Ensemble Methods for Airbnb Price Prediction in NYC

Felipe GARAVANO Yasmina HOBEIKA Ian MOON Pallav SAHU
Akshay SHASTRI

March 20, 2023

Abstract

This project aims to predict the optimal price for Airbnb listings in New York City using ensemble learning techniques. With over 40,000 listings in the city, hosts face difficulties in pricing their properties fairly while renters may find it challenging to assess whether a listing is well-priced or not. In this project, we tried different ensemble methods such as bagging, AdaBoost, CatBoost, XGBoost, and Random Forest to then stack them together. The stacking model resulted in an MSE of 0.151, RMSE of 0.387, and a R^2 of 0.656.

Key Words: Ensemble Learning; Decision Tree; Random Forest; Boosting; Gradient Boosting; XGBoost; AdaBoost; Weak Learner; Stacker Models.

1 Introduction

Airbnb is a popular online platform that allows individuals to list their own properties as rental places for travelers. It has disrupted the traditional hospitality industry by offering a wide range of accommodation options. In New York City alone, there are around 40,000 listings on Airbnb. However, setting an optimal price for a listing can be challenging for hosts, as they have to consider various factors such as demand, supply, location, amenities and competition. Similarly, renters may find it difficult to assess whether a listing is fairly priced or not.

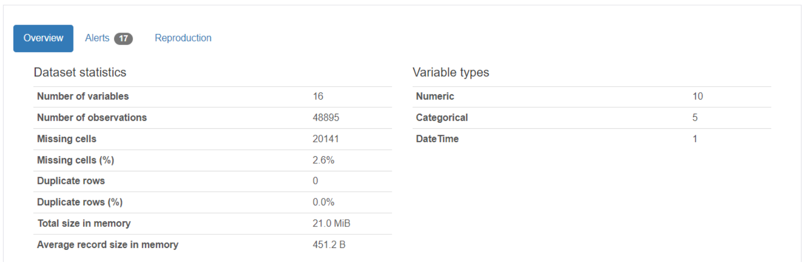
In this project, we aim to predict the price of Airbnb listings in New York City using ensemble learning methods. Ensemble learning is a technique that combines multiple models to improve prediction accuracy and reduce overfitting. We will use different types of ensemble methods such as bagging, boosting and stacking to compare their performance on our data.

We will first conduct a pre-processing step where we will apply various transformations and feature engineering techniques. Then we will build our models by performing cross-validation and hyper-parameter tuning to optimize them. Finally we will evaluate our models using different metrics such as root mean squared error (RMSE), mean squared error (MSE) and R-squared (R^2) and present our findings.

2 Data Exploration

For data exploration we used two automatic profiling libraries $ydata\_profiling$ and SweetViz which evaluate all the variables and assess whether they have high or low cardinality, missing values, distinct values and relationships like we can see in figure 1 where we have an overview of the dataset.

Then on figure 2 where it shows both the relationship between numeric and categorical variables and their correlation.



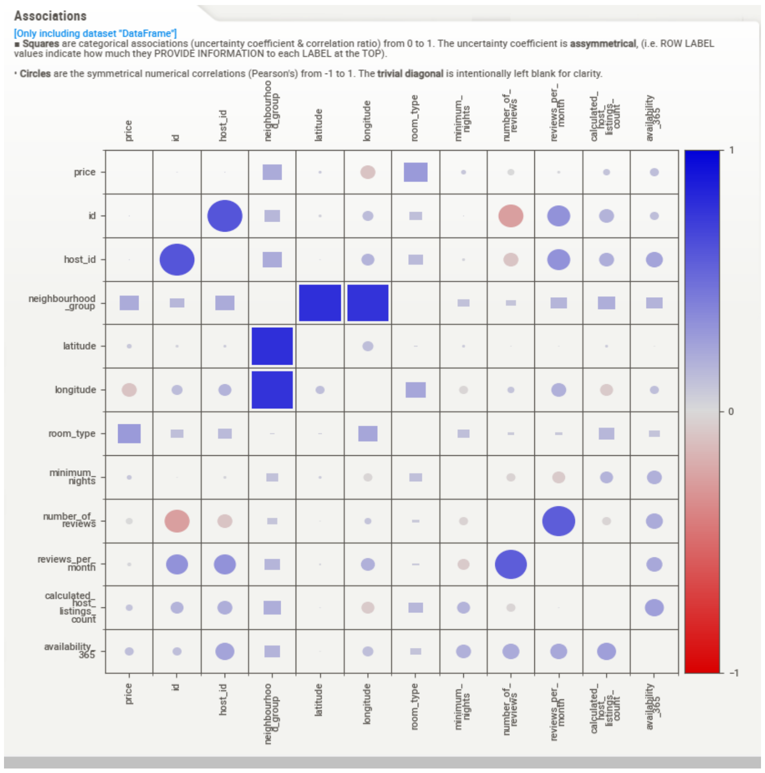
Dataset statistics

Number of variables	16
Number of observations	48895
Missing cells	20141
Missing cells (%)	2.6%
Duplicate rows	0
Duplicate rows (%)	0.0%
Total size in memory	21.0 MiB
Average record size in memory	451.2 B

Variable types

Numeric	10
Categorical	5
DateTime	1

Figure 1: Overview of Dataset



Associations

[Only including dataset "DataFrame"]
■ **Squares** are categorical associations (uncertainty coefficient & correlation ratio) from 0 to 1. The uncertainty coefficient is **assymmetrical**, (i.e. ROW LABEL values indicate how much they PROVIDE INFORMATION to each LABEL at the TOP).

• **Circles** are the symmetrical numerical correlations (Pearson's) from -1 to 1. The **trivial diagonal** is intentionally left blank for clarity.

Figure 2: Associations between Variables

Here the most interesting findings relate to the correlation with the variable price which will be our target for the prediction. Furthermore, a more in depth analysis of the data can be seen in the EDA files in the Github.

3 Methodology

In this section, we conducted a pre-processing of the data, which included dropping NAs values, removing outliers, one-hot encoding variables, creating new variables, and standardizing numerical columns. The section also describes some models such as decision tree, bagging, random forest, xgboost, catboost and adaboost that are used to predict the price of Airbnb listings in New York City.

3.1 Pre-Processing and Feature Engineering

We conducted a pre-processing step for our data analysis using a separate .py file where we applied all the necessary transformations and feature engineering. We read the data file with pandas and defined three functions: two for sentiment analysis and one for pre-processing.

The pre-process function took the data as input and returned a final file after applying all the transformations and feature engineering.

In our pre-processing function, we started by dropping the NAs values from the "name" and "host name" columns, and removing outliers in the price variable using a z-score. We then one-hot encoded the neighbourhood group variable, and kept only the top 25 neighbourhoods as dummy variables since they represented most of the data.

We also one-hot encoded the room type, filled NAs in "reviews per month" with 0s, and created a new variable called "is rated". Another variable we created was length name, which counted the words in each listing name. We used sentiment analysis functions to generate some features related to polarity analysis, such as the "sentiment" feature, which measured how positive or negative each listing name was using the TextBlob package, and the "price keyword ratio" feature, which calculated ratio of "high" price words to "low" ones.

Furthermore, based on host information, we computed the number of listings per host and host activity level, and calculated the distance to the center grouped by each region. We also extracted from the last review date, days since last review and day, month, and year of the last review. Finally, we took the log transformation of the price variable, squared the minimum nights variable to capture a possible non-linear relationship, standardized all numerical columns, and dropped the neighbourhood and neighbour columns.

3.2 Models

In this section, we will discuss how we implemented decision tree, bagging and boosting models. For each model, we followed the same methodology of splitting the data into a training and testing set, running a grid search to tune the hyperparameters, and then computing the evaluation metrics as well as the permutation feature importance to better understand the results.

Specifically, we split the data into train and test using an 80/20 ratio. We initialized the model with a random state of 42 for reproducibility. The grid search for the hyperparameters were done using a 5-fold cross-validation on the training data to minimize the negative mean squared error (MSE). After the grid search, we fit our best model to the training data and made predictions on the test data. We evaluate our model performance using metrics such as root mean squared error (RMSE), R-squared (R^2) and MSE.

3.2.1 Regression Tree

A regression tree is a type of decision tree that splits the data into smaller groups based on certain criteria, and then predicts the mean value of the target variable for each group. Regression trees can handle both numerical and categorical features, and are easy to interpret and visualize. The specific hyperparameters that we tuned and the results of the best model can be seen in Table 1. The MSE, RMSE, and R^2 can be seen in Table 2.

3.2.2 Bagging

To improve our regression tree model we tried Bagging. Bagging, also known as bootstrap aggregating, is an ensemble learning method that combines multiple models to reduce the variance and overfitting of a single model. Bagging works by creating multiple subsets of data from the original training set using sampling with replacement, and then training a base estimator (such as a decision tree) on each subset. The final prediction is obtained by averaging the predictions of all the base estimators.

The specific hyperparameters that we tuned and the results of the best model can be seen in Table 1. The MSE, RMSE, and R^2 can be seen in Table 2.

3.2.3 Random Forest

Random Forest expands on the concept of bagging by training on a random subset of the data and also a random subset of the features for each split. During training, the algorithm randomly selects a subset of the available features to use for each tree, which helps to reduce overfitting and increase generalization performance. Once the trees are trained, they are combined to make predictions on new data. In our regression final prediction is the average of the predictions of all the individual trees.

The specific hyperparameters that we tuned and the results of the best model can be seen in Table 1. The MSE, RMSE, and R^2 can be seen in Table 2.

3.2.4 Extreme Random Forest

Extreme Random Forest further expands on the concept of random forests by randomly selecting the split instead of choosing the best one. Additionally, the entire dataset is used instead of a bootstrapped method.

The specific hyperparameters that we tuned and the results of the best model can be seen in Table 1. The MSE, RMSE, and R^2 can be seen in Table 2.

3.2.5 CatBoost

CatBoost is a machine learning algorithm that uses gradient boosting on decision trees. It can handle both numerical and categorical features, and is known for its ability to handle noisy data and prevent overfitting. Catboost has a novel algorithm for handling missing values called "ordered choice", which helps to improve the accuracy of the model when there are missing values in the dataset. It also includes built-in methods for handling imbalanced data and for calculating feature importance.

The specific hyperparameters that we tuned and the results of the best model can be seen in Table 1. The MSE, RMSE, and R^2 can be seen in Table 2.

3.2.6 XGBoost

XGBoost is an ensemble method that combines multiple weak models (decision trees) to create a strong model that can make accurate predictions. The algorithm uses a gradient boosting technique that iteratively improves the model by adding new decision trees that correct the errors of the previous ones. XGBoost has several advantages over traditional decision trees, such as faster training speed, higher accuracy, and better handling of missing values and outliers.
The specific hyperparameters that we tuned and the results of the best model can be seen in Table 1. The MSE, RMSE, and R^2 can be seen in Table 2.

3.2.7 AdaBoost

AdaBoost works by iteratively reweighting the data, so the subsequent models focus on the instances that were misclassified by previous models. AdaBoost can handle both numerical and categorical features, and its ability to adapt to the data makes it powerful for solving complex problems.

The specific hyperparameters that we tuned and the results of the best model can be seen in Table 1. The MSE, RMSE, and R^2 can be seen in Table 2.

3.2.8 Stacking Models

We decided to combine a set of five individual models, including BaggingRegressor, AdaBoostRegressor, CatBoostRegressor, XGBRegressor, and RandomForestRegressor. Each of these models has a set of hyperparameters that have been tuned to improve performance.

We created a StackingRegressor object, which is a type of ensemble model that combines the predictions of multiple individual models to produce a more accurate prediction. The stacked model is then trained on the training data and used to make predictions on the test data. The performance of the stacked model is evaluated using two metrics: the coefficient of determination (R^2) and the root mean squared error (RMSE). By combining the best models, we got an RMSE of 0.387 and a R^2 of 0.656.

Overall, this code combines multiple individual models using the StackingRegressor algorithm to produce a more accurate prediction than any single model could achieve on its own.

4 Results and Evaluations

As can be seen in table 1 the values which resulted for grid search for the different models were quite diverse. We should also note that when 'n/a' is provided it means the parameter does not correspond to the model or the default from the package was used. Something to mention is that we also tried to run the Decision Tree we implemented from scratch on the dataset and got a performance of around 0.56 R^2 when using the default configuration which is much better compared to the default regression tree from Sklearn with only 0.26 R^2. Regarding the performances in table 2 the best performing models were XGBoost, AdaBoost with XGboost, CatBoost and Gradient Boosting. We can see that the Stacking of these models resulted in the best performance.

Models Hyperparameters								
Model	DT	Bagging	RF	ERF	GradB	CatB	XGB	AdaB
N Estimators	n/a	300	200	210	160	n/a	250	60
Max Samples	n/a	1	1	1	n/a	n/a	n/a	n/a
Min Samples Split	2	n/a	5	5	n/a	n/a	n/a	n/a
Min Samples Leaf	4	n/a	n/a	n/a	n/a	n/a	n/a	n/a
Max Features	n/a	1	0.7	0.7	n/a	n/a	n/a	n/a
Bootstrap	n/a	True	True	True	n/a	n/a	n/a	n/a
Bootstrap Features	n/a	True	True	True	n/a	n/a	n/a	n/a
Learning Rate	n/a	n/a	n/a	n/a	0.1	0.15	0.08	0.1
Iterations	n/a	n/a	1	1	280	280	n/a	n/a
Max Depth	8	n/a	7	18	7	8	7	n/a

Table 1: Tuning by Model

Models comparison			
Model	MSE	RMSE	R^2
Regression Tree from Scratch	n/a	0.434	0.597
Regression Tree	0.176	0.419	0.596
Bagging	0.153	0.391	0.649
Random Forest	0.168	0.410	0.613
Extra Random Forest	0.155	0.394	0.644
Gradient Boosting	0.151	0.389	0.653
CatBoost	0.152	0.390	0.651
XGBoost	0.151	0.388	0.654
ADABoost with DT	0.166	0.408	0.618
ADABoost with CatBoost	0.153	0.391	0.648
ADABoost with XGBoost	0.151	0.389	0.653
Stacking	0.151	0.387	0.656

Table 2: Performance and Loss by Model

For our best individual model, XGBoost we can also see the top 5 permutation feature importance in the figure 3 where location variables and those relating to the size where most useful for prediction. Interestingly availability which could indicate how in demand the listing is was also quite significant.

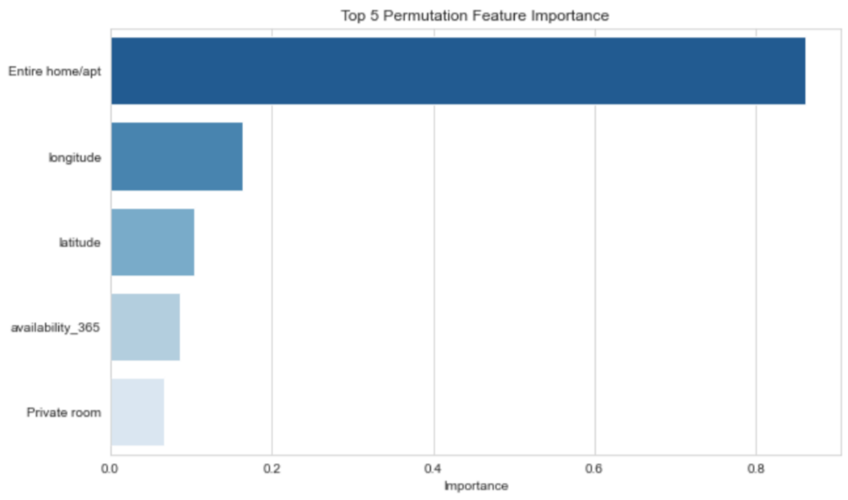


Figure 3: Feature Importance XGBoost

5 Conclusion

This was a quite challenging task since there were really a few columns with valuable information for price prediction. While we engineered new features using the data we had like sentiment analysis on name and geo-location ones, we believe that additional variables like square feet, number of rooms would help improve the prediction considerably. As can be seen in table 2 our best models have a fairly similar performance with an R^2 of around 0.65. After stacking our best models, we did see a marginal improvement achieving our best results of 0.656 R^2 and the lowest error.

Appendix: Implementation of Decision Tree from Scratch

Implementation

This appendix provides a brief overview of the implementation of a decision tree (DT) from scratch using Python and NumPy libraries. We first define the Node_cls class that initializes the DT nodes for classification and regression problems. The Node_cls constructor stores and initializes the feature index, left and right child nodes, node splitting threshold value, information value gain, and output value of the node. The is_leaf_node() function is used to check if a node is a leaf node.

Next, we implement the DecisionTreeClassifier class that builds a DT model for classification problems. The DecisionTreeClassifier constructor stores and initializes the maximum tree depth, minimum samples required for splitting nodes, number of features to be used for the tree, cost function/loss criterion for splitting nodes, root node, and tree structure for generating a visualization. The train() method of the DecisionTreeClassifier takes input features and output labels and uses the _build_tree() function to grow the DT structure. The _dominant_class() function finds the most common class/label, and the _split() function splits the data into left and right child nodes. The information_gain() function calculates the information gain for splitting nodes.

Similarly, we implement the DecisionTreeRegressor class that builds a DT model for regression problems. The DecisionTreeRegressor constructor stores and initializes the maximum tree depth, minimum samples required for splitting nodes, number of features to be used for the tree, cost function/loss criterion for splitting nodes, and tree structure for generating a visualization. The train() method of the DecisionTreeRegressor takes input features and output labels and uses the _build_tree() function to grow the DT structure. The Node_reg class constructor initializes the tree nodes for regression problems and stores the feature index, left and right child nodes, node splitting threshold value, variance loss value, and output value of the node. The is_leaf_node() function is used to check if a node is a leaf node. The DecisionTreeClassifier and DecisionTreeRegressor classes also have a tree attribute that stores the tree structure to generate tree graph visualization.

Finally, we compare the performance of our DT with that of the DT implemented in the Scikit-learn library on four datasets: Breast Cancer, Iris, Diabetes, and Airbnb. The table 3 presents the execution time and **accuracy** (for classification) or **R-squared** (for regression) values. Our DT has almost the same performance as the Scikit-learn DT in classification experiments, while it overperforms the Scikit-learn DT in regression experiments.

Performance and comaprison with SKlearn

Dataset	Our DT (time)	Our DT	SKlearn DT (time)	SKlearn DT
Breast Cancer (Classification)	0.6694 seconds	0.937	0.6808 seconds	0.958
Iris (Classification)	0.0662 seconds	0.973	0.0682 seconds	1
Diabetes (Regression)	0.0547 seconds	0.485	0.0596 seconds	-0.22
Airbnb (Regression)	168.0722 seconds	0.567	168.9311 seconds	0.274

Table 3: DT Comparison execution time and performance [accuracy for classification and R^2 for regression]

Next Steps

- Implement pruning to prevent overfitting.
- Optimize for parallel processing so that this DT can be used for bagging and random forest implementations efficiently.