Ensemble Learning Project 2023

Daris CHUA
MSc DSBA student 2022-2023
ldarius.chua@student-cs.fr

Yanjie Ren
MSc DSBA student 2022-2023
yanjie.ren@student-cs.fr

Konstantina TSILIFONI
MSc DSBA student 2022-2023
konstantina.tsilifoni@student-cs.fr

Chih-Tung CHEN
MSc DSBA student 2022-2023
chihtung.chen@student-cs.fr

Hao XU
MSc DSBA student 2022-2023
hao.xu@student-cs.fr

PART I. AirBnb Price Prediction

1. Problem statement and motivation

In the field of accommodation, traditional hotel accommodation still occupies a huge market share, whereas Airbnb is growing as one of the largest accommodation providers. The supply of the accommodation market is highly dynamic and influenced by various factors such as economic conditions, government policies, and external factors like natural disasters or pandemics. Additionally, the features of an apartment or a room such as an area, construction quality, location, and amenities can also significantly impact its price. As a result, it can be challenging for real estate players or property owners to estimate the accurate price of a hotel room on the Airbnb platform.

Real estate players such as developers, investors, and agents rely on accurate pricing models to make informed decisions regarding the pricing strategy of hotel rooms or entire apartments. Estimating the price of an apartment involves analyzing various factors such as location, features, and market trends. A robust pricing model can assist real estate players in making informed decisions and ensure that the property is rented out at a fair price. Furthermore, having an accurate pricing model can help travelers make informed decisions and ensure that they do not overpay for stays in an apartment. Therefore, the development of a robust prediction model for accommodation prices on Airbnb is crucial for both property owners and travelers in the accommodation market. The dataset used to perform this task was taken from kaggle and refer to airbnb prices in New York city in 2019.

2. Preprocessing

2.1 Exploratory Data Analysis

This section will provide a comprehensive analysis of the Airbnb-NYC_2019 dataset, including descriptive statistics, visualizations, and outlier detection analysis and an explanation of how missing data was handled

Descriptive Statistics:

Summary statistics were computed for the numerical variables in the dataset. The table [appendix 1] shows the mean, standard deviation, minimum, and maximum values for each variable.

Data Visualization:

To gain a better understanding of the distribution of the variables in the dataset and identify any patterns or trends, several visualizations were created. Useful insights for instance include [appendix 2], where we can clearly see that the log(price) follows a normal distribution, in comparison to the distribution of price which is significantly skewed. Therefore, it was decided to use the log(price) for the model and the predictions to be made on that.

Outlier Detection:

Outlier detection was performed with a focus on the price columns. After outliers were identified, those accommodations were mapped to examine whether a certain region showed more outliers regarding the price. From the resulting map [appendix 3] we can see that there is a clear congestion of price outliers (purple circles) in the Manhattan area firstly and in the Brooklyn area secondly.

Missing Data Treatment:

Missing data was also identified as a necessary step and was filled in appropriately. Missing data was located in the name, host_name, reviews_per_month and last_review, out of those the first 2 include text, reviews per month is numerical and the last_review is a date column. The missing entries for text features were filled in with ‘unknown’ and for numerical features they were filled with 0. Before filling in the missing entries of reviews_per_month and last_review it was confirmed that a missing value in one of those columns coincides with a missing value in the other column, an information which was later used in the feature engineering step.

Overall, the exploratory data analysis provided valuable insights into the Airbnb dataset and helped gain a better understanding of the data. These insights will be used to inform the modeling approach and feature engineering strategies.

2.2 Feature Selection

After exploring and analyzing the dataset follows the step of feature selection. Firstly, certain features were dropped and not used any further, those features are ['id', 'name', 'host_id', 'host_name', 'last_review']. Features like the host_id, host_name cannot contribute to a regression model as they do not hold any important information on the listing. The column name, which describes the listing, even though interesting to explore, would require Natural Language Processing models, which is out of the scope of this project. The column last_review was incorporated into the model by other engineered features that were created based on it.

Finally, from the correlation matrix [appendix 4] it follows that there is no high correlation between the rest of the variables, therefore it is safe to include them all into the model.

2.3 Feature Engineering

As a next step, feature engineering was performed to extract meaningful information that can help in predicting the prices of accommodations. Firstly, a binary column "new_listing" was created that categorizes listings as new or existing. This was done based on whether or not the listing had a previous review. After creating this feature, we tested whether there is a statistically significant difference on the average price of new and old listings. The average price of old listings is 142.31 and of new listings 192.91, which based on a hypothesis test that was performed is indeed a significant difference and therefore the feature was considered important to keep. The data was then visualized on a map [appendix 5], where it can be observed that most of the new listings are located in Manhattan and Brooklyn, whereas Staten Island has the least number of new listings.

After creating this new feature, the missing values in the "last_review" column were filled with zero and converted to datetime format. Next, a new column "review_recency" was created that represents the number of days since the last review of a listing.

Additionally, two more columns "all_year_avail" and "low_avail" are generated to categorize listings into those that are available year-round and those that have low availability, respectively. This feature was inspired by the distribution of the feature “availability_365” [appendix 6], where it can be seen that there are two peaks in the figure of distribution, meaning that the number of days most listings can be booked is either less than 12 days or more than 350 days a year. Hence, it was decided that this characteristic should be captured by grouping the listing into these groups and not by the absolute number of days a listing is available.

Furthermore, to capture the number of months a listing has been on the platform, a new column "months" was created by dividing the number of reviews by the reviews per month. The resulting values were then filled with zero where there were missing values.

To incorporate categorical features, one-hot encoding was performed on the "room_type" and "neighbourhood_group" columns, resulting in new binary columns that represent the categories of each feature. Finally, the "neighborhood" column was encoded using label encoding to convert the categorical values into numeric codes.

Overall, these engineered features provide valuable insights that can be used to develop a model for predicting Airbnb accommodation prices.

Feature scaling

Before the data were ready to be inserted into the model they required standardization, because the features were currently in significantly different scales. All numerical features were standardized apart from the binary (0,1) features as those were already in an acceptable scale. The standardization was performed after the split on train and test data to avoid data leakage from the test data into the training data. In general, performing scaling before the train-test split, might lead to scaling the test data as well, which could result in overfitting and poor generalization performance. So, the approach followed was to first split the data into training and test sets and then perform scaling on the training set only, using the scaling parameters obtained from the training set, and then transform the test set using those parameters.

3. Methodology

For the prediction task at hand, a variety of ensemble algorithms was selected, such as decision trees, random forests and different boosting algorithms. The reason behind choosing ensemble algorithms is that they can improve the performance of individual models by combining multiple weak learners.

First of all, the dataset was split into training and test sets using a 70/30 ratio. The training set was used to train the models, while the test set was used to evaluate their performance.

3.1 Decision Tree

As a starting point, a basic model was trained using a simple Decision Tree with default hyperparameters. The decision tree is a simple yet powerful algorithm that is intuitive to visualize and interpret. However, it can easily overfit and lead to a large and unpruned tree with small bias but large variance. For such reason, grid search with 5-fold cross-validation was used to tune the hyperparameters of the model, such as setting up the maximum depth of the tree, the minimum number of samples required to split an node, and the minimum number of samples to required to be at a leaf node. This has an essential effect of smoothing the model, especially in a regression task. Then the best hyperparameters were selected based on the performance of the MSE score of all fits. The hyperparameters used to find the best

Decision Tree model and the best resulting parameters can be seen in appendix 7.

3.2 Bagging

Being aware of the potential issue of high variance in the decision tree model, we decided to use Bagging as a means of mitigating this concern. More specifically this approach involves creating multiple subsets of the original training data by randomly sampling observations with replacement. A separate model, such as a decision tree, is then trained on each of these subsets. The predictions of each model are combined through averaging or voting to obtain the final prediction.

A Bagging Regressor is a specific implementation of this technique for regression problems. It uses an ensemble of decision trees to make predictions by averaging the outputs of each individual tree. This can help to reduce the variance of the model and improve its accuracy. The hyperparameters used to find the best Bagging model and the best resulting parameters can be seen in appendix 8.

3.3 Random Forest and Extremely Random Forest

Despite the powerfulness of bagging, there are two reasons why bagging may not be a good choice in the case at hand. The main problem of Bagging stems from the different bootstrap trees, which are correlated with each other, as the variance of the bagging model cannot shrink below a certain threshold that is proportional to the correlation coefficient between different trees. This might potentially result in an outcome with high variance and vulnerable to minor fluctuations.

Moreover, bagging may not be a suitable model since there is already a sufficient number of observations in our dataset, and therefore there is no need to take any risks by creating duplicate observations in the subsets.

Hence, the next implemented model was Random Forest, which uses a random choice of input features to split each node. Random forests are an extension of decision trees that can reduce overfitting and improve the performance of the model. Several random forest models were trained with different hyperparameters using grid search with 5-fold cross-validation. The varied parameters were the number of trees, the maximum depth of the trees, and the minimum number of samples required to split a node.

Moreover, extremely random forest models were also trained to compare their performance with random forests. Extremely random forests are a variant of random forests that introduce more randomness by randomly choosing a split point, which can further reduce overfitting and improve the performance of the model.The hyperparameters used to find the best Random Forest model and the best resulting parameters can be seen in appendix 9.

3.4 Boosting

Apart from the aforementioned models several boosting models were also used and trained, including AdaBoost, Gradient Boosting, and XGBoost, with different hyperparameters. Boosting is another ensemble technique that can improve the performance of weaker models. For each boosting algorithm, different hyperparameters were tuned to find the best ones for the models.

AdaBoost: at each step data are reweighted to minimize the sum of the weighted absolute errors.

Gradient Boosting: a generalization of AdaBoost that uses a gradient descent algorithm to minimize the loss function.

XGBoost: an optimized implementation of gradient boosting that uses regularization and parallel processing to improve performance.

The hyperparameters used to find the best AdaBoost model and the best resulting parameters can be seen in appendix 10, the ones for the best Gradient Boosting model in appendix 11 and the ones for XGBoost in appendix 12.

In addition to using a default Decision Tree as a weak learner for the AdaBoost model, Random Forest model was also used as a weak learner to train our AdaBoost model. Nevertheless, in comparison to the results obtained from the random forest model, the improvement garnered from this approach was minimal. Considering the significant computational expense incurred by using the random forest as a weak learner, the output obtained was not particularly noteworthy.

3.5 Stacking

Stacking is an ensemble learning method that involves combining several models to improve the predictive performance. Stacking involves training several base models on the same dataset, then using the predictions of these models as input to a higher-level model, which makes the final predictions.

One of the advantages of stacking is that it can capture more complex interactions between features than individual models, leading to better predictive performance. Additionally, it can mitigate the weaknesses of individual models by combining their strengths.

In our case, three models were stacked : Bagging, XGBoost, and Gradient Boosting. This allowed for leverage the strengths of each individual model while mitigating their weaknesses. Specifically,

Bagging was used to reduce variance, XGBoost to capture nonlinear interactions, and Gradient Boosting to improve the accuracy of the predictions. After that, a meta-model was trained on the predictions of these models to make the final predictions.

4. Evaluation

In terms of model evaluation, 5-fold cross-validation was used for all models to avoid the significant impact that comes from the characteristics of some individuals. After training the models, the performances of the models were evaluated on the testing set using several metrics, including mean squared error (MSE), root mean squared error (RMSE), and R-squared (R2). R2 measures how well the algorithm fits the data and ranges from 0 to 1, with 1 being a perfect fit. RMSE measures the difference between the predicted and actual values, with lower values indicating better performance. RMSE after transform measures the RMSE after exponentiating the predicted price values back to their normal scale. Since this is a regression task, the objective is to minimize the impact of significant errors, it would be more appropriate to use metrics such as Mean Squared Error (MSE) or Root Mean Squared Error (RMSE) to evaluate the performance of the models.

5. Results and comparison

The performance results of all the models used are summarized and presented in the table below.

Model	**R2**	**RMSE train**	**RMSE test**	**RMSE test after transform**
Decision Trees	55,44 %	0,434	0,458	182,01
Bagging	60,78 %	0,204	0,430	177,19
Random Forest	59,98 %	0,302	0,434	179,18
Extremely random forest	57,96 %	0,442	0,445	183,49
XGBoost	61,08 %	0,369	0,428	175,10
Adaboost	59,67 %	0,370	0,436	179,42
Gradient Boosting	60,94 %	0,402	0,429	177,14
Stacking	61,78 %	0,324	0,424	176,02

From this table it can be concluded that the best performing algorithm is the Stacking Regressor, in which 3 models where stacked, the bagging, the xgBoost and the gradient boosting regressors. This does not pose a surprise since the stacking of certain base algorithms should on average perform better than those algorithms individually.

Our results showed that all algorithms had an R2 value greater than 0.5, indicating a good fit to the data. The highest R2 value was achieved by the Stacking algorithm with a value of 0.6178, followed closely by XGBoost with a value of 0.6108.

In terms of RMSE on the train, Bagging had the lowest value at 0.204, followed however by a RMSE on the test data of 0,43, which poses the biggest gap between train and test performance of all algorithms, giving an indication of probable overfitting.

Based on the test RMSE values, the Stacking Regressor (0.424) exhibited the best performance among the evaluated algorithms. This model involved stacking three individual models, namely, Bagging, XGBoost, and Gradient Boosting Regressors. It is not surprising that the Stacking Regressor outperformed the individual models, as the stacking of multiple base algorithms is known to result on average in improved predictive performance.

In terms of the RMSE after being transformed back to the original price scale, it can be observed that the XGBoost algorithm had the lowest value at 175.1, followed closely by the Stacking algorithm with a value of 176.02. The Decision Tree algorithm yielded the highest value of 182.01, which aligns with expectations as the Decision Tree is considered to be the weakest learner among the employed algorithms.

6. Conclusion

Based on our results, we conclude that Stacking and XGBoost are the most effective algorithms for this particular dataset, achieving high R2 values and low RMSE and RMSE after transform values. However, it is worth noting that the performance of the algorithms may vary depending on the specific characteristics of the dataset, and further research is necessary to validate our findings on other datasets. Moreover, the Stacking algorithm, which consists of a combination of three base algorithms, performed remarkably well, and its success can be attributed to the process of stacking, which averages the outputs of several models to make a more robust prediction. This highlights the advantage of ensemble learning methods over individual models in producing more accurate predictions.

PART II. Implement a DT from scratch

1 Introduction

Decision tree is a popular non-parametric and supervised machine learning algorithm used for both classification and regression tasks. It works by recursively splitting the data based on the values of the independent variables until the samples in each branch belong to the same class or a stopping criterion is met.
In this project, we built a decision tree from scratch to fit both regression and classification tasks. The tree was constructed using the CART methodology and a greedy approach. The tree was later used to make predictions for the regressor task by predicting the values and for the classifier task by predicting the class labels. Lastly, accuracy was evaluated based on respective metrics.

Our github repository mainly composes four different files: algorithm.py that provides the decision tree algorithm built from scratch, two .ipynb notebooks that respectively solve a classification and regression problem using our self-designed decision tree and a csv file that contains the data we used for both problems.

2 Decision tree building

2.1 Decision tree algorithm

Before we dive into the details of how decision trees are able to solve different kinds of prediction problems, we first need to understand step by step how the algorithm is implemented and which metrics are used in the process of designing the algorithm. Under this goal, we will first look into the algorithm in details:

Step 1: Define the metric of accuracy

The primary step is to choose a proper impurity metric of the dependent variable which will serve as the measure of accuracy of our decision tree. If the dependent variable is continuous, calculate the mean squared error (MSE) of the samples. If the dependent variable is categorical, calculate the Gini impurity or entropy of the samples.

Step 2: Define the metric of information gain

In order to compare different split options, calculate the impurity reduction or information gain for each independent variable using a proper metric as the splitting criterion.

Step 3: Find the best split

Choose the independent variable with the highest impurity reduction or information gain as the first node of the decision tree and then split the data based on the values of the selected independent variable. Create a new branch for each value of the variable in order to find the best splitting value for it. If either of the child datasets is empty, it creates a leaf and adds it to the list of leaves.

Step 4: Repeat the process unless stopping criterion

For each branch, repeat the process from step 1, using the subset of samples that belong to that branch. The decision tree algorithm may stop splitting the data when one of the following conditions is met:

- All samples in a branch belong to the same class.
- All independent variables have been used as nodes in the tree.
- The max_depth of the tree reaches a predefined value.

Once the decision tree is built, we can then apply it to make predictions on new data.

2.2 Decision tree classifier and regressor

According different types of questions, there will be some nuances between the decision tree algorithm:

For classification problems, it is suggested in our algorithm to use the Gini index (see below for formula) as a metric to measure both model accuracy and information gain at each potential split.

$$Gini(S) = 1 - \sum_{i=1}^{n} (p_i)^2$$

$$Overall\ Gini = p_L * Gini(L) + p_R * Gini(R)$$

where p_L is the probability of nodes occurring on the left and p_R is the probability of nodes occurring on the right after splitting.

For regression problems, we used MSE which proved to be a better metric for continuous data.

$$MSE(S) = \frac{1}{n} * \sum_{i=1}^{n} (y_i - \hat{y})^2$$

$$Overall\ MSE = p_L * MSE(L) + p_R * MSE(R)$$

3 Implementation and results

3.1 Dataset

For evaluating the performance of the decision tree, the dataset was chosen from Kaggle, containing 500 records of people's gender, height, weight, and body mass index.

The gender attribute and BMI index are categorical, whereas the height attribute and weight attribute are continuous. For classification task, 'Index' was chosen as the target, and for regression task, 'Height' was chosen to be the independent variable.

3.2 Result

The results of the decision tree model built from scratch and the optimized algorithm from package sklearn.tree were compared based on utilizing the same hyperparameters (min_samples_leaf=2, max_depth=6, random_state=42).

Algorithm	Accuracy (Classifier)	MSE (Regressor)
Decision Tree from scratch	0.81	**10.79**
Sklearn Decision Tree	**0.87**	12.93

Table.1. Comparison of results

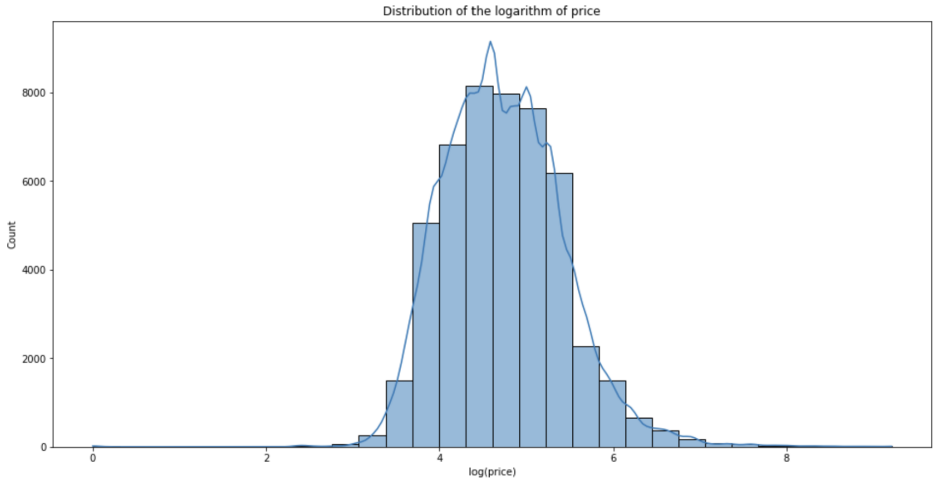
The results showed that the optimized algorithm achieved a higher accuracy, but a lower prediction error compared to the algorithm built from scratch. These results demonstrate that the optimized algorithm has a potential to be more efficient than the algorithm built from scratch.

Appendix

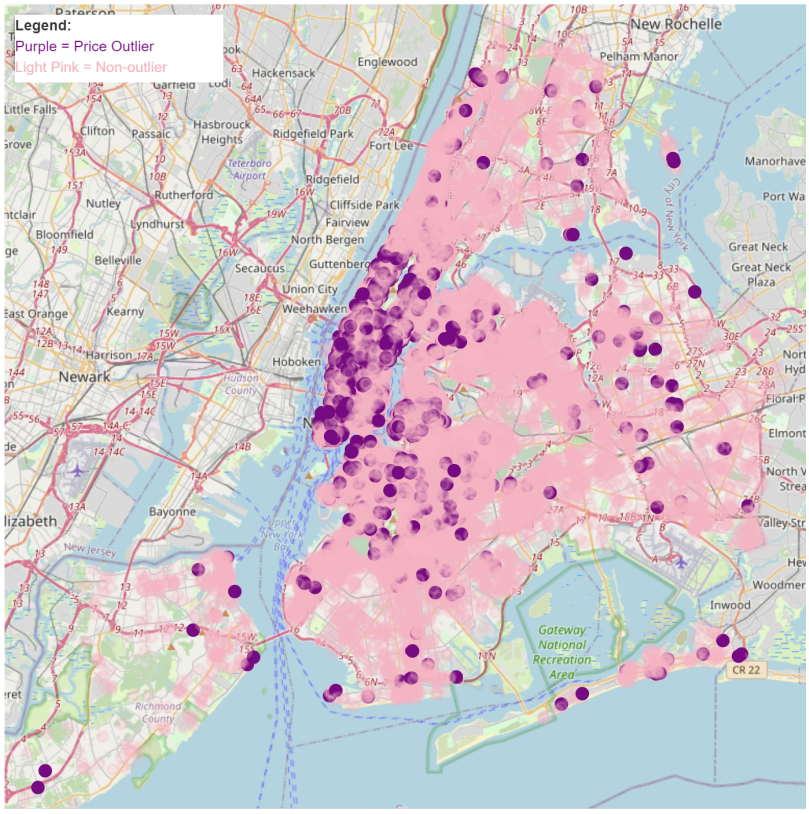
[1] Descriptive analytics table of numerical features

	latitude	longitude	price	minimum_nights	number_of_reviews	reviews_per_month	calculated_host_listings_count	availability_365
count	48895.000000	48895.000000	48895.000000	48895.000000	48895.000000	38843.000000	48895.000000	48895.000000
mean	40.728949	-73.952170	152.720687	7.029962	23.274466	1.373221	7.143982	112.781327
std	0.054530	0.046157	240.154170	20.510550	44.550582	1.680442	32.952519	131.622289
min	40.499790	-74.244420	0.000000	1.000000	0.000000	0.010000	1.000000	0.000000
25%	40.690100	-73.983070	69.000000	1.000000	1.000000	0.190000	1.000000	0.000000
50%	40.723070	-73.955680	106.000000	3.000000	5.000000	0.720000	1.000000	45.000000
75%	40.763115	-73.936275	175.000000	5.000000	24.000000	2.020000	2.000000	227.000000
max	40.913060	-73.712990	10000.000000	1250.000000	629.000000	58.500000	327.000000	365.000000

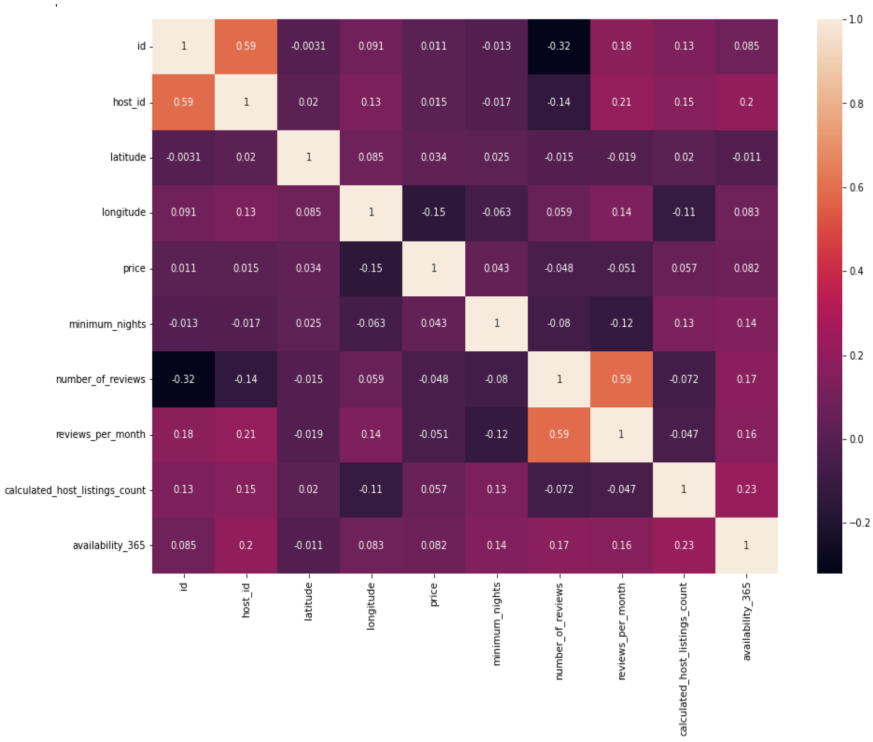
[2] Distribution of log(price)



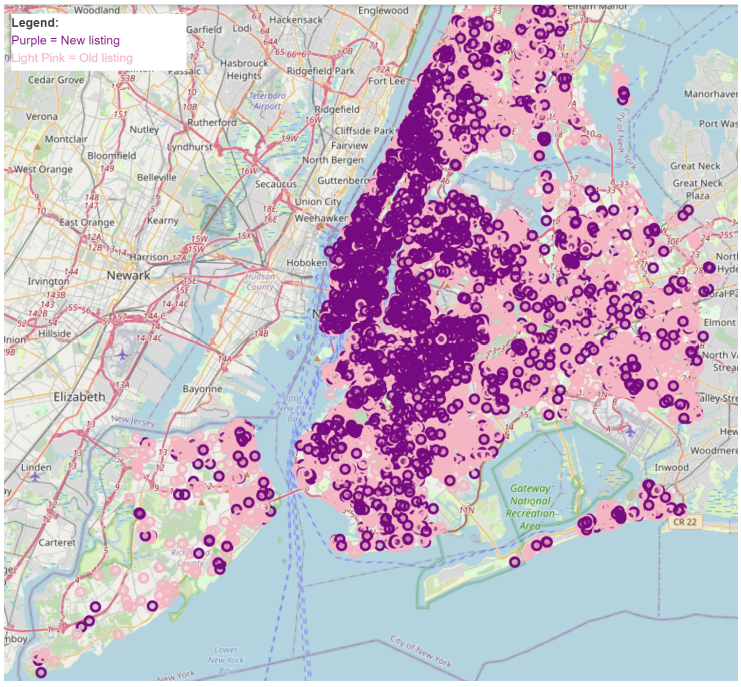
[3] Mapping of the listings that are price outliers



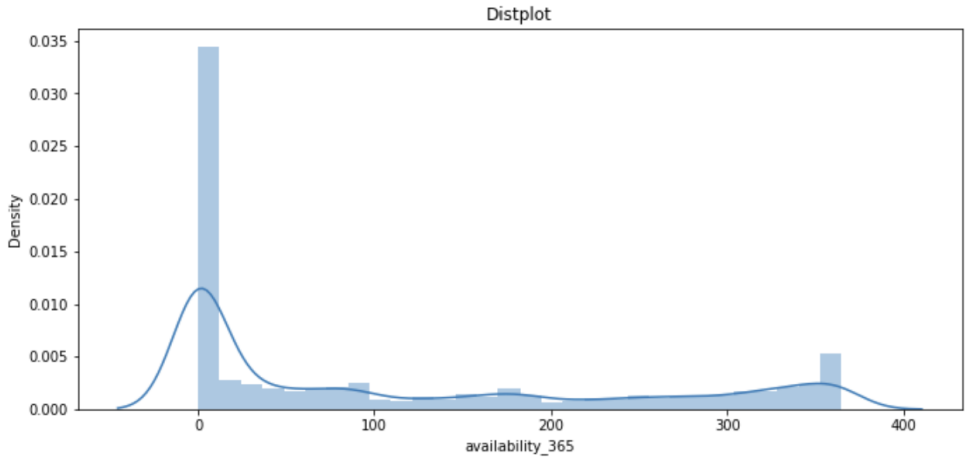
[4] Correlation matrix



[5] Mapping of the new listings



[6] Distribution of “availabilty_365 column”



[7] Hyper-parameter tuning of Decision Tree

Parameter	**Grid**	**Best parameter**
Max_depth	7, 8, 9, 10, 15	10
Min_simple_split	6, 8, 10, 15, 20	6
Min_simple_leaf	4, 5, 8, 10	10
Criterion	'friedman_mse', 'squared_error', 'poisson'	squared_error
Max_features	None, 'sqrt', 'log2'	None
Max_leaf_nodes	None, 10, 20, 30	None

[8] Hyper-parameter tuning of Bagging

Parameter	**Grid**	**Best parameter**
N_estimators	100, 150, 200	150
Max_samples	0.8, 1.0	0,8
Max_features	0.8, 1.0	0,8
Oob_score	True, False	False

[9] Hyper-parameter tuning of Random Forest

Parameter	**Grid**	**Best parameter**
Max_depth	2, 4, 6, 8, 10	10
Min_samples_split	2, 4, 6, 8, 10	2
Min_samples_leaf	1, 2, 3, 4, 5	4

[10] Hyper-parameter tuning of AdaBoost

Parameters	**Grid**	**Best parameter**
base_estimator_max_depth	2, 4, 6, 12	12
n_estimators	50, 100, 200	100
learning_rate	0.01, 0.1, 0.5, 1	0.01

[11] Hyper-parameter tuning of Gradient Boosting

Parameter	**Grid**	**Best parameter**
n_estimators	50, 100, 200	200
max_depth	2, 4, 6	6
learning_rate	0.1, 0.01, 0.05	0.1
min_samples_split	2, 4, 6	6
min_samples_leaf	1, 2, 4	4
max_features	0.3, 0.6, 0.9	0.3

[12] Hyper-parameter tuning of XGBoost

Parameter	**Grid**	**Best parameter**
min_child_weight	1, 5, 10	5
gamma	0, 0.5, 0.7, 1, 1.5	0.7
max_depth	5, 10, 15	10
n_estimators	70,100,110	90
learning_rate	0.1, 0.05, 0.01	0.1