Ensemble Learning Final Group Project

OLUWAFISAYOMI BADARU[1], TUNÇ EREN[2], VALENTIN MÄURER[3], HASSAN AMRAOUI[4], AND HOUSSAM FOUKI[5]

[1] *oluwafisayomi.badaru@student-cs.fr*
[2] *tunc.eren@student-cs.fr*
[3] *valentin.maurer@student-cs.fr*
[4] *hassan.amraoui@student-cs.fr*
[5] *houssam.fouki@student-cs.fr*

Compiled March 18, 2023

Abstract — **This report presents the results of a final group ensemble learning project, which focused on two topics. The first part of the project aimed to predict Airbnb prices in New York City using various machine learning algorithms. The second part of the project focused on implementing a decision tree from scratch and testing it for both classification and regression tasks. In the first part, we preprocessed the data to extract relevant features. We then trained and evaluated several regression algorithms, including decision tree regression, random forest regression, and gradient boosting regression, to predict Airbnb prices. In the second part, we implemented a decision tree from scratch and tested it for both classification and regression tasks. We used the decision tree to classify the Iris dataset and predict house prices from the Boston Housing dataset. Our results showed that the decision tree performed well in both classification and regression tasks, achieving high accuracy and robustness. This project demonstrates the effectiveness of ensemble learning techniques including decision tree algorithms in classification and regression tasks.**

Part I

Predicting Airbnb prices in New York City

1. INTRODUCTION

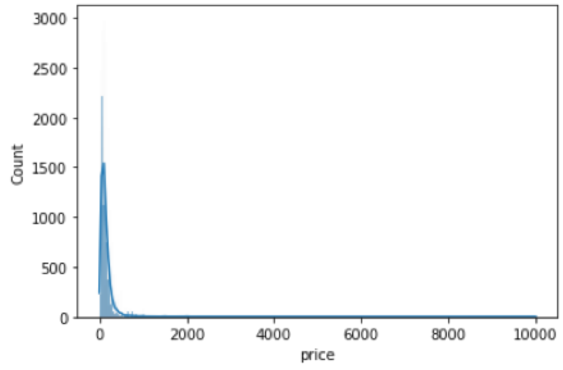
Airbnb has emerged as a popular alternative to conventional hotels, bringing about a disruption in the hospitality industry. It allows individuals to offer their own properties as rental places, resulting in approximately 40,000 listings in New York City alone. Unlike traditional hotels, hosts do not have dedicated teams to analyze demand and supply and adjust pricing accordingly. Thus, determining the optimal price for a listing can be challenging. Moreover, the varying types of listings can make it challenging for renters to get a clear idea of reasonable pricing. In this project, we will use ensemble learning techniques to forecast the price of Airbnb listings in New York City.

2. THE DATASET

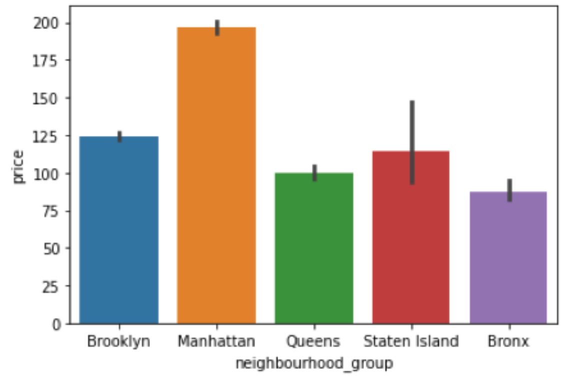
The dataset used for this project has been sourced from Kaggle and comprises information on the listings in New York City in the year 2019. It encompasses 15 features of listings, which include the name of the listing, neighborhood, price, review details, and availability. The dataset consists of approximately 47,000 listings.

3. EXPLORATORY DATA ANALYSIS (EDA)

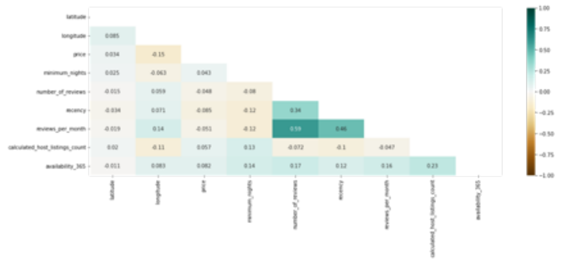
The histogram of the target variable 'price' indicates that the distribution of prices is highly skewed to the right, with the majority of listings having prices below $1000 and a few listings having prices over $10,000. This suggests that the majority of Airbnb listings in New York City are relatively affordable. However, the presence of some very expensive listings could potentially skew any analysis of the data. Alternatively, we could use a transformation such as the logarithm or square root to reduce the effect of extreme values on the analysis. It's important to carefully consider the approach chosen and its potential impact on the results of the analysis.



The barplot of the categorical variable 'neighbourhood_group' against the target variable 'price' shows that Manhattan has the highest average price of listings, while Bronx has the lowest. This could be due to the fact that Manhattan is a popular tourist destination with high demand for accommodation, while Bronx is less popular and has fewer tourist attractions. However, further analysis is necessary to explore the range of prices and the distribution of listings within each neighborhood group.



The correlation matrix provides insights into the relationships between different features in the dataset. From the heatmap, it can be seen that there are several features that are strongly correlated with each other. For example, the number of reviews and recency are positively correlated.



4. PRE-PROCESSING

A. One hot encoding

We one hot encoded the categorical variables (neighborhood group, neighborhood, room type). In order to simplify the distribution of the 'availability_365' feature, we split it into three main buckets - low availability (0-29 days), medium availability (30-300 days), and high availability (301-365 days) - and encoded them as categorical variables.

B. Multi-listing

We created a new binary feature called 'multi_listing' based on the 'calculated_host_listings_count' feature, which represents the number of listings that a host has on Airbnb.If the host has only one listing, the 'multi_listing' feature is set to 0, indicating that it is a single listing. If the host has more than one listing, the 'multi_listing' feature is set to 1, indicating that it is a multi-listing host.

C. Log of target variable

To make the target variable 'price' less susceptible to the effect of outliers, we applied a logarithmic transformation to it. This reduces the impact of extreme values and makes the distribution of prices more normal. After applying the logarithmic transformation, we removed any listings with prices below 20$orabove$3000, as these prices were considered to be outliers. Finally, we replaced the original 'price' feature with the logarithmically transformed version, which can be used in our analysis. This will improve the accuracy of our predictions by reducing the influence of outliers and making the distribution of prices more suitable for modeling.

D. Short Stay vs. Long Stay

We created a new binary feature called 'rentals' based on the 'minimum_nights' feature, which represents the minimum number of nights that a listing can be rented for. If the minimum number of nights is less than 30, the 'rentals' feature is set to 1, indicating that it is a short-term rental. If the minimum number of nights is 30 or greater, the 'rentals' feature is set to 0, indicating that it is a long-term rental. This feature can be useful in our analysis, as short-term rentals may have different pricing strategies or different characteristics compared to long-term rentals.

E. Dimensionality reduction

To reduce the dimensionality of the feature set and improve computation time, we applied standardization and principal component analysis (PCA). The resulting reduced feature set was reduced from 238 features to 204 features, which can be used for further analysis and modelling. This approach can improve computational efficiency while retaining most of the relevant information in the dataset.

5. MODELING

It should be noted that all model hyperparameters underwent optimization through GridSearchCV, with the best_params_ chosen for the final results. To evaluate model performance, we employ the R^2 and mean absolute percentage error (MAPE) metrics. A higher R^2 value indicates a superior model, while a lower MAPE (closer to zero) signifies a better model. Additionally, we have provided the metrics for both logged and anti-logged predictions.

A. Baseline model

Elastic Net: We trained an Elastic Net model with alpha=0.001 and l1_ratio=0.8 using the New York City Airbnb Open Data. The model was fit to the training data and evaluated on the test data using mean absolute percentage error (MAPE) and R-squared metrics. We observed that the y values were log-transformed before being used to fit the model and make predictions.
Decision Tree: We trained a decision tree regressor with a maximum depth of 12, minimum samples per leaf of 10, and using the New York City Airbnb Open Data. The model was fit to the training data and evaluated on the test data using mean absolute percentage error (MAPE) and R-squared metrics. We observed that the y values were log-transformed before being used to fit the model and make predictions.
Random Forest: We trained a random forest regressor with 500 trees, maximum depth of 30, minimum samples per leaf of 5, and using the New York City Airbnb Open Data. The model was fit to the training data and evaluated on the test data using mean absolute percentage error (MAPE) and R-squared metrics.

B. Boosting

We trained an XGBoost regressor with 500 trees, learning rate of 0.05, maximum depth of 6, minimum child weight of 1, and using the New York City Airbnb Open Data. The model was fit to the training data and evaluated on the test data using mean absolute percentage error (MAPE) and R-squared metrics. We observed that the y values were log-transformed before being used to fit the model and make predictions. Overall, the XGBoost regressor is expected to perform well in predicting the Airbnb prices in New York City, given its ability to handle complex relationships between features and the target variable.

We also trained an AdaBoost regressor on top of the decision tree regressor with 50 estimators, learning rate of 1, and using the New York City Airbnb Open Data. The model was fit to the training data and evaluated on the test data using mean absolute percentage error (MAPE) and R-squared metrics. Overall, the AdaBoost regressor is expected to improve the accuracy of the decision tree regressor by combining multiple weaker models to create a stronger ensemble model.

C. Bagging

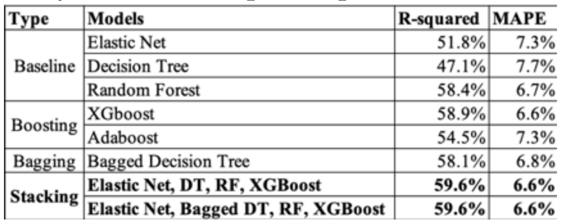
We trained a bagging regressor on top of the decision tree regressor with 40 estimators, maximum features of 0.8, and using the New York City Airbnb Open Data. The model was fit to the training data and evaluated on the test data using mean absolute percentage error (MAPE) and R-squared metrics. We observed that the y values were log-transformed before being used to fit the model and make predictions. Overall, the bagging regressor is expected to improve the accuracy and reduce the variance of the decision tree regressor by combining multiple models trained on random subsets of the data and features.

D. Stacking

We trained a stacking regressor on top of four base models (ElasticNet, Decision Tree Regressor, Random Forest Regressor, and XGBoost Regressor) using the New York City Airbnb Open Data. The model was fit to the training data and evaluated on the test data using mean absolute percentage error (MAPE) and R-squared metrics. We observed that the y values were log-transformed before being used to fit the model and make predictions. Overall, the stacking regressor is expected to improve the accuracy and reduce the variance of the base models by training a meta-model on their predictions.

6. CONCLUSION

In this project, we used ensemble learning techniques to forecast the price of Airbnb listings in New York City. We preprocessed the data by applying one hot encoding, log transformation of the target variable, and creating new features such as ’multi_listing’, ’rentals’, and performing dimensionality reduction using PCA. We then trained various models, including Elastic Net, Decision Tree Regressor, Random Forest Regressor, XGBoost Regressor, AdaBoost Regressor, Bagging Regressor, and Stacking Regressor on the preprocessed data. We evaluated the performance of each model using R-squared and mean absolute percentage error (MAPE) metrics. The results showed that the XGBoost and Stacking Regressor models outperformed the other models, which are expected to improve the accuracy and reduce the variance of the base models. Overall, this project demonstrates the effectiveness of ensemble learning techniques in predicting Airbnb prices, and it can be used by hosts to set optimal prices for their listings.



Type	Models	R-squared	MAPE
Baseline	Elastic Net	51.8%	7.3%
	Decision Tree	47.1%	7.7%
	Random Forest	58.4%	6.7%
Boosting	XGboost	58.9%	6.6%
	Adaboost	54.5%	7.3%
Bagging	Bagged Decision Tree	58.1%	6.8%
Stacking	**Elastic Net, DT, RF, XGBoost**	**59.6%**	**6.6%**
	Elastic Net, Bagged DT, RF, XGBoost	**59.6%**	**6.6%**

As can be seen from the table, the best models for predicting Airbnb prices were from stacking multiple models. We could improve this by doing more feature engineering.

Part II

Implementation of a Decision Tree from scratch

1. DECISION TREE MODEL IMPLEMENTATION

The decision tree algorithm implemented here is a simple, yet effective approach that follows the standard recursive partitioning strategy. The algorithm starts by considering the entire input dataset and recursively splits the data based on the values of the input features. The splitting criterion is chosen based on the cost function, which measures the quality of the split based on the distribution of the target variable in the resulting subsets. The cost function used in this implementation can be either the Gini index for classification problems or the mean squared error for regression problems.

PL

```python
import numpy as np

class DecisionTree:

    def __init__(self, min_samples_split=2, max_depth=5, task="classification", max_features=None, random_state=None):
        self.min_samples_split = min_samples_split
        self.max_depth = max_depth
        self.task = task
        self.max_features = max_features
        self.random_state = random_state


    class Node:
        def __init__(self, feature=None, threshold=None, left=None, right=None, value=None):
            self.feature = feature
            self.threshold = threshold
            self.left = left
            self.right = right
            self.value = value

    def _split(self, dataset, feature, threshold):
        left = dataset[dataset[:, feature] <= threshold]
        right = dataset[dataset[:, feature] > threshold]
        return left, right

    def _gini(self, labels):
        _, counts = np.unique(labels, return_counts=True)
        probs = counts / labels.size
        return 1 - np.sum(probs ** 2)

    def _mse(self, values):
        return np.mean((values - np.mean(values)) ** 2)

    def _cost(self, left_labels, right_labels):
        if self.task == "classification":
            return self._gini(left_labels) + self._gini(right_labels)
        elif self.task == "regression":
            return self._mse(left_labels) + self._mse(right_labels)

    def _choose_best_split(self, dataset, feature_indices):
        best_cost, best_feature, best_threshold = np.inf, None, None

        for feature in feature_indices:
            feature_values = dataset[:, feature]
            possible_thresholds = np.unique(feature_values)

            for threshold in possible_thresholds:
                left, right = self._split(dataset, feature, threshold)
                cost = self._cost(left[:, -1], right[:, -1])

                if cost < best_cost:
                    best_cost, best_feature, best_threshold = cost, feature, threshold

        return best_feature, best_threshold

    def _build(self, dataset, depth):
        n_samples, n_features = dataset.shape

        if n_samples >= self.min_samples_split and depth <= self.max_depth:
            feature_indices = np.random.choice(n_features - 1, self.max_features, replace=False) if self.max_features else np.arange(n_features - 1)
            feature, threshold = self._choose_best_split(dataset, feature_indices)
            if feature is not None:
                left, right = self._split(dataset, feature, threshold)
                left_child = self._build(left, depth + 1)
                right_child = self._build(right, depth + 1)
                return self.Node(feature=feature, threshold=threshold, left=left_child, right=right_child)

        value = np.mean(dataset[:, -1]) if self.task == "regression" else np.bincount(dataset[:, -1].astype(int)).argmax()
        return self.Node(value=value)

    def fit(self, X, y):
        dataset = np.column_stack((X, y))
        self.root = self._build(dataset, 1)

    def _predict(self, x, node):
        if node.value is not None:
            return node.value

        if x[node.feature] <= node.threshold:
            return self._predict(x, node.left)
        else:
            return self._predict(x, node.right)

    def predict(self, X):
        return np.array([self._predict(x, self.root) for x in X])

    def get_params(self, deep=True):
        return {
            "min_samples_split": self.min_samples_split,
            "max_depth": self.max_depth,
            "task": self.task,
            "max_features": self.max_features,
            "random_state": self.random_state,
        }

    def set_params(self, **params):
        for key, value in params.items():
            setattr(self, key, value)
        return self
```

Listing 1. The proposed Decision Tree Model

The stopping criterion for the recursive partitioning is defined by two parameters: the minimum number of samples required to split an internal node and the maximum depth of the tree. These parameters control the trade-off between bias and variance in the model. A smaller minimum number of samples will lead to a more complex decision tree, which can potentially overfit the training data. On the other hand, a larger minimum number of samples will result in a simpler tree, but may not capture all the relevant information in the data. Similarly, a larger maximum depth will result in a more complex tree, but may also overfit the data, while a smaller maximum depth will lead to a simpler tree, but may not be able to capture all the relevant features in the data. To further control the complexity of the decision tree, an additional parameter, max_features, is included in the implementation. This parameter controls the maximum number of features that can be considered at each split. This can be useful when dealing with high-dimensional data, where not all features may be relevant to the prediction task. The implementation also includes a random_state parameter, which can be used to control the randomness in the algorithm. This is important when the algorithm is run multiple times, as the results may differ due to the randomness in the algorithm. To evaluate the performance of the decision tree, the implementation includes a fit() method to train the model and a predict() method to make predictions on new data. The model can be used for both classification and regression tasks, depending on the value of the task parameter.

This code defines a Decision Tree classifier/regressor using the Gini impurity for classification and Mean Squared Error (MSE) for regression. The Decision Tree can be used for both classification and regression tasks. The class takes several hyperparameters as input, including the minimum number of samples required to split an internal node, the maximum depth of the tree, the type of task to perform (classification or regression), the maximum number of features to consider when looking for the best split, and a random state for reproducibility. The Node class is used to represent each node in the Decision Tree. Each Node object contains information about the feature, threshold, left and right child nodes, and a value (in case the node is a leaf node). The _split method splits the dataset into two subsets based on a specified feature and threshold. The _gini method computes the Gini impurity of a set of labels. The _mse method computes the mean squared error of a set of values. The _cost method computes the cost of splitting a set of labels based on the Gini impurity or MSE. The _choose_best_split method selects the feature and threshold that minimize the cost of splitting the dataset. The _build method recursively builds the Decision Tree by splitting the dataset at each node and stopping when the maximum depth is reached or the number of samples is too small. The fit method takes in the training data and labels and builds the Decision Tree. The _predict method recursively traverses the Decision Tree to predict the label of a single sample. The predict method takes in a matrix of samples and returns an array of predicted labels. The get_params and set_params methods are used to get and set hyperparameters of the Decision Tree object, respectively.

2. EVALUATION OF THE PERFORMANCE OF OUR DECISION TREE ON THE AUTO MPG DATASET

To assess our decision tree regressor's performance on the Auto MPG dataset, we computed its MSE and R^2 score. This dataset predicts the fuel efficiency of vehicles based on their features. The calculated MSE of 22.95 implies that the model has learned some patterns in the data and provides a reasonable estimate, but there is still some scope for improvement as the predicted fuel efficiency values differ from the actual values. The R^2 score of 0.5503 indicates that our decision tree model provides a reasonable approximation of fuel efficiency. This score suggests that the model can explain approximately 55.03% of the variance in the target variable. These outcomes highlight that the optimized hyperparameters, namely min_samples_split (17), max_depth (19), and max_features (1), enable the model to perform relatively well on this regression task. To sum up, our decision tree regressor is a dependable model for basic regression tasks.

3. TESTING THE DECISION TREE ON A CLASSIFICATION TASK USING THE IRIS DATASET

To evaluate the performance of our decision tree classifier on the Iris dataset, we computed its cross-validated accuracy. This classic dataset is widely used for classification tasks. The average accuracy score of 96% implies that our decision tree performed exceptionally well and could accurately predict the species of iris flowers based on their features. Moreover, the low standard deviation of 0.025 indicates that the model's predictions were consistent across different splits of the dataset. These outcomes also demonstrate that the default values we selected for min_samples_split (2), max_depth (5), and max_features (None) are appropriate for simple classification tasks. In conclusion, the results affirm that our custom-built decision tree classifier is a dependable model for classifying iris flowers in the given dataset.

4. CONCLUSION

In conclusion, decision trees are a powerful and widely used machine learning algorithm that can be used for both classification and regression tasks. The algorithm is easy to interpret and can handle both categorical and numerical data, making it a popular choice in many industries. The DecisionTree class we implemented in this project allows for customization of key hyperparameters, such as the minimum number of samples required to split a node, the maximum depth of the tree, and the type of task being performed (classification or regression). We also demonstrated how to use cross-validation to evaluate the performance of our model and how to tune hyperparameters using RandomizedSearchCV. Overall, decision trees are a versatile and effective tool in any machine learning practitioner's toolbox.