ENSEMBLE LEARNING PROJECT REPORT

Abhishek Tiwari
MSc DSBA 2022-2023
abhishek.tiwari@student-cs.fr

Deepesh Dwivedi
MSc DSBA 2022-2023
deepesh.dwivedi@student-cs.fr

Namrata Tadanki
MSc DSBA 2022-2023
namrata.tadanki@student-cs.fr

Swaraj Bhargava
MSc DSBA 2022-2023
swaraj.bhargava@student-cs.fr

I. PREDICTING AIRBNB PRICES IN NEW YORK CITY

1 INTRODUCTION

In this project, we aim to analyze the New York City Airbnb dataset and build a regression model to predict the price of a property based on various descriptive features such as location, host information, and amenities. To achieve this goal, we will use ensemble learning methods like bagging and boosting, which combine multiple models to improve the accuracy of predictions.

2 DATASET DESCRIPTION

The New York City Airbnb Open Data is a dataset containing information about Airbnb listings in New York City. The dataset is sourced from Kaggle, and it includes detailed information about the Airbnb properties, their location, host information, pricing, and availability.

The dataset contains around 49,000 rows and 16 columns, with each row representing a unique Airbnb listing. The columns in the dataset include:

1. id: Unique identifier for the listing
2. name: Title of the listing
3. host_id: Unique identifier for the host
4. host_name: Name of the host
5. neighbourhood_group: Borough of the listing
6. neighbourhood: Specific neighbourhood of the listing
7. latitude: Latitude coordinate of the listing
8. longitude: Longitude coordinate of the listing
9. room_type: Type of listing (e.g. Entire home, Private room, Shared room)
10. price: Price per night for the listing
11. minimum_nights: Minimum number of nights required for a booking
12. number_of_reviews: Total number of reviews for the listing
13. last_review: Date of the last review for the listing
14. reviews_per_month: Number of reviews per month
15. calculated_host_listings_count: Number of listings for the host
16. availability_365: Number of days in the year that the listing is available for booking

3 PREPROCESSING

The preprocessing of the dataset included 3 steps – Exploratory Data Analysis, Dataset Cleaning, and Feature Engineering.

Performing Exploratory Data Analysis (EDA) is a crucial step in any project. It sets the foundation for understanding the data, any inherent patterns, identifying outliers and anomalies, missing values, and what features can be engineered to improve the performance of the models.

The first step in our EDA and preprocessing the data was to check for null values. We found that the 'last_review' and 'reviews_per_month' features had 20% missing values occurring in the same records. Since having no last review data implies the property never got reviewed, the null values in the 'reviews_per_month' were filled with 0 (since the property has no reviews), and the null values in the 'last_review' were imputed with the earliest date that occurs in the dataset.

The next step was to check for duplicate records and drop them if any. We also dropped the 'id', 'name', 'host_name', and 'host_id' columns as we believe they are irrelevant in predicting the property price.

We then checked if any numerical features had a skewed distribution. Log normalization may be appropriate if the feature has a high degree of skewness. For example, we observed that the 'minimum_nights' feature had a skewed distribution, and when log normalized, it resembled a normal distribution. Therefore, we performed log normalization of the 'minimum_nights'.

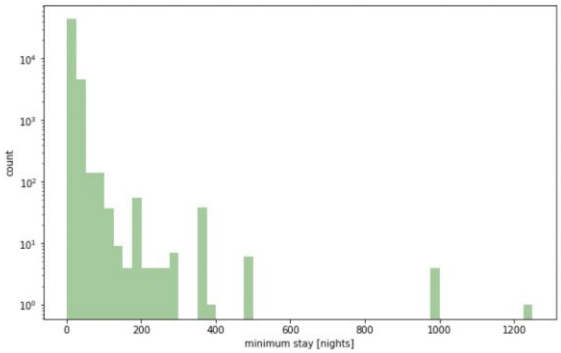

Fig.1. Distribution of minimum nights

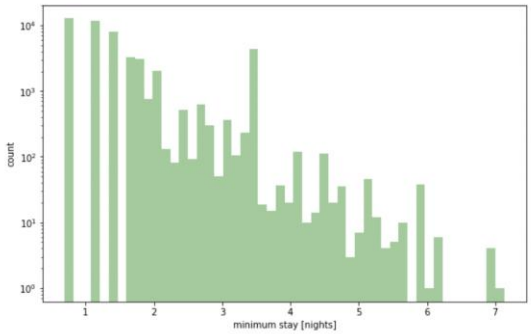

Fig.2. Distribution of minimum nights after log transformation

We also observed that the 'availability_365' which tells us how many nights the property is available ranged from 0-365.

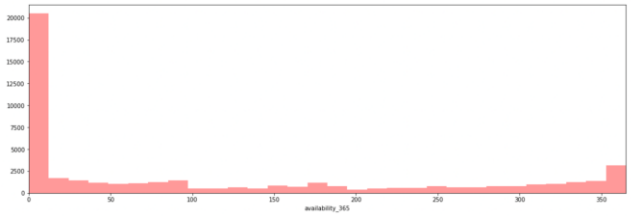

Fig.3. Availability of properties

Most properties are available for less than ~20-30 days or more than ~350 days. Hence, we converted this feature into three buckets based on availability - Low availability (below 30 days), Available (30 days to 350 days), and High availability (more than 350 days).

We also created a new feature called 'No_review', which takes boolean values depending on whether the property has no reviews.

The target variable, 'price', was observed to have a wide range, with values ranging from 0-10000. To deal with this, we performed log normalization of the target variable.

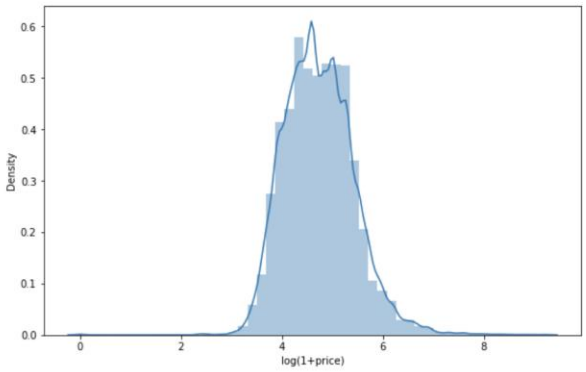

Fig.4. Target variable, Price, after log normalization

Most of the data lie in the range ~3-8. Hence, we decide to narrow down our dataset to those records.

Finally, we performed one-hot encoding of the three categorical features, 'neighbourhood_group', 'neighbourhood', and 'room_type', and robust scaling on the numerical features.

4 ENSEMBLE METHODS

4.1 BAGGING ALGORITHMS

4.1.1 RANDOM FOREST REGRESSOR

Random Forest Regressor is used to solve regression problems by constructing multiple decision trees and averaging their predictions. We trained our model with 500 estimators, squared error as the criterion, a maximum depth of 30, and a minimum of 5 samples for the leaf and split. The obtained results suggest that the model is capable of accurately predicting prices with a test error of ~17% and test R2 score of 61%.

4.1.2 BAGGED DECISION TREES

Next, we implemented Bagged Decision Trees using the BaggingRegressor Module in which multiple decision trees are generated using bootstrap sampling on the training dataset, and then aggregation is done by averaging their predictions. Unlike Random Forest, there is no random subsampling of features for each individual tree.

To do this, we first trained a baseline decision tree model with a maximum depth of 12, minimum sample leaf of 10 and square error criterion. This baseline tree performed with a training and test error of ~24% and training and test R2 score of ~46%. The bagged regression model was trained using 50 base decision trees and a maximum feature and sample size of 0.7 and 0.5, respectively. The obtained results suggest that the model is capable of accurately predicting prices with a training and test error of ~23% and training and test R2 score of ~50%.

4.1.3 BAGGED ELASTIC NETS

Elastic Nets are regularized linear regression models that leverage both L1 and L2 penalties to regularize the model and handle multicollinearity and perform variable selection. The BaggingRegressor module allows for bagging of different base regressors such as Elastic Net.

To create a bagged elastic net model, we first trained a baseline elastic net with hyperparameters alpha and l1_ratio set to 0.001 and 0.5, respectively, to control the strength and balance of the L1 and L2 regularization penalties. This baseline model performed with a training and test error of ~20% and training and test R2 score of ~56%. The bagged regression model was then trained using 50 base estimators and a maximum feature and sample size of 0.5 each. The obtained results suggest that the model is capable of accurately predicting prices with a training and test error of ~21% and training and test R2 score of ~53%.

4.1.4 EXTRA TREES REGRESSOR

Extra Trees Regressor is a bagging method that is similar to Random Forest and Bagged Decision Trees because it aggregates the results of multiple decision trees to obtain final predictions.

However, the main difference is that Extra Trees have a high degree of randomness in the attribute and split point selection, and the splitting thresholds are selected randomly instead of looking for the best split.

The obtained results suggest that the model overfit the training data with a training error of almost 0 and training R2 score of 99%. But it has a test error of 19% and test R2 score of 56%.

4.2 BOOSTING ALGORITHMS

4.2.1 RIDGE REGRESSION

Ridge regression is a type of regularized linear regression model that aims to prevent overfitting by adding a penalty term to the cost function. This penalty term, called the L2-norm penalty, shrinks the coefficients towards zero, reducing their variance and improving the model's generalization performance.

The obtained results give a train error of 0.19 and a train R2 of 0.58 whereas there is a test R2 of 0.57.

4.2.2 XGBOOST

XGBoost is an ensemble learning algorithm that uses a gradient boosting framework to build and combine multiple weak decision trees into a strong predictive model. It is designed to optimize performance and scalability, making it suitable for handling large datasets with high-dimensional features. XGBoost is known for its speed, accuracy, and ability to handle a wide range of data types, making it a popular choice for a variety of machine learning tasks. Its implementation includes features such as regularization, cross-validation, and parallel processing, making it a powerful tool for ensemble learning.

In our case, we got very encouraging results, with a 0.72 training R2 score and a 60.6 test R2 score.

4.2.3 CATBOOST

CatBoost uses an ordered boosting strategy that combines decision trees with a set of categorical-specific features to make accurate predictions. CatBoost is known for its ability to handle high-dimensional data, its automatic feature selection capability, and its ability to handle missing data.

In our case, we get training R2 of 0.68 and test R2 of 0.6123.

5 RESULTS AND CONCLUSION

The results obtained are shown below

5.1 Bagging Algorithms

	algorithm	CV error	CV std	training error	test error	training_r2_score	test_r2_score
0	Decision Tree - Base Model	0.264577	0.014911	0.241688	0.248268	0.473226	0.460607
0	ElasticNet - Base Model	0.198815	0.003982	0.198051	0.200588	0.568336	0.564198
0	Random Forest Regressor	0.176186	0.003676	0.079652	0.177473	0.826394	0.614417
0	Bagged Decision Trees	0.229394	0.007902	0.228333	0.233723	0.502335	0.492208
0	Bagged ElasticNet	0.224099	0.007103	0.214662	0.216622	0.532130	0.529363
0	Extra Trees Regressor	0.200175	0.005814	0.000002	0.198782	0.999997	0.568122

Table.1. Evaluation metrics of bagging algorithms

5.2 Boosting Algorithms

	algorithm	CV error	CV std	training error	test error	training_r2_score	test_r2_score
0	Ridge Regression	0.194570	0.003577	0.191916	0.000000	0.581708	0.573136
0	AdaBoostRegressor	0.298307	0.014713	0.312945	0.326759	0.317917	0.290076
0	XGBRegressor	0.178277	0.002888	0.126205	0.181032	0.724928	0.606685
0	CatBoostRegressor	0.174953	0.003059	0.151723	0.178623	0.669310	0.611920

Table.2. Evaluation metrics of boosting algorithms

Some general observations were made on the performance of different models.

1. In contrast to Bagged Decision Trees, in Random Forest, the algorithm generates multiple decision trees using bootstrap sampling. However, it also randomly selects a subset of features for each tree, which reduces the correlation between the trees and improves the model's accuracy.
2. Extra trees regressor can sometimes perform worse than random forest because of its high level of randomness. While this randomness can lead to increased diversity and variance reduction in the ensemble, it can also result in an increased number of poorly performing trees, particularly if the number of trees is relatively low.
3. The main reason Bagged Decision Trees might perform better than Bagged Elastic Nets is that decision trees can capture nonlinear relationships between the features and the target variable, while Elastic Net is a linear model and may not be able to capture such relationships as effectively. Moreover, bagged decision trees are less sensitive to outliers than linear models such as Elastic Net. They partition the feature space into regions where the data has similar characteristics, and outliers are less likely to affect the final prediction.
4. For Elastic Net and Bagged Elastic Net, the benefit of bagging multiple base models is that it controls instability and regularizes prediction. Since elasticnet already has regularization terms and generally not much instability, bagging elastic nets did not significantly affect the output scores.

II. IMPLEMENTING A DECISION TREE FROM SCRATCH

1 INTRODUCTION

For the implementation of a Decision Tree from scratch, the idea was to take the CART algorithm and use Information Gain to calculate the splits in the decision tree. For this, a DecisionTreeClassifier and DecisionTreeRegressor classes have been built which take fit the data and then predict the required results for the dataset. To store the information on each node, there is another node class that has been created which stores information on the root node, the value, the threshold, the feature that was used for splitting, and if it is a leaf node, the value associated with it.

2 DECISION TREE CLASSIFIER

2.1 Class, Methods, and Working

The DecisionTreeClassifier class has a root node that is initialized to None until the fit method is called, at which point the root node is set to the result of the _grow_tree method, which recursively grows the tree.

The _grow_tree method takes as input the input data (X) and target labels (y), and also has two optional parameters: depth (which is initialized to 0) and a flag n_features (which is set to None by default). The depth parameter keeps track of the depth of the current node, and the n_features parameter specifies the number of features to use for each split.

The _grow_tree method first calculates the number of samples, features, and labels in the input data. It then checks if the stopping criteria for growing the tree has been met. If any of the following conditions is true, the _grow_tree method returns a Node object with the value set to the most common label in the target labels:

1. The depth of the current node is greater than or equal to the maximum depth of the tree
2. The number of unique labels in the target labels is equal to 1
3. The number of samples in the current node is less than the minimum number of samples required to split a node

The _best_split method calculates the best split for the current node based on information gain. It first selects a random subset of features (n_features) to split on. For each feature, it calculates the information gain for each unique value of the feature and selects the value with the highest information gain as the split threshold. The feature index and split threshold with the highest information gain are returned as the best split.

The information gain is calculated as:

$$Information\ Gain = entropy(parent_{node}) - weighted\ average\ of\ the\ entropy\ of\ the\ children\ node$$

Where, $entropy = \sum_i p_i \log(p_i)$ and

$$p_i = \frac{number\ of\ instances\ of\ i\ in\ the\ node}{total\ samples\ in\ the\ node}$$

This information gain is then used to decide how best to split the nodes. The tree is split till any of the stopping conditions are met:

1. Max depth – tell how deep a tree should be
2. Minimum samples split – the minimum number of samples in a node that are required in a node
3. Number of labels in a node – if there are only one labels in a node, then it does not require splitting any further

To decide the value of the leaf nodes, the most common label in each of the nodes is taken, and that value is given to those nodes.

2.2 Result

For prediction, the predict function takes this tree, and then traverses the tree to find the labels for each of the samples. As an example, we ran the algorithm on the breast cancer dataset provided by sklearn.

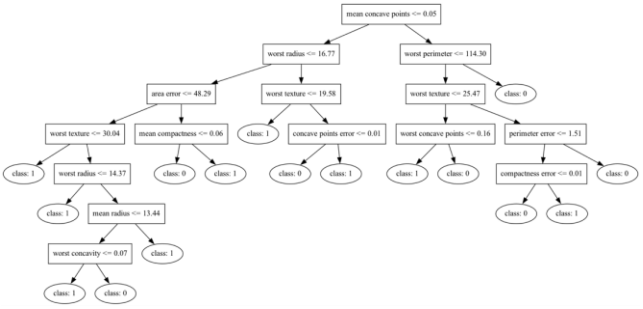

Fig.5. Tree formed after fitting the model on our algorithm

We also compared the results obtained from our classifier model to sklearn's optimized decision tree algorithm.

Algorithm	Accuracy	Time Taken
Decision Tree Classifier from scratch	94.737%	4.094s
Sklearn Decision Tree	94.737%	0.0138s

Table.3. Comparison of results

3 DECISION TREE REGRESSOR

The given code is an implementation of the decision tree algorithm for regression. The algorithm builds a binary decision tree to predict a continuous target variable. The implementation is done using Python and NumPy libraries.

3.1 Class, Methods, and Working

The main class in the code is DecisionTreeRegressor, which contains the following methods:

1. init(self, max_depth=5, min_samples_split=2):

This method initializes the DecisionTreeRegressor object with maximum depth and minimum number of samples required to split a node.

2. mse(self, y):
   This method calculates the mean squared error of the given target variable y.
3. split(self, X, y, split_feature, split_value):
   This method splits the given feature X and target variable y based on a split feature and its value.
4. best_split(self, X, y):
   This method finds the best feature and value to split the given feature X and target variable y using mean squared error.
5. fit(self, X, y):
   This method builds the decision tree model using the given feature X and target variable y.
6. _build_tree(self, X, y, depth=0):
   This method recursively builds the decision tree based on the given feature X and target variable y.
7. _predict_sample(self, tree, x):
   This method predicts the target variable value for a given sample x based on the decision tree.
8. predict(self, X):
   This method predicts the target variable value for a given set of samples X based on the decision tree.

The algorithm works as follows:

1. The fit method is called with the feature matrix X and target variable y.
2. The _build_tree method recursively builds the decision tree based on the given feature X and target variable y.
3. The _predict_sample method predicts the target variable value for a given sample x based on the decision tree.
4. The predict method predicts the target variable value for a given set of samples X based on the decision tree.
5. The algorithm finds the best feature and value to split the given feature X and target variable y using mean squared error. It continues to split the tree until the maximum depth is reached or the minimum number of samples required to split a node is not met. Finally, the algorithm predicts the target variable value for a given sample x based on the decision tree.

3.2 Result

As an example, we ran the algorithm on the diabetes dataset provided by sklearn.

We also compared the results obtained from our regression model to sklearn's optimized decision tree algorithm.

Algorithm	**R2**	**MSE**	**Time Taken**
Decision Tree Regressor from scratch	0.41	3171.87	0. 0583s
Sklearn Decision Tree	0.060	4976.79	0. 0015s

Table.4. Comparison of results