Ensemble Learning Project 2023

A. Helburg, P. Michel, M. Pellet, E. Riguidel, E. Serfaty

SECTION I

Predicting Airbnb Prices in New York City using Ensemble Learning Based Regression Models

Airbnb has become a popular alternative to traditional hotels and has disrupted the hospitality industry. Through Airbnb, individuals can list their own properties as rental places. In New York City alone, there are around 49,000 listings. While traditional hotels have teams that carefully measure demand and supply to adjust pricing, as a host, it can be challenging to determine the optimal price for a listing. The variation in types of listings can also make it difficult for renters to get an accurate sense of fair pricing. In this paper, we will present our project that aims to predict the price of Airbnb listings in New York City in the most accurate and efficient way possible, using ensemble learning methods. To solve this regression problem, we will use different approaches taught in the course and practiced in lab sessions, namely *Decision Trees*, *Random forests*, *AdaBoost* and *XGBoost.* We will also be discussing Ensemble Stacking.

1 The dataset

Our dataset is obtained from Kaggle and contains the Airbnb listings in New-York City in 2019. It includes 15 features per listing, among which the name of the listing, the neighborhood, the price, the review information, and the availability. The dataset contains 48,895 listings and can be downloaded from Kaggle here.

2 The Pre-Processing

To apply the ensemble methods, we started by preprocessing the data in 2 steps: transforming the variables to obtain data in a usable format, and removing outliers.

We started by removing unuseful variables for the prediction. To do so, we used the function *feature importances*, and we found out that the following features did not provide valuable information to predict prices: *name*, *host_name*, *last_review*, *host_id*, and *id.* The second step was to transform categorical variables into dummy variables for *room_type,* which contained 2 different values, and *neighborhood_group* which contained 5 different values. The task was harder for the column *neighborhood*, as it contained 215 unique values. We decided to encode it using a hashing technique. A first attempt was to convert the 215 distinct *neighborhood* values into 10 columns of hashed values representing the neighborhoods. Eventually, after several tests we noticed that even if the neighborhood information was valuable, using an increasing number of columns did not increase the accuracy of our models. We therefore converted the 215 distinct *neighborhood* elements into 3 columns of hashed values. This allows to represent the neighborhood information in a way that is easily usable by the models, and to keep the model as simple as possible.

The second and probably most important step was to remove the outliers from our dataset. Removing them ensures that the model is trained on a representative sample of data and that it is not overly influenced by unusual data points. To do so, we used the target column *price*. First, we removed the listings with a price below $10. We noticed from the beginning that prices presented a long-tail distribution, which seems quite logical as prices can increase fast if the airbnb is luxurious. However, as the dataset did not provide information about the flat/house surface, our first attempt presented very bad performance: for instance, the RMSE of random forest was 177. To lower it, we choose to remove outliers which influenced the model very badly. We started with basic methods such as z-score or interquartile range, but it gave us a huge amount of outliers, around 10% of data, as it assumed that the distribution was normal. Finally, we decided to perform an *Isolation Forest* on the price column, removing only the most obvious anomalies, meaning 3% of the dataset. If we had had the surface information, this step would have been more precise.

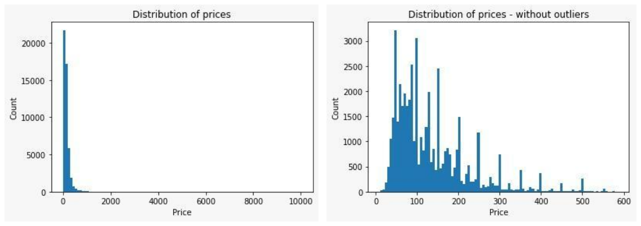

Figure 1. Distribution of prices with and without outliers

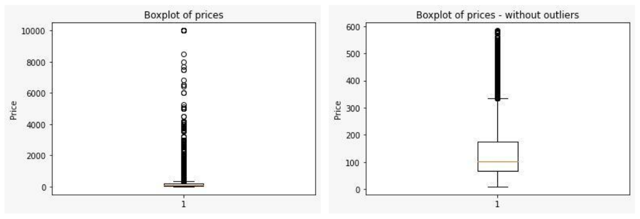

Figure 2. Boxplots distribution of prices with and without outliers

Price characteristics	Dataset with outliers	Outliers	Final dataset
Count	48,895	1,448	47,430
Mean	152	944	128
Std	240	1,040	82
Min	0	0	11
1st quartile	69	500	68
Median	106	650	100
3rd quartile	175	900	170
Max	10,000	10,000	452

Figure 3. Descriptive statistics for column *Price*

We can see on *Figure 3* that only by removing 1,448 values, the std is divided by 3, and the median is twice closer than the mean value of the column. *Figure 1 and 2* shows us that even by removing the most aberrant values, we kept the long tail distribution, typical for prices data. Finally, after preprocessing the data, we obtain a dataset of a little more than 47,430 data points and only the most relevant feature to predict prices.

3 Methodology

To compare the performance of each model, we will use various metrics learned in class. To ensure a fair comparison between our models, we chose to use the Root-Mean-Square Error (RMSE) as our main evaluation metric. It is a common measure of performance for regression models, with a lower value indicating a better fit to the data. Unlike the Mean Squared Error (MSE), the RMSE is expressed in the same units as our target variable. which in our case is the price of a listing (in \$). This makes the interpretation of the RMSE more intuitive and easier to understand, as it is directly in dollars. Therefore, a smaller RMSE value indicates a better predictive performance of our models for the Airbnb prices in New York City.

$$RMSE = \sqrt{\sum_{i=1}^{n} \frac{(\hat{y}_i - y_i)^2}{n}}$$

In addition to the RMSE, we will also use the coefficient of determination (R-squared), which measures the proportion of variance in the target variable that is explained by the model.

$$R^2 = 1 - \frac{SS_{RES}}{SS_{TOT}} = 1 - \frac{\sum_i (y_i - \hat{y}_i)^2}{\sum_i (y_i - \overline{y})^2}$$

It is a measure of how well the model fits the data, with a value between 0 and 1, where a higher value indicates a better fit.

.4 Models

In this project, we explored the predictive power of four ensemble learning techniques on the Airbnb dataset. Specifically, we started by implementing the *Decision Tree* and *Random Forest* models, using bagging. We then focused on Boosting, which involves sequentially adding models to the ensemble, with each model learning from the mistakes of the previous model. It is a popular method for ensemble learning in regression problems because it can handle complex relationships between the features and the target variable, and can reduce the variance of the model. We used *AdaBoost* and *XGBoost*, two popular boosting algorithms, to improve the performance of our regression models. *AdaBoost*, short for Adaptive Boosting, is a boosting algorithm that focuses on improving the accuracy of the model by adjusting the weights of misclassified samples in the training set. This process is repeated until the model can accurately classify all samples in the training set. In our case, we used AdaBoost with decision trees as the base estimator. *XGBoost*, short for Extreme Gradient Boosting, is another boosting algorithm that is widely used in regression problems. It is an extension of the Gradient Boosting algorithm and can handle large datasets and missing values. *XGBoost* also has built-in regularization to prevent overfitting, which is a common problem in boosting methods.

It is worth noting that finding the perfect parameters that prevent overfitting is a trade-off, as the objective is to find the balance between underfitting and overfitting. That is why we used cross-validation to evaluate our models on a separate validation set and make sure they were not overfitting, as well as techniques such as grid search or random search to find the optimal parameters for the model. With these methods, we can ensure that the model is generalizing well to new unseen data and minimize the risk of overfitting.

In addition to the individual models described above, we also tested an ensemble approach called Stacking. It involves training multiple models and then combining their predictions to obtain a final output. In our implementation, we combined the predictions of the four models using a simple linear regression model. We trained the linear regression model on the predictions of the four models on the training set, and used it to predict the prices on the test set. While this approach did not result in significant improvements in accuracy over the best individual model, it can be useful in situations where multiple models with complementary strengths are available.

5 Evaluation

We will now examine the performance of each of the four algorithms. First, before comparing the different scores, let us summarize the optimal values of the hyperparameters for each model, i.e. the values that led to the lowest RMSE. *Figure 4* presents a summary of the hyperparameters that led to the best predictions. Note that only those parameters whose optimal value is different from the default value are mentioned, although many others were tested by our cross-validation function.

Model	Best Hyperparameters

Decision Trees	max_depth = 8
	min_samples_leaf = 14
	min_samples_split = 2

Random forests	bootstrap: True
	max_depth = 50
	max_features = sqrt
	min_samples_leaf = 5
	min_samples_split = 4

AdaBoost	base estimator: . max_depth = 5 . min_samples_leaf = 12 . min_samples_split = 4 learning_rate = 0.01 loss = 'linear' n_estimators = 50

XGBoost	colsample_bytree = 0.55
	learning_rate = 0.045
	max_bin = 300
	max_depth = 9
	min_child_weight = 1
	n_estimators = 140
	reg_alpha = 4
	reg_lambda = 1
	scale_pos_weight = 0
	subsample = 1

Figure 4. Optimal Hyperparameters for each model

Let us now compare the performance of each model, summarized in *Figure 5*.

Model	Best Score (lowest RMSE)	R2
Decision Trees	58.57	49%
Random forests	55.4	55%
AdaBoost	59.8	47%
XGBoost	55.1	55%
Stacking	**54.9**	**56%**

Figure 5. Best Performance for each model

We can see that the best performing algorithm for our problem is obtained through stacking the models, with the lowest RMSE at 54.9. The corresponding R-squared was about 56%. Overall, the ensemble methods did not perform amazingly on the dataset.

6 Conclusion

In this project, our goal was to predict the price of a listing in the Airbnb dataset using four different algorithms: *Decision Trees*, *Random forests*, *AdaBoost* and *XGBoost*. Our results show that stacking the models is the best performing algorithm to tackle this class of problems, as it showed the lowest RMSE. The other methods *XGBoost* and *Random Forest* performed slightly less well, but they require less computational time which can be interesting in some contexts. Overall, the accuracy is not very high, as we choose to remove only the most extreme values. This limited accuracy can also be explained by the fact that we had no data concerning the surface of the flat/house, which is obviously significant information to predict the price of an airbnb.

SECTION II

Implementing a Decision Tree from scratch suitable for both regression and classification tasks

In this section, we will provide a detailed explanation of the steps involved in building decision trees for classification and regression tasks. Decision trees are widely used in machine learning as a simple and interpretable model for making predictions based on a set of features. A decision tree consists of a root node, internal nodes, and leaf nodes. The root node represents the entire dataset, while internal nodes represent a feature and a corresponding decision rule, and leaf nodes represent a predicted outcome.

1 Introduction

The goal of a decision tree is to split the dataset into homogeneous groups, where each group corresponds to a unique class label. The purity of a group is measured by a criterion adapted to a specific task. Decision trees can solve two types of problems: regression task & classification tasks. We started by developing two distinct models at the beginning of the decision tree implementation process: one for classification and the other for regression. Subsequently, we discovered that both architectures shared a significant core. We choose to merge these two jobs into a single model for this reason.

The criterion and metrics, however, are significant distinctions between these two algorithms despite their substantial similarities.

2 Architecture

To build a decision tree, we start at the root node and choose the feature that maximally reduces the criterion function. The best split is the one that results in the lowest weighted sum of the criterion function of the child nodes, where the weight is proportional to the size of each child node. We then repeat the process recursively for each child node until a stopping criterion is met, such as a maximum depth or a minimum number of samples per leaf node.

The *DecisionTree* class makes up the majority of the model that we have created. The *fit()* function in this class permits task identification, checks whether splitting should be stopped, loops over all features to determine the best split, splits the data, and builds subtrees recursively. The *fit* method calls on several other functions that differ depending on the task at hand.

Finally, when we call the method to construct the decision tree, we need to precise the task, and other parameters such as *max_depth*, *min_samples_leaf*, *min_samples_split* that take default parameters that we defined otherwise.

To test the performance of the decision tree, we can use a validation set or cross-validation to estimate the generalization error. The generalization error is the difference between the error on the training set and the error on the test set. Depending on the task, the error can be measured by different metrics.

a) Classification task:

For a classification task, we have chosen to use Gini Impurity as our splitting criterion corresponding to *_gini()* method in our architecture:

$$G = \sum_{i=1}^{C} p(i) * (1 - p(i))$$

where C is the number of classes and *p(i)* is the proportion of samples in the i^{th} group that belongs to a certain class.

Moreover, the new leaf's identity will be decided by majority vote referring to the *most_common_label* method. There are also numerous metrics to measure the error in relation to the performance measure for a classification problem: *accuracy, Recall, precision, or F1-score.*

b) Regression task

For a regression task, The quality of a split is measured by the mean squared error (MSE) corresponding to *_mse()* method in our architecture:

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

where n is the number of samples, y is the true value of the target variable and $\hat{y}$ is the predicted value of the target variable.

In this case, the new leaf's identity will be decided by the average values of the samples belonging to it. Concerning the metrics to measure the error, we have the *MSE* and/or *R2*.

3 Testing the Decision Tree on simple Datasets

To illustrate the implementation of decision trees on a regression task, we used the fetch_california_housing dataset which is a well-liked dataset for regression analysis and machine learning models. It includes details on housing costs and other factors for several Californian districts in the USA. The dataset has more than 20,000 instances but we kept only 200 of them to decrease computation time. Each instance represents a district and 8 attributes, including the median income, the typical number of people living in a home, and the distance from the ocean, among others. The median house price for each district serves as the target variable.

Figure 6 compares the results obtained on a regression task with our decision tree and the python decision tree:

Model	MSE	RMSE	R2
Our Decision Tree	0.338	0.581	69.8 %
Python Decision Tree	0.259	0.509	74.5 %

Figure 6. Performances of our Decision Tree versus python's one

Then, to illustrate the implementation of decision trees on a classification task, we chose to use the load_wine dataset which is frequently used for classification problems in machine learning. It provides information about several Italian wine types. The dataset consists of 178 instances, each of which represents a separate sample of wine, and 13 features, including, among others, the amount of alcohol, the concentration of malic acid, and the amount of ash. The class of wine, which stands for one of three kinds, is the target variable.

Figure 7 compares the results obtained on a classification task with our decision tree and the python decision tree:

Model	Accuracy	Precision	Recall	F1_score
Our Decision Tree	94.4 %	96.2 %	94.4 %	94.7 %
Python Decision Tree	88.8 %	89.0 %	88.8 %	88.5%

Figure 7. Performances of our Decision Tree versus python's one

4 Evaluation and conclusion

Overall, we can observe that for each individual task, the outcomes produced by our own model and those produced by the decision tree of the Python sklearn.tree module are quite comparable. We can also see that the performance for classification task outperforms the one of regression task. But since the results are similar to Python implementation, we can presume that this is due to the dataset used. We can also assume that the difference of performance between our model and Python implementation are due to the randomness of decision tree building. Overall, we can conclude that our model works as expected and performs as well as Python implementation.