Hooman HALABICHIAN
MSc AI
GitHub link: https://github.com/Ensemble-learning-project/Ensemble-learning

Predicting Airbnb Prices in New York City

Introduction:

New York City, a bustling metropolis with over 8 million residents, is famous for its iconic landmarks, diverse culture, and vibrant nightlife. The online platform Airbnb offers unique accommodations in the city, allowing travelers to experience local life. To analyze New York City's Airbnb market, a dataset containing detailed listings is imported, along with necessary packages for effective data processing and analysis. It is important for travelers to ensure their Airbnb bookings comply with the city's strict short-term rental regulations.

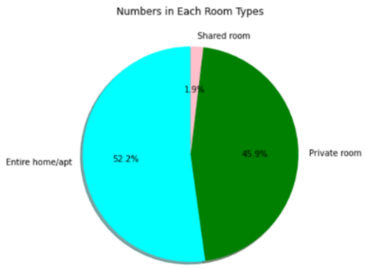
Exploratory Data Analysis:

In this analysis of the New York City Airbnb dataset, several crucial steps are taken to gain valuable insights into the market trends and patterns. This involves an in-depth exploration of the data, addressing various aspects such as preprocessing, handling outliers, and visualizing categorical data.

Data Preprocessing: 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.

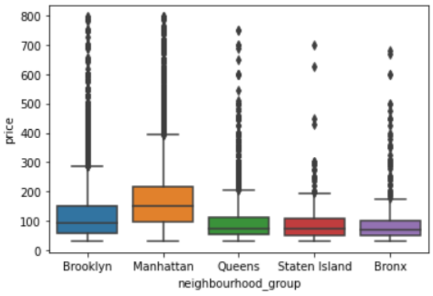
Outlier Handling: Outliers can skew the results of the analysis and negatively impact the performance of predictive models. Therefore, it is essential to identify and address any potential outliers within the dataset. Various statistical techniques and visualizations, like box plots and scatter plots, 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.

Categorical Data Visualization: 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.

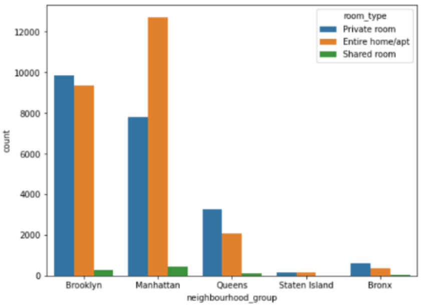


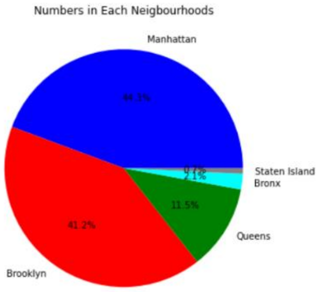
Neighborhood Analysis: The dataset is further examined to understand the distribution of listings and prices across five neighborhood groups in New York City - Bronx, Brooklyn, Manhattan, Queens, and Staten Island. Manhattan and Brooklyn emerge as the most sought-after neighborhoods, boasting a higher 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.

By conducting a comprehensive exploration of the dataset, several trends and patterns emerge that provide valuable insights into the factors influencing the supply and demand of accommodations within the New York City Airbnb market. This analysis enables a better understanding of the preferences of travelers and the performance of different neighborhoods and room types.

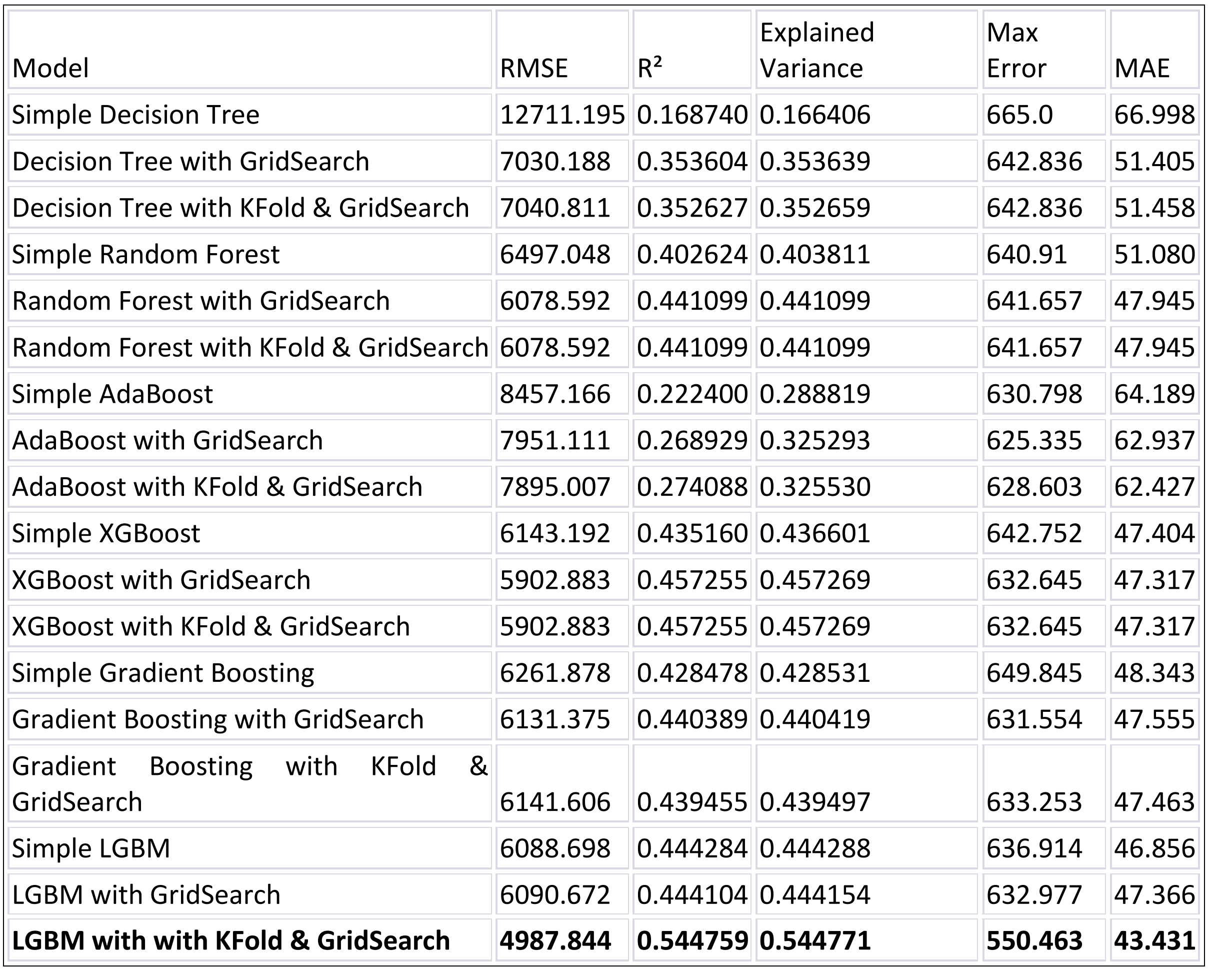
In conclusion, the in-depth analysis of the New York City Airbnb dataset sheds light on various aspects of the market, such as the distribution of room types, price dynamics across neighborhoods, and the factors that contribute to price variation. By addressing data quality issues, handling outliers, 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.

Data Preprocessing, models, 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.

During preprocessing, 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.

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. 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 K-fold cross-validation for robust performance evaluation. Performance metrics were used to compare the models, such as Root Mean Square Error (RMSE), Coefficient of Determination (R²), 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.



Model	RMSE	R²	Explained Variance	Max Error	MAE
Simple Decision Tree	12711.195	0.168740	0.166406	665.0	66.998
Decision Tree with GridSearch	7030.188	0.353604	0.353639	642.836	51.405
Decision Tree with KFold & GridSearch	7040.811	0.352627	0.352659	642.836	51.458
Simple Random Forest	6497.048	0.402624	0.403811	640.91	51.080
Random Forest with GridSearch	6078.592	0.441099	0.441099	641.657	47.945
Random Forest with KFold & GridSearch	6078.592	0.441099	0.441099	641.657	47.945
Simple AdaBoost	8457.166	0.222400	0.288819	630.798	64.189
AdaBoost with GridSearch	7951.111	0.268929	0.325293	625.335	62.937
AdaBoost with KFold & GridSearch	7895.007	0.274088	0.325530	628.603	62.427
Simple XGBoost	6143.192	0.435160	0.436601	642.752	47.404
XGBoost with GridSearch	5902.883	0.457255	0.457269	632.645	47.317
XGBoost with KFold & GridSearch	5902.883	0.457255	0.457269	632.645	47.317
Simple Gradient Boosting	6261.878	0.428478	0.428531	649.845	48.343
Gradient Boosting with GridSearch	6131.375	0.440389	0.440419	631.554	47.555
Gradient Boosting with KFold & GridSearch	6141.606	0.439455	0.439497	633.253	47.463
Simple LGBM	6088.698	0.444284	0.444288	636.914	46.856
LGBM with GridSearch	6090.672	0.444104	0.444154	632.977	47.366
LGBM with with KFold & GridSearch	**4987.844**	**0.544759**	**0.544771**	**550.463**	**43.431**

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. The LightGBM model with KFold and GridSearch achieved a lower mean squared error, higher coefficient of determination (R²), 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.

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. When building a decision tree model, we must consider handling missing values, avoiding overfitting through dimensionality reduction, and balancing the dataset to prevent bias. 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.

Mathematical approach:

Our training data consists of independent X variables and a dependent y variable, the dataset comprises N samples and M features. The feature set is represented by the column vectors $\{x_f\}$ that make up X, where f=1…M. The objective of splitting the dataset is to identify the feature f^*, and cutoff value x_{f*} for this feature, such that the impurity metric is minimized. We define the total data matrix as the combination of X and y, then the splitting procedure can be expressed as follows:

$$\mathbf{D}_{left} = \mathbf{D}|\mathbf{x}_{f^*} \leq x_{f^*}$$

$$\mathbf{D}_{right} = \mathbf{D}|\mathbf{x}_{f^*} > x_{f^*}$$

The impurity for this split is measured using the chosen impurity metric I_p:

$$I_p^{node} = \frac{N_{left}}{N_{node}} I_p(\mathbf{D}_{left}) + \frac{N_{right}}{N_{node}} I_p(\mathbf{D}_{right})$$

The lowest possible I_p^{node} :

$$\{f^*, x_{f^*}\} = argmin_{\{f, x_f\}} I_p^{node}$$

The selection of $\boldsymbol{I_p}$ will vary based on whether we are solving a regression or classification problem. In regression trees, the label **y** can take on continuous values, whereas classification trees only allow for discrete values.

When it comes to classification, we can start by defining the proportion of class **c** at the current node:

$$p_c^{node} = \frac{1}{N_{node}} \sum_{i \in N_{node}} \mathbf{I}(y_i = c)$$

In the above equation, **I** represents the identity matrix and **i** is the index over all samples at the current node. For classification problems, the gini impurity is a commonly used impurity metric:

$$I_p(\mathbf{D}_{node}) = \sum_c p_c^{node}(1-p_c^{node})$$

The gini impurity intuitively measures the probability of misclassifying a datapoint at the given node. A value of 0 indicates a perfect fit to the data. Another measure we could use is the Shannon entropy:

$$I_p(\mathbf{D}_{node}) = -\sum_c p_c^{node} \log_2(p_c^{node})$$

In the case of regression, we begin by calculating the mean of the label values y:

$$y^{node} = \frac{1}{N_{node}} \sum_{i \in N_{node}} y_i$$

The next step is to calculate the mean squared error (MSE) of each sample label in the node:

$$I_p(\mathbf{D}_{node}) = \frac{1}{N_{node}} \sum_{i \in N_{node}} (y_i - y^{node})^2$$

The mean absolute error (MAE) can also be used to measure the impurity of each sample label in the node with respect to:

$$I_p(\mathbf{D}_{node}) = \frac{1}{N_{node}} \sum_{i \in N_{node}} |y_i - y^{node}|$$

Explication of the code:

The functions available in the Node class are:

- init(self): This function is the initializer and is executed automatically when an instance of the class is declared.
- set_params(self, Bs, Bf): This function is used to assign values to the split parameters for the current node.
- get_params(self): This function returns the split parameters for the current node.
- set_chi(self, left, right): This function assigns nodes to the current node. The input arguments "left" and "right" are also expected to be of the Node type.
- get_left_node(self): This function returns the left node.
- get_right_node(self): This function returns the right node.

Next, we will define classes to construct our trees. The structural similarities between regression and classification trees will be leveraged, with the primary differences lying in the impurity metrics and computation of leaf node values. There will be:

- One class that encompasses the shared functionality for both classification and regression use cases.
- Additionally, a separate class will be created for the unique functionality of the classification use case and another for the regression use case.

Within the **class DecisionTree**, the constructor includes two hyperparameters for the model: **max_depth** and **min_samples_split.**

The max_depth parameter determines the maximum number of levels that can be reached while building the tree. Similarly, min_samples_split specifies the minimum number of samples required to split a node.

These hyperparameters work to prevent overfitting of the model to the training data and should be determined via cross-validation techniques applied to the dataset.

DecisionTreeClassifier:

The class **DecisionTreeClassifier** requires the DecisionTree class as an input argument. Below are the member functions of the DecisionTreeClassifier class:

- __init__(self, max_depth, min_samples_split, loss): This function is an initializer that is automatically executed when a class instance is created. Model parameters are initialized in this function. Additionally, an extra hyperparameter is included, loss, which can take values of either 'gini' or 'entropy'.
- __gini(self, D): This private function is utilized to implement the gini impurity metric.
- __entropy(self, D): This private function is utilized to implement the entropy impurity metric.
- _impurity(self, D): This protected function defines an implementation for the abstract method in the base class. The choice of impurity metric is determined in this function.
- _leaf_value(self, D): This protected function defines an implementation for the abstract method in the base class. Leaf node values are determined in this function.

DecisionTreeRegressor:

The DecisionTreeRegressor class contains the following member functions:

- __init__(self, max_depth, min_samples_split, loss): This function is an initializer that automatically executes when a class instance is created. Model parameters are initialized in this function. Additionally, an extra hyperparameter is included, loss, which can take values of either 'mse' or 'mae'.
- __mse(self, D): This private function is utilized to implement the mean squared error impurity metric according to an equation.
- __mae(self, D): This private function is utilized to implement the mean absolute error impurity metric according to an equation.
- _impurity(self, D): This protected function defines an implementation for the abstract method in the base class. The choice of impurity metric is determined in this function.
- _leaf_value(self, D): This protected function defines an implementation for the abstract method in the base class. Leaf node values are determined in this function.

Now that the tree models are implemented, we can proceed to test their performance.

For classification we use, the breast cancer dataset from the scikit-learn library using the load_breast_cancer() function; we test the model and we have this performance below:

- accuracy: 0.93
- precision: 0.99
- recall: 0.92

For regression we use, we generated a toy dataset:

- rmse: 70.04
- mae: 56.44
- r2: 0.68