Ensemble Learning Project Report

Amandine Allmang, Nicolas Bourriez, Marie Bouvard, Jules Giraud, Arthur Nardone

March 20, 2023

Contents

GitHub Link

I AirBnB Price Prediction

In this project, we used the New York City AirBnB Price dataset to build a regression task in which the goal was to predict the price of a listing. The dataset contains 48,895 AirBnB listings. Below, we will be presenting the data pre-processing applied to the dataset, the various Ensemble Learning models implemented, as well as our recommendation on the best performing model.

I.1 Data Pre-processing

We started this project by investigating the relations between the features of our data set and the price of the listing. Some features were spuriously correlated with the price of a listing and had to be removed. Some features were also excluded because they did not bring relevant information about the listing. The excluded features are listing id, host name, date of last review, listing name, and host id. The correlation between the remaining numerical features can be seen in Fig. 1.

The `utils.py` file that can be found in the `AirBnB_Price_Prediction` folder of our GitHub page contains the pre-processing function used in this project.
This function allows us to remove the `NaN` from the dataset, replacing them with appropriate values such as `0` when the `NaN` is associated to the number of reviews.
We performed one-hot encoding on the categorical features and seperated the data set into input features (X) from the price output feature (y). Lastly, we standardized the input features using a `RobustScaler`, which is resistant to outliers, and applied it seperately on the train and test set to avoid data leakage.

Furthermore, we trained our models both on the price variable (y) with and without `MinMaxScaler` normalization. However, the performances of the models in both of these approaches were seemingly equal.

I.2 Implemented and Trained Models

The best parameters for all models were found with a `GridSearchCV`. The prediction results can be found in Table 1.

In these sections, we will be going over the bagging and boosting models that we implemented and trained on our pre-processed data. The functions used for the creation of these models can be found in the `bagging.py` and `boosting.py` file on the GitHub page.
Each function contains the option to run a `GridSearchCV` in order to find the best model parameters for the given data. All models were trained using a 5-fold cross validation and a train-test split of 80%/20%.

I.2.1 Decision Tree and Bagging models

Decision Tree

The best parameters found for this model were for the criterion is squared error, a maximum depth of 25, a minimum of two sample per leaf and for a split.

Random Forest

The best parameters found for this model were for the criterion is squared error, a maximum depth of 25, a minimum of two sample per leaf and for a split, and 100 estimators.

Extremely Randomized Forest

The best parameters found for this model were for the criterion is squared error, a maximum depth of 25, a minimum of two sample per leaf and for a split, and 100 estimators.

I.2.2 Boosting models

Gradient Boosting Regressor

One of the boosting models that we implemented and trained on our dataset was a `GradientBoostingRegressor` model. The best parameters found for this model where 50 estimators, a maximum depth of 3, a learning rate of 0.1, a minimum number of samples per leaves of 1 and lastly a minimum number of samples per split of 2. All other parameters where kept to their default values.

eXtreme Gradient Boosting Regressor

The eXtreme Gradient Boosting (XGBoost) model, being a very popular and strong implementation of a regularized gradient boosting model, we decided to try and implement it for our task as hand. The best parameters found for this model where with 100 estimators, a maximum depth of 3, a learning rate of 0.1 and a gamma value of 0. All other parameters where kept to their default values.

Adaboost Regressor

Lastly, the third boosting model that we implemented and trained was the `AdaBoostRegressor`. The best parameters found for this model where 50 estimators and a learning rate of 0.01. All other parameters where kept to their default values.

I.2.3 Model comparison and best model

Based on the performances presented in Table 1, we can observe a significant overfitting in the bagging models, especially with the Extremely Randomized Forest and the Decision Tree. In comparison, the boosting models present almost no overfitting with very close performances on the train and test set. Overall, the best performance is obtained when using the XBGoost Regressor. Indeed, this model renders a train MAE of 65.45, a test MAE of 67.01, a train RMSE of 204.97 and a test RMSE of 221.63. We consider this model to be the best performing as it renders the lowest errors and smallest difference between train and test metrics.

II Decision Tree Implementation

In this part we presented an implementation of a Decision Tree algorithm in Python. The implementation consists of a base tree class, a node class, and two child classes for classification and regression tasks. For this task we followed the CART implementation from Breiman et al.

II.1 Code structure

The implementation consists of the following four files:

- base_tree.py: Contains the BaseTree class, which serves as a foundation for building Classification and Regression Trees. It contains methods for fitting, predicting, and building the tree.
- node.py: Contains the Node class, which represents a single node in the tree. It holds information about the splitting point, left and right child nodes, whether it is a leaf node and the fitted value.
- classification_tree.py: Contains the ClassificationTree class, which inherits from the BaseTree class and implements methods specific to classification tasks.
- regression_tree.py: Contains the RegressionTree class, which inherits from the BaseTree class and implements methods specific to regression tasks.

II.2 Base Tree & Node

The BaseTree class serves as a foundation for building Classification and Regression Trees. It includes methods for fitting, predicting, and building the tree. The BaseTree class takes three arguments at initialization, namely min_sample_leaf, max_depth, and min_sample_split, which set the minimum number of samples required to be at a leaf node, the maximum depth of the tree, and the minimum number of samples required to split an internal node, respectively. These arguments are the stopping criterions of the tree.

The fit() method of the BaseTree class builds the tree from the root node and the current depth. It retrieves the different classes from the dataset and stores them into self.classes. Then, it assigns a new value to self.node with self.create_node() and builds the tree from the new root and current depth using the build_tree() method. Finally, it returns the root of the tree.

The predict() method of the BaseTree class predicts the class of the given test dataset from a pandas DataFrame and returns an array of integers or a string (class) depending on whether the tree is used for regression or classification tasks.

The build_tree() takes a Node and the current depth of the tree as input. It checks whether the current depth is less than the max depth and whether the node is a leaf node or not. If these conditions are met, it creates left and right child nodes using the create_node() method and recursively builds the tree. If the conditions are not met, it sets the node as a leaf node.

The split_dataset() method of the BaseTree class splits the dataset into left and right datasets from the value of the splitting point as input. It returns a dictionary with left and right datasets.

The create_node() method of the BaseTree class is an abstract method that needs to be implemented in specific child classes. It creates a new node from the left or right region.

The node.py file contains the Node class, which represents a single node in the tree. It includes information about the splitting point, left and right child nodes, and whether it is a leaf node. The Node class takes two arguments at initialization, namely value and is_leaf. Value is the value to split the data, and is_leaf is whether the node is a leaf node or not.

II.3 Classification Tree

The ClassificationTree class is a child class of BaseTree that implements methods specific to classification tasks. It implements the following methods:

- gini_index(): Calculates the Gini index for a given dataset.

$$G = 1 - \sum_{i=1}^{j} P(i)^2$$

- weighted_gini_index(): Calculates the weighted Gini index for a given split.

$$Gini = \frac{|R_L|}{R} \times G(R_L) + \frac{|R_R|}{R} \times G(R_R)$$

- majority_vote(): Returns the class index of the class which has the most samples in a node.

$$c(k) := argmax \hat{p}_{k,c}$$
$$with \;\; \hat{p}_{k,c} = \frac{1}{n_k} \sum_{X_i \in R_k} \mathbb{1}_{\{y_i = c\}}$$

- create_node(): Creates a new node based on the provided data.

The create_node method first checks if the provided data has enough samples to be split again. If not, it creates a leaf node with the predicted class based on the majority vote of the samples. If there are enough samples, it calculates the weighted Gini index for all possible splits and selects the one with the lowest index. We chose to implement the Gini Index over the cross-entropy index because there is not much difference in terms of performance and Gini is slightly faster to compute. The selected split becomes the new node, and the method is called recursively on the left and right child nodes.

II.4 Regression Tree

The RegressionTree class is a child class of BaseTree that implements methods specific to regression tasks. It implements the following methods:

- create_node(data): Creates a new node based on the provided data.
- mse(y): Computes the Mean Square Error between the target (y) of a given region and the predicted value($\hat{y}$).

$$MSE = \frac{1}{n} \sum (y - \hat{y})^2$$

- risk_regression(child): Computes the risk function for the regression from the two regions.

Same as the ClassificationTree, the RegressionTree creates a new node, by checking the stopping criterions and splitting data in two regions computing the risk function for all possible splits and selects the one with the lowest risk value. Risk function used for RegressionTree is the mse() method which computes the Mean Square Error between the target of a given region and the predicted value.

II.5 Conclusion

In conclusion, we have presented an implementation of a Decision Tree algorithm in Python for both classification and regression tasks. The implementation allows for customization of parameters such as the minimum number of samples required to be at a leaf node, the maximum depth of the tree, and the minimum number of samples required to split an internal node. The implementation also includes methods for fitting, predicting, and building the tree. As it is described in the course, CART is a greedy algorithm and can takes some time to be fitted on the data. Thus, with our implementation, we achieved the exact same splitting points and value as those returned by a Decision Tree from scikit-learn, but.

III Annex

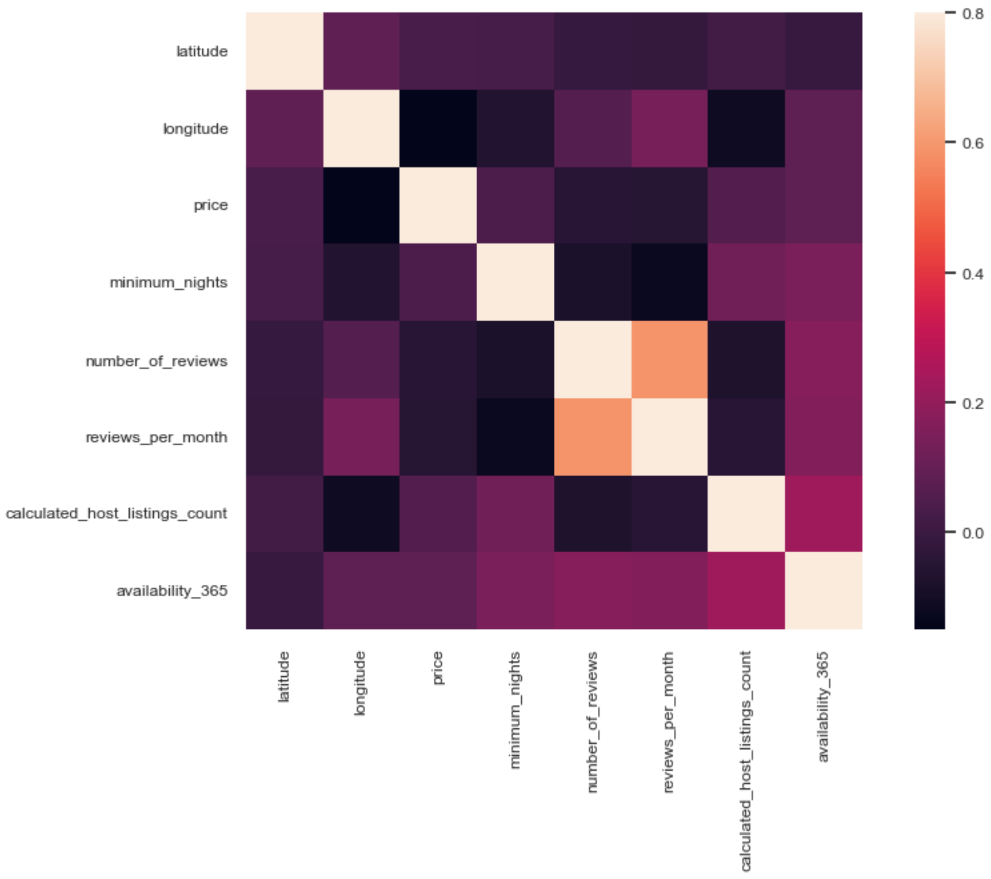

Figure 1: Correlation of numerical features in the New-York AirBnB dataset.

Model	Set	Mean Absolute Error (MAE)	Root Mean Squared Error (RMSE)
Decision Tree	Train	0.01 ± 0.0006	0.11 ± 0.09
	Test	87.09 ± 2.60	318.04 ± 17.94
Random Forest	Train	24.85 ± 0.33	84.73 ± 2.90
	Test	67.11 ± 1.16	228.36 ± 22.21
Extremely Random Forest	Train	0.002 ± 0.0004	0.15 ± 0.05
	Test	68.09 ± 1.25	233.03 ± 24.77
Gradient Boosting	Train	66.45 ± 0.68	209.88 ± 3.70
	Test	67.59 ± 1.42	223.16 ± 23.35
Adaboost Regressor	Train	74.66 ± 0.90	222.23 ± 4.85
	Test	75.09 ± 1.72	228.23 ± 25.29
XGBoost Regressor	Train	65.45 ± 0.57	204.97 ± 4.07
	Test	67.01 ± 1.48	221.63 ± 23.29

Table 1: Performance on training and test sets for all models