McGill University

Implementing Decision Trees, Random Forests, and Gradient Boosting for Classification

Final Project Report

MGSC 695: Optimization for Data Science

Winter 2024

Dr. Sanjith Gopalakrishnan
McGill University

Lakshya Agarwal
Michelle Barabasz
Arham Anwar
Jared Balakrishnan

March 4, 2024

Contents

1 Introduction to Tree-Based Methods

This group project was focused on the implementation of three different **tree-based** methods, a collection of statistical learning methods that can be used for regression as well as classification tasks.

At a very high level, these methods divide the predictor space into a number of regions. The prediction for a given observation is then made based upon the mean or mode of the target value for the training data in the region to which it belongs.

The rules that go into segmenting the predictor space can be visualized in the form of a tree structure, because of which these methods are called *tree-based* or *decision-tree* based methods.

In this project, we implemented classifiers based on tree-based algorithms, including **Decision Trees**, **Random Forests**, and **Gradient Boosting**.

1.1 Decision Trees

Decision Trees [1] are a class of machine learning algorithms capable of performing classification, prediction (regression), as well as multi-output tasks. These are the fundamental building blocks of **Random Forests** discussed in the following section. The two main kinds of decision trees are **Regression Trees** and **Classification Trees**.

1.1.1 Regression Trees

When a decision tree is used to predict a numerical value, we refer to it as a Regression Tree. A regression tree is built by the following two steps:

- The predictor space is divided into j distinct and non-overlapping regions, $R_1, R_2, \cdots, R_j$
- For every observation that falls into region R_j, we make the same prediction, which is simply the mean of the response values for the training observations in R_j.

We divide the predictor space into high-dimensional boxes, akin to a 2-dimensional rectangle or a 3-dimensional cuboid.

The goal is to find $R = \{R_1, R_2, \cdots, R_j\}$ that minimize the **Residual Sum of Squares** given by

$$\sum_{j=1}^{J} \sum_{i \in R_j} (y_i - \hat{y}_{R_j})^2$$

where $\hat{y}_{R_j}$ is the mean response for the training observations within the j^{th} box.

However, this approach turns out to be computationally expensive, as considering every possible partition of the feature space into j boxes does not scale very well. This necessitates a **top-down, greedy** approach called **recursive binary splitting**.

1.1.2 Classification Trees

A classification tree is very similar to a regression tree, except that it is used to predict a categorical variable as opposed to a numeric one. In the case of a regression tree, it was seen that the predicted response for an observation is given by the mean response of the training observations that belong to the same terminal node. In contrast, a classification tree is used to predict the class of observations that each instance in the testing set belongs to.

In interpreting the results of a classification tree, we are interested not only in the class prediction corresponding to a particular terminal node region, but also in the class proportions among the training observations that fall into that region.

1.1.3 Advantages and Disadvantages of Decision Trees

The advantages and disadvantages of using decision trees over other standard methods are enlisted in Table 1.

Table 1: Advantages and Disadvantages of Decision Trees

Advantages	**Disadvantages**
1. **Easily understood:** The tree structure visualizes the decision-making process, interpretable by both non-technical and technical individuals.	1. **Accuracy challenge:** Decision trees exhibit lower accuracy compared to leading supervised learning methods due to high variance and low bias.
2. **Handles complex datasets:** Easily manages mixed numerical and categorical predictors without requiring dummy variables.	2. **Sensitivity to orientation:** Optimal performance relies on orthogonal decision boundaries, favoring linearly separable datasets.
3. **Systematic approach:** Mirror human decision-making process with rule-based methodology.	3. **Variance sensitivity:** Decision trees are highly sensitive to variance, with slight hyperparameter changes leading to significant model variations.

1.2 The Optimization Problem

As we saw previously, in the case of regression problems, the binary splits in the decision tree growing processes are based on **minimizing the RSS** values.

In the case of classification problems, the RSS is no longer an effective criterion to make the binary splits in the tree-growing process. A better alternative is the **classification error rate**. It is known that an instance in a particular region is going to be assigned to the most commonly occurring label/class of training instances in that region. Therefore, the **classification error rate** can be defined as the number of training observations in a region that do not belong to the most represented label/class, given by:

$$E = 1 - \max_k(\hat{p}_{mk})$$

where $\hat{p}_{mk}$ refers to the proportion of training instances in the m^{th} region that are from the k^{th} class.

However, these classification errors are not sensitive to the tree-growing process, because of which two other metrics are considered preferable, namely the Gini Index and the Entropy.

1.2.1 Gini Index

The **Gini Index** is given by:

$$G = \sum_{k=1}^{K} \hat{p}_{mk}(1 - \hat{p}_{mk})$$

It is a measure of total variance across the K classes. The value of this index will be small if all the $\hat{p}_{mk}$ values are close to zero or one, because of which the Gini Index is considered a measure of node purity.

Smaller values of the Gini Index indicate higher purity, signifying that a node contains predominantly observations from one single class.

1.2.2 Entropy

A concept that originates from contemporary information theory, entropy is calculated as:

$$D = -\sum_{k=1}^{K} \hat{p}_{mk} \log \hat{p}_{mk}$$

Considering $\hat{p}_{mk}$ takes a value between 0 and 1, it is safe to say that $0 \leq \hat{p}_{mk} \log \hat{p}_{mk}$.

Since the classifiers in this group project are tested against a binary classifier ($K = 2$). In the case of such a binary classification, the value of entropy can be further simplified as:

$$H(X) = -p \log_2 p - (1 - p) \log_2 (1 - p)$$

where p refers to $\mathrm{P}(X = 1)$.

Upon plotting H(X) against the probability of class representation, we get this curve:

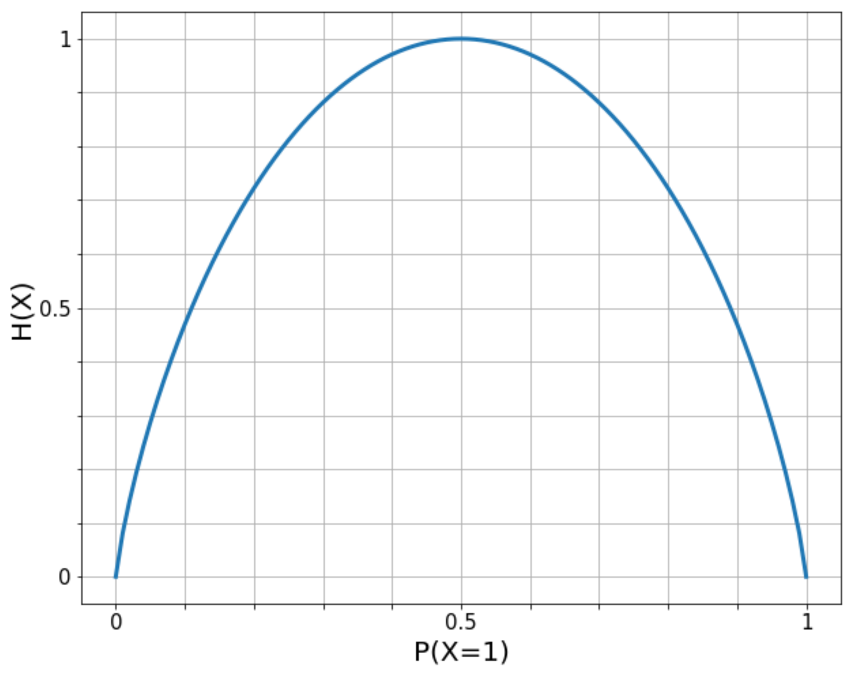

Figure 1: Entropy vs. Purity

Entropy is minimized when all samples in a given node belong to the same class. This can be seen from the fact that the value of entropy will take on a value closer to zero when all the $\hat{p}_{mk}$ values are near zero or one. That is, the purer the node, lesser the value of entropy. Conversely, entropy is maximized when there is a uniform class distribution. Surprisingly enough, this definition of entropy clearly seems to align with the literal thermodynamic interpretation of entropy.

When splitting a node, it is preferred to minimize the value of entropy so that the node is as pure as possible, and the classification is as certain as possible.

1.2.3 Information Gain

Information gain is defined as a measure of how much information is gained when a node is split at a particular value. In practice, it is the comparison of the entropy values before and after a split.

The specific formula of information gain is given by:

$$IG(D) = I(D_p) - \frac{N_{\text{left}}}{N_p} I(D_{\text{left}}) - \frac{N_{\text{right}}}{N_p} I(D_{\text{right}})$$

where $D_p, D_{\text{left}}, D_{\text{right}}$ represent the datasets from the parent, left, and right children nodes respectively, $N_p, N_{\text{left}}, N_{\text{right}}$ represent the number of observations in the parent, left and right children nodes, and $I(D)$ denotes the entropy for that particular node. At a high level, this formula simply distills down to:

$$IG_{A,B} = S_B - S_A$$

S_B is a measure of how uncertain we were with our data **before** we made the split, and S_A is a measure for how uncertain we are **after** we split the data. In terms of optimization, we seek to to split a node at a value in a way such that the information gain is maximized.

This optimization problem is a combinatorial optimization problem. The process involved in finding the best decision tree is considered to be a combinatorial problem since it involves selecting the best combination of splits from a given set of finite splits. This optimization problem is also inherently non-convex since there is no smooth function that we need to explicitly maximize or minimize. Considering the fact the number of potential trees can grow exponentially with the number of features and the depth of the tree, finding an optimal decision tree is an NP-complete problem.

Heuristic methods like greedy algorithms are used to build decision trees, recursively choosing the best split based on a criterion, namely the information gain for classification trees and MSE for regression trees. Each split is chosen to best reduce the impurity of the resulting child nodes, which can be seen as a local optimization problem; however, this greedy approach wouldn't lead to a globally optimal tree.

To summarize, in the case of a regression tree, we seek to find the split that minimizes the mean squared error within each node, whereas in the case of classification problems, we seek to choose a split that results in the maximum information gain (or maximum reduction in entropy) to reduce uncertainty in the class labels.

1.3 Random Forests

Random Forest [2] [3] [4] is an *ensemble learning* algorithm (one that combines multiple learning algorithms to obtain a better model) that combines the output of many several decision trees (multiple trees = forest!) to get a single result, typically using the process of **bagging** and **bootstrapping**. In order to better understand the concept of Random Forest Models and how they work, it is imperative to understand the concepts of **Bootstrapping** and **Bagging**, which are discussed in the appendix.

In summary, **Random Forest** models effectively harness the concepts of bagging and bootstrapping creating a bunch of decision trees with an element of randomness baked into them. This randomness is critical to ensure that constituent individual trees have low correlations with each other, leading to reduced bias. In addition, the creation of multiple decision trees reduces the risk of overfitting caused by incorporating too much noise from the training dataset. The mechanics of building a Random Forest are summarized in Figure 2.

[1]Source: KDnuggets

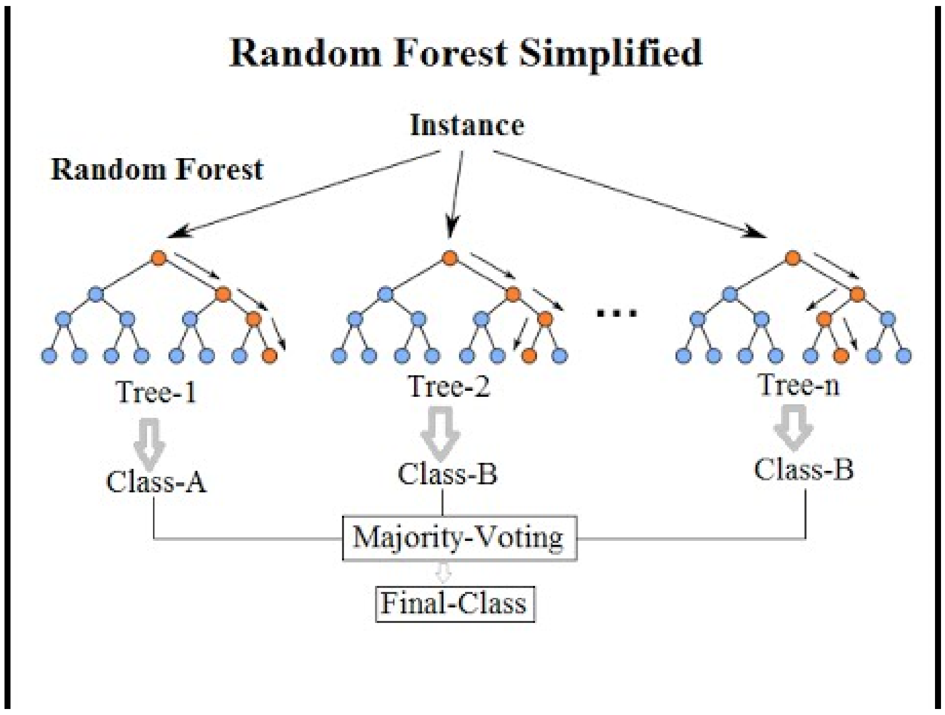

Figure 2: Random Forests, simplified.[1]

1.3.1 Advantages and Disadvantages of Random Forests

The advantages and disadvantages of using random forests are enlisted in Table 2.

Table 2: Advantages and Disadvantages of Random Forests

Advantages	**Disadvantages**
1. Performs well for both regression and classification.	1. Accuracy outperforms decision trees but lags behind boosted ensemble methods.
2. Handles missing values effectively without compromising accuracy.	2. Slower than boosted tree ensembles with an increase in the number of trees.
3. Can handle very large datasets with thousands of features and identify important features.	3. Interpreting the results of a Random Forest model can be challenging.

1.4 Gradient Boosting

Gradient Boosting [5], an ensemble learning technique, is deeply rooted in the tree-based methods family, sharing its foundation with decision trees. Widely embraced for tasks spanning regression and classification domains, Gradient Boosting stands out for its superior predictive performance. Gradient Boosting enhances decision trees through an iterative optimization process. The approach involves sequentially fitting *weak learners*, often shallow decision trees, to the residuals of the ensemble's preceding models. This iterative refinement systematically minimizes errors, contributing to enhanced predictive accuracy.

1.4.1 Fitting Subsequent Trees on Residuals

Let $\mathcal{D}$ represent the dataset, $\mathbf{X}$ the feature matrix, $\mathbf{y}$ the true labels. Given the current model's prediction $\hat{\mathbf{y}}$, the residuals (r) are calculated as:

$$r = \mathbf{y} - \hat{\mathbf{y}}$$

This process can be interpreted through the lens of gradient descent, where the residuals represent the negative gradient of the loss function with respect to the current model's prediction. The subsequent tree is then fitted to approximate this negative gradient, minimizing the loss function.

1.4.2 Calculation of Residuals using MSE

The Mean Squared Error (MSE) is utilized to quantify the residuals' discrepancy between the true labels and the current prediction:

$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

Where N is the number of data points, y_i is the true label, and $\hat{y}_i$ is the current model's prediction for the i-th data point.

To control the contribution of each tree to the overall model, a learning rate (η) is introduced. The update to the current model's prediction is then performed as:

$$\hat{\mathbf{y}} \leftarrow \hat{\mathbf{y}} + \eta \times \text{Tree}(\mathbf{X})$$

Here, $\text{Tree}(\mathbf{X})$ represents the prediction of the fitted tree. The learning rate scales the contribution of each tree, influencing the reduction of residuals and subsequent model refinement.

This iterative process of fitting subsequent trees on residuals, with the incorporation of the learning rate, provides a mechanism for gradual improvement in predictions, contributing to the effectiveness of the gradient boosting algorithm.

1.4.3 Regularization and Early Stopping

Regularization techniques, such as controlling tree depth, minimum samples per leaf, or maximum features considered during tree fitting, play a role in preventing overfitting. Early stopping criteria are employed to halt the boosting process when further iterations do not significantly improve model performance, contributing to enhanced generalization.

1.4.4 Advantages and Disadvantages of Gradient Boosting

The advantages and disadvantages of using gradient boosting are enlisted in Table 3.

Table 3: Advantages and Disadvantages of Gradient Boosting

Advantages	Disadvantages
1. Excels in predictive accuracy, making it a robust choice for both regression and classification problems.	1. Its performance may be overshadowed by other boosted ensemble learning methods, such as XGBoost or LightGBM.
2. Adept at handling missing values within the dataset without compromising accuracy, showcasing resilience even when dealing with substantial data gaps	2. The computational cost of Gradient Boosting increases with the number of trees generated, resulting in slower model training compared to some boosted tree ensembles.
3. Gradient Boosting models demonstrate scalability, proving effective in managing large datasets with numerous features. Moreover, they provide insights into the importance of different features within the training dataset.	3. Gradient boosting models are sensitive to their hyperparameters, and selecting the right combination of hyperparameters can be challenging.

2 Implementation

The subsequent sections are involved in describing the implementation process for the **Random Forest** and **Gradient-Boosting** classifiers, as well as the interpretation of their performance in comparison to the available black-box implementations via **scikit-learn**.

2.1 A Note on Regularization Parameters

The following subsections discussing the implementation of the classifiers explain what parameters are associated with them. This is very important to understand because Decision Trees make very few assumptions about the training dataset, as compared to linear models that assume the underlying data is linear. This implies that decision trees are non-parametric models, since the number of parameters is not determined prior to training.

This lack of constraining the model implies that the tree structure will adapt itself to the training data, most likely overfitting it. In contrast, a parametric model such as a linear model has a pre-determined number of parameters meaning the risk of overfitting is reduced due to the limited degrees of freedom.

Decision Trees can be prevented from overfitting the training data by imposing a set of limitations upon them, called regularization. The parameters used to influence this regularization process are called **regularization parameters**, and are a part of every classifier implemented as part of this project.

2.2 Decision Tree Implementation

The implementation of this classifier can be found in the *DecisionTreeClassifier.py* file.

The implementation of the custom implementation of the decision tree classifier was implemented in the

Object-Oriented (OO) paradigm using **Python**. The technical specifics of this classifier implementation can be found in subsection B.1

2.3 Random Forest Classification Implementation

The implementation of this classifier can be found in the *RandomForestClassifier.py* file.

Similar to the implementation of the Decision Tree Classifier, the Random Forest Classifier was also implemented using the OO paradigm in Python. The class builds a specified number of decision trees, each of which is trained on a bootstrapped sample of the input data. **For predictions, it aggregates the predictions of all trees using majority voting**. In addition, it also calculates the Out-Of-Bag score as an additional metric to estimate model performance.

The Random Forest Classifier has the same hyperparameters as the Decision Tree Classifier, considering they both involve controlling how much a tree can grow. In addition, it has a few more hyperparameters seen to control the ensemble of trees itself. The technical specifics of this classifier implementation can be found in subsection B.2

2.4 Gradient Boosting Classification Implementation

The implementation of this classifier can be found in the *GradientBoostingClassifier.py* file.

Developed under the Object-Oriented (OO) paradigm in Python, this classifier is designed for binary classification tasks, leveraging an ensemble of boosted decision trees. The technical specifics of this classifier implementation can be found in subsection B.3

3 Results

3.1 Overall Summary

The overall results from testing the custom and blackbox models for the three classifiers are summarized in Table 4:

Table 4: Overall Testing Results

Model	**Winner**
Decision Tree Classifier	Custom Implementation
Random Forest Classifier	Scikit-Learn Blackbox
Gradient Boosting Classifier	Scikit-Learn Blackbox

The code for the tests conducted below are found in the *scratch.ipynb* file attached along with this submission.

Overall, the tests of the custom classifiers seem to echo the theoretical understanding that the Gradient Boosting method performs the best, followed by Random Forest Models and

simple Decision Trees. The specifics of the testing process as well as the performance of each classifier are discussed in the subsequent subsections:

3.2 Testing Approach

The custom implementations of the decision tree, random forest and gradient boosting classifiers were trained and tested on a dataset generated using the `scikit-learn` package's `make_classification` dataset. The specifics of the generated dataset are summarized in Table 5:

Parameter	**Value**
Number of Samples	1000
Number of Features	30
Number of Classes	2
Number of Clusters per Class	1
`n_informative`	15

Table 5: Dataset Parameters

The dataset was split into testing and training datasets in an 80-20 split, leading to 800 training samples and 200 testing samples.

3.2.1 Decision Tree Classifier

The custom model and the scikit-learn model were both initialized with the same hyperparameters described below:

Parameter	**Value**
Maximum Depth	5
Maximum Features	7
Minimum Samples to Split	15
Minimum Impurity Decrease	0.001

Table 6: Decision Tree Model Parameters

The results from training both of the models on the dataset are summarized in Table 7:

	Custom Implementation	**Scikit-Learn Blackbox Implementation**
Accuracy	0.86	0.85
Precision	0.86	0.85
Recall	0.86	0.85
F1 Score	0.86	0.85

Table 7: DT Models - Performance

From the above summary, it can be seen that the **custom model** performed slightly better than the **scikit-learn blackbox** implementation.

The custom implementation of the decision tree can be seen in Figure 3.

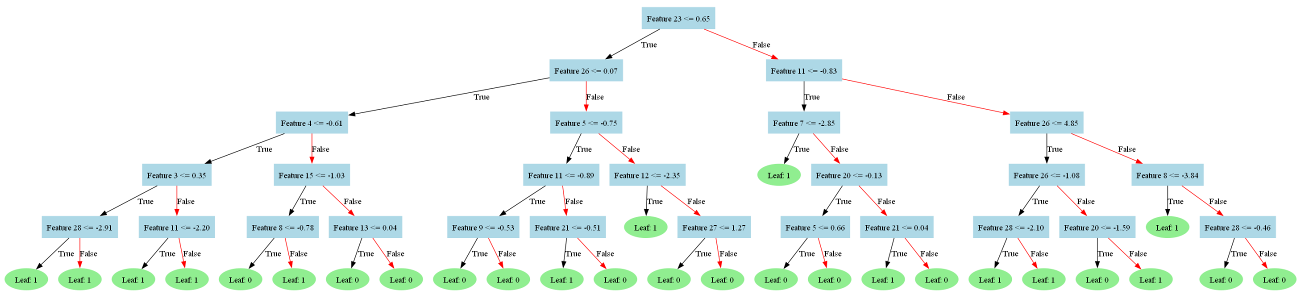

Figure 3: Custom Decision Tree

3.2.2 Random Forest Classifier

The custom model and the scikit-learn model were both initialized with the same hyperparameters described below:

Parameter	Value
Number of Estimators	100
Maximum Depth	5
Maximum Features	7
Minimum Samples to Split	15
Minimum Impurity Decrease	0.001

Table 8: RF Model Parameters

The results from training both of the models on the dataset are summarized in Table 9:

	Custom Implementation	Scikit-Learn Blackbox Implementation
Accuracy	0.92	0.95
Precision	0.92	0.95
Recall	0.92	0.94
F1 Score	0.91	0.95
OOB Score	0.93	0.95

Table 9: RF Models - Performance

As mentioned before, around one-third of the observations are left out when fitting a bagged tree. These observations are called **Out-Of-Bag** observations (**OOB** for short). The **OOB Score** in the above table refers to the percentage of correct predictions made on OOB samples taken for validation.

Both the custom implementation as well as the blackbox implementation have high OOB scores, meaning both of the models performed well on predicting the class associated with out-of-bag samples. However, the **scikit-learn blackbox** model came on top with an OOB Score of 95% compared to the 93% of the custom implementation.

A similar trend is seen across the other metrics as well, with the **scikit-learn blackbox** model having a slight edge over the **custom model**. However, it should be noted that the custom model has performed extremely well.

3.2.3 Gradient Boosting Classifier

Parameter	Value
Number of Estimators	100
Learning Rate	0.3
Maximum Depth	5
Maximum Features	7
Minimum Samples to Split	15
Minimum Impurity Decrease	0.001

Table 10: GB Model Parameters

The results from training both of the models on the dataset are summarized in Table 11:

	Custom Implementation	Scikit-Learn Blackbox Implementation
Accuracy	0.92	0.96
Precision	0.92	0.96
Recall	0.92	0.96
F1 Score	0.92	0.96

Table 11: GB Models - Performance

The scikit-learn blackbox implementation of the Gradient Boosting Classifier consistently outperformed the custom model across multiple evaluation criteria, including accuracy, precision, recall, and F1 score. These findings underscore the effectiveness and robustness of the scikit-learn Gradient Boosting model in capturing complex relationships within the dataset.

4 Challenges, Learnings & Future Scope

4.1 Challenges

Developing custom tree-based algorithms presented two distinct hurdles compared to models covered in this course.

Firstly, tree-based models involve a different optimization problem at their core, seeking optimal splits at each node based on purity or information gain. This contrasts with class-covered models, which optimize coefficients once to minimize a predefined loss function or maximize likelihood using techniques like gradient descent. Understanding concepts such as entropy, information gain, bagging, boosting, and ensemble learning required extensive self-study and research.

Secondly, translating theoretical knowledge into functional code demanded overcoming implementation complexities, including designing efficient data structures, handling categorical variables, and optimizing computational performance. Tuning hyperparameters and optimizing model performance without predefined guidelines added further challenges, necessitating iterative experimentation and meticulous testing.

Despite these hurdles, proactive learning, collaborative problem-solving, and iterative refinement enabled us to overcome challenges and achieve successful outcomes.

4.2 Key Learnings

Key learnings from the project include understanding the optimization challenges inherent in decision tree construction, recognizing the trade-offs between tree-based methods such as Random Forests and Gradient Boosting, and appreciating the importance of hyperparameters in fine-tuning model performance and avoiding overfitting.

4.3 Future Scope

Looking ahead, we aim to augment our optimization problem by incorporating advanced techniques such as model stacking. Additionally, we plan to explore the mathematics of state-of-the-art tree-based models such as LightGBM and XGBoost. These advancements will provide opportunities to navigate complex data landscapes with greater ease and accuracy.

5 References

[1] Ander Fernandez. *How to Code a Decision Tree in Python from Scratch*. 2021. URL: https://anderfernandez.com/en/blog/code-decision-tree-python-from-scratch/ (visited on 02/25/2024).

[2] Jason Brownlee. *How to Implement Random Forest From Scratch in Python*. https://machinelearningmaste com/implement-random-forest-scratch-python/. 2018. (Visited on 02/25/2024).

[3] Carbonati. *Random Forests from Scratch*. https://carbonati.github.io/posts/random-forests-from-scratch/. 2016. (Visited on 02/25/2024).

[4] Enozeren. *Building a Random Forest Model From Scratch*. https://medium.com/@enozeren/building-a-random-forest-model-from-scratch-81583cbaa7a9. 2023. (Visited on 02/25/2024).

[5] Zepu Zhang. *Understanding Gradient Boosting Tree for Binary Classification*. 2018. URL: https://zpz.github.io/blog/gradient-boosting-tree-for-binary-classification/ (visited on 02/23/2024).

[6] Bhiksha Raj. *Decision Trees*. 2016. URL: https://www.cs.cmu.edu/~bhiksha/courses/10-601/decisiontrees/ (visited on 02/25/2024).

[7] Matthew N. Bernstein. *Random Forests*. 2017. URL: https://pages.cs.wisc.edu/~matthewb/pages/notes/pdf/ensembles/RandomForests.pdf (visited on 02/25/2024).

6 Deliverables and Code

The submission contains the following files:

- `Project-Report.pdf`: This PDF report.
- `DecisionTreeClassifier.py`: A python script that implements the Decision Tree Classifier.
- `RandomForestClassifier.py`: A python script that implements the Random Forest Classifier.
- `GradientBoostingClassifier.py`: A python script that implements the Gradient Boosting Classifier.
- `scratch.ipynb`: A Jupyter notebook that contains the code to run the custom implementations, generate visualizations and compare the results with the `scikit-learn` implementations.
- `Node.py`: Overarching class that represents a node in a decision tree.
- `utils.py`: Helper script to compute metrics and generate visualizations whenever necessary.

Appendices

A Random Forest Models

A.1 Bootstrapping

Random Forest Models are made powerful as a result of the randomness injected into each tree by the boostrapping process. Each individual decision tree is constructed on a **bootstrapped** subset of the available data: that is, if we have a dataset containing n observations, **bootstrapping is the process of sampling n instances with replacement**.

Further, if we have a dataset containing p features, bootstrapping can also be used to randomly select a subset of the available features to be used in the tree-building process, which allows the random forest model to be less susceptible to overfitting.

In practice, this means that some instances in the available data will be selected more than once whereas some instances won't be selected at all. In fact, the probability of an instance being dropped entirely due to bootstrapping is given by $(1 - \frac{1}{n})^n$, the value of which can be proven to be equal to approximately $\frac{1}{3}$. **This implies that a third of the instances in the original data will be left out in each tree when bootstrapping n samples with replacement.**

One of the biggest positives resulting from the bootstrapping process is that considering the fact that each tree only uses two-thirds of the available data, trees generated in the random forest models usually differ significantly from each other.

The bootstrapping process gives us a good metric to judge the performance of a Random Forest model, namely the **Out of Bag Error** (more commonly shortened as OOB Error). As mentioned before, about a third of the instances in the original data are not selected for the tree-building process. After building a tree with the n bootstrapped observations, one can test the model against each of the instances in the third of observations that were left out, and calculate the average prediction error from those samples. The overall out of bag error for the model can subsequently be calculated by averaging the OOB score for each tree generated by the random forest model.

A.2 Bagging

Bagging is the process of growing a tree where each node in the tree looks at every value in the boostrapped sample in every feature to find the best split in the data at that particular node. This process is then repeated for every single tree generated in the random forest model.

Random Forests incorporate bagging with a subtle difference: While bagging considers every feature to find the best split in the data at a given node, each tree in a Random Forest model only looks at a subset of the available features. As opposed to using the same set of features in every single tree, choosing a subset of the available features helps inject more randomness into the model thereby making it less susceptible to overfitting whilst experimenting with interactions between varying sets of features.

B Implementation Detail

B.1 Decision Tree Implementation

The implementation of this classifier can be found in the *DecisionTreeClassifier.py* file attached along with this submission.

The implementation of the custom implementation of the decision tree classifier was implemented in the **Object-Oriented** (OO) paradigm using **Python**.

A Decision Tree Classifier object created using the implementation has the following parameters:

- `max_depth`: This refers to the maximum depth of the tree. If set to None, the tree will grow until all leaves are pure, or until it reaches the minimum samples split.
- `min_samples_split`: The minimum number of samples required to split an internal node or branch.
- `max_features`: The number of features to be considered when looking for the optimal split. If None, all features are considered.
- `min_impurity_decrease`: A node will be split if this split induces a decline in impurity that is greater than or equal to this value.
- `random_state`: Controls the randomness of the bootstrapping in the samples used when the trees are built.
- `debug`: Flag to set the logging information specificity.

The **attribute(s)** associated with the object include:

- `root` (Node): Root node of the decision tree post-fitting.

The **methods** applicable to the object include:

- `fit(X,y)`: Fits the decision tree model to the given dataset.
- `predict (X)`: Predicts the class labels for the given dataset.
- `visualize_tree(feature_names, class_names)`: Generates an image of the decision tree using the *graphviz* library.

Technical specification of the classifier's parameters are summarized in Table 12.

Parameter	Data Type	Optional?	Default Value
max_depth	int	yes	None
min_samples	int	yes	2
max_features	int	yes	None
min_impurity_decrease	float	no	0
random_state	int	yes	42
debug	bool	yes	FALSE

Table 12: DT - Parameter Settings

B.1.1 Pseudocode

In general, decision trees are synthesized using the training data available. The decision tree algorithm recursively builds the tree [6] by:

- Step 1: All available observations in the training dataset are assigned to the root node of the tree. The current node is then set to the root node.
- Step 2: For every single feature available, training data instances are separated by their values for the associated feature. The information gain is then calculated for this separation or partition.
- Step 3: The current node is then set to be the feature which resulted in the greatest information gain in the previous step. When the optimal information gain is equal to 0, that node can be considered to be a leaf node.
- Step 4: Repeat the separation process according to the value of the best feature, just as in step two of this algorithm described above.
- Step 5: Mark every layer of separation to be a child node to the current node.
- Step 6: If a child node contains only instances from a singular class, then mark it as a leaf, and return. If not, set the current node to be this child note and repeat from step two recursively until all options are exhausted.

A generalized flowchart of such a decision tree classifier is shown in Figure 4:

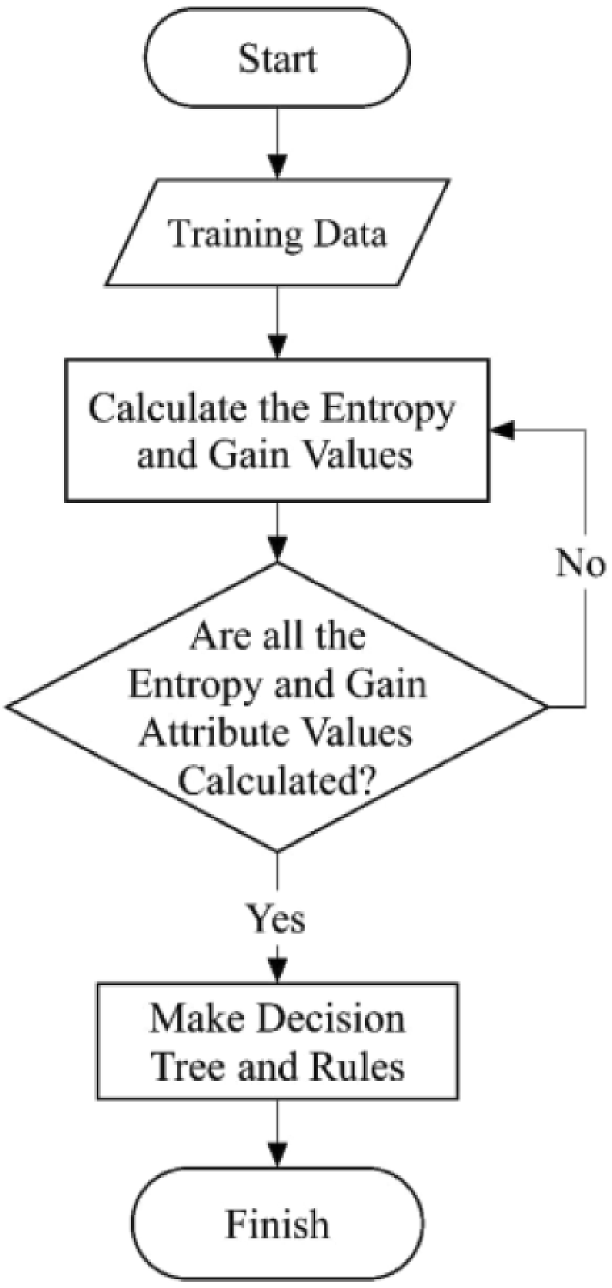
Start
Training Data
Calculate the Entropy and Gain Values
Are all the Entropy and Gain Attribute Values Calculated?
No
Yes
Make Decision Tree and Rules
Finish

Figure 4: Training a Decision Tree

B.2 Random Forest Classification Implementation

The implementation of this classifier can be found in the *RandomForestClassifier.py* file attached along with this submission.

Similar to the implementation of the Decision Tree Classifier, the Random Forest Classifier was also implemented using the OO paradigm in Python. The class builds a specified number of decision trees, each of which is trained on a bootstrapped sample of the input data. **For predictions, it aggregates the predictions of all trees using majority voting**. In addition, it also calculates the Out-Of-Bag score as an additional metric to estimate model performance.

The Random Forest Classifier has the same hyperparameters as the Decision Tree Classifier considering they both involve controlling how much a tree can grow. In addition, it also has the hyperparameters seen in a typical bagging classifier to control the ensemble of trees in itself:

- `n_estimators`: The number of trees in the forest.

- `max_depth`: This refers to the maximum depth of the tree. If set to None, the tree will grow until all leaves are pure, or until it reaches the minimum samples split.
- `min_samples_split`: The minimum number of samples required to split an internal node or branch.
- `max_features`: The number of features to be considered when looking for the optimal split. If None, all features are considered.
- `min_impurity_decrease`: A node will be split if this split induces a decline in impurity that is greater than or equal to this value.
- `random_state`: Controls the randomness of the bootstrapping in the samples used when the trees are built.
- `debug`: Flag to set the logging information specificity.

The **attribute(s)** associated with the object include:

- `oob_score`: A floating point value, referring to the model's performance on unseen data.
- `trees` : A list of DecisionTreeClassifier objects to represent the list of fitted decision tees within the forest.

The **methods** applicable to the object include:

- `fit(X,y)`: Fits the Random Forest Classifier to the given dataset.
- `predict (X)`: Predicts the class labels for the given dataset.
- `_bootstrap_sample(X, y)`: Generates a bootstrapped sample from the input dataset and identifies OOB indices.
- `_calculate_oob_score(oob_votes, y)`: Calculates the OOB score based on the aggregated OOB predictions.

Technical specifics of the classifier's parameters are summarized in Table 13.

Parameter	**Data Type**	**Optional?**	**Default Value**
n_estimators	int	yes	100
max_depth	int	yes	None
min_samples	int	yes	2
max_features	int	yes	None
min_impurity_decrease	float	no	0
random_state	int	yes	42
debug	bool	yes	FALSE
n_jobs	int	yes	-1
verbose	int	yes	10

Table 13: RF - Parameter Settings

B.2.1 Pseudocode

As discussed before, the Random Forest model is an ensemble learning method that combines the output of many decision trees to get a single result, using the processes of bagging and bootstrapping. In general, Random Forests are synthesized using multiple decision trees constructed from randomly selected subsets of the training data. The Random Forest algorithm builds these trees and aggregates their predictions by [7]:

- Step 1: For each tree in the Random Forest:
  - Step 1.1: Randomly select a subset of the training data with replacement (bootstrapping).
  - Step 1.2: Build a decision tree using the selected subset:
    - Step 1.2.1: Assign all observations in the subset to the root node of the tree.
    - Step 1.2.2: For a randomly selected subset of features, calculate the information gain for each feature.
    - Step 1.2.3: Choose the feature with the highest information gain as the current node.
    - Step 1.2.4: Repeat the separation process according to the value of the best feature.
    - Step 1.2.5: Mark each layer of separation as a child node to the current node.
    - Step 1.2.6: If a child node contains only instances from a single class, mark it as a leaf. Otherwise, set the current node to this child node and repeat from Step 1.2.2.
- Step 2: For each new observation, make a prediction by aggregating the predictions from all the trees in the Random Forest (e.g., majority vote for classification or averaging for regression).

This is summarized visually in Figure 5:

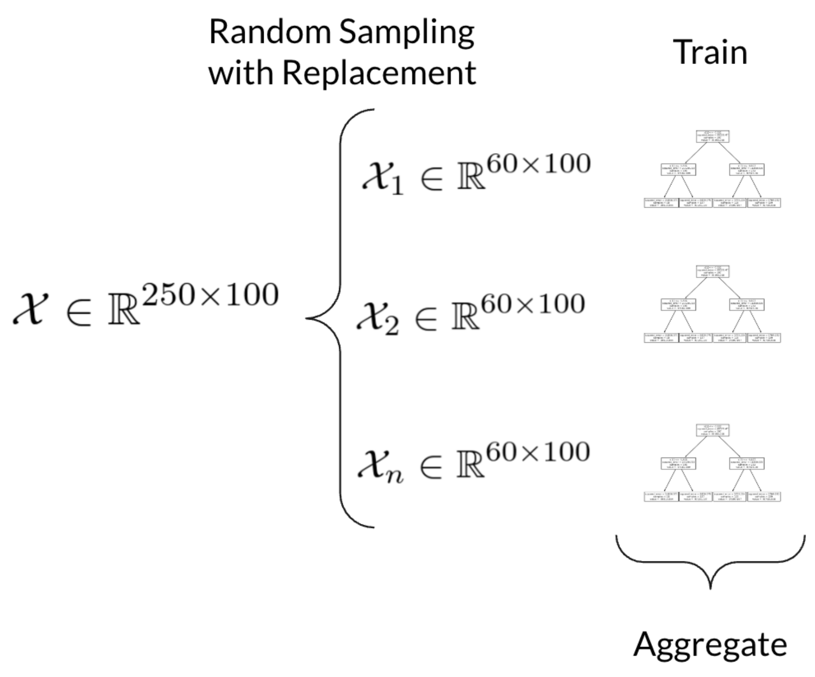
Random Sampling with Replacement
Train
$\mathcal{X} \in \mathbb{R}^{250\times 100}$
$\mathcal{X}_1 \in \mathbb{R}^{60\times 100}$
$\mathcal{X}_2 \in \mathbb{R}^{60\times 100}$
$\mathcal{X}_n \in \mathbb{R}^{60\times 100}$
Aggregate

Figure 5: Illustration of training a Random Forest Model.[2]

B.3 Gradient Boosting Classification Implementation

The implementation of this classifier can be found in the *GradientBoostingClassifier.py* file attached along with this submission.

Adhering to the Object-Oriented (OO) paradigm in Python, this classifier is constructed to address binary classification tasks through an ensemble of decision trees.

The class GradientBoostingClassifier is equipped with the following parameters:

- `n_estimators`: The number of trees to build. Default is 100.
- `learning_rate`: The learning rate controls the contribution of each tree. Default is 0.1.
- `max_depth` (optional): The maximum depth of each tree. If None, the trees will grow until all leaves are pure.
- `min_samples_split`: The minimum number of samples required to split an internal node. Default is 2.
- `max_features` (optional): The number of features to consider when looking for the best split. If None, all features are considered.
- `min_impurity_decrease`: A node will be split if this split induces a decrease of the impurity greater than or equal to this value. Default is 0.0.
- `random_state` (optional): Controls the randomness of the bootstrapping of the samples used when building trees. Default is None.
- `debug`: If True, the logging level will be set to DEBUG, providing more detailed logging information. Default is False.

The **attribute(s)** associated with the object include:

- `trees` (List[DecisionTreeClassifier]): The list of fitted trees.
- `f0` (float): The initial prediction of the model.

The **methods** applicable to the object include:

- `fit(X, y)`: Fits the gradient boosting model to the given dataset.
- `predict(X)`: Predicts the class labels for the given dataset.

Additionally, the `sigmoid`, `log_loss`, and gradient methods handle the mathematical computations necessary for the model's training process.

[2]Source: Wikipedia

Technical specifics of the classifier's parameters are summarized in Table 14.

Parameter	**Data Type**	**Optional?**	**Default Value**
n_estimators	int	yes	100
learning_rate	float	no	0.1
max_depth	int	yes	None
min_samples_split	int	no	2
max_features	int	yes	None
min_impurity_decrease	float	no	0.0
random_state	int	yes	None
debug	bool	yes	False

Table 14: GB Parameter Settings