ENSEMBLE LEARNING PROJECT

Vanille BOURRE Victor MAILLOT Mélodie MIRVAL
Adel REMADI Amine ZAAMOUN

1. PREDICTING AIRBNB PRICES IN NEW YORK CITY

1.1 Introduction

Airbnb has become a popular alternative to traditional hotels, allowing individuals to list their properties as rental places. However, determining the optimal price for an Airbnb listing can be challenging for hosts, especially in large cities like New York, where the number of listings is substantial. To help hosts set competitive prices and improve their occupancy rates, accurate prediction of Airbnb prices is crucial.

In this project, we aimed to predict the price of Airbnb listings in New York City using Ensemble Learning techniques on a Kaggle dataset. Our goal was to train and tune the hyperparameters of 14 methods (such as Decision trees, Random Forest or XGBoost, etc.) and combine their predictions using Stacking and Voting algorithms, two popular Ensemble techniques, that we developed ourselves and which performed better in comparison to Scikit-Learn's implementation. We have evaluated the performance of our Ensemble models using several metrics such as MAE (Mean Absolute Error), RMSE (Root Mean Squared Error), or R^2. As part of the analysis, we also developed decision tree algorithms that were compared with Scikit-Learn's implementation on the task of predicting prices of Airbnb listings as well as on 4 other datasets.

The results of this project confirmed that Stacking, Voting and Boosting are very interesting ensemble techniques that, if associated with proper feature engineering, could allow to provide valuable insights to Airbnb hosts in New York City for better decision making.

1.2 The Dataset

The "New York City Airbnb Open Data", which can be accessed through Kaggle, is a dataset comprising information on Airbnb listings within New York City. Additionally, the dataset includes related data such as host details and reviews. In 2019, the data contained in the dataset was extracted from the publicly available Airbnb listings in New York City. This dataset serves as a valuable resource for analyzing Airbnb trends and patterns in the city, and can also be used to develop predictive models pertaining to pricing, availability, and occupancy.

The dataset contains a total of 48,895 records and is comprised of 16 columns of data. Each listing is assigned a unique identifier, with the name of the listing and the host name provided. The host's unique identifier is also included in the dataset. The borough and neighborhood where the listing is situated are identified by the "neighbourhood_group" and "neighbourhood" columns respectively. Additionally, the dataset includes the latitude and longitude of each listing. The type of room being offered is listed under the "room_type" column, with the price per night specified in the price column. The minimum number of nights a guest must stay is denoted by the "minimum_nights" column. The number of reviews the listing has received and the date of the last review are specified in the "number_of_reviews" and "last_review" columns respectively, with the average number of reviews per month provided under "reviews_per_month". The "calculated_host_listings_count" column specifies the number of listings the host has, while the "availability_365" column indicates the number of days per year the listing is available for booking.

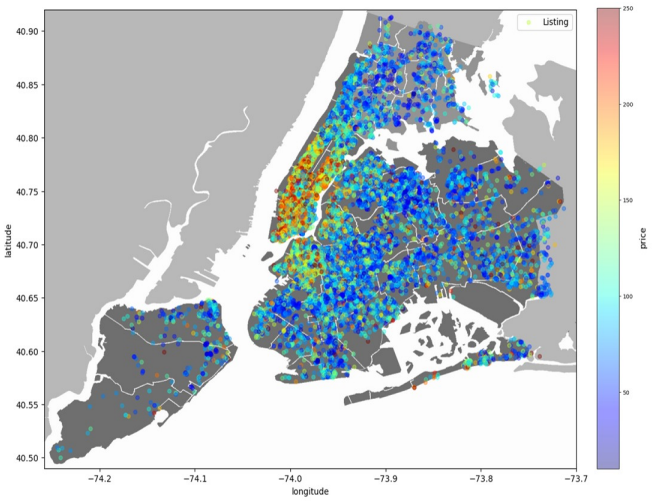

Figure 1. Map of the listings with their range of price in New-York

1.3 Data Preprocessing and Feature Engineering

The first step was to conduct an exploratory data analysis of the dataset to have a better understanding of the nature of each field, the distribution of the target variable 'Price', and the correlations across features. This was followed by a feature engineering step which consisted in embedding the textual data, dealing with categorical features of high cardinality and identifying and integrating external features that would be relevant to predict Airbnb prices. Lastly, more than 600 lines of helper functions were de-

veloped to perform the pre-processing in an efficient and scalable way in order to be in the best conditions to train and tune 14 Ensemble models.

The target variable 'Price' is heavily skewed towards lower prices and has a long right tail. Exceptionally expensive hostings correspond to out-of-the norm sites that are rare. Considering that fact along with the fact that very few features seemed to have a significant correlation with the target variable (see heatmap example in Appendix), outliers were trimmed out of the considered data for training. In that aim, a double log transformation was performed on the Price field to get its distribution closer to a normal, as illustrated by the histogram and QQ-plot in Figure 3 and 4.

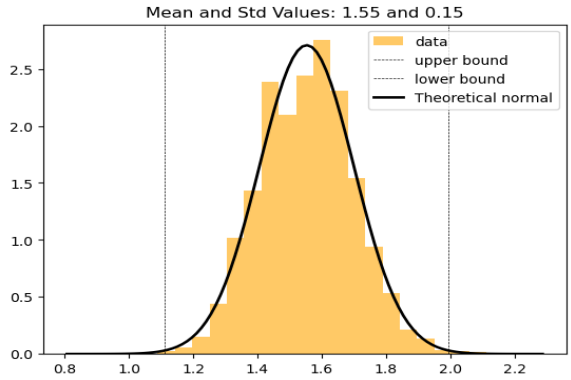

Figure 2. Distribution of log(log(Price))

As can be noticed on both plots, the double log of price seems to match well with a theoretical normal distribution, at least outside of the extreme values. We defined the upper and lower bounds (displayed in dashed lines on both Figures 3 and 4) as the points located 3 standard deviations away from the mean. In the case of a Normal Distribution, events are extremely unlikely to occur outside these boundaries (close to zero probability). Using this approach, about 1000 outliers were trimmed out of the considered dataset.

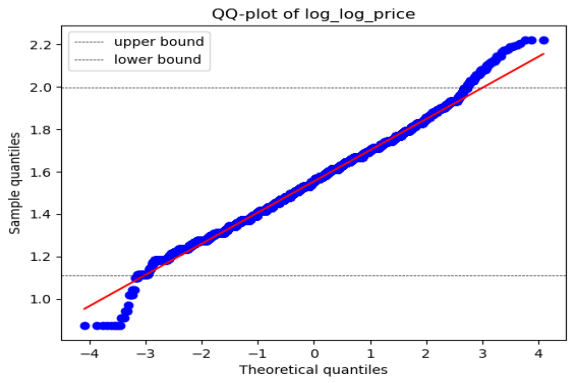

Figure 3. QQ-plot of log(log(Price))

In order to increase the performance of our models, we integrated a set of external features through an open data Airbnb dataset, available at this link. This dataset provides a wide range of additional information about 2000 of the listings. These features include the number of beds, rooms, and bathrooms, as well as the square meters and number of visitors each listing accommodates. The host response rate is also provided, which can indicate the responsiveness and overall quality of the host. Additionally, the type of bed is included, which can be a crucial factor for some travellers. The integration of these external features provided valuable information for our models to better predict and understand the characteristics of the listings.

In addition, many textual descriptions provided in the external dataset offered valuable insights about the unique characteristics of each listing. We therefore used a seq2vec encoding approach on these fields to run some clustering and dimensionality reduction analysis on the encodings. As can be noticed in Figure 4, the clusters obtained with K-means (colors on the plot) did not seem to significantly distinct when projected using T-SNE. A PCA projection led to the same result. However, we saved both PCA and T-SNE embeddings of all the encoded texts and it eventually turned out that using the T-SNE embeddings had a positive impact on the performance of our models.

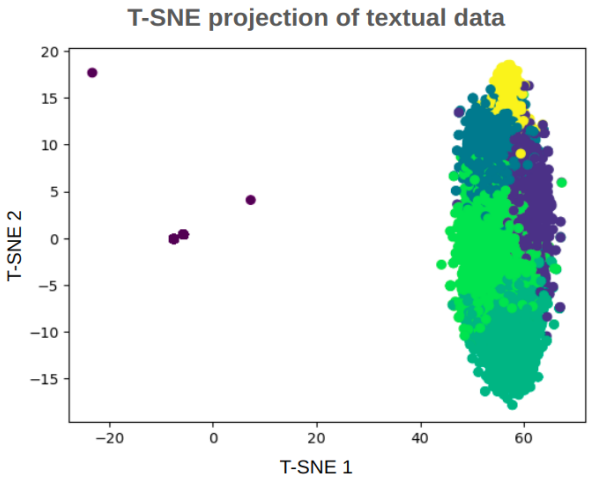

Figure 4. Clusters obtained via K-means on encoded texts

As a last significant feature engineering approached, we implemented a target encoding function to address several categorical variables with high cardinality. In order to reduce the impact of under represented categories and prevent overfitting, we implemented a sigmoid blending factor as described by Micci-Barreca (2001). The function also accounts for hierarchical relationships between categorical variables.

Overall, 600 lines of code were developed to allow us to scale the training of tuning of 14 models. The main helper function was called 'preprocess'. It is a versatile tool for preparing data and splitting train and test sets. Given the file path and optional arguments, it reads the relevant CSV files from the specified paths and performs a series of preprocessing steps on the data. It loads external features, removes rows with no "price" nor "availability", handles missing values, performs one-hot encoding etc. The 'preprocess' function relies on a series of other functions that

would for example, trim out the outliers, remove stopwords, reformat text, encode text into vectors, compute PCA embeddings, fetch T-SNE embeddings, compute target encoding with blending factors, and convert latitude and longitude coordinates to cartesian. The output of this function is a feature matrix and target values for both the train and test sets, making it easy to use for training our models.

Last but not least, we assisted the final features selection process with the use of additional analyses, such as a performance plotting function or a heatmap of correlations to between features and the target variable. *See figure in appendix.*

1.4 Models

For predicting the Airbnb prices in New York using Ensemble Learning, we trained and tuned a total of 14 algorithms: Decision Tree (sklearn), Decision Tree (Coded from scratch), Bagging, Random Forest, Extremely Randomized Trees, Gradient Boosting, AdaBoost, Histogram Gradient Boosting, XGBoost, LightGBM, CatBoost, Voting (sklearn), Voting (From scratch), Stacking (sklearn), Stacking (From scratch). Several of the best models were indeed combined using advanced ensemble techniques such as Stacking and Voting. In our project, we used all the previously listed Ensemble Learning models and developed our own algorithms for Voting and Stacking. We compared our algorithms with the implementations provided by Scikit-Learn and found that our algorithms performed equally or better and were faster, particularly for Stacking.

1.4.1 Tuning

For each model, we optimized the hyperparameters in order to achieve the best possible performance on the dataset, while avoiding overfitting or underfitting. Since the hyperparameters are set before any training and cannot be learned from the data, e.g. the number of trees in a random forest for example, tuning these hyperparameters can have a significant impact on the performance of the models. We have optimized the hyperparameters of all the models with the hyperopt package, which relies on Bayesian Optimization. It searches for the best set of hyperparameters by minimizing the loss function. The algorithm creates a probability distribution over the hyperparameters and then samples from that distribution to generate new sets of hyperparameters to try. It evaluates each set of hyperparameters using cross-validation and updates the probability distribution based on the results. It can also handle multiple hyperparameters at once, making it a powerful tool for tuning models.

1.4.2 Evaluation method of the results

We trained and fine-tuned each model and evaluated its performance before and after tuning. We calculated evaluation metrics for both 'Price' and 'log(Price)' but we focused on 'Price' as it is the target variable of interest ('log(price)' has no meaningful interpretation and can wrongly boost performance metrics significantly,e.g. 0.67 in R^2). In order to visualize the models' performance, we defined a dedicated helper function that computes several regression evaluation metrics such as MAE, RMSE, MSE, R^2, and MAPE. Additionally, it can plot the predicted vs actual and residuals vs predicted graphs for visualization purposes. The models were compared and ranked based on these evaluation metrics between the target variable 'Price' and the generated predictions on a test set.

1.4.3 Decision Tree (Scikit-Learn and from scratch)

Decision Trees are tree-like models where each node represents a feature or attribute, and each branch represents a possible value or outcome of that feature. The leaves of the tree represent the predicted outcome or class. The Decision Tree algorithm works by recursively splitting the data based on the most significant gain with respect to a given loss function (most generally MSE) until it reaches a point where the data is perfectly classified or a stopping criterion is met. We tuned many of the model's hyperparameters, such as the max depth of the tree, the minimum number of sample per leaf, etc. and obtained the following results. As can be seen in the below table, the decision tree algorithm that was coded from scratch (See details in Section 2.) performed better than Scikit-Learn's implementation. However, it was computationally more expensive.

	Before tuning (sklearn)	**After tuning(sklearn)**	**From scratch**
R^2	0.1083	0.4781	0.4901
MAE	56.227	45.826	44.814
MSE	7977.60	4669.26	4562.14
RMSE	89.317	68.332	67.544
MAPE	0.4252	0.3712	0.3517

1.4.4 Bagging

Bagging (Bootstrap Aggregating) involves training multiple instances of the same model on different subsets of the training data and then combining their predictions to reduce the variance of the overall prediction. A set of decision trees is typically trained on random subsets of the training data, with replacement. Each tree is trained independently, and the final prediction is the average of the predictions of all the trees. Bagging can improve the performance of a model by reducing overfitting and increasing stability. It is particularly useful when the dataset has high variance or when the model is sensitive to small changes in the training data. We tuned many of the model's hyperparameters, such as the number of estimators and obtained the following results.

	Before tuning	**After tuning**
R^2	0.5114	0.5545
MAE	43.263	41.776
MSE	4370.95	3985.59
RMSE	66.113	63.131
MAPE	0.3403	0.3321

1.4.5 Random Forest

Random Forest is an algorithm relying on Boostrap Aggregating that combines multiple decision trees to create a more accurate and robust model. Each decision tree is trained on a different random subset of the training data sampled with replacement. What distinguishes random forests from Bagging is that before each split, only a random subset of features are considered. This aims at reducing the correlation between trees, while keeping the variance low. The final prediction is then made by aggregating the predictions of all the individual trees in the forest. We tuned many of the model's hyperparameters, such as the number of trees, the number of features to consider at each split, etc. and obtained the following results.

	Before tuning	**After tuning**
R^2	0.5440	0.5627
MAE	42.044	41.077
MSE	4076.87	3911.80
RMSE	63.874	62.544
MAPE	0.3320	0.3221

1.4.6 Extremly Randomized Trees

Extremely Randomized Trees are similar to Random Forest, but with two main differences: the decision trees are constructed with randomly selected features at each split, and the splitting thresholds are selected randomly instead of being determined based on the features' values. ExtraTrees regression is a bagging-based technique, which involves training multiple decision trees on different subsets of the training data and aggregating their predictions to make a final prediction. The randomness in ExtraTrees helps to reduce overfitting and improve the model's generalization ability. The method can be applied to both small and large datasets, and it can handle a mix of categorical and numerical data. We tuned many of the model's hyperparameters, such as the number of trees, the minimum number of samples per leaf, etc. and obtained the following results.

	Before tuning	**After tuning**
R^2	0.5490	0.5617
MAE	41.436	41.070
MSE	4034.77	3920.70
RMSE	63.520	62.615
MAPE	0.3295	0.3252

1.4.7 Gradient Boosting from Scikit-Learn

Gradient Boosting uses a series of decision trees to make predictions. Trees are built sequentially, with each tree trying to correct the errors of the previous tree. At each iteration, the algorithm fits a decision tree to the negative gradient of the loss function with respect to the predicted values, which are updated after each tree. The GradientBoostingRegressor class in Scikit-learn offers a range of hyperparameters to control the number of trees, the learning rate, the maximum depth of each tree, and the subsampling rate of the data and features. These hyperparameters were tuned and led us to the following results:

	Before tuning	**After tuning**
R^2	0.5189	0.5360
MAE	43.625	42.522
MSE	4304.32	4151.14
RMSE	65.607	64.429
MAPE	0.3451	0.3376

1.4.8 Histogram Gradient Boosting from Scikit-Learn

Histogram Gradient Boosting is a variant of Gradient Boosting that uses histogram-based algorithms to speed up the training process. It builds decision trees using histograms to bucket the training data into discrete bins, rather than sorting the data and splitting it into continuous intervals as in traditional Gradient Boosting. This allows for faster computations since histograms can be computed in linear time, whereas sorting can take much longer. It also uses other optimizations such as subsampling of the data and features to further speed up training. It can handle missing values and categorical features by treating them as separate bins in the histograms. As for the case of Gradient Boosting, a whole range of hyperparameters to control the number of trees, the learning rate, the maximum depth of each tree, the subsampling rate, etc. were tuned and led us to the following results:

	Before tuning	**After tuning**
R^2	0.5532	0.5590
MAE	41.810	41.268
MSE	3997.61	3945.12
RMSE	63.227	62.810
MAPE	0.3274	0.3194

1.4.9 AdaBoost

In AdaBoost (Adaptative Boosting) for regression, the weak classifiers are decision trees, and the algorithm works by sequentially adding decision trees that focus on the areas of the data where the previous trees performed poorly. At each iteration, the samples are weighted based on their errors, and the decision tree is trained to minimize the exponential loss. The final prediction is then made by aggregating the predictions of all the individual decision trees. It's a simple yet powerful algorithm that can achieve high accuracy with a small number of weak classifiers. However, it can be sensitive to noisy data and outliers, and the performance may degrade if the weak classifiers are too complex and overfit the data. It was actually the case with the Airbnb dataset, on which even a tuned Adaboost performed poorly (even compared to Decision Trees).

	Before tuning	**After tuning**
R^2	0.3295	0.4260
MAE	60.201	48.832
MSE	5998.33	5134.83
RMSE	77.449	71.658
MAPE	0.5783	0.4034

1.4.10 XGBoost

XGBoost (Extreme Gradient Boosting) is an optimized implementation of Gradient Boosting that is specifically

designed for efficiency and speed. XGBoost for regression works by sequentially adding decision trees to correct the errors of the previous trees. However, unlike traditional Gradient Boosting, XGBoost uses a more regularized approach to prevent overfitting and improve generalization. The algorithm applies L1 and L2 regularization to the weights of the decision trees, and it also uses a shrinkage technique to reduce the impact of each tree and prevent overfitting. It is particularly effective on large datasets with many features. The tuned XGBOOST actually offered the third best performance, just after the Voting and Stacking algorithms that we coded from scratch.

	Before tuning	**After tuning**
R^2	0.5434	0.5699
MAE	42.268	40.757
MSE	4085.24	3847.96
RMSE	63.916	62.032
MAPE	0.3310	0.3154

1.4.11 LightGBM

LightGBM for regression works similarly to other gradient boosting algorithms by adding decision trees sequentially to correct the errors of the previous trees. However, LightGBM uses a unique technique called "gradient-based one-side sampling" that enables it to handle large datasets with high-dimensional features. The algorithm samples only a portion of the data and uses only the gradient information of the subset for splitting, instead of scanning the entire data. This technique reduces the computation time significantly while maintaining the accuracy. LightGBM also includes other optimizations, such as leaf-wise tree growth, which grows the tree leaf by leaf instead of level by level, and histogram-based feature bundling, which groups features with similar values into the same bin to reduce the number of split candidates. In addition to being a fast algorithm, the tuned LightGBM also displayed one of the fourth best performance, just below XGBOOST's.

	Before tuning	**After tuning**
R^2	0.5548	0.5675
MAE	41.509	40.894
MSE	3983.03	3869.23
RMSE	63.111	62.203
MAPE	0.3244	0.3160

1.4.12 CatBoost

CatBoost is a gradient boosting algorithm that is designed to handle categorical features efficiently, making it particularly useful for tasks involving datasets with a mix of categorical and numerical features. It works similarly to other gradient boosting algorithms but it uses a technique called "ordered boosting" that considers the order of categorical variables during training, which can lead to better accuracy on datasets with many categorical features. CatBoost also includes other optimizations, such as symmetric trees, which reduce the overfitting of the decision trees, and gradient-based feature bundling, which groups features based on their importance to reduce the number of split candidates.

	Before tuning	**After tuning**
R^2	0.5620	0.5648
MAE	41.157	40.905
MSE	3918.08	3893.10
RMSE	62.595	62.395
MAPE	0.3194	0.3170

1.4.13 Voting

Voting is a technique that is commonly used for classification problems, but it can also be adapted for regression tasks. Here, we want to predict a continuous numerical output, rather than a categorical label. The final prediction is made by averaging the predictions of multiple base models. Each base model is trained on a different subset of the training data or with a different algorithm, and once the models are trained, they are used to make individual predictions on the test data. The predictions of all the models are then combined by taking the average, and the final prediction is made based on this average value. Regression voting can be performed in two ways: either by an average of the models' prediction or by a weighted sum. Voting can be an effective approach for improving the accuracy of a regression model, especially when the base models are diverse and perform well individually. However, it may not be the best choice when the base models are highly correlated or when there are outliers in the data.

We **developed our own Voting algorithm** and applied it to our top-performing models: XGBoost, LightGBM, Histogram Gradient Boosting, Extremely Randomized Trees, and Random Forest. After comparing our approach to Scikit-learn's Voting algorithm, we found that our algorithm was faster and achieved better performance. Our algorithm had a run time of 91.64 seconds, while the Scikit-learn algorithm had a run time of 94.71 seconds. To further improve the performance of our Voting algorithm, we fine-tuned the weights assigned to each model using the hyperopt optimization framework. By adjusting these weights, we were able to enhance the overall performance of our algorithm and achieve even better results.

	Scikit-learn	**Before tuning (our model)**	**After tuning (our model)**
R^2	0.5720	0.5720	0.5733
MAE	40.564	40.564	40.506
MSE	3829.15	32829.15	3817.81
RMSE	61.880	61.880	61.788
MAPE	0.3157	0.31557	0.3144

1.4.14 Stacking

Stacking is a technique that combines multiple base models to improve the accuracy of predictions. It involves training a meta-model, which takes the outputs of several base models as input and learns to make a final prediction

based on those inputs. In stacking, the training data is split into multiple folds, and each base model is trained on a subset of the data. The predictions of the base models are then combined to create a new dataset, which is used to train the meta-model. The meta-model takes the predictions of the base models as input and learns to make a final prediction based on those inputs. It is more complex than other ensemble techniques, such as bagging or boosting, and requires careful tuning to achieve the best results.

By developing **our own stacking algorithm** and comparing it to Scikit-learn's model, we were able to achieve superior performance in a shorter amount of time. We found out that concatenating the meta-features with the original features in the final step of training the metamodel resulted in the best performance, surpassing all of our previous attempts. Our approach used a straightforward design with a final linear regression, and we only stacked four models: XGBoost, LightGBM, Extremely Randomized Trees, and Random Forest.

	Scikit-Learn	**Our model**
R^2	0.5736	0.5740
MAE	40.403	40.410
MSE	3814.84	3811.06
RMSE	61.764	61.734
MAPE	0.3104	0.3121

1.5 Results and conclusion

Our homemade stacking and voting algorithms showed the best results on all metrics, with XGBoost completing the top three. Scikit-Learn's Stacking algorithm performed poorly compared to the homemade one, most probably due to the fact that the meta-feature matrix cannot be appended to the initial features in Scikit-Learn's implementation.

Model	R2	MAE	MSE	RMSE	MAPE
Homemade stacking	0.574006	40.409499	3811.056886	61.733758	0.312093
Homemade Voting	0.573251	40.506392	3817.807805	61.788412	0.314359
XGBoost	0.569880	40.756571	3847.962707	62.031949	0.315426
LightGBM	0.567503	40.893731	3869.227835	62.203118	0.316006
CatBoost	0.564835	40.904796	3893.103788	62.394742	0.316990
Random Forest	0.562745	41.077431	3911.796941	62.544360	0.322141
Extremely Randomized Forest	0.561750	41.069603	3920.700182	62.615495	0.325224
sklearn Hist Gradient Boosting	0.559020	41.267632	3945.121306	62.810201	0.319405
Bagging	0.554497	41.776126	3985.585007	63.131490	0.332119
sklearn Stacking	0.550037	41.366705	4025.484549	63.446706	0.317317
sklearn Gradient Boosting	0.535991	42.522334	4151.144567	64.429377	0.337570
Homemade Decision Tree	0.490051	44.813514	4562.136115	67.543587	0.351701
Decision Tree	0.478077	45.825598	4669.256527	68.331958	0.371195
Adaboost	0.426036	48.832359	5134.831774	71.657741	0.403429

Figure 5. Comparative table of results (R^2 decreasing)

To conclude, this project on the Airbnb dataset allowed us to work on several Ensemble Learning algorithms, to discover their specificities, and to tune them to obtain the best possible performance. Another aspect that was just as important as the models was the preprocessing of the data, especially on the work and the selection of features.

2. BUILDING A DECISION TREE FROM SCRATCH

This part will detail the second part of the project consisting in coding from scratch a decision tree able to manage classification and regression tasks. As commented previously in Section 1.4.3, the implemented algorithm was also used on the Airbnb dataset and showed promising results.

2.1 Implementation

In order to facilitate the task we decided to build our decision tree using object oriented programming. The first class *Node* is responsible for representing a node in the tree, which allows access, for example, to the feature and the value that have been chosen in order to split. Moreover, this class is recursive because it allows access from the node in question to its two children who are also of the *Node* class.

The classification and regression tasks are implemented and a tree is used like that of scikit-learn using a *fit* and *predict* function which will respectively train the decision tree on the data passed in input and produce the tree's prediction for a particular element.

In addition to this, a *visualize_tree* function is present in order to be able to graphically represent a decision tree thus facilitating the understanding of its operation.

2.2 Performance

In order to evaluate the performance of our implementation of a decision tree, we decided to compare it with the version present on scikit-learn. We can compare our implementation and that of scikit-learn through two aspects: the results of the evaluation metrics as well as the time complexity.

Concerning the evaluation metrics (accuracy for the classification task, RMSE and coefficient of determination for the regression task) the two decision trees achieve substantially the same results :

	Our model	**scikit-learn**
Accuracy (load_digits)	0.8556	0.7666
Accuracy (load_iris)	1.0	1.0
RMSE (load_diabetes)	66.0775	67.8823
R^2 (load_diabetes)	0.2712	0.2309
RMSE (fetch_california_housing)	0.8087	0.8087
R^2 (fetch_california_housing)	0.5084	0.5084

These results were obtained by training the two trees on the same datasets with the same hyperparameters. For the classification task we chose the scikit-learn *load_digits* dataset and the *load_iris* dataset. For the regression task, whe chose two more datasets, *load_diabetes* and *fetch_california_housing*. All these tests are present in the *Test_of_homemade_decision_tree.ipynb* notebook file.

As expected, the biggest difference is on the time complexity where our implementation is much slower than the one present on scikit-learn :

	Our model	**scikit-learn**
Classification load_digits (s)	3.3778	0.0215
Classification load_iris (s)	0.0817	0.0023
Regression load_diabetes (s)	0.3840	0.0020
Regression fetch_california_housing (s)	86.9535	0.0539

2.3 Visualization

As explained previously, our implementation of a decision tree includes a *visualize_tree* function allowing to display all the nodes of a tree. Let's take the example of the famous *load_iris* dataset and observe the graphical representation of our tree :

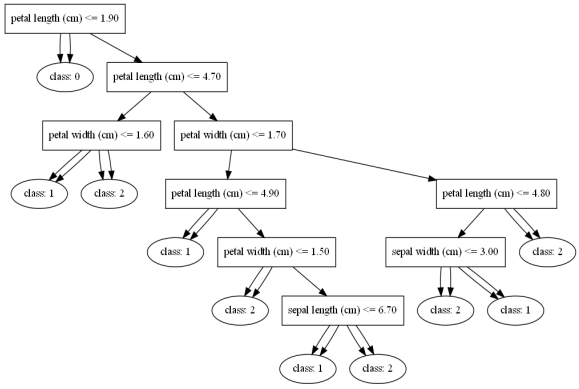

Figure 6. Visualization of our homemade decision tree trained on *load_iris* dataset.

Note that for each split, the chosen feature as well as the associated value is indicated within the node in question which points to its children. For terminal nodes, they are easily identifiable on the graph thanks to a double arrow pointing towards them. Additionally, we are here on a classification task so the class predicted by the tree is also displayed.

Appendix: Correlation of Features