Hooman HALABICHIAN, MSc AI
Hooman.halabichian@student-cs.fr
GitHub link: Here

Predicting Airbnb Prices in New York City

Introduction:

New York City, known for its iconic landmarks and vibrant culture, has embraced Airbnb as an alternative to traditional hotels. Travelers should ensure compliance with the city's short-term rental regulations. Determining optimal pricing for hosts can be challenging due to listing variety. This project aims to predict Airbnb listing prices in NYC using ensemble learning methods.

The dataset:

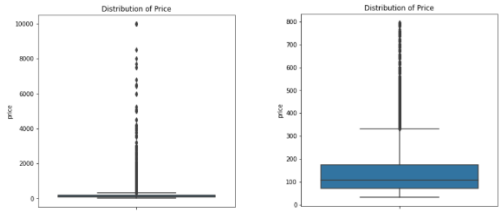
This dataset consists of around 49,000 rows and 16 columns, representing individual Airbnb listings with various attributes. These include unique identifiers for listings and hosts, titles, host names, and geographical details such as borough and neighborhood. The dataset also provides coordinates, room types, prices, minimum stays, review statistics, host-managed listings, and yearly availability.

Data Preprocessing:

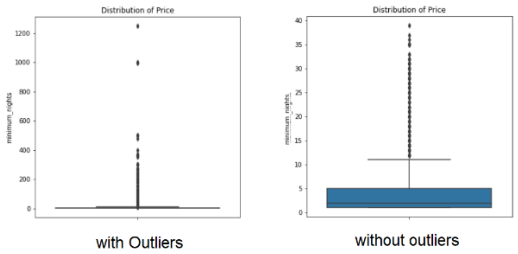
In this analysis, we will delve into various aspects of data preprocessing, specifically tailored to address the following tasks: identifying and handling duplicates, managing missing values, detecting, and handling outliers, deleting irrelevant columns, data correlation, augmenting the dataset with additional features, encoding of some categorical variables, and conducting exploratory data analysis.

Missing values and irrelevant columns: The first step in the analysis is preprocessing the dataset to ensure its quality and reliability. This involves addressing missing values and dropping unnecessary columns that do not contribute significantly to understanding the factors influencing price. Missing values are replaced with 0 to maintain consistency, while irrelevant columns such as 'name', 'host_id', 'host_name', 'latitude', and 'longitude' are removed this step helps in reducing the noise and improving the accuracy of the analysis.

Outlier Handling: Outliers can skew the results of the analysis and negatively impact the performance of predictive models. Therefore, it is essential to identify and address any potential outliers within the dataset. Various statistical techniques and visualizations, like box plots and scatter plots, are employed to detect and manage outliers. In this analysis, percentiles are used to identify and remove extreme values in the 'price' and 'minimum_nights' columns.

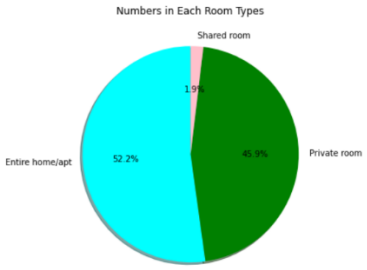


Boxplots distribution of price column with and without outliers



Boxplots distribution of minimum nights with and without outliers

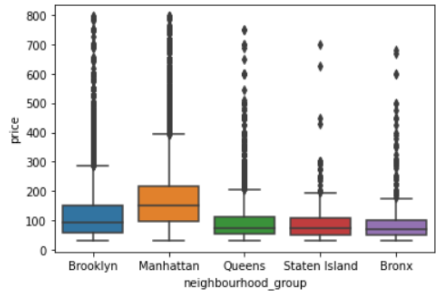
Exploratory Data Analysis: The dataset contains three distinct types of accommodations: private rooms, whole homes, and shared rooms. Analyzing the distribution patterns of these room types helps understand their respective demand and market dynamics within the Airbnb landscape. The majority of listings are found in the entire homes and private rooms categories, while shared rooms make up a smaller portion of available accommodations.



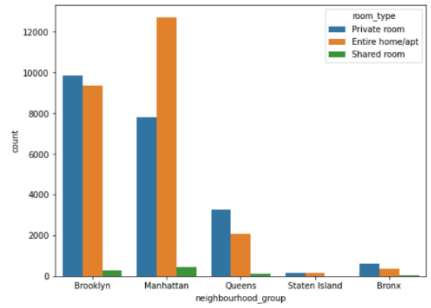
Neighborhood Analysis: The dataset is further examined to understand the distribution of listings and prices across five neighborhood groups in New York City - Bronx, Brooklyn, Manhattan, Queens, and Staten Island. Manhattan and Brooklyn emerge as the most sought-after neighborhoods, boasting a higher number of listings compared to other boroughs. This reflects the popularity of these areas among tourists and locals due to their iconic attractions and vibrant neighborhoods.

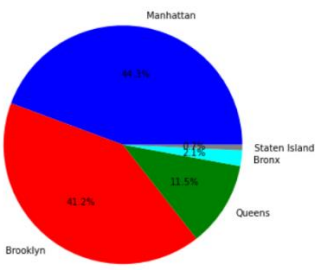
Price Distribution: The average price of accommodations in Manhattan is higher than in other boroughs, indicating a premium for staying in this bustling and iconic location. A summary of the price distribution is provided for each

neighborhood group, detailing the count of listings, mean price, standard deviation, minimum price, and various percentiles.



Room Type Distribution: The distribution of room types across different neighborhoods is analyzed to understand traveler preferences and market performance. Brooklyn exhibits a relatively balanced distribution between "private room" and "entire home" listings, while Manhattan has a higher proportion of entire home or apartment listings. In less popular areas like Staten Island and the Bronx, there is a more uniform distribution of listings across all categories.





Supply and Demand: The concentration of listings in Manhattan and Brooklyn highlights the high demand for accommodations in these areas, emphasizing their significance within the city's overall Airbnb landscape. These neighborhoods cater to a wide range of traveler preferences, offering diverse options in terms of privacy, comfort, and amenities.

In conclusion, the in-depth analysis of the New York City Airbnb dataset sheds light on various aspects of the market, such as the distribution of room types, price dynamics across neighborhoods, and the factors that contribute to price variation. By addressing data quality issues, handling outliers, and visualizing categorical data, the analysis paves the way for more accurate insights and improved model performance. Understanding the trends and variations across different neighborhoods allows for better decision-making in the development of effective marketing strategies, pricing policies, and property management approaches for hosts and the Airbnb platform.

Model deployment, and Evaluation:

The goal was to predict Airbnb listing prices in New York City using various Ensemble learning models. The dataset was preprocessed to facilitate efficient model training and evaluation.

The dataset was limited to 10,000 randomly selected rows to reduce computation time. Numerical and categorical columns were separated and processed differently. Categorical columns were encoded as integers using LabelEncoder, while numerical columns were standardized using StandardScaler to make them comparable in magnitude.

Models: The preprocessed dataset was split into training and testing sets, with 70% used for training and 30% for testing. Six machine learning models were evaluated for price prediction, including Decision Tree, RandomForest, AdaBoost, XGBoost, Gradient Boosting, and LightGBM.

Hyperparameters and cross-validation: Each model was assessed under three conditions: simple training and testing, training and testing with GridSearch for hyperparameter tuning, and training and testing with both GridSearch and cross-validation(K-fold=5) for robust performance evaluation. The table below shows the hyperparameters used for each model:

Model	Hyperparameters
Decision tree	'max_depth': [3, 4, 5, None], 'min_samples_split': [2, 5, 10], 'min_samples_leaf': [1, 2, 4]
Random Forest	'n_estimators': [50, 100, 200], 'max_depth': [None, 10, 20], 'min_samples_split': [2, 5, 10], 'min_samples_leaf': [1, 2, 4]
AdaBoost	'n_estimators': [50, 100, 200], 'learning_rate': [0.1, 0.5, 1.0]
XGBoost	'max_depth': [3, 4, 5], 'learning_rate': [0.1, 0.01, 0.001], 'n_estimators': [50, 100, 200]
Gradient Boosting	'learning_rate': [0.1, 0.01, 0.001], 'n_estimators': [50, 100, 200], 'max_depth': [3, 4, 5], 'min_samples_split': [2, 5, 10], 'min_samples_leaf': [1, 2, 4]
LGBM	'learning_rate': [0.1, 0.01, 0.001], 'n_estimators': [50, 100, 200], 'max_depth': [3, 4, 5], 'min_child_samples': [1, 5, 10], 'reg_alpha': [0.0, 0.1, 1.0], 'reg_lambda': [0.0, 0.1, 1.0],

Performance metrics: Were used to compare the models, such as Root Mean Square Error (RMSE), Coefficient of Determination (R^2), Explained Variance Score, Maximum Error, and Mean Absolute Error (MAE). These metrics allowed for a comprehensive assessment of each model's accuracy, reliability, and computational efficiency.

Evaluations:

Model	RMSE	R^2	Explained Variance	Max Error	MAE
Simple Decision Tree	12711.19	0.1687	0.1664	665.00	66.99
Decision Tree with GridSearch	7030.18	0.3536	0.3536	642.83	51.40
Decision Tree with KFold & GridSearch	7040.81	0.3526	0.3526	642.83	51.45
Simple Random Forest	6497.04	0.4026	0.4038	640.91	51.08
Random Forest with GridSearch	6078.59	0.4410	0.4410	641.65	47.94
Random Forest with KFold & GridSearch	6078.59	0.4410	0.4410	641.65	47.94
Simple AdaBoost	8457.16	0.2224	0.2888	630.79	64.18
AdaBoost with GridSearch	7951.11	0.2689	0.3252	625.33	62.93
AdaBoost with KFold & GridSearch	7895.00	0.2740	0.3255	628.60	62.42
Simple XGBoost	6143.19	0.4351	0.4366	642.75	47.40
XGBoost with GridSearch	5902.88	0.4572	0.4572	632.64	47.31
XGBoost with KFold & GridSearch	5902.88	0.4572	0.4572	632.64	47.31
Simple Gradient Boosting	6261.87	0.4284	0.4285	649.84	48.34
Gradient Boosting with GridSearch	6131.37	0.4403	0.4404	631.55	47.55
Gradient Boosting with KFold & GridSearch	6141.60	0.4394	0.4394	633.25	47.46
Simple LGBM	6088.69	0.4442	0.4442	636.91	46.85
LGBM with GridSearch	6090.67	0.4441	0.4441	632.97	47.36
LGBM with with KFold & GridSearch	**4987.84**	**0.5447**	**0.5447**	**550.46**	**43.43**

Based on the evaluation, the LightGBM model with KFold and GridSearch emerged as the best-performing model for predicting Airbnb listing prices in New York City. The LightGBM model is a gradient boosting method that builds an ensemble of weak learners (shallow decision trees) sequentially. Each new tree in the ensemble attempts to correct the errors made by the previous trees. The final prediction is the weighted sum of the predictions made by all individual trees.

In conclusion: The LightGBM model with KFold and GridSearch achieved a lower mean squared error, higher coefficient of determination (R^2), and lower mean absolute error compared to the other models. This indicates that the LightGBM model provides more accurate and reliable predictions, making it the best choice for predicting Airbnb listing prices in New York City while balancing performance and computational efficiency.

References:

1. Online: https://www.kdnuggets.com/2022/09/decision-tree-pruning-hows-whys.html
2. Online: New York City Airbnb Open Data | Kaggle

Implement a Decision Tree from scratch on Python allowing to deal with both a regression task and a classification task.

Introduction:

Decision trees are non-parametric supervised learning models that use Boolean decision rules to predict outcomes based on independent variables. They are suitable for both classification and regression problems. The algorithm splits the feature space into rectangles until a stopping criterion is met and fits a simple model to estimate the label y. Decision trees consist of root nodes, interior nodes, and leaf nodes. Root nodes initiate the tree, interior nodes create sub-decisions, and leaf nodes determine the final partition and output. The choice of metric depends on the problem type, and impurity metrics are used to quantify the information content of each feature. At each node, the most informative feature is selected for splitting, along with an optimized split value.

Decision tree architecture:

The architecture of this Decision Tree model is based on a tree structure, where each internal node represents a decision based on a feature's value, and each leaf node represents the predicted output or class. The model is implemented using object-oriented programming in Python, with classes for TreeNode, DecisionTree (abstract base class), DecisionTreeClassifier, and DecisionTreeRegressor.

1. TreeNode (class):
   This class represents a single node in the decision tree. Each node has a split value, split feature, left and right children, and a leaf value. Functions are provided to set the parameters (split_value and split_feature) and children (left and right) of a TreeNode.

2. DecisionTree (abstract base class):
   This is an abstract base class for both the classifier and regressor. It initializes the decision tree with a root node, maximum depth, and minimum samples required for a split. It contains abstract methods for impurity and leaf value calculations, as well as functions for growing the tree, traversing the tree, training, and predicting. The DecisionTree class will not be instantiated directly, but rather inherited by the classifier and regressor classes.
   Abstract methods: _impurity and _leaf_value for impurity calculation and leaf value determination, respectively. These methods are implemented in the

inheriting classes (DecisionTreeClassifier and DecisionTreeRegressor).

Methods:

_grow: Recursively constructs the decision tree, determining the optimal split point for each node based on the impurity measure.

_traverse: Recursively traverses the decision tree to predict the output for a single input instance.

train: Trains the decision tree using input data X and target labels y.

predict: Predicts the output for input data X using the trained decision tree.

3. DecisionTreeClassifier (class):
   This class inherits from the DecisionTree class and is specifically designed for classification tasks. It calculates Gini impurity or entropy-based impurity, depending on the 'criterion' parameter. The leaf value is calculated using the mode of the target labels in the leaf.

Impurity Criterion

Gini Index

$$I_G = 1 - \sum_{j=1}^{c} p_j^2$$

p_j: proportion of the samples that belongs to class c for a particular node

Entropy

$$I_H = -\sum_{j=1}^{c} p_j log_2(p_j)$$

p_j: proportion of the samples that belongs to class c for a particular node.

*This is the the definition of entropy for all non-empty classes ($p \neq 0$). The entropy is 0 if all samples at a node belong to the same class.

4. DecisionTreeRegressor (class):
   This class inherits from the DecisionTree class and is specifically designed for regression tasks. It calculates mean squared error (MSE) or mean absolute error (MAE) impurity, depending on the 'criterion' parameter. The leaf value is calculated using the mean of the target values in the leaf.

$$\text{MAE} = \underbrace{\frac{1}{n}\sum_{i=1}^{n}}_{\text{test set}} \underbrace{|y_i}_{\text{predicted vaue}} - \underbrace{\hat{y}_i|}_{\text{actual value}}$$

$$MSE = \frac{1}{n}\Sigma\underbrace{\left(y - \hat{y}\right)^2}_{\text{The square of the difference between actual and predicted}}$$

The tree structure is built using a top-down, recursive approach called the "Recursive Binary Splitting" algorithm. The algorithm starts at the root node and recursively splits the data into left and right subsets based on the feature values and impurity measure. The splitting process continues until a stopping criterion is met, such as reaching the maximum depth or having too few samples to split a node. At that point, the node becomes a leaf, and the leaf value is calculated based on the target labels or values in the leaf.

Random Forest architecture:

The architecture of this Random Forest model is based on an ensemble of decision trees, where each tree is trained on a bootstrap sample of the dataset and a random subset of features. The main idea is to combine multiple weak learners (decision trees) to create a more robust model that can generalize better and reduce overfitting. This implementation of Random Forest is based on the implementation of DecisionTreeClassifier and DecisionTreeRegressor that we implemented previously. The RandomForest class has the following components:

RandomForest class:

1. Initialization: Initializes a RandomForest model with the number of estimators (decision trees), maximum depth, minimum samples required to split a node, and the number of features to consider when looking for the best split (max_features).
2. _bootstrap_sample: A method to create a bootstrap sample of the data (with replacement) of the same size as the original data.
3. _fit_tree: A method to fit a single decision tree on a bootstrap sample of the data. The number of features to be considered is determined based on the max_features parameter. The tree is fitted on the selected features, and the tree and the selected features are returned.
4. train: A method to train the Random Forest model. This method initializes the trees and tree_features lists. For each estimator, it creates a bootstrap sample of the data, fits a single decision tree on the sample, and appends the tree and the selected features to the corresponding lists.
5. predict: A method to make predictions using the trained Random Forest model. This method iterates through the trees and their corresponding features, predicts the output for the input data using the selected features, and appends the predictions to the predictions list. The final output is calculated by taking the mode of the predictions across all trees.

The code demonstrates how to create a custom Random Forest implementation by creating an ensemble of decision trees and combining their predictions. The decision trees are trained on bootstrap samples of the data, and a subset of the features is considered when

making a split at each node. The final prediction is determined by aggregating the predictions of all the trees in the forest, which helps to improve the overall performance and reduce overfitting compared to a single decision tree.

Evaluations: Now that the models are implemented, we can proceed to test their performance. For classification we use, the “breast cancer” dataset from the scikit-learn; and for regression we use, the "California Housing" datasets from scikit-learn:we test the model and we have this performance below:

Decision Tree:

Our Implementation	Sklearn
Classification accuracy: 0.924 Regression (MSE): 0.7361	Classification accuracy: 0.959 Regression (MSE): 0.632

Random Forest:

Our Implementation	Sklearn
Classification accuracy: 0.9508 Regression (MSE): 0.6877	Classification accuracy: 0.9691 Regression (MSE): 0.6531

The performance differences between our models and that of scikit-learn can be explained by several reasons:

Implementation: Scikit-learn is a highly optimized library and uses efficient algorithms to build decision trees. Our implementation is quite simple and might not be as optimized, which can lead to differences in the quality of the trees built and, therefore, in the performance of the model.

Special case handling: Scikit-learn more robustly handles special cases and implementation details, such as balanced splits and stopping conditions. Our simplified implementation may not handle these cases as effectively, which may affect model performance.

Default Settings: The default settings for decision tree models in scikit-learn may be different from the ones we used in our implementation. Differences in parameters, such as maximum depth, minimum number of samples for splitting, and impurity criterion, can affect model performance.

Stability: Decision trees are sensitive to variations in training data. A small change in the training data can result in a very different decision tree. Scikit-learn can use techniques to make the decision tree more stable, which can improve model performance.

References:

1. Decision tree methods: applications for classification and prediction, Yan-yan SONG and Ying LU, 2015
2. Decision Tree Learning, Vili Podgorelec and Milan Zorman, 2019