Semester Project

Exploring Gradient Boosting

Author
Andrey Nosov

June 17, 2024

Contents

1 Introduction

This report aims to describe the work done during the "Exploring Gradient Boosting" semester project. The project was done under supervision of Motonobu Kanagawa and Nugzar Gognadze at the EURECOM university. It was also done in parallel with Lucie Moutardier, who worked on the same tasks. Meetings were conducted each week (with the exception of holidays and force majeur situations) and status of the project as well as its current goals were discussed with Nugzar Gognadze.

1.1 Project Goals

The project had the following **objective**:
Gradient Boosting algorithms perform best on tabular data. This project aims to delve into their development, trace their evolution, and if possible make marginal contributions. Students will gain hands-on experience in understanding and implementing boosting algorithms, comparing them with peer algorithms.

Following steps were suggested at the beginning of the project:

1. Weak Learners
   - Explore decision tree construction.
   - Build decision trees for real data.
   - Implement decision trees from scratch in Python.
2. Bagging of decision trees (Random Forest)
   - Understand how bagging enhances predictions.
   - Build a random forest for real data.
   - Develop a simple random forest from scratch in Python.
3. Implementing AdaBoost
   - Walkthrough the implementation of AdaBoost using a basic decision tree as the weak learner.
4. Gradient Boosting
   - Explore the concept behind gradient boosting.
   - Implement gradient boosting for real data.
   - Develop a simple gradient boosting algorithm from scratch in Python.
5. XGBoost
   - Understand the idea behind XGBoost.
   - Build XGBoost for real data.
   - Implement a basic gradient boosting algorithm from scratch in Python.
6. LightGBM and CatBoost
   - Understand the concepts behind LightGBM and CatBoost.
   - Build LightGBM and CatBoost for real data.
7. Additional tasks
   - Propose innovative ideas to enhance gradient boosting techniques.

It was discussed at the beginning of the project that the mentioned steps should serve as a guide and that actual outcomes of the project will highly depend on our current knowledge of machine learning (ML) algorithms and experiences of implementing such algorithms in python. As an electrical engineering student I did not have much experience with ML before this project apart from one course on the basics of ML taken in the previous semester. Nevertheless I found this field fascinating and that was the reason for choosing this project, which allowed me to deepen my understanding of decision trees and ensemble techniques used to improve them. That said, unfortunately in 3 months of work not all steps were completed due to various reasons, mainly due to long debugging processes when implementing the ML algorithms in python. The steps 1. - 4. were fully completed and will be described in the following chapters.

In addition to this report jupyter notebook files are provided that demonstrate results of the implemented models in comparison to equivalent models that are part of the scikit-learn library.

2 Model comparison

All models implemented in python were tested to ensure their correct function and compared to the equivalent reference models from the sckit-learn library. The process of testing and comparison will be the same for all models and consists out of the following steps.

1. Split the dataset into train and test datasets with 80:20 ratio
2. Fit the models and compare train and test prediction score using overfit hyper-parameters
   - Used to ensure the algorithm was implemented correctly
   - Custom algorithm should perfectly fit train data when overfitted
   - Should show similar performance on the test data as the reference model
3. Fit the models and compare train and test prediction score with randomly chosen hyper-parameters
   - Again used to ensure the custom algorithm was implemented correctly
   - Should show similar performance on the test and train data as the reference model
4. Fit the models and compare test prediction scores with hyper-tuned hyperparameters
   - Shows best ability of the models to predict test data
   - Hyper-tuning is done using 5-Fold cross-validation on a predefined parameter space
   - In case of random forest models the hyper-tuning is done using bagging datasets and OOB scores

Metrics used to evaluate the models are accuracy for the classification problems

$$\text{Accuracy} = \frac{\text{Number of Correct Predictions}}{\text{Total Number of Predictions}}$$

and R2 for regression problems

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

where y_i : actual value, $\hat{y}_i$: predicted value, $\bar{y}$: mean of the actual values, n : number of observations.

It is worth mentioning that all scikit-learn models feature a parameter called `random_state`, which serves as a seed for pseudo-random processes inside of the models. To keep the results consistent a value of 42 was used in all testings. In case of the custom models all possible random functions do not have a parameter for the seed value and thus the results slightly variate from one fit to another. We use our tests only to prove that the models were implemented correctly and to show general advantages/disadvantages of each model, thus these slight variations can be ignored.

2.1 Wine Quality Dataset

To compare classifier models the Wine Quality dataset from the UCI Machine Learning repository was chosen. It contains 1599 samples with 11 features and 6 classes. The structure of the dataset can be seen in figure 1.

The main reasons for choosing this particular dataset were following:

- **Size**: Is reasonably sized and can be easily used on a local machine

- **Features**: Has big enough set of features to sample from when creating ensembles
- **Complexity**: Data cannot be trivially separated allowing us to see improvements as we move from simple trees to ensembles

The main drawback of this particular dataset is uneven distribution of classes, which would need to be dealt with using various techniques if we wanted to build a quality prediction model. However in our case the datasets are used only for model comparison and thus we can ignore this aspect.

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol	quality
0	7.4	0.700	0.00	1.9	0.076	11.0	34.0	0.99780	3.51	0.56	9.4	5
1	7.8	0.880	0.00	2.6	0.098	25.0	67.0	0.99680	3.20	0.68	9.8	5
2	7.8	0.760	0.04	2.3	0.092	15.0	54.0	0.99700	3.26	0.65	9.8	5
3	11.2	0.280	0.56	1.9	0.075	17.0	60.0	0.99800	3.16	0.58	9.8	6
4	7.4	0.700	0.00	1.9	0.076	11.0	34.0	0.99780	3.51	0.56	9.4	5
...	...	...	...	...	...	...	...	...	...	...	...	...
1594	6.2	0.600	0.08	2.0	0.090	32.0	44.0	0.99490	3.45	0.58	10.5	5
1595	5.9	0.550	0.10	2.2	0.062	39.0	51.0	0.99512	3.52	0.76	11.2	6
1596	6.3	0.510	0.13	2.3	0.076	29.0	40.0	0.99574	3.42	0.75	11.0	6
1597	5.9	0.645	0.12	2.0	0.075	32.0	44.0	0.99547	3.57	0.71	10.2	5
1598	6.0	0.310	0.47	3.6	0.067	18.0	42.0	0.99549	3.39	0.66	11.0	6

Figure 1: Structure of the Wine Quality dataset

2.2 Scikit Diabetes Toy Dataset

To compare regression models the Scikit Diabetes Toy dataset was selected. It comprises 442 samples with 10 features and 1 continuous target variable. The dataset's structure is illustrated in figure 2.

The reasons for choosing this dataset are essentially the same as in case of the Wine Quality Dataset and are its size, number of features and non trivial complexity.

	age	sex	bmi	bp	s1	s2	s3	s4	s5	s6	0
0	0.038076	0.050680	0.061696	0.021872	-0.044223	-0.034821	-0.043401	-0.002592	0.019907	-0.017646	151.0
1	-0.001882	-0.044642	-0.051474	-0.026328	-0.008449	-0.019163	0.074412	-0.039493	-0.068332	-0.092204	75.0
2	0.085299	0.050680	0.044451	-0.005670	-0.045599	-0.034194	-0.032356	-0.002592	0.002861	-0.025930	141.0
3	-0.089063	-0.044642	-0.011595	-0.036656	0.012191	0.024991	-0.036038	0.034309	0.022688	-0.009362	206.0
4	0.005383	-0.044642	-0.036385	0.021872	0.003935	0.015596	0.008142	-0.002592	-0.031988	-0.046641	135.0

Figure 2: Structure of the Scikit Diabetes dataset

3 Decision Trees

Decision trees are a popular and intuitive method for classification and regression tasks in data science and machine learning. They work by recursively splitting a dataset into subsets based on feature values, forming a tree-like structure where each internal node represents a decision based on a feature, each branch represents the outcome of the decision, and each leaf node represents a final prediction or decision outcome.

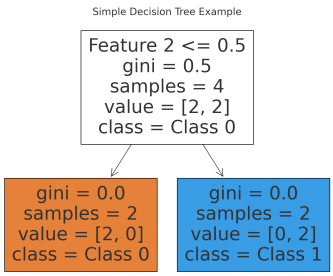

Figure 3: Example of a simple decision tree with one node and two leaves

At each node the algorithm selects the feature that best splits the data according to a specific criterion, such as Gini impurity, entropy, or mean squared error. The goal is to create partitions that are as pure as possible in terms of the target variable.

3.1 Splitting Criteria

The choice of splitting criteria is crucial in building an effective decision tree. Some common criteria include:

- **Gini Impurity**: Used in classification tasks, Gini impurity evaluates the probability that a randomly chosen data point will be misclassified when its class label is determined according to the class distribution within a specific node. A Gini impurity of 0 indicates perfect purity while a Gini impurity of 0.5 indicates maximum impurity.
- **Entropy**: Also used for classification, entropy measures how similar the distribution of the classes in the node is to the uniform distribution (worst case scenario). Higher entropy indicates that the distribution of classes is close to uniform, while lower values indicate higher node purity.
- **Mean Squared Error (MSE)**: Used in regression tasks, MSE measures the average of the squares of the errors, i.e., the difference between the observed and predicted values. The aim is to minimize MSE for better predictions.

To better understand the concepts of the decision tree algorithm a classification and a regression models were implemented in python and compared to the equivalent scikit-learn models.

3.2 CustomDecisionTreeClassifier

CustomDecisionTreeClassifier is a decison tree classifier and therefore is suitable to solve classification problems. It can be initialized and used with the following integer type parameters:

- **max_depth** - the maximum depth of the tree
- **min_samples_split** - the minimum number of samples required to split an internal node.

- **min_samples_leaf** - the minimum number of samples required to be at a leaf node. A split point at any depth will only be considered if it leaves at least `min_samples_leaf` training samples in each of the left and right branches.

To measure the quality of each split the model uses weighted Gini impurity. First Gini impurity $G_{L,R}$ is calculated for the left and right nodes that were created after the split using the formula

$$G_{L,R} = 1 - \sum_{i=1}^{C} p(i)^2 \tag{1}$$

where C is the number of classes present in the data and $p(i)$ is the probability of picking an element of class i in the current node. After obtaining the impurity measurements for both nodes a weighted sum is performed to compute the final quality measurement of the split.

$$G_{split} = p_L * G_L + p_R * G_R \tag{2}$$

The weights p_L and p_R represent a fraction of elements that were split into the left and right nodes respectively.

To stop the splitting process from going infinitely the model uses the following stopping criteria:

- All the samples in the node belong to the same class.
- The current depth of the tree is equal to the maximum depth.
- The number of samples is less than the minimum number of samples required to split an internal node.
- No split yielded a new node with more samples than the minimum required.

3.3 CustomDecisionTreeRegressor

CustomDecisionTreeRegressor is a decison tree regressor and therefore is suitable to solve regression problems. It can be initialized and used with the same set of parameters as the CustomDecisionTreeClassifier.

To measure the quality of each split, the model uses weighted Mean Square Error (MSE). First, $MSE_{L,R}$ is calculated for the left and right nodes created after the split using the formula

$$MSE_{L,R} = \frac{1}{N_{L,R}} \sum_{i \in L,R} (y_i - \hat{y}_i)^2 \tag{3}$$

where $N_{L,R}$ is the number of samples in the left and right nodes, y_i is the true target value of the i-th sample, and $\hat{y}$ is the predicted target value for nodes L and R. After obtaining the MSE measurements for both nodes, a weighted sum is performed to compute the final quality measurement of the split.

$$MSE_{split} = p_L \cdot MSE_L + p_R \cdot MSE_R \tag{4}$$

The weights p_L and p_R represent the fraction of elements that were split into the left and right nodes, respectively.

Similar to the classification tree, to prevent the tree from growing infinitely, the model uses the following stopping criteria:

- All the samples in the node have the same target value.
- The current depth of the tree equals the maximum depth.
- The number of samples is less than the minimum number required to split an internal node.
- No split yielded a new node with more samples than the minimum required.

3.4 Results of CustomDecisionTreeClassifier

The CustomDecisionTreeClassifier was evaluated and compared against the equivalent DecisionTreeClassifier from the scikit-learn library using the Wine Quality dataset. For a fair comparison the default Gini impurity split quality measurement was used in the scikit model and all parameters not present in the CustomDecisionTreeClassifier model were left at their default values. The tests were performed as described in section 2.

First the ability of the model to perfectly fit the train data was tested. For this purpose a `max_depth` parameter was set to 100, which would ensure the big enough tree to leave only pure leaves. The accuracy scores can be seen in table 1.

Table 1: Tree Classifier Comparison for overfit parameters

Model	Accuracy on Test Data	Accuracy on Training Data
Scikit-learn	0.56	1.00
Custom	0.57	1.00

The results show that both the scikit-learn's DecisionTreeClassifier and our CustomDecisionTreeClassifier achieve perfect accuracy (1.00) on the training data when forced to overfit with a high `max_depth`. The performance on the test data is expectendly worse, however both models exhibit similar accuracy, with the custom implementation slightly outperforming the scikit-learn model, which is most probably given by the effect of the `random_state` parameter. This suggests that the custom model works correctly and shows very similar behaviour to the reference model.

To further strengthen this argument the models were tested on randomly chosen parameters with the results shown in table 2.

Table 2: Tree Classifier Comparison for random parameters

max_depth=2, min_samples_split=3, min_samples_leaf=4

Model	Test Data Accuracy	Training Data Accuracy
Scikit-learn model	0.53	0.56
Custom model	0.53	0.56

In this test with more reasonable parameters, both models achieve the exact same accuracy on both test and training datasets, further supporting that the custom implementation is robust and aligns closely with the scikit-learn implementation.

Finally, to see the best potential results of our models they were hyper-tuned. The achieved results are shown in table 3 with accompanying confusion matrices obtained from the predictions of both models in figure 4.

Previous tests and comparisons can be found in the `tree_classifier_comparison.ipynb` file.

Table 3: Tree Classifier Comparison for best parameters

Model	Test Data Accuracy
Scikit-learn	0.56
Custom	0.57

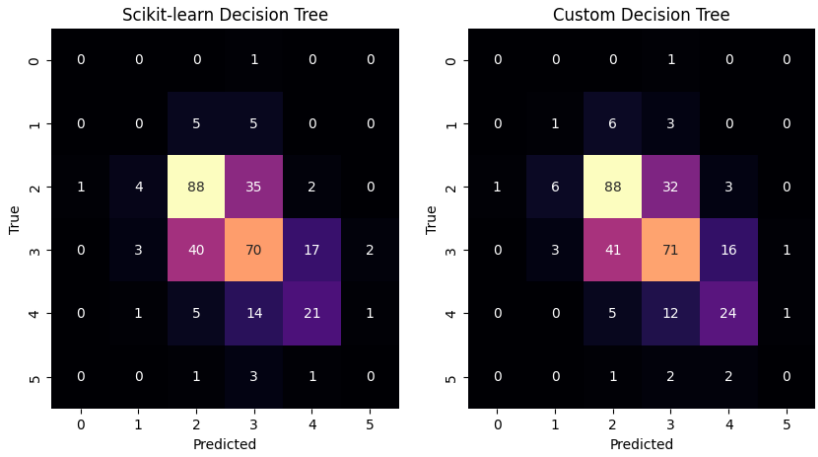

Figure 4: Comparison of confusion matrices for tree classifiers

3.5 Results of CustomDecisionTreeRegressor

CustomDecisionTreeRegressor was evaluated and compared against the equivalent DecisionTreeRegressor from the scikit-learn library using a regression task as described in section 2. The performance metrics were assessed on both training and test datasets. The results for different parameter settings are summarized below.

Table 4: Tree Regressor Comparison for overfit parameters

Model	R2 on Test Data	R2 on Training Data
Scikit-learn	0.06	1.00
Custom	-0.05	1.00

In this test, both models were trained with parameters aimed at overfitting the training data, specifically with `max_depth`=100. Both models fit the train data perfectly and both perform very poorly on the test data, showing very similar behaviour when overfitted as can be observed in table 4

Table 5: Tree Regressor Comparison for random parameters

max_depth=2, min_samples_split=3, min_samples_leaf=4

Model	R2 on Test Data	R2 on Training Data
Scikit-learn model	0.29	0.45
Custom model	0.27	0.45

For random parameters the results are again very similar for both models (table 5), again suggesting that the custom implementation works as intended. This time they were not able to fit the train data perfectly, as the depth parameter is much lower, decreasing the R2 score for the train data prediction. Test score however improved, which is given by better generalization as we prevent overfitting of the models.

Table 6: Tree Regressor Comparison for best parameters

Model	R2 on Test Data
Scikit-learn	0.39
Custom	0.41

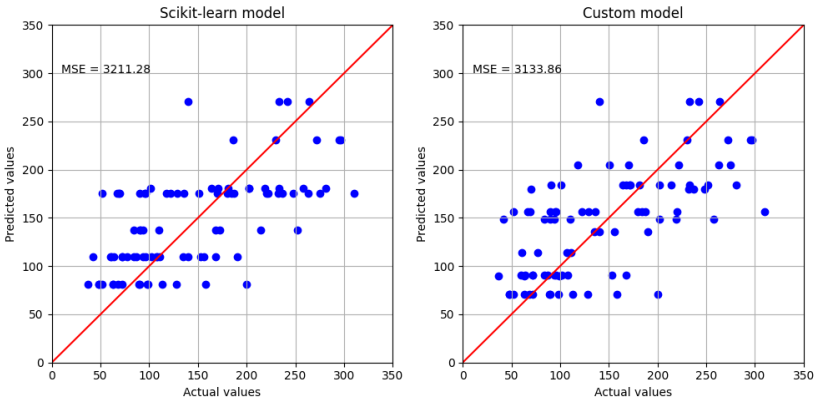

Figure 5: Comparison of test prediction results for tree regressors

When hyper-tuned to find the best parameters, both models achieve comparable R2 scores on the test data (table 6, figure 5), indicating similar predictive performance.

These results highlight that the custom implementation of DecisionTreeRegressor closely mirrors the behavior of the scikit-learn implementation across different parameter settings. While there are slight variations in performance metrics most probably given by the pseudo random split selection in the scikit model, particularly evident in the overfitting scenario, overall consistency in model behavior supports validity of the custom implementation of the model.

Previous tests and comparisons can be found in the `tree_regressor_comparison.ipynb` file.

4 Random Forests

Random forests build on the concepts of decision trees, offering a way to improve prediction accuracy. A random forest is an ensemble method, meaning it combines multiple decision trees to create a stronger overall model.

The key idea behind random forests is to generate many different decision trees by introducing randomness. This randomness comes from two main sources: using random subsets of features for each tree and creating diverse training datasets through a method called bootstrap sampling. By doing this, each tree in the forest is slightly different, which helps to avoid overfitting and improves the model's ability to generalize to new data.

When it comes to making predictions, a random forest aggregates the outputs of all its individual trees. For classification tasks, it takes a majority vote, while for regression tasks, it averages the predictions.

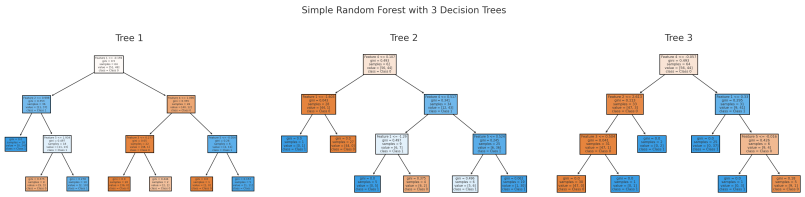

Figure 6: Example of a simple random forest with three trees

4.1 Bagging

For each classification or regression model (predictor) an expected generalization error can be decomposed into three different parts:

$$\mathbb{E}_{x,y,\mathcal{D}}\left[(h_{\mathcal{D}}(x)-y)^2\right] = \underbrace{\mathbb{E}_{x,\mathcal{D}}\left[(h_{\mathcal{D}}(x)-\bar{h}(x))^2\right]}_{\text{Variance}} + \underbrace{\mathbb{E}_{x,y}\left[(\bar{y}(x)-y)^2\right]}_{\text{Noise}} + \underbrace{\mathbb{E}_{x}\left[(\bar{h}(x)-\bar{y}(x))^2\right]}_{\text{Bias}^2} \tag{5}$$

- $\mathbb{E}_{x,y,\mathcal{D}}$: Expected generalization error given $h_{\mathcal{D}}$
- x: Input data
- y: True output value
- $\mathcal{D}$: Training dataset
- $h_{\mathcal{D}}(x)$: Predictor learned from training dataset $\mathcal{D}$
- $\bar{h}(x)$: Expected predictor
- $\bar{y}(x)$: Expected true output value
- Variance: Measures the variability of the model prediction $h_{\mathcal{D}}(x)$ around its mean $\bar{h}(x)$
- Noise: Represents the inherent noise in the data, i.e., the variability of the true output y around its mean $\bar{y}(x)$
- Bias: Measures the error due to the difference between the expected model prediction $\bar{h}(x)$ and the expected true output $\bar{y}(x)$

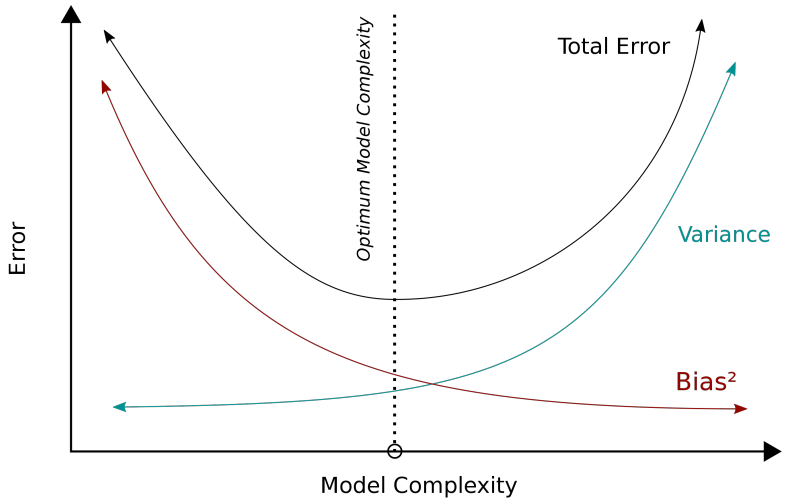

Figure 7: Illustration of the variance-bias trade-off

Usually when working with prediction algorithms we encounter a so called variance-bias trade-off, which tells us that by decreasing one we increase the other as illustrated in figure 7.

Bagging is an ensemble method which allows us to reduce variance without affecting the bias of the model. In case of the equation 5 it means $h_{\mathcal{D}}(x) \to \bar{h}(x)$ which will be achieved by averaging multiple models.

The idea of bagging comes from the weak law of large numbers which says that for a set of i.i.d. (independent and identically distributed) random variables x_i with mean $\bar{x}$

$$\frac{1}{M}\sum_{i=1}^{M} x_i \to \bar{x} \quad \text{as} \quad M \to \infty. \tag{6}$$

We apply the weak law of large numbers to classifiers:

1. M datasets available $\mathcal{D}_1, \ldots, \mathcal{D}_M$ drawn from an unknown data probability distribution
2. Train a classifier on each dataset and then average.

Ensemble of classifiers H:

$$H = \frac{1}{M}\sum_{i=1}^{M} h_{\mathcal{D}_i}(x) \to \bar{h}(x) \quad \text{as} \quad M \to \infty \tag{7}$$

The problem of this approach is that only one training dataset is available. The solution to this is bootstrap sampling, which given a set $\mathcal{D}$ containing N training samples creates new dataset $\mathcal{D}_{new}$ by drawing N samples at random with replacement from $\mathcal{D}$. By repeating the process M times we will obtain M datasets. The i.i.d. condition of course does not hold, which means $h_{\mathcal{D}}(x) \not\to \bar{h}(x)$. Despite this it was shown that in praxis this still reduces variance and improves the generalization error of the model.

4.2 CustomRandomForestClassifier

The CustomRandomForestClassifier is a random forest classifier that can use both scikit-learn DecisionTreeClassifier and the CustomDecisionTreeClassifier to build its trees. It can be used only for classification problems, meaning the model will predict the test sample class based on the majority vote of the trained ensemble. Each tree in the forest supports all of the following CustomDecisionTreeClassifier integer type parameters that the model can be initialized with:

- **max_depth** - the maximum depth of the tree
- **min_samples_split** - the minimum number of samples required to split an internal node.
- **min_samples_leaf** - the minimum number of samples required to be at a leaf node. A split point at any depth will only be considered if it leaves at least `min_samples_leaf` training samples in each of the left and right branches.

As well as some new parameters which modify the behaviour of the model:

- **tree_type** - (str) the tree classifier type used: `'scikit'` for scikit-learm DecisionTreeClassifier and `'custom'` for CustomDecisionTreeClassifier.
- **'n_estimators'** - (int) number of classifiers used to build the forest.
- **'max_features'** - (str/int) number of randomly selected features used to build classifiers: an int number to give a specific value, `'sqrt'` to use square root of the total number of features in the dataset, `'log2'` to use log with base 2 of the total number of features in the dataset.

4.3 CustomRandomForestRegressor

The CustomRandomForestRegressor is a random forest regresor that can use both scikit-learn DecisionTreeRegressor and the CustomDecisionTreeRegressor to build its trees. It can be used only for regression problems, meaning the model will predict the test sample class based on the mean value taken from all the predictions of the trained ensemble. The model supports the same set of parameters as CustomRandomForestClassifier.

4.4 Modifications to CustomDecisionTreeClassifier and CustomDecisionTreeRegressor

To make CustomDecisionTreeClassifier and CustomDecisionTreeRegressor more suitable for being used as a part of the random forest, the **'max_features'** parameter was added. This parameter is used to limit the number of features when searching for the best split, increasing the variability of the trees in the ensemble.

The models were also improved in terms of their computational performance. Multiple loops were exchanged for numpy array operations, which improved the training time significantly without affecting the accuracy.

4.5 OOB Score

As an essential part of the random forest, bagging is performed by the CustomRandomForestClassifier and the CustomRandomForestRegressor. Using the bootstrapping technique the model randomly samples n datasets with replacement and builds n trees using those datasets. Each sample that was not chosen becomes part of the out-of-bag (OOB) dataset for that tree. After all trees are built, each sample from the original dataset becomes a test sample for the sub-forest of the built ensemble for which it was part of the OOB. The accuracy of those sub-forests provides us with an OOB score that can be used for model validation without the need of a validation dataset.

4.6 Results of CustomRandomForestClassifier

The CustomRandomForestClassifier was evaluated and compared against the equivalent RandomForestClassifier from the scikit-learn library using the Wine Quality dataset. Since the CustomRandomForestClassifier provides an option to choose between scikit and custom trees to be used for the ensemble, both versions were tested. Due to the randomness of the forests we do not expect the models to show the exact same results even for the train dataset, but rather the same magnitude of values.

First we overfit all three models.

Table 7: Forest Classifier Comparison for overfit parameters

Model Comparison	Accuracy on Test Data	Accuracy on Training Data
Scikit-learn forest - Scikit-learn tree	0.65	1.00
Custom forest - Scikit-learn tree	0.63	1.00
Custom forest - Custom tree	0.64	1.00

Then we use a set of random predefined parameters to fit the models.

Table 8: Forest Classifier Comparison for random parameters

max_depth=3, min_samples_split=4, min_samples_leaf=5, n_estimators=6

Model Comparison	Test Data Accuracy	Training Data Accuracy
Scikit-learn forest vs Scikit-learn tree	0.54	0.62
Custom forest vs Scikit-learn tree	0.56	0.60
Custom forest vs Custom tree	0.54	0.60

From tables 7 and 8 we can see, as expected, slight variations in the results. Nevertheless, it is clear that all three models behave in a very similar manner and we can state that the custom implementation was done correctly. Further we can observe big improvements in terms of accuracy of prediction of the test data. Even overfitted forests perform significantly better than stand alone decision trees, which provides evidence that the bagging technique is indeed working.

To obtain even better results the hyper parameters of all three models were tuned.

Table 9: Forest Classifier Comparison for best parameters

Model Comparison	Test Data Accuracy
Scikit Forest - Scikit Tree	0.67
Custom Forest - Scikit Tree	0.66
Custom Forest - Custom Tree	0.68

As expected, prediction of the test data is further improved when fitting hyper-tuned models as can be seen in table 9 and accompanying confusion matrices 4.

Previous tests and comparisons can be found in the `forest_classifier_comparison.ipynb` file.

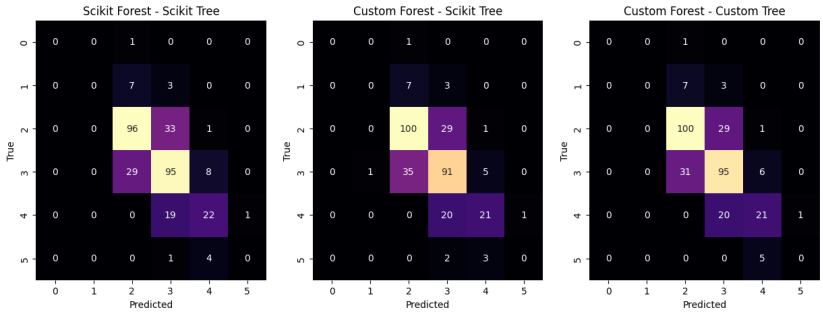

Figure 8: Comparison of confusion matrices for forest classifiers

4.7 Results of CustomRandomForestRegressor

The CustomRandomForestRegressor was evaluated and compared against the equivalent RandomForestRegressor from the scikit-learn library using the scikit Diabetes dataset. Again, as in case of CustomRandomForestClassifier, CustomRandomForestRegressor provides option to build the forest out of scikit or custom decision trees. Both versions of the cutom forest were tested and compared to the scikit-learn RandomForestRegressor.

First we try to overfit the models. Since the predictions are made as a mean value of the whole ensemble it will highly depend on the dataset and the randomness of the forest whether or not the training data can be fit perfectly. In case of the Diabetes dataset it was possible to fit the train data with 0.92 R2 score as shown in 10.

Table 10: Forest Regresor Comparison for overfit parameters

Model Comparison	R2 on Test Data	R2 on Training Data
Scikit-learn forest - Scikit-learn tree	0.43	0.92
Custom forest - Scikit-learn tree	0.40	0.92
Custom forest - Custom tree	0.44	0.92

Then we used a set of predefined random parameters to fit the models.

Table 11: Forest Regressor Comparison for random parameters

max_depth=3, min_samples_split=4, min_samples_leaf=5, n_estimators=6		
Model Comparison	Test Data R2	Training Data R2
Scikit-learn forest vs Scikit-learn tree	0.41	0.52
Custom forest vs Scikit-learn tree	0.45	0.53
Custom forest vs Custom tree	0.41	0.53

Results in both tables 10 and figure 11 show consistency of the models. This suggest that the custom model works similarly to the reference scikit model and was implemented correctly.

All three models were hyper-tuned to obtain the best results.

Table 12: Forest Regressor Comparison for best parameters

Model Comparison	Test Data R2
Scikit Forest - Scikit Tree	0.47
Custom Forest - Scikit Tree	0.45
Custom Forest - Custom Tree	0.46

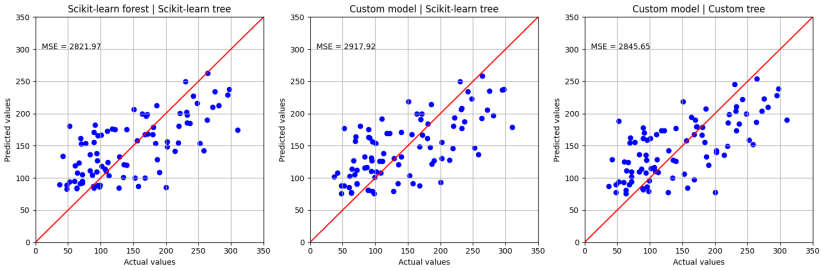

Figure 9: Comparison of test prediction results for forest regressors

From the results in table 12 and figure 9 we see an improvement of R2 score in comparison to the single decision tree regression models again showing how bagging allows to lower the test error.

Previous tests and comparisons can be found in the `forest_regressor_comparison.ipynb` file.

5 AdaBoost

AdaBoost, short for Adaptive Boosting, is another ensemble learning method. It is designed to improve the performance of weak classifiers (that predict slightly better than random guessing) by combining them into a strong classifier (which predictions are well correlated with true labels). The core idea of AdaBoost is to focus on instances that are difficult to classify and adjust the weights of the classifiers accordingly. Unlike random forests, which generate multiple trees independently and aggregate their results, AdaBoost builds trees sequentially, with each new tree attempting to correct the errors made by the previous ones.

5.1 The algorithm

AdaBoost works by iteratively training a sequence of weak classifiers, typically decision stumps (a one-level decision tree). Each classifier is trained on the same dataset but with adjusted weights to emphasize the instances that were misclassified by previous classifiers. The algorithm works as follows (we assume binary classification example with $y \in \{-1, +1\}$):

1. Initialize the weights w_i for each training instance i to be equal.

$$w_i = \frac{1}{N}, \tag{8}$$

where N is the number of training instances.

2. For $t = 1$ to T (number of classifiers):
   1. Train a weak classifier $h_t(x)$ using the weighted training data.
   2. Compute the error ϵ_t of the classifier:

$$\epsilon_t = \sum_{i=1}^{N} w_i I(y_i \neq h_t(x_i)), \tag{9}$$

   where I is the indicator function taking value 1 when $y_i \neq h_t(x_i)$ and 0 otherwise, making only the misclassified values count.

   3. Compute the classifier's weight α_t:

$$\alpha_t = \frac{1}{2} \ln \left(\frac{1 - \epsilon_t}{\epsilon_t} \right). \tag{10}$$

   4. Update the weights for the next iteration:

$$w_i \leftarrow w_i \exp\left(-\alpha_t y_i h_t(x_i)\right). \tag{11}$$

   Normalize the weights so that they sum to 1.

3. The final strong classifier is a weighted majority vote of the T weak classifiers:

$$H(x) = \text{sign} \left(\sum_{t=1}^{T} \alpha_t h_t(x) \right). \tag{12}$$

6 Gradient Boosting

Gradient boosting is also an ensemble technique that builds on the concepts of decision trees that aims to improve model performance through a sequential approach. Just like the AdaBoost algorithm, gradient boosting builds trees iteratively, with each new tree correcting the errors of the previous ones.

The key idea behind gradient boosting is to again combine weak learners, typically shallow decision trees, in a chain-wise fashion to create a strong predictive model. Each tree is trained to minimize the pseudo-residual errors of the combined ensemble of trees that came before it. Gradient of a specified loss function measures the difference between the predicted and the true values and guides the predictions towards the true values.

In gradient boosting, the process starts with an initial model, usually a simple one like the mean of the target variable. At each subsequent step, a new tree is added to the model to predict the pseudo-residuals of the current ensemble. The predictions of these pseudo-residuals are then scaled by a learning rate and added to the ensemble, gradually improving its performance.

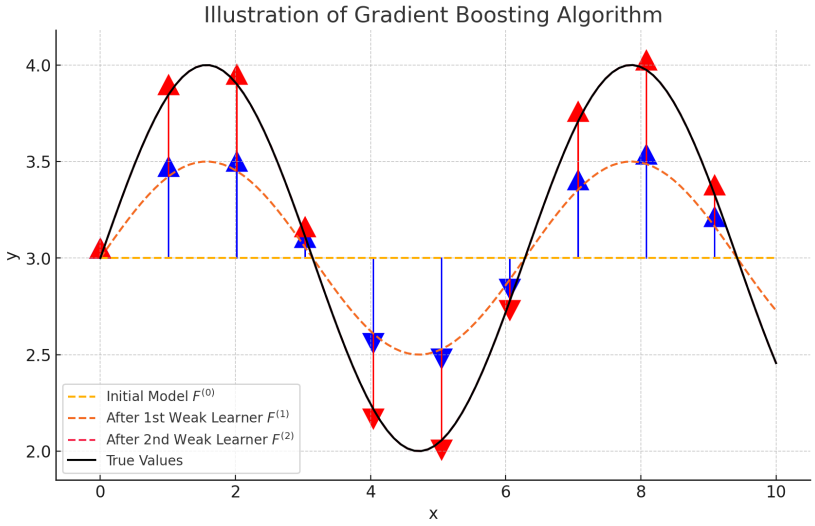

Figure 10: Illustration of the gradient boosting process

6.1 Boosting Algorithm

The boosting process involves the following steps:

1. Initialize the model with a constant value like the mean of the target variable (this will be our initial model $H^{(0)}$).
2. For each iteration $m = 1$ to M:
   (a) Compute the pseudo-residuals $r_i^{(m)}$ for each data point i:

$$r_i^{(m)} = -\left[\frac{\partial L(y_i, H(x_i))}{\partial H(x_i)}\right]_{H=H_{m-1}} \tag{13}$$

   Here, $L(y_i, H(x_i))$ is the loss function, y_i is the actual target value, $H(x_i)$ is the predicted value by the model at iteration $m-1$, and H_{m-1} represents the model at iteration $m-1$.

   (b) Train a new weak learner $h^{(m)}$ on the residuals $r^{(m)}$.

(c) Update the model:

$$H^{(m)}(x) = H^{(m-1)}(x) + \alpha \cdot h^{(m)}(x) \tag{14}$$

where α is the learning rate, a value between 0 and 1 that controls the contribution of each weak learner.

3. Predict new values using the final ensemble:

$$H^{(m)}(x) = H^{(0)}(x) + \sum_{m=1}^{M} \alpha \cdot h^{(m)}(x) \tag{15}$$

6.2 CustomGradientBoostingRegressor

For this task only regression model was implemented. CustomGradientBoostingRegressor is a gradient boosting regressor that can use both scikit-learn DecisionTreeRegressor and the CustomDecisionTreeRegressor to build its trees. It supports the same set of parameters as the CustomRandomForestRegressor model:

- **max_depth** - (int) the maximum depth of the tree
- **min_samples_split** - (int) the minimum number of samples required to split an internal node.
- **min_samples_leaf** - (int) the minimum number of samples required to be at a leaf node. A split point at any depth will only be considered if it leaves at least `min_samples_leaf` training samples in each of the left and right branches.
- **tree_type** - (str) the tree regressor type used: `'scikit'` for scikit-learm DecisionTreeRegressor and `'custom'` for CustomDecisionTreeRegressor.
- **'n_estimators'** - (int) number of trees used in a sequence.
- **'max_features'** - (str/int) number of randomly selected features used to build tree regressors: an int number to give a specific value, `'sqrt'` to use square root of the total number of features in the dataset, `'log2'` to use log with base 2 of the total number of features in the dataset.

As described in the previous sub-section we need to define a loss function for our model. Since scikit-learn GradientBoostingRegressor uses Mean Square Error loss it seemed to be the obvious choice, considering the scikit implementation is going to serve as the reference model.

We define the loss as:

$$L(y, H(x)) = \frac{1}{2}(y - H(x))^2 \tag{16}$$

And calculate pseudo-residuals for step m as:

$$r_i^{(m)} = y_i - H_{m-1}(x_i) \tag{17}$$

6.3 Results of CustomGradientBoostingRegressor

The CustomGradientBoostingRegressor was evaluated and compared against the equivalent GradientBoostingRegressor from the scikit-learn library using the scikit Diabetes dataset as for the previous regression models. Again as in the case of random forests we compare three models: scikit-learn GradientBoostingRegressor, CustomGradientBoostingRegressor with scikit-learn trees used as weak learners and CustomGradientBoostingRegressor with custom trees used as weak learners.

We begin by overfitting all three models:

Table 13: Gradient Boosting models comparison for overfit parameters

Model Comparison	R2 on Test Data	R2 on Training Data
Scikit-learn model - Scikit-learn tree	0.28	1.00
Custom model - Scikit-learn tree	0.30	1.00
Custom model - Custom tree	0.29	1.00

Then we use a set of predefined random parameters to fit the models:

Table 14: Gradient Boosting models comparison for random parameters

max_depth=3, min_samples_split=4, min_samples_leaf=5, n_estimators=6

Model Comparison	Test Data R2	Training Data R2
Scikit-learn model vs Scikit-learn tree	0.29	0.37
Custom model vs Scikit-learn tree	0.29	0.36
Custom model vs Custom tree	0.29	0.36

From the very consistent results in tables 13 and 14 we can, as in previous comparisons, conclude that the models behave very similarly and the custom implementation is valid.

To achieve the best results the parameters were hyper-tuned. Since building of the trees is done sequentially, the training times for the gradient boosting models are higher than for the previous algorithms. This is especially the case for the custom implementation which is not optimized. For that reason the hyper-tuning was done only on the scikit-learn model and than the same tuned parameters were used to build the two custom ones.

Table 15: Gradient Boosting models comparison for best parameters

Model Comparison	Test Data R2
Scikit Forest - Scikit Tree	0.47
Custom Forest - Scikit Tree	0.47
Custom Forest - Custom Tree	0.47

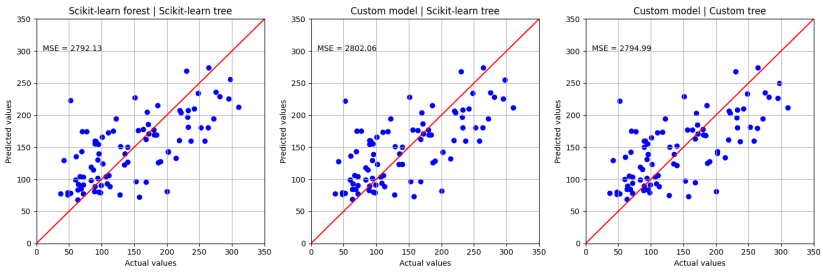

Figure 11: Comparison of test prediction results for gradient boosting models

All three models show the same results and an evident improvement in comparison to the regression tree models. The results are quite similar to the ones of the random forest regressors, which might not be the case when working with different datasets.

Previous tests and comparisons can be found in the `gradient_boosting_regressor_comparison.ipynb` file.

7 Conclusion

This project aimed to delve into the development and implementation of gradient boosting algorithms, comparing their performance with other algorithms such as decision trees and random forests. Throughout this semester, significant progress was made in understanding the underlying mechanisms and practical implementation of these machine learning techniques.

First, the custom implementations of decision trees were validated against their scikit-learn counterparts, showing consistent and comparable results. This validation was important for ensuring that the custom models were correctly implemented and could serve as reliable weak learners for the ensemble algorithms.

Subsequently, the theoretical exploration and implementation of random forests demonstrated the power of bagging techniques. The random forest models showed significant improvements in prediction accuracy compared to standalone decision trees by combining the results of multiple predictors.

Finally, a gradient boosting algorithm was developed. By sequentially building trees to correct the errors of their predecessors, the gradient boosting model again achieved better performance metrics than the individual decision trees. The custom gradient boosting implementation, although more time-consuming due to lack of optimization, closely mirrored the results of the scikit-learn models, confirming the validity of the approach.

Unfortunately the time constrains of the project did not allow to dive into the more advanced topics such as implementation of XGBoost algorithm.

In summary, the project successfully achieved its goals of implementing and understanding various ensemble learning techniques. The consistent performance of custom models compared to established scikit-learn implementations validated their implementations.