4.2 Random Forest Regressor Hyperparameter Tuning

For the **Random Forest Regressor** (`RandomForestRegressor`), I established the following parameter grid:

```
rf_param_grid = {
    'n_estimators': [500, 700, 900],
    'max_depth': [None, 10, 20, 30],
    'min_samples_split': [2, 5, 10, 15],
    'min_samples_leaf': [1, 2, 4, 8],
    'max_features': ['sqrt', 'log2', None],
    'bootstrap': [True, False]
}
```

Similarly, I applied `RandomizedSearchCV` with 100 iterations and 5 folds to explore this parameter space efficiently. This method enabled the identification of optimal settings for critical hyperparameters such as the number of trees (`n_estimators`), maximum depth of the trees (`max_depth`), and the minimum number of samples required to split an internal node (`min_samples_split`).

4.3 Performance Evaluation and Visualization

To visualize the impact of hyperparameter tuning, I focused on the relationship between the `max_depth` parameter and the model's predictive accuracy, measured by Root Mean Squared Error (RMSE). For the XGBoost model, I varied `max_depth` from 1 to 10 while keeping other parameters fixed at the optimal values identified through the grid search:

```
{
    'reg_alpha': 2.5,
    'n_estimators': 900,
```

```
    'learning_rate': 0.03,
    'grow_policy': 'depthwise',
    'objective': 'reg:squarederror',
    'verbosity': 0
}
```

The resulting plot, shown in Figure 2, illustrates that RMSE rapidly decreases as `max_depth` increases, reaching a minimum at a depth of 8. Beyond this point, further increases in depth do not yield significant improvements and may lead to overfitting.

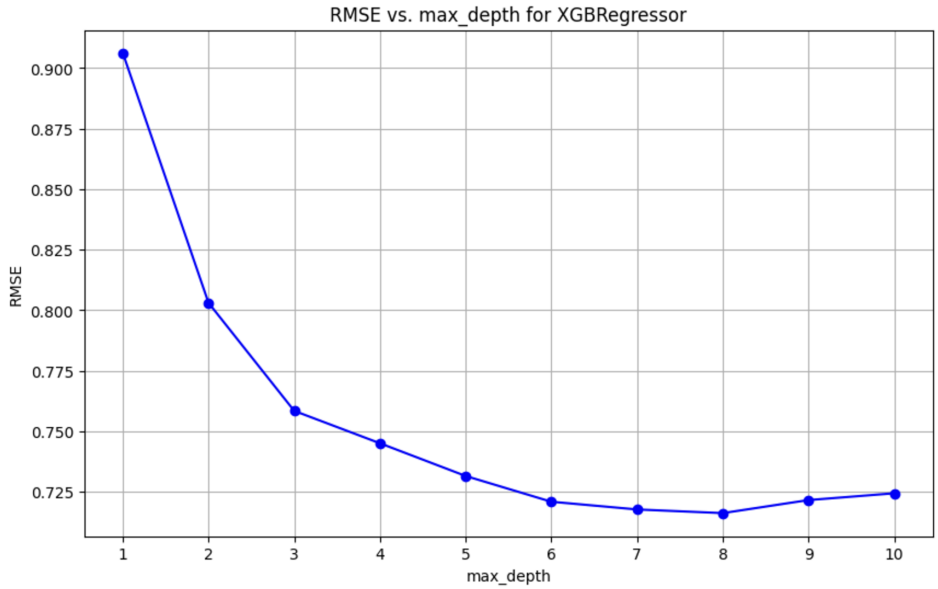

Figure 2: Effect of `max_depth` on RMSE for XGBoost Regressor

5 Data Splits

To prevent overfitting and ensure that my models generalize well to unseen data, I implemented an 80/20 train-test split using Scikit-learn's `train_test_split` function. Specifically, 80% of the dataset was used to train the models, while the remaining 20% served as a validation set to evaluate model performance using Root Mean Squared Error (RMSE). This split allowed me to assess the models' predictive accuracy on unseen data effectively. Once a model achieved a satisfactory RMSE on the validation set, I retrained it using the entire dataset to maximize its performance before submitting it to the competition. This approach ensured that the models were both well-validated and fully trained, minimizing the risk of overfitting.

6 Reflection on Progress

Mistakes? Missteps? Bugs? There were many. In no particular order:

- My first approach involved using classification models instead of regression models. These underperformed relative to regression, presumably because the labels are inherently ordered.
- I had the idea to tune the rounding thresholds for my label predictions. I figured that there may be a smarter way to decide how to round the reggesor's point predictions other than up/down at .5. The first method I tried was simply optimizing the rounded RMSE by shifting the thresholds, but this optimization happened after the tuning of the regressor, so it severely overfit the thresholds to the predictions themselves. The next idea was to use a multilabel logistic regressor on the predictions, but this often increased the post-rounding RMSE score as well, although not as much as the simple optimizer.
- I didn't do all of my preprocessing and feature engineering before all of the modeling, and as I would update the data I was training on, I would often forget to change the datafram in the model notebooks, which slowed me down.
- I looked into if there were any models perfectly suited for this regression/rounding business and came across a `OrdinalGBT` library, an ordinal regressor using the statistical concept of ordinal regression. Although ordinal regression seemed like just what I needed, `OrdinalGBT` did not perfrom better than `XGBRegressor`.
- There were a couple of times I trained on all of the data and checked my score on `X_train`, but since it had been part of the training data, the score was way better than it should have been had I trained appropriately.
- PCA didn't end up being useful given the fact that my models were tree based.
- Training on the x most important features or features with importance $> c$ never seemed to really help my model's score on data outside of training set. I am curious to know better approaches for this.

The hardest part of this competition was figuring out which features to include or exclude. As my models started to plateau in their scores at around .73, it was hard to imagine where to go next, or if the score could really be improved without overfitting. I would love to see how the TA's would approach this problem, as well as the top scorers.

7 Predictive Performance

7.1 Performance Comparison

The effectiveness of the models was evaluated using the Root Mean Squared Error (RMSE) metric on the Kaggle leaderboard. The results for my submissions are as follows:

- **XGBoost Regressor**: Achieved an RMSE of **0.73756**.
- **Random Forest Regressor**: Achieved an RMSE of **0.77028**.

Based on these results, the **XGBoost Regressor** outperformed the Random Forest model, demonstrating superior predictive accuracy for Airbnb price prediction in this competition. Note: These results are on half of the test data as detailed by the submission site.

7.2 Feature Importance Analysis

To gain insights into the factors driving the model predictions, I analyzed the top 20 feature importances for both models. The feature importance scores highlight the most influential variables in predicting Airbnb prices.

7.2.1 XGBoost Regressor Top 20 Features

Feature	Importance
room_type_Entire home/apt	0.517189
shared_bathrooms	0.036048
minimum_nights	0.032009
property_type_Private room in resort	0.029792
property_type_Room in hotel	0.024477
property_type_Room in boutique hotel	0.013779
location_cluster_296	0.012283
location_cluster_48	0.010214
neighbourhood_group_cleansed_Manhattan	0.008131
host_response_time_nan	0.006906
accommodates	0.006413
bathrooms	0.006305
neighbourhood_cleansed_Gramercy	0.006284
has_dishwasher	0.005685
host_total_listings_count	0.005481
location_cluster_71	0.005019
bedrooms	0.004813
room_type_Hotel room	0.004695
property_type_Private room in rental unit	0.004648
neighbourhood_cleansed_Arrochar	0.004137

Table 1: Top 20 Feature Importances for XGBoost Regressor

7.2.2 Random Forest Regressor Top 20 Features

Feature	Importance
room_type_Private room	0.244749
minimum_nights	0.129754
longitude	0.062244
host_listings_count	0.044434
accommodates	0.042020
room_type_Entire home/apt	0.039385
latitude	0.035906
bathrooms	0.023918
shared_bathrooms	0.023563
calculated_host_listings_count_private_rooms	0.023071
bedrooms	0.020575
host_total_listings_count	0.020121
property_type_Room in hotel	0.014816
availability_90	0.014809
amenities_count	0.014049
availability_60	0.012659
calculated_host_listings_count_entire_homes	0.011884
host_since	0.011634
availability_30	0.011216
calculated_host_listings_count	0.011068

Table 2: Top 20 Feature Importances for Random Forest Regressor

7.3 Interpretation of Feature Importances

The feature importance analysis provides valuable insights into the key drivers of Airbnb pricing:

- **Room Type**: Both models highlight the significance of room type, with entire homes/apartments and private rooms being the most influential features. This aligns with the intuitive understanding that the type of accommodation directly impacts price.
- **Minimum Nights**: The minimum number of nights required for a booking is another critical factor, as longer stays may be priced differently compared to shorter ones.
- **Location Clusters and Geographical Features**: Features related to location, such as longitude, latitude, and location clusters, play a substantial role in determining prices. This reflects the importance of geographical positioning within New York City.

The engineered clusters themselves are inherently interpretable because we can find them on our scatterplot map mentioned in section 1.

- **Property Type**: Specific property types, such as rooms in hotels or resorts, also significantly influence pricing, indicating that the type of property offers different levels of amenities and experiences.
- **Amenities and Host Listings Count**: The availability and count of amenities, along with the number of listings a host has, contribute to the pricing, suggesting that more amenities and experienced hosts may charge higher prices. The importance of having a dishwasher for XGBoost was neat to see after identifying the average price discrepency in the exploratory analysis chart described in section 1.

This feature importance analysis not only enhances the interpretability of the models but also provides actionable insights into what factors most strongly influence Airbnb pricing. Understanding these key factors can inform both hosts and guests in making informed decisions, which is neat to see.

7.4 Kaggle Submission

My submissions were uploaded to Kaggle under the username `Ethan Ouellette`. The XGBoost Regressor achieved an RMSE of 0.73756, while the Random Forest Regressor achieved an RMSE of 0.77028. The XGBoost model was selected as the final submission due to its superior performance.

8 Code

https://github.com/Ethan-Ouellette/airbnb-price-prediction/upload/main

References

[1] Tianqi Chen and Carlos Guestrin. XGBoost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 785–794. ACM, 2016.

[2] Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

[3] *Scikit-learn: Machine Learning in Python*. https://scikit-learn.org/, 2024.