Statistical Machine Learning using Tree-Based Methods

Pattanin (Mill) Luangamornlert

April 26, 2021

Abstract

This piece of work is a result of my own work except where it forms an assessment based on group project work. In the case of a group project, the work has been prepared in collaboration with other members of the group. Material from the work of others not involved in the project has been acknowledged and quotations and paraphrases suitably indicated.

This project aims to look at methods of Machine Learning, which has been described as the 4th Industrial Revolution (4IR) by the World Economic Forum at Davos in 2016 [Schwab, 2016].

We will look at the mathematics behind Decision Trees, famously described in Leo Breiman's landmark book Classification and Regression Trees [Breiman et al., 1984] which is one of the most cited sources for Statistical Machine Learning, with over 48,000 citations.

Breiman, Friedman, Olshen, and Stone noted that CART had its flaws in the book itself and have always strived to improve the method, Breiman introducing Random Forests a few years later, and Friedman created Gradient Boosted Machines in 1999. Since then, many mathematicians have tried to improve upon the work and this project will try to explore these methods.

Key Words: Classification Decision Tree, Regression Decision Tree, Cost Function, Evaluation Metrics, Ensemble, Optimization, Interpretability

Contents

List of Algorithms

List of Figures

Chapter 1

An Introduction to Machine Learning

Machine Learning is a subset of Artificial Intelligence (AI) where we create a mathematical model which can be "trained" using previously known data to create a more robust AI programme which can be deployed in the real world. We achieve this by using Statistics to create models which we then feed into the machine for it to predict outcomes based on prior events.

1.1 Notation

Before we proceed, it is necessary to state the notation that will be used in this paper.

- n the number of values at each point
- i an observation in the data
- M the number of groups we form after splitting/partitioning
- $\{x_i, y_i\}_{i=1}^{n}$ the full dataset
- $x_i \in X$ an input vector
- $y_i \in Y$ the dependent factor
- $\hat{y}_i$ the predicted value y_i based on a relation with X

1.2 Supervised and Unsupervised Learning

In statistics, there are two objectives which become possible when you look at a data set. Given observations X :

- **Supervised Learning** where we observe how X affects Y, the output, as we change the variable X
- **Unsupervised Learning** we investigate the characteristics of X and how it behaves relative to the rest of the data

The methods we will explore will be a Supervised Learning method.
There are two main ways to deal with data under Supervised Learning:

- **Classification** - classification is suited when you want to group data together to make predictions which are factoral. These methods are listed in Table 1.1

Logistic Regression
Support Vector Machines
Linear Discriminant Analysis
Classification Decision Tree
Random Forests
AdaBoost
Neural Networks

Table 1.1: Classification Methods

- **Regression** - Regression is the main form of statistical analysis for numerical forms of data as they can be used to build a trend line to estimate an output Y from X based on previously known information. These methods are listed in Table 1.2

Linear Regression
Non-Linear Regression
LASSO
Regression Decision Trees
Random Forests
Gradient Boosting

Table 1.2: Regression Methods

For more information, please refer to **An Introduction to Statistical Learning** [James et al., 2013], or **Elements of Statistical Learning** [Hastie et al., 2009].

1.3 What are Decision Trees?

Decision Trees are essentially a flow chart that is shaped like a tree which is used to make predictions. They are a form of non-parametric statistics and a common and easy to interpret alternative for when data is non-linear. The aim is to input multiple variables (often historical data) into the model, and under supervised learning, we are able to predict the outcome of a future test using the same variables as in the training data.

Each decision point is called a node, where at each node, we split the set of data into root nodes - called “Child” nodes. This is done recursively until a stopping point has been achieved. Decision Trees are used widely in industry, such as in medicine where a tree is often used to predict if a patient has cancer or not.

Decision Trees provide many advantages over other methods:

- They are very easy to understand
- They do not require lots of effort with regards to data processing
- Data can be left as is - no normalization or scaling is needed
- Missing data can be accommodated as they do not affect the learning of the model
- They can be used in both Classification and Regression settings and are sometimes referred individually as Classification or Regression Trees

1.3.1 Brief History of Decision Trees

The idea of Decision Trees first came out of Operations Research as an optimization problem. This was first developed in 1963 [Morgan and Sonquist, 1963] and the method is called Automatic Interaction Detection (AID). AID splits the model into subsets and does that recursively until a stopping criteria is met. However, AID is a regression tree algorithm and so development for classification naturally followed.

This ended up being developed into Theta Automatic Interaction Detection (THAID) [Messenger and Mandell, 1972]. THAID splits data using classification methods but had its flaws and was superseded by Chi-square Automatic Interaction Detection (CHAID) [Kass, 1980]. In this case we split data into intervals and merge the groups which are not as significant together. From here, Bonferroni corrections are used.

Unfortunately, this method was still not optimal and thus a better approach was needed. Thus, the idea of metrics were introduced as a splitting criterion [Loh, 2014]. These methods will be covered in detail in Chapter 3.

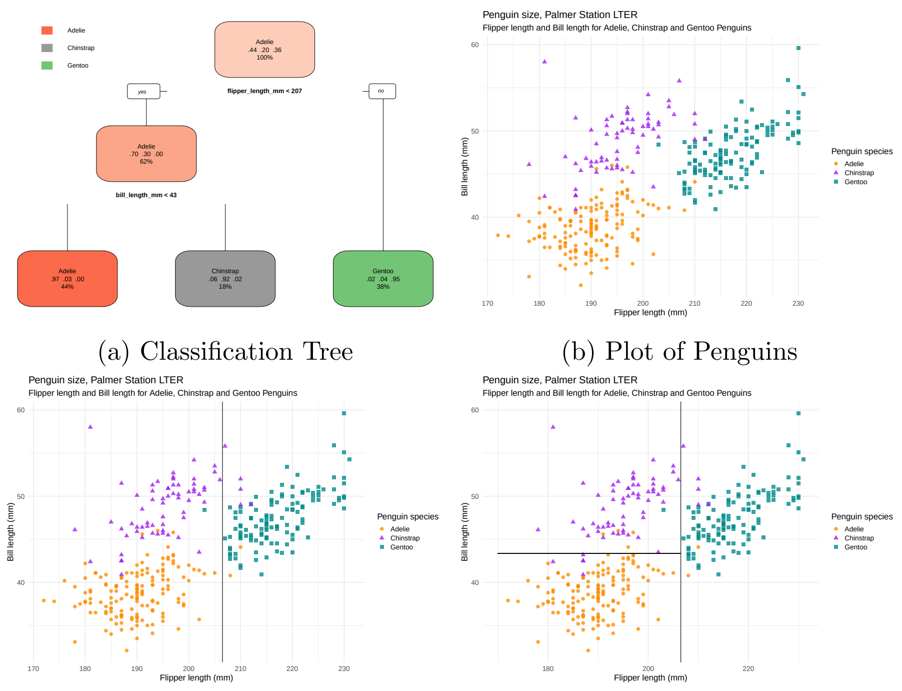
1.3.2 The Classification and Regression Tree method - and where do we go on from it?

Multiple algorithms can be used to create a Decision Tree. The most famous of these, Classification and Regression Trees (CART) [Breiman et al., 1984] was the first developed by Leo Breiman, Jerome Friedman and others in the 1984 book, Classification and Regression Trees. An alternative, Iterative Dichotomiser 3 (ID3) by Ross Quinlan [Quinlan, 1986], was introduced in 1985 and uses similar ideas but with minor differences which will be discussed later. It is another popular method and indeed, some may prefer it to CART while others may find its limitations too restrictive.

Firstly, we will look at how the CART algorithm actually works and show some examples for both classification and regression problems. From here we are able to look at the multitude of methods which have been used to improve upon the CART method, such as Random Forest, Gradient Boosting and Optimal Decision Trees. These methods will be covered in Chapter 3.

A Basic Example of CART in action

Here we have a graphical example of how CART is used on the Penguins Dataset [Horst et al., 2020] to classify penguin species: We see in Fig 1.1 how we can classify



(a) Classification Tree (b) Plot of Penguins

(c) x = 206.5 (d) On left split, y = 43.35

Figure 1.1: An Example of CART using Penguins

data on Penguins based on Bill Length and Flipper Length. In this case, we first look at 1.1a to see how the Decision Tree splits the data. We then are able to easily interpret it by drawing lines on 1.1b to form decision boundaries. The first split at the top is shown in 1.1c which divides the group into two segments. This split shows that Gentoo penguins are mainly grouped to the right of this split while the other two are to the left. Thus we can now classify the right hand side as Gentoo Penguins. The split on the left is still equally spread out between Adelie and Chinstrap Penguins so this group is further split up into two, shown in 1.1d. Now, we see two subgroups which each clearly can classify which species a penguin is, between Adelie and Chinstrap. This is how we end up with three groups which each classify what species of penguins by seeing what is the majority in the group.

1.4 Programming Language

For this project, we will be mainly using R as the main programming language although occasionally we might encounter situations where other languages such as Python or Julia turn out to be more convenient and easier to use. This is entirely dependant on the person and their personal preferences.

In many cases, it is possible to use at least 2 different languages for the entire process due to the convenience that each bring for the step needed. Often this occurs when we use one language for data cleansing and another for the analysis and sometimes a third for producing the plots. There is no set order to this and you may use any language you wish. Note that you will have to install external packages which do not come pre-installed with the programme. In such cases, you will have to install the package before running the code.

R

In `R` [R Core Team, 2020], this will be through the Comprehensive R Archive Network (CRAN) which maintains all packages that will be required for this project. When you need to install a new package, run the following line *once*, the first time:

```
install.packages("package")
```

To work with data sets we will be using the `dplyr` [Wickham et al., 2020] package which is part of the Tidyverse Library and is used for data manipulation. We will also be using `ggplot2` [Wickham, 2016] to produce plots as this package produces better plots than the regular plot function.

1.5 Dataset

We will be using some data from the sport of baseball to demonstrate how the Tree-Based methods we are looking at work. The data we are using is from a library called the Lahman Dataset. The Lahman Dataset is the largest publicly available dataset on baseball. It is part of the Baseball Archive set up by Sean Lahman in 1995 and the Baseball Dataset itself first appeared in 1996 and has been updated annually and maintained by a team of support staff [Friendly et al., 2020] [1]. There is a .csv version, an SQL version, and R version (which is available through the CRAN). We will be using the R version to simplify the necessary data preparation.

[1]Sean Lahman, Chris Dalzell, Michael Friendly, Dennis Murphy, Martin Monkman, Vanessa Foot, Justeena Zaki-Azat

Description

The following is the description of the data which has been sourced from the Lahman reference manual.[Friendly et al., 2020]
"*This database contains pitching, hitting, and fielding statistics for Major League Baseball from 1871 through 2019. It includes data from the two current leagues (American and National), the four other "major" leagues (American Association, Union Association, Players League, and Federal League), and the National Association of 1871-1875. This database was created by Sean Lahman, who pioneered the effort to make baseball statistics freely available to the general public. What started as a one man effort in 1994 has grown tremendously, and now a team of researchers have collected their efforts to make this the largest and most accurate source for baseball statistics available anywhere.*"[2]

How to use this Data

In this project, we will use different parts of the overall dataset to create Tree-Based Models. One of the models will be a classification model and another will be a regression model. This will be covered in more detail in Chapter 6.

1.6 Testing Model Performance

When building models, we have to give prior information to teach the model what it needs to know. What happens is that we will split the data into a learning class and a prediction class. In the learning class, we will further split the data into *training* and *testing* data, normally with around 70% of the data being for *training*. This is usually done automatically, and randomly, by the algorithm for each model. For each algorithm, the training data will be used to teach the model how each factor affects the output and the testing data will be used to verify the results to make sure that the model is within the bounds of error.

In this project, we have two data sets to work with for classification and regression respectively, where I showed the code for extracting the data in the Appendix.

1.6.1 Data Splitting for Error Estimation

Training and Testing Data

An obvious method to test how well the model does is to look at how well the model performs when tested with real world data.

Clearly, the Train Data is used as a set of examples to teach the model where to fit the boundaries for the model. Without restrictions, this would allow us to draw perfect boundaries around the data set. However, new data would be poorly fit so the model

[2]CRAN reference manual: https://cran.r-project.org/web/packages/Lahman/Lahman.pdf

would often draw more generalized boundaries which roughly fit the training data.

On the other hand, testing data is used as a test once the model has been fitted. This testing data is a completely new sample and is used to check the performance of the model given new data which the model has not seen previously. The measure of model performance is how well the model does when given the testing data.

Validation

An alternative to using training and testing is to use validation. This data is used to test the data as it learns and adjusts the *tuning parameters* as well as give an error estimate of the final model once learning is completed.

To avoid bias from subset selection, we use **cross-validation** (CV) as a technique to reduce bias from selection. Most commonly, 10-fold Cross-Validation is the preferred method used in Statistical Machine Learning as it provides a good number of validation trials while not being as computationally taxing as higher folds would result in. With 10-fold CV, we split the data into 10 "folds" - P_i - of equal size and then train the model 10 times, leaving out fold P_i in each run and using said fold to test the accuracy of the model at that stage before tuning for the next model. For each run, each fold has an accuracy F_i and the overall accuracy of the training model is given, for any K-fold, as follows:

$$\texttt{CV} = \frac{\sum_{i=1}^{K} F_i}{K} \tag{1.1}$$

Where most commonly, $K = 10$

1.7 Checking for Accuracy

The aim for this project is to show different tree-based methods to show how advancements in computing and different techniques have allowed for improvements in accuracy for tree-based models. Thus it is crucial that we use the same datasets for each model that we build. To compare models, we now have to check on the performance of each model. In order to test for performance, we have some methods for this:

1.7.1 Checking Accuracy for Regression

For Regression, it is trivial to use the errors in prediction as a variable to check accuracy. In this case, we use the **Root-Mean-Square Error** (RMSE) which is defined as follows:

Root-Mean-Square Error

Definition 1 (RMSE)**.** *The RMSE value is defined as the square root of the expected value of the square of the difference between $\hat{y}_i$ estimator and y_i actual value*

Thus, the RMSE is as follows:

$$RMSE = \sqrt{\frac{\sum_{i=1}^{n}(\hat{y}_i - y_i)^2}{n}} \tag{1.2}$$

Where we have n observations, and y_i and $\hat{y}_i$ are defined as above.

Conveniently the `accuracy` command in `R` calculates and provides the RMSE value for us so this is easy to extract.

1.7.2 Checking Accuracy for Classification

For Classification, we create an error matrix referred to as the **Confusion Matrix**, which shows how "confused" the model is when making predictions. This is a visual representation of the accuracy rate of the model which is calculated as such:

$$\texttt{accuracy} = \frac{\texttt{correct predictions}}{\texttt{total predictions}} \tag{1.3}$$

Confusion Matrix

A Confusion Matrix classifies how well a model did into a table:

		True Value	
	Total	Actual Positive	Actual Negative
Predicted Value	Predicted Positive	True Positive	False Positive
	Predicted Negative	False Negative	True Negative

We note that typically, a False Positive is a `Type I Error`, and a False Negative is a `Type II Error`.

Receiver Operating Characteristic

A better method to show how well the model performs is to use the **Receiver Operating Characteristic Curve** (ROC Curve). It is a graphical plot which uses probability predictions to show how accurate our model is. We plot the True Positive Rate (TPR) on the y-axis against the False Positive Rate (FPR) on the x-axis. These are generated by the confusion matrix above and is defined as the probabilities one would accept/reject H_0.

$$\texttt{TPR} = \frac{\texttt{True Positive}}{\texttt{True Positive + False Negative}} \tag{1.4}$$

$$\texttt{FPR} = \frac{\texttt{True Positive}}{\texttt{True Positive + False Negative}} \tag{1.5}$$

A good model aims to maximise the **Area Under the ROC Curve** (AUC), which is the entire area below the ROC Curve on the graph. The aim here is to get as close to `AUC = 1` as possible.

Conversely, a poor model would have `AUC = 0.5`. For reference, we plot the line $y = x$, the "chance line", and if the curve "follows" this line, we consider this a bad model and a poor predictor which we would not be able to use.

For this we will be using `ROCit` [Khan and Brandenburger, 2020] which is the new ROC plotting function for R. A feature that `ROCit` provides is that it calculates the **Youden J Index** [Ruopp et al., 2008]:

$$\mathtt{J} = \frac{\mathtt{True\ Positive}}{\mathtt{Actual\ Positive}} + \frac{\mathtt{True\ Negative}}{\mathtt{Actual\ Negative}} - 1 \tag{1.6}$$

Where the optimal point is shown on the plot. Each J Index calculates the likelihood that a Positive result is actually a True Positive. While this is not as useful for our data, it is very useful for medical data as it estimates the probability of an informed decision.

1.8 Summary of Project

We have now introduced the general set-up for our statistical model. Over the next few chapters, we will begin by defining, conceptually, how each of the models we will be using are constructed. In the next chapter, we will start with the old models which, while they have been supplanted by newer algorithms, are still nice to look into to and see as a motivation, why the current models have been built as they have been. In Chapters 3 and 4, we will look into the current crop of Tree-Based Methods that most people are using right now. In Chapter 5, we will introduce a new method and an entirely new concept which will look into an attempt to reinvent Tree-Based Models and compare them with the current generation of models. We will then implement them into an example in Chapter 6. We have briefly looked into this earlier in this chapter and will further expand on that in Chapter 6.

Chapter 2

Original Decision Trees

The idea of decision trees is to split up the data into segments (this is called *segmenting*) and then keep breaking them down until we get suitable groups. This can then be summarized up into a tree which can be displayed to see the paths taken. We will briefly explain the mathematics behind these two original Decision Trees as motivation to see why current modern Tree based methods are formulated as they are today.

2.1 Basics on Decision Trees

Trees are basically graphs to show structure and flow. A general idea is to think of a numerical variable y_i and some x_i, $i = 0, ..., n$, observations. Lets say for simplicity, we have for two variables X_1, X_2 (thus creating a 2-D plane), we are able to create two separate regions from a partition, which we then model Y on. This partition is on either X_1 or X_2.

Then, we are able to repeat the process for either one or both regions and model Y on the new partitions and continue to do so until we get to a stopping point.

The first question we ask is how do we know where to split each branch the tree?

2.2 AID

Automatic Interaction Detection (AID) was first developed in 1963 by Morgan and Sonquist as a social science problem [Morgan and Sonquist, 1963] expressed in an optimization problem to sequentially partition data in an observation matrix into a Regression Tree [Ali et al., 1975]. However, since then, it has proven so useful that it has become the basic building block for Decision Tree methods.

2.2.1 AID algorithm

ANOVA

Recalling Analysis of Variance (ANOVA) from Statistical Methods, we can calculate the following [Ritschard, 2013]:

For a value y_{im}, $i \in [1, \ldots, n]$ observations, $m \in [1, \ldots, M]$ groupings.

Residual Sum Squares (RSS) - The sum of the squares of residuals for each group (Also known as Within Sum Squares (WSS))

$$RSS = \sum_{m=1}^{M} \sum_{i=1}^{n} (y_{im} - \bar{y}_m)^2 \tag{2.1}$$

Total Sum Squares (TSS) - The sum of squares for all values

$$TSS = \sum_{m=1}^{M} \sum_{i=1}^{n} (y_{im} - \bar{y})^2 \tag{2.2}$$

Steps for AID algorithm

The AID algorithm is performed stepwise - it starts with a single cluster $M = 1$ and calculates the splits one step at a time until a stopping point (e.g. minimum node size) is reached.

Algorithm 1: AID Algorithm

```
1. At each step, for each predictor, either of the following occurs [Wilkinson, 1992]:
    • Continuous Predictors - For all $n$ observations, we have $n - 1$ ways to
      split the cluster. Calculate the RSS (eq. 2.1) value and select the optimal
      value.
    • Categorical Predictors - All $2^k - 1$ possible splits between $k$ categories
      are checked. Calculate the RSS (eq. 2.1) value and select the optimal value.
2. After calculating all RSS values we select the point with the smallest RSS value.
3. We then consider the interaction between predictors as we move down the levels
   as we no longer consider the same predictors for separate branches of the tree for
   the next stepwise calculation.
4. Repeat the process until stopping point is reached.
```

2.2.2 Benefits of AID

AID was a breakthrough in figuring out interactions between variables. It made possible, the idea of interaction between variables which helped solve many social science problems such as average earnings which previously have been difficult to solve if categorical variables have been included. It is also one of the least taxing computationally but does struggle to compete with newer techniques as advancements in computing power has made this method relatively trivial.

Most importantly, AID was easy to interpret by non mathematicians and is highly visual, thus making them appealing in presentations.

2.3 CHAID

CHAID was developed by Kass in 1980 [Kass, 1980]. It is essentially an extension of the AID algorithm suited for classification and stands for Chi-square Automatic Interaction Detection.

CHAID uses chi-squared statistical significance to test if there is an association between different categorical variables:

$$\chi^2 = \sum_{i=1}^{n} \frac{\sqrt{(y_i - E(y_i))^2}}{E(y_i)} \tag{2.3}$$

Where $E(y_i)$ is the expected value of y_i. We continue merging the groups until no significant χ^2 value can be found.

CHAID Algorithm

Here we state Kass's CHAID algorithm [Kass, 1980, pp. 121].
For a given data table with a contingency table, we perform Algorithm 2 on the table:

Algorithm 2: CHAID Algorithm

```
1. For each predictor, perform cross-tabulation (e.g. Continuous Predictors) of
   categories and then do the following:
   (a) Pair up categories of predictors which are the least significantly different.
       Merge both if they do not reach the critical value of the $\chi^2$ (eq. 2.3)
       significance test above
   (b) Repeat this step until all similar pairs have been merged
   (c) For categories formed from **at least two** merges, find the most significant
       binary split and perform a $\chi^2$ test to see if this split is significant
   (d) Repeat until splits are no longer significant
2. Calculate the significance of each merged predictor and select the highest
   chi-squared value. If that value turns out to be larger than a criterion value, split
   the data into subcategories
3. Repeat until all partitions have been analysed
```

Note: Here, we check the criterion value against a contingency table. An issue arises when we reduce the table size when merging groups as this means we can no longer use the same contingency table as before. Kass noted this when formulating the algorithm and proposed using Bonferroni corrections on the splitting criterion which he found to have improved the results. We will not go into detail of that but the reader may consult [Kass, 1980, pp. 122-126] if they would like to look into this.

2.3.1 Benefits of CHAID

CHAID brings all the benefits of AID but applies them in a classification setting. While it has been superseded by other models which are much more accurate, CHAID is still one of the fastest and easiest to explain mathematically and interpret.

2.4 Downsides of AID/CHAID

AID and CHAID, while useful, suffers from some issues:

1. They are heuristically greedy - The algorithm works in a top-down approach. Unfortunately, this is a computational constraint and comes up quite often in future methods.
2. Stopping point not defined - The nature of the algorithm means that there is no defined stopping point unless defined by the user when initializing the algorithm. This leads to unnecessarily large trees which require pruning.
   This will be solved in CART and will be discussed in the next chapter.
3. Focus not on prediction - The creators of AID/CHAID were more interested in segmenting data than making predictions. Therefore the splits are more associative than a data scientist would prefer.
   Nevertheless, we should not ignore AID/CHAID based on this alone and it is still useful, however, not optimal.

2.5 Summary

We now see how Decision Trees such as AID and CHAID are easily formulated using techniques available back when they were invented. We notice that they are still usable techniques, however, they do have their flaws which has lead to these algorithms being superceeded by more modern algorithms which we will explore in the next chapter.

Chapter 3

Modern Decision Trees

To address flaws in optimizing AID/CHAID, modern methods to form decision trees were invented in the 1980s and are still used today. **CART** was first introduced by Leo Breiman an others in the 1984 book *Classification and Regression Trees* (CART) [Breiman et al., 1984]. Since then, it has become one of the most cited methods for machine learning since it is easy to interpret while maintaining its accuracy.

An alternative method was developed in 1986 by Ross Quinlan [Quinlan, 1986] called **ID3**, which grew to become **C4.0, C4.5, C5.0 etc**. The method is very similar to Breiman's and really only differs in terms of the metrics used.

In this chapter, we will explore the different ways we evaluate each node and how to grow Decision Trees. We note that due to the complexity, we also will look at how trees are trimmed down to make them easy to interpret for people to use.

3.1 Classification

To find the best split for Classification Trees, we try to find the split which gives the least errors. This can be defined as an evaluation metric, or cost function, to simplify calculations. There are **two** major metrics, and many more minor ones, that are used as cost functions in deciding where to split each branch.

1. ***Gini Impurity***: Used by the CART method [Breiman et al., 1984],
2. ***Information Gain***: Used in methods such as C4.5, ID3 [Quinlan, 1986]

A metric is considered a good metric if it gives more information to the overall model by splitting into smaller nodes. In each instance, we have $m = 1, \ldots, M$ classes made up of $k = 1, \ldots, K$ categories for each $i = 1, \ldots, n$ observations to be grouped into.

Misclassification Error

Before we look at the other two methods, one has to consider why the classification alternative to RSS, **Misclassification Error**, where we sort the data into groups and consider the observations that do not belong in that group relative to the total number of observations.

Definition 2 (Misclassification Error). *A binary measure of node impurity is called Misclassification Error. Given that p_{mk} is the proportion that observations in region m are in class k,*

$$I_M(p_{mk}) = 1 - \max_k(p_{mk}, 1 - p_{mk}) \tag{3.1}$$

The Misclassification Error line is the boundary in deciding whether the metric performs better than random guessing. In this case, the majority voting group of the region is marked as p_i and everything else is considered as misclassification. One would think that this method would be the best method but we find that using this criterion, one ends up with relatively useless splits compared to the methods we will be introducing below. The reason may not be obvious, but one has to consider the main objective that we are trying to see whether we improve prediction if one were to split up a node.

When splitting, one tries to find the Loss Function L which improves the model the most. Here, we define $L_{misclass}$ for misclassification error as

$$L_{misclass} = I_M(p_{mk}) = 1 - \max p_{mk}$$

and one tries to find the split which minimises L as much as possible before and after splitting.

To show why misclassification does not always work, one can construct many splits based off the same starting information, and achieve the same misclassification error where one happens to be more useful than the other.. Lets say we have a binary data in one node which is divided $\{400, 100\}$ and try to split it. One can achieve multiple splits which give groups, of equal L, which are considered the same by misclassification, but of different results, i.e.

1. $\{400, 100\} \to \{200, 100\} + \{200, 0\}$
2. $\{400, 100\} \to \{200, 50\} + \{200, 50\}$
3. $\{400, 100\} \to \{375, 25\} + \{25, 75\}$

All three splits give the same loss $L = 100$. However, the first instance gives us a "pure" node which no longer needs to be split, the second would likely require both nodes to be further split to create a better predictor and the third gives us two different predictions after splitting. In most cases, the second node is relatively useless and one might prefer the first or third node instead. However, computationally, misclassification would see all three splits as equivalent and may result in the second split more often than not. Thus, one needs to find a better metric to split on.

3.1.1 Calculating metrics

These Metrics are based on the idea of Entropy, which itself is part of **Information Theory**, defined by Claude E. Shannon in 1948 in his landmark paper *A Mathematical Theory of Communication.* [1] [Shannon and Weaver, 1948]. The aim here is to calculate a value, which we can then use to see if splitting at that node would improve the accuracy of our decision tree.

Entropy

Definition 3 (Entropy). *[Shannon and Weaver, 1948] The Entropy of a random variable is defined as the information uncertainty in the variable's possible outcomes.*

In fact, Shannon had formulated this into a theorem for entropy which we will now state:

Theorem 1 (Entropy). *[Based on source [Shannon and Weaver, 1948, Section 6, p. 10]]*
For a measure $H(p_1, \dots, p_J)$, J events (in the node), where $p_1 + \dots + p_J = 1$ and p_j, the probabilities for each event, which hold the following conditions:

1. *H is continuous in p_j*
2. *If all p_c's are equal, then H is monotonic*
3. *If a choice is broken down into two successive choices, then H becomes the sum of each individual values.*
   For Example: $H(\frac{1}{2}, \frac{2}{5}, \frac{1}{10}) = H(\frac{1}{2}, \frac{1}{2}) + \frac{1}{2}H(\frac{4}{5}, \frac{1}{5})$

If the above is satisfied, we can define entropy as:

$$H(T) = I_E(p_1, \dots, p_J) = -\sum_{j=1}^{J} p_j \log_b p_j \tag{3.2}$$

We can derive two forms of entropy from the above:

1. Entropy of one attribute (Parent Entropy):
$$H(T) = -\sum_{j=1}^{J} p_j \log_2 p_j \tag{3.3}$$
   This is the full data
2. Entropy of two attributes (Child Entropy):
   For an attribute a in the set of observations, the entropy of H given a is defined as,
$$H(T \mid a) = -\sum_{j=1}^{J} P(j \mid a) \log_2 P(j \mid a) \tag{3.4}$$

[1] http://cm.bell-labs.com/cm/ms/what/shannonday/shannon1948.pdf

> We have that $P(c \mid a)$ the proportion of the group that was split into one child node from the parent node. This is the filtered data and the sum of each group of Child Entropy's is equal to the Parent Entropy.

We can now use the two entropy's to calculate information gain.

Information Gain

Adapting from above into our class observations of m regions and k classes, we can now define the first of the two evaluation metrics.

Definition 4 (Information Gain)**.** *Information Gain is a measure to see how much information can be added if we were to split the tree node T compared to keeping the node as it is:*

$$I_{IG}(T,a) = H(T) - H(T \mid a) = -\sum_{k=1}^{K} p_{mk} \log p_{mk} + \sum_{k=1}^{K} P(k \mid a) \log_2 P(k \mid a) \tag{3.5}$$

With p_{mk} and $P(k \mid a)$ the proportions of the group that was split into one child node a from the parent node.

Simply, we can subtract 3.4 from 3.3 to get the Information Gain Value.

Information Gain is used to decide on which feature to split on. We want it so that each split provides the most "information" and is therefore the value with this highest $IG(T,a)$. If $IG(T,a) = 0$, then we have found a terminal node and stop calculating Information Gain, else repeat the process until we arrive at a leaf node. This metric is used in ID3 and C4.5 [Quinlan, 1986].

Gini Impurity

An alternative, and simpler, method used to evaluate Decision Trees will now be described here. CART uses a method called **Gini Impurity** which measures how often we would incorrectly sort a random element from the set. While not directly formulated from Entropy above, it comes from very similar ideas as we have seen previously.

Definition 5 (Gini Impurity)**.** *The Gini Impurity (also called Gini Index), with proportions p_{mk} with m regions and k classes, can be defined as the following:*

$$I_G(p_{mk}) = \sum_{k=1}^{K} p_{mk}(1 - p_{mk}) \tag{3.6}$$

[Breiman et al., 1984]

Our aim is to try to minimise $I_G(p_{mk})$ such that the number is as close to 0 as possible.

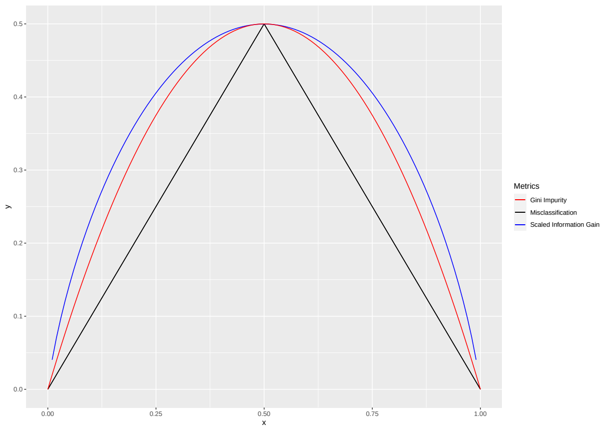


Figure 3.1: Comparison of Metrics, Gini vs Information vs Misclassification

Whats the difference?

While the two methods require different calculations, a paper by Murthy comparing these two methods among others [Murthy and Salzberg, 1995] found that there is no significant difference between Information Gain and Gini Impurity in the results to determine which is more optimal (Fig 3.1).

Referring to Fig 3.1, one can see clearly that there is no discernible difference between both Gini and Information Gain and thus the use case really depends on the preference of the user. In our case, we will prefer Gini Impurity as they lead on nicely to the techniques we will look at in Chapter 4.

Before we proceed onto the algorithm, we note a final metric which is useful to know but not essential towards forming Decision Trees.

Twoing Criterion

This final criterion was introduced by Breiman in CART [Breiman et al., 1984] along with Gini Impurity.

Definition 6 (Twoing Criterion)**.** *Twoing is the grouping of all $j \in J$ classes into two superclasses using the following:*

Given we send a proportion P_L of the whole learning class to the left, and $P_R = 1 - P_L$ to the right, we find the best twoing split s at node t to be as follows:

$$I_{TC}(p) = \frac{P_L P_R}{4} \left[\sum_{j=1}^{J} |p_{j,L} - p_{j,R}| \right]^2 \tag{3.7}$$

Where $p_{j,L}$ and $p_{j,R}$ the probabilities that p_j goes left or right respectively.

Twoing balances purity and the creation of equal size nodes and is thus the best method since this allows us to consider the problem as a two class problem and lets us build a more balanced - and much better - tree.

However, this method is much slower than calculating Gini Impurity and is thus not often used and we are unable to perform this in `R`, only in the original `Salford Systems'` which CART was originally written for.

3.1.2 Classification Tree Algorithm

For each observation x_i, $i \in [i, \dots, n]$, leading to a response observed y_i, we compute Algorithm 3 as follows:

Algorithm 3: CART for Classification

```
1. For each variable at every point, split the data into two regions $R_1, R_2$ where:
    (a) $R_1 = x_i \leq s$
    (b) $R_2 = x_i > s$
2. Calculate the Gini Impurity (eq. 3.6) for each split and compare all values
   against each other
3. Select the value with the highest Impurity as the choice of split
4. Repeat until a stopping criterion (e.g Node Size) is met
```

3.2 Regression Trees

Unlike classification, regression does not result in natural groupings which help us to choose where to split each branch of the decision tree, therefore an alternative method for partitioning needs to be found. The method is to minimize the Residual Sum of Squares (RSS), the same to the one used for AID (eq 2.1) but with some minor differences.

3.2.1 CART Regression Calculations

Suppose we have a space R and we have M partitions, we first partition the space into M regions $R_1, \dots, R_M$. Our aim is to predict which region the values x fall into and take the average value of the region to make a predictor. To construct the regions, we try to find regions which minimize RSS. Similar to Equation 2.1, we have that:

$$RSS = \sum_{m=1}^{M} \sum_{i \in R_m} (y_i - \hat{y}_{R_m})^2 \tag{3.8}$$

where $\hat{y}_{R_m}$ is the mean response for each region [Hastie et al., 2009, Section 9, p. 307].

Modelling Response Regions

For each observation x_i we have a corresponding response y_i, $i \in [1, \dots, n]$ for n observations. We now model the predicted response variable to the data. Similar to before, we model the "predicted response" as a constant c_m in each region $R_1, \dots, R_M$:

$$f(x) = \sum_{m=1}^{M} c_m I(x \in R_m) \tag{3.9}$$

Where we can obtain a optimum $\hat{c}_m$ by minimising the RSS value which we find to be the average of y_i for each region R.

$$\hat{c}_m = av(y_i | x_i \in R_m) \tag{3.10}$$

Using this, we can now formulate an algorithm to predict all the $\hat{c}_m$'s.

3.2.2 Regression Tree Algorithm

Algorithm 4 is the algorithm for regression trees. The algorithm itself is based off both the AID algorithm and the Classification Tree algorithm.

Algorithm 4: CART Regression Algorithm

```
1. For each variable at every point, split the data into two regions $R_1, R_2$ where:
    (a) $R_1 = x_i \leq s$
    (b) $R_2 = x_i > s$
2. Calculate the following for each split:
```

$$\sum_{x_i \in R_1} (y_i - \hat{y}_{R_1})^2 + \sum_{x_i \in R_2} (y_i - \hat{y}_{R_2})^2 \tag{3.11}$$

```
3. Select the minimal value in (2) for each variable and compare with all variables
   selecting the minimal value as the splitting variable.
4. Repeat until stopping criterion (e.g. Node Size) is met
```

3.2.3 Comparisons with AID

We notice that both Regression CART and AID calculate the RSS values to figure out the residuals if we were to split at a point. In fact, the algorithm is pretty much identical, except that Regression CART is a bit more modern and has been kept up to date, essentially replacing AID.

3.3 Pruning

Often we get trees which are way too big and unnecessarily complicated when simpler trees often do the trick. This leads to overfitted trees which provide low bias and high variance and is therefore unsuitable for regular use.

Ideally, we would grow a small tree with a minimum split value at certain point. However, the heuristically greedy nature of the algorithm would hide a very good split behind a seemingly weak split. To get around this, the alternative is to grow a very large tree T_0 and *prune* them back to obtain a subtree T instead where $T \subset T_0$. We call this method *Cost-Complexity Pruning* [Breiman et al., 1984].

3.3.1 Cost-Complexity Parameter

A quick explanation on the Cost-Complexity Parameter is useful to have before we proceed onto pruning. The **Complexity Parameter** (cp) is another method used to control how big the tree can grow. The cp parameter is there so that if adding another split to this node leads to an increase in cp, the tree does not grow that node. Knowing this, we can now work out the Cost-Complexity Pruning.

3.3.2 Cost-Complexity Pruning

The idea of pruning is to minimise the cost-complexity criterion:

$$C_\alpha(T) = R(T) + \alpha\,|T| \tag{3.12}$$

Where $R(T)$ is the learning error, $|T|$ the number of terminal nodes in tree T, and α the regularization parameter.

$R(T)$ differs depending on the type of tree we are pruning:

- For Classification:

$$R(T) = \sum_{m=1}^{|T|} (1 - \max_k(p_{mk}, 1 - p_{mk})) \times \frac{n(t)}{n} \tag{3.13}$$

  Where $n(t)$ is the number of x_i's in each t leaf node and n the total number of observations in that node and n the total number of observations

- For Regression:

$$R(T) = \sum_{m=1}^{|T|} \sum_{i \in R_m} (y_i - \hat{y}_{R_m})^2 \tag{3.14}$$

Pruning a Subtree

To prune a subtree, lets look at an individual subtree T_t at node t. The aim here is to minimise $C_\alpha(T - T_t) - C_\alpha(T)$, the variation of the cost-complexity function:

$$\begin{aligned} C_\alpha(T - T_t) - C_\alpha(T) &= R(T - T_t) + \alpha|T - T_t| - (R(T) + \alpha|T|) \\ &= R(T - T_t) - R(T) + \alpha(|T - T_t| - |T|) \\ &= R(T) - R(T_t) + R(t) - R(T) + \alpha(|T| - |T_t| + 1 - |T|) \\ &= R(t) - R(T_t) + \alpha(1 - |T_t|) \end{aligned} \tag{3.15}$$

Where we can find an optimal α when we set $C_\alpha(T - T_t) - C_\alpha(T) = 0$, resulting in:

$$\alpha = \frac{R(t) - R(T_t)}{|T_t| - 1} \tag{3.16}$$

Note: If $\alpha = 0$, then we return the whole tree unpruned so we aim to look at cases when $\alpha \neq 0$.

Algorithm

The Algorithm for pruning is as follows [MLWiki, 2020]:

Algorithm 5: Pruning Algorithm

```
1. Initialization - Let $T_1$ be the tree obtained with $\alpha_1 = 0$ by minimizing $R(T)$
2. Step 1.1 - Select $t \in T_1$ which minimizes

       $g_1(t) = \frac{R(t) - R(T_t^1)}{|T_t^1| - 1}$

3. Step 1.2 - Let $t_1$ be this node and let $\alpha_2 = g_1(t_1)$ which results in $T_2 = T_1 - T_{t_1}^1$
4. Repeat - Repeat the process $i$ times until at cost $k$
```

This results in the following outputs:

- a sequence of trees up to the root node: $T_1 \supseteq T_2 \supseteq \cdots \supseteq T_k \supseteq \cdots \supseteq T_{root}$
- a sequence of parameters $\alpha_1 \leq \alpha_2 \leq \cdots \leq \alpha_k \leq \ldots$

We now choose α^k which is our optimal value and then obtain the corresponding tree T^k as our result.

3.4 Implementation of CART

There have been multiple implementations of CART over the years. The most famous of these is rpart [Therneau and Atkinson, 2019], which is what we have used for this project. Here we will use the two sets of data which were put together in the introduction and I will provide a summary of it below.

3.4.1 Implementing CART in R

Using rpart without applying any control metrics gives us the following Decision Tree:

```
lgwinfit <- rpart(LgWin ~ ., data = alplayoff1)
#Using all available variables as possible predictors

lgwinfit
```

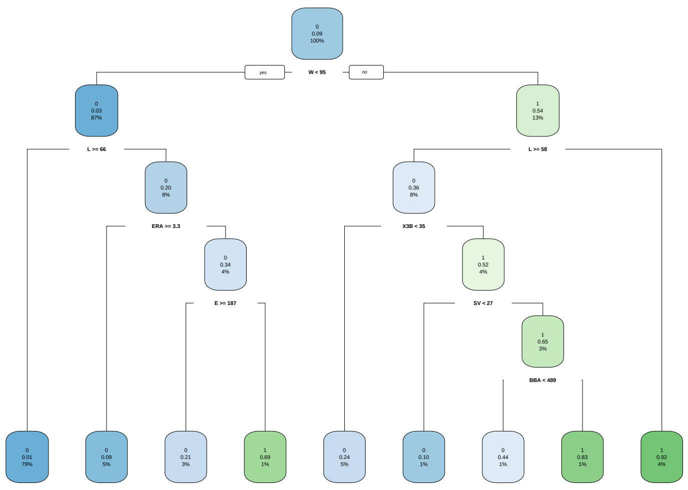


Figure 3.2: Example of a Decision Tree using CART

```
n= 1265

node), split, n, loss, yval, (yprob)
      * denotes terminal node

 1) root 1265 118 0 (0.90671937 0.09328063)
   2) W< 94.5 1108  33 0 (0.97021661 0.02978339)
     4) L>=65.5 990  12 0 (0.98787879 0.01212121) *
     5) L< 65.5 118  21 0 (0.82203390 0.17796610)
      10) ERA>=3.295 71   5 0 (0.92957746 0.07042254) *
      11) ERA< 3.295 47  16 0 (0.65957447 0.34042553)
        22) E>=186.5 34   7 0 (0.79411765 0.20588235) *
        23) E< 186.5 13   4 1 (0.30769231 0.69230769) *
   3) W>=94.5 157  72 1 (0.45859873 0.54140127)
     6) L>=57.5 106  38 0 (0.64150943 0.35849057)
      12) X3B< 34.5 62  15 0 (0.75806452 0.24193548) *
      13) X3B>=34.5 44  21 1 (0.47727273 0.52272727)
        26) SV< 26.5 10   1 0 (0.90000000 0.10000000) *
        27) SV>=26.5 34  12 1 (0.35294118 0.64705882)
          54) BBA< 488.5 16   7 0 (0.56250000 0.43750000) *
          55) BBA>=488.5 18   3 1 (0.16666667 0.83333333) *
     7) L< 57.5 51   4 1 (0.07843137 0.92156863) *
```

Which results in the following Decision Tree (Fig 3.2):

3.4.2 Using `rpart` for CART and ID3

One of the great features of `rpart` is that it allows you to create many different styles of Decision Trees based on what restrictions you give the model.

By default, `rpart` uses Gini Impurity as the main metric for splitting in classification. However, one can decide to implement ID3 in this scenario by selecting the parameter `parms=list(split="information")` which makes `rpart` apply Information Gain as the splitting metric. Using Information Gain, we end up building a different tree, however, by [Murthy and Salzberg, 1995], it has been proven that both perform similarly and we will revisit this in Section 6.

3.5 Tree Depth

The default parameter when growing a tree has a minimum number of variables before splitting at `minsplit = 20` and a maximum tree depth `maxdepth = 10` among other variables. Applying different control parameters lead to different size decision trees and one may prefer to increase the maximum tree depth if the dataset is large enough that our stopping criterion is hiding major splits.

The change in tree due to changes in tree depth leads to changes in accuracy. One finds that a larger tree depth leads to much more accurate trees, but going too far can lead to over-fitting which counter-intuitively leads to a reduction in accuracy with very large trees, which then have to be pruned to achieve generalisabililty anyway. Thus, when building trees, one should carefully select the size of the tree which is the best predicting model while maintaining interpretability.

3.6 Flaws of CART

CART, while a major improvement over other methods, does have its flaws like all methods do:

1. Heuristically greedy - Similar to AID/CHAID, the top-down approach leads to greedy splits where good splits are hidden behind a split which looks strong, but is actually a terrible split as it masks other more appropriate splits.

2. Complexity of Trees - This is both taxing in terms of computer memory and in terms of size. Unfortunately, pruning is the only method to achieve a desirable tree without losing any major splits if we were to grow a very small tree instead.

3. Flexibility - While normally, a flexible model is great for many use cases, one finds that sometimes this model is too flexible and would overfit the model in such a way that it becomes useless.

3.7 Summary

We have now seen the theory behind Breiman's Classification and Regression Tree, along with similar alternative algorithms developed by Quinlan. We also note that very large trees can be "cut back" with the use of pruning which allow us to create very accurate trees while still maintaining the ease-of-use that one can expects from smaller Decision Trees. Noting the above, we can now look at some more advanced techniques which use Decision Trees as their base learner as part of a much larger algorithm.

Chapter 4

Ensemble Methods - Bagging and Boosting

While being easy to interpret as well as flexible for many problems, there have been many attempts to improve upon decision trees. This has led to the development of two different tree based methods which when combined, create a much better predictor:

1. **Bagging/Random Forest** - Based on the idea of bootstrapping techniques to create many trees from randomly selected samples [Efron and Tibshirani, 1993], with replacement, from the full learning set
2. **Boosting** - Using weak learning trees to learn from incorrect predictions when forming successive trees

Both use the idea of "black-box" learning of trees but differ in the way they use decision trees when learning. Simplistically, Bagging/Random Forests grow trees co-currently in parallel with each other whereas Boosting grows trees successively, learning from previous mistakes.

4.1 Bagging vs Boosting

Bagging/Random Forests are used to reduce the variance of Decision Trees by aggregating lots of trees together to create a stronger predictor (Fig 4.1). Since each sample has a lower correlation of one another, we can combine each individual model together to reduce the overall variance of the model. However, we have not introduced any bias to the model by bootstrapping the samples.

Conversely, Boosting is used to reduce bias in Decision Trees since they aim to "learn" from the errors of previously grown weak learners (Fig 4.2). Boosting also combines results as it grows each tree and thus each new prediction is somewhat related to previous predictions from older weak learners.

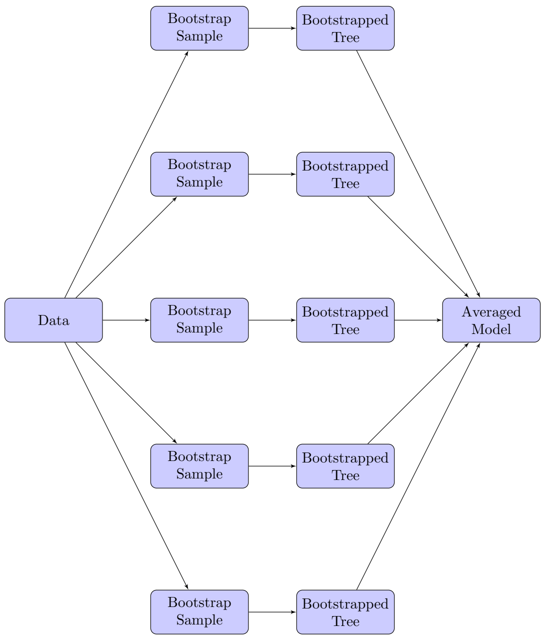


Figure 4.1: Flow of Bagging/Random Forests

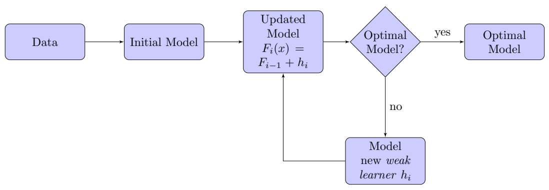

Data
Initial Model
Updated Model $F_i(x) = F_{i-1} + h_i$
Optimal Model?
yes
Optimal Model
no
Model new *weak learner* h_i

Figure 4.2: Flow of Boosting

We note that clearly, both methods lose the interpretability of CART as it is not feasible to show how all trees are grown. However, this is often offset by the shear superior accuracy that these models produce compared to regular Decision Trees. Therefore, we find these methods are often used when the prescriptive aspect of the model is not essential and one only cares for the predictive results from this model.

4.2 Bagging/Random Forests

4.2.1 Bagging

Bootstrap **Agg**regat**ing** (Bagging) [Breiman, 1996] uses multiple learning samples from the $\{x_i, y_i\}_{i=1}^n$ dataset, we collectively denoted these as $\mathcal{L}$. Each bootstrap involves random sampling from the original dataset with replacement, and is treated individually despite all bootstrap samples coming from the same data. We denote each bootstrap sample as $\mathcal{L}_k$ where $k \in [1, \dots, B]$.

Aside: The paper was written by Leo Breiman who invented CART and edited by Ross Quinlan who invented ID3

The aim here is to get an aggregated predictor, made up of individual predictors φ_k, which have been fitted from the bootstrap samples, and are better than one individual predictor. We will denote the final aggregated predictor as φ, defined as:

$$\varphi = \frac{1}{B}\sum_{k=1}^{B}\varphi_k \tag{4.1}$$

Bagging Algorithm

Here we present the Classification Algorithm (The Regression Algorithm is similar but adjusted accordingly for non-categorical data):

Algorithm 6: Classification Bagging

```
1. A seed is set to allow for replication
2. 50 bootstrap sample from a learning set $\mathcal{L} = \{x_i, y_i\}_{i=1}^n$ are drawn randomly, and
   each make a $\phi$ classifier to form $\phi_1, \dots, \phi_{50}$
3. All 50 $\phi$ classifiers are compared with each other and the most common classifier
   is selected, else, the class with the lowest error relative to the full Classification
   Tree is selected and this becomes $\varphi_k$
4. Repeat steps 4-5 $k$ times and average the $\varphi_k$ values to obtain $\varphi$
```

One can implement this algorithm using the `ipred` library in `R`

4.2.2 Random Forests

Random Forests are a natural development from Bagging as they use very similar techniques for growing ensembles. In Breiman's paper *Random Forests* [Breiman, 2001], which will be replicated here, he noted that while both Bagging and Random Forests are similar, Bagging is akin to "...darts thrown at random boxes ...", Random Forests are much more civilized in nature as they are formed by voting "...for the most popular class."

We begin by using Breiman's definition of what a Random Forest is [Breiman, 2001]:

Definition 7 (Random Forests)**.** *A tree-based method formed from k, $k \in \mathbb{N}$, independent and identically distributed ensemble of classifiers, $h_1, \dots, h_k$.*
Each tree casts one vote for the most popular class which is to be selected.

Random Forest Algorithm

Using the above definition, we can adapt the Bagging Algorithm into a Random Forest Algorithm with some minor changes as below.

Algorithm 7: Random Forest Algorithm

```
1. A seed is set to allow for replication
2. Draw $B$ bootstrapped samples from the learning set $\mathcal{L}$
3. Grow a random forest as follows:
    (a) Select $m$ variables randomly from the sample $\mathcal{L}_b$ where $b \in [1, \dots, B]$
    (b) Form a regular decision tree using only the selected features and the
        relevant metric described above
    (c) Output the classifiers $\phi_1, \dots, \phi_B$
4. Make a prediction as follows
   Regression $\frac{1}{B}\sum_{b=1}^{B}\phi_b$
   Classification $majority\{\phi_b\}_{b=1}^{B}$
```

Here, we note the following defaults when the value of m is to be selected (For which the library used for implementation, `randomForest`, does this automatically if undefined by the user):

- **Classification** - $m = \lfloor\sqrt{p}\rfloor$ with minimum node size one
- **Regression** - $m = \lfloor\frac{p}{3}\rfloor$ with minimum node size five

4.2.3 Out of Bag Samples

The **Out of Bag Samples** (OOB) is used to check how good how well a model is fitted. A feature of selecting only a certain sample for bootstrapping is that we can use the samples not selected for a certain bootstrap to be a testing variable. Thus if variable $\{x_i, y_i\} \notin \mathcal{L}$ we can construct a predictor for this variable from the trees in which this variable is not a training data in.

We are able to use this value to tell us right away if a model is good as a good model would have a low OOB error. This is helpful as `R` calculates the OOB value for us automatically and if a bad model were to be found to have been fitted, we can discover this early on before proceeding onwards towards predicting values.

4.2.4 A Note on Implementation

One can see from these algorithms that Bagging and Random Forests are very similar. In fact, one can see from the algorithm for Random Forests that if one sets $m = p$, which means all variables are selected in this Random Forest, one is practically performing Bagging. Thus we have that **Bagging** $\subset$ **Random Forest**.

When implementing this in `R`, one can use this fact to reduce the number of libraries loaded into `R`. Here, instead of using `ipred`, one can instead use `randomForest` setting `mtry = p`, where p is the number of variables one has in the training dataset. One finds that there is no substantial difference between these two methods, besides sampling bias which one always expects to occur when bootstrapping.

4.3 Boosting

While the previously seen methods of Bagging and Random Forests work on Bootstrapping samples and running many trees at the same time before aggregating/voting for results, Boosting involves using many "weak learners" which learn from incorrect predictions. Here, incorrect learners are sampled more often and given more weight when learning to improve on these mistakes and try to minimize the errors that these weak learners produce. The final model is formed from combining all the weak learners into a much stronger single model.

4.3.1 AdaBoost

The first popular boosting algorithm is **Ada**ptive **Boost**ing [Freund and Schapire, 1997] (AdaBoost). AdaBoost uses $t = [1, \ldots, T]$ *weak learners*, made up of small one-step decision trees which individually are not accurate, but which combined makes a much better model.

AdaBoost gives a weight $w_t(i)$ to each predictor x_i to help decide which variable to focus on after each iteration. A predictor which has been incorrectly classified previously is given more weighting in subsequent weak learners compared to previous weak learners.

Weak Learners

We have a weak hypothesis $h_t : X \to \{-1, +1\}$ for $x_i \in X$. How good the hypothesis is can be tested by its error ε_t.

$$\varepsilon_t = \sum_{h_t(x_i) \neq y_i} w_t(i) \tag{4.2}$$

In computations there exists a function called **WeakLearn** which achieves this by using *decision stumps*.

AdaBoost Algorithm for Classification

Combining the above, we can now generate the AdaBoost Algorithm as follows:

AdaBoost for Regression

The algorithm and mathematics behind AdaBoost for regression work similarly to above. Refer to [Freund and Schapire, 1997] to see the AdaBoost Algorithm for Regression.

AdaBoost is a Random Forests

A conjecture noted by Breiman in *Random Forests* [Breiman, 2001] was that both AdaBoost and Random Forests are so similar that they are essentially equivalent.

Algorithm 8: AdaBoost

```
1. Set $\mathbf{w}_1 = \frac{1}{n} \; \forall \; n$ training sets
2. for $t = 1, \ldots, T$ do
     Pick $h_t$ which minimizes $\varepsilon_t = \sum_{h_t(x_i) \neq y_i} w_t(i)$
     Compute weight of chosen classifier
         $$\alpha_t = \frac{1}{2} \log \frac{1 - \varepsilon_t}{\varepsilon_t}$$
     Update weights
         $$w_{t+1}(i) = \frac{w_t(i) \exp -\alpha y_i h_t(x_i)}{Z_t}$$
     Where $Z_t$ is the normalization factor
     Break if $\varepsilon_t = 0$ or $\varepsilon_t \geq \frac{1}{2}$
   end
3. Output final hypothesis
         $$h_f(x) = \text{sign}\left(\sum_{t=1}^{T} \alpha_t h_t(x)\right)$$
```

4.3.2 Gradient Boosting

A more generalized version of AdaBoost came along in 1999 by Jerome Friedman (The second author in CART) in *Greedy Function Approximation: A Gradient Boosting Machine* [Friedman, 1999a]. Here, we maintain the idea of using **Weak Learner** from AdaBoost while introducing two further components:

Loss Function - A function to estimate how good a model performs

Additive Model - The method of adding Weak Learners to each other one at a time

This can be summarised into three simple steps:

1. An initial model F_0
2. A new model h_1 is fitted based on the errors of the previous model - this is the weak learner
3. The previous model and the weak learner are combined to get an updated model $F_1(x) = F_0(x) + h_1(x)$

These three steps are repeated for m until the errors have been minimized as much as possible which gives us the following iterative steps:

$$\begin{array}{rcl} F_1(x) & = & F_0(x) + h_1(x) \\ F_2(x) & = & F_1(x) + h_2(x) \\ \vdots & \vdots & \vdots \\ F_m(x) & = & F_{m-1}(x) + h_m(x) \end{array}$$

So $F(x) = F_0(x) + \sum_{j=1}^{m} h_j(x)$.

The final goal here is to find an approximate $\hat{F} \approx F_m(x)$ which minimize the the expected loss function which would minimize the bias of the final model.

$$\hat{F} = \operatorname*{argmin}_F \mathbb{E}[L(y, F(x))]$$

We will introduce this in a more general sense before showing how Tree-Based Methods can be fitted and used.

Loss Function

To start, we need to establish $F_0(x)$. Here, we establish a function which minimizes the cost function as such:

$$F_0(x) = \operatorname*{argmin}_\gamma \sum_{i=1}^{n} L(y_i, \gamma) \tag{4.3}$$

An example of a Loss Function is the **Mean-Squared Error** where we set $\gamma = \hat{y}$.

We will use a method called **gradient descent** to estimate the loss function.

Gradient Descent

Gradient Descent, also referred to as *steepest-descent*, is a method to find a local minimum of a function. It is one of the simplest minimization methods used and is great for boosting as we will be applying this method each time you grow a tree. One defines the gradient descent for boosting as follows:

$$g_m(x) = \left[\frac{\partial L(y, F(x))}{\partial F(x)}\right]_{F(x)=F_{m-1}(x)} \tag{4.4}$$

An intuition for this is to imagine that the loss function is shaped like a bowl and the idea of gradient descent is to reach the lowest point of the bowl which happens to be the lowest cost. This has been achieved by use of partial differentiation on the Loss Function as seen in equation 4.4.

Friedman found that the best loss function is $-g_m(x)$, which can be referred to as the "pseudo-responses" $\tilde{y}_m$ [Friedman, 1999a]. One uses the responses from $\tilde{y}$ to fit a weak learner h_m onto it.

Scaling Multiplier

The last value one needs to find γ_m when updating the model is a multiplying scaling factor ρ_m which gives us the optimal steepest-descent value when updating the model:

$$\gamma_m = \rho_m h_m(x)$$

where ρ_m is the following:

$$\rho_m = \underset{\gamma}{\operatorname{argmin}} \sum_{i=1}^{n} L(y_i, F_{m-1}(x_i) + \gamma h_m(x_i)) \tag{4.5}$$

Intuitively, this scales each weak learner so that it leads to <u>minimal</u> changes to the model while correcting previous errors, which is the intent of a weak learner. One can finally solve for $F_m(x)$ using the iterative model

$$F_m(x) = F_{m-1}(x) + \rho_m(x) h_m(x) \tag{4.6}$$

Adjustment for Trees

An issue with the current method of construction is that we end up losing the disjoint regions that Decision Trees give when combining two separate models. Having written both CART [Ripley, 2019] and [Friedman, 1999a], Friedman knew of this and suggested the following, noting that the additive form of Decision Trees, following Friedman, are as follows:

$$h(x; \{b_j, R_j\}_1^J) = \sum_{j=1}^{J} b_j \mathbb{1}(x \in R_j)$$

For disjoint regions $\{R_j\}_1^J$ and base learner parameters $\{b_j\}_1^J$ with an indicator function $\mathbb{1}(x \in R_j)$ to indicate whether predictor y_i is in the region or not. Since each region in each tree is disjoint, if $x \in R_j$ then $h(x) = b_j$.
Thus one can update 4.6 to become:

$$\begin{aligned} F_m(x) &= F_{m-1}(x) + \rho_m \sum_{j=1}^{J} b_{jm} \mathbb{1}(x \in R_{jm}) \\ &= F_{m-1}(x) + \sum_{j=1}^{J} \gamma_{jm} \mathbb{1}(x \in R_{jm}) \end{aligned} \tag{4.7}$$

From here, one can establish the Gradient Boosting Algorithm adapted for Tree-Based learners.

Gradient Boosting Algorithm

Combining all the processes above together, we achieve the following algorithm for Tree-Based Boosting. We will show the case for the Regression Algorithm although one can manipulate this for Classification if one encodes categorical variables numerically.

Before initializing the algorithm, we need to find a starting model which is easy to calculate. In the regression setting, one finds that setting $F_0 = median\{y_i\}_1^n$ is the easiest and most effective starting point, although others do exist.

Algorithm 9: Gradient Boosting

```
1. Set $F_0(x) = \text{argmin}_\gamma \sum_{i=1}^n L(y_i, \gamma)$
2. for $m = 1, \dots, M$ do
    Compute
        $$g_m(x) = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1}(x)}$$
    Fit $h_m$ to $g_m$ to obtain nodes $\{R_{jm}\}_1^J$
    Compute multiplier $\gamma_{jm}$
    Set
        $$F_m(x) = F_{m-1}(x) + \sum_{j=1}^{J} \gamma_{jm} \mathbb{1}(x \in R_{jm})$$
   end
3. Output $F_m$ as our $\hat{F}$ model
```

Note: Observe that this algorithm does not implement well in a classification setting and especially using Classification Trees as gradient descent relies on continuous or numerical data to function. In [Friedman, 1999a], the method recommended is to use Logistic Regression and log-likelihood functions. This allows us to simulate classification, while maintaining continuity of the function by using Logistic Regression which is similar enough numerically to regression that the algorithm is able to function One can also just run the classification as a Regression Tree by *one-hot encoding* the predictions and then round the final estimator to achieve a classification prediction although this really only works up to a certain extent and is not as accurate as just using Logistic Regression for classification here.

4.3.3 xgBoost

Extreme Gradient Boosting, referred to as xgBoost, [Chen and Guestrin, 2016] is a modern implementation of Gradient Boosting which applies Gradient Boosting as a "scalable machine learning system for tree boosting" [Chen and Guestrin, 2016].

The biggest difference between both methods is that xgBoost uses the Second-Order derivative of the Loss Function, i.e. differentiate twice instead of once. The idea here is to find more information about the loss more quickly than previous implementations.

Another difference between regular boosting introduced in *Greedy Function Approximation* [Friedman, 1999a] and xgBoost was the idea of **Stochastic Boosting** introduced in Friedman's follow up *Stochastic Gradient Boosting* [Friedman, 1999b]. Here, one subsamples rows and columns before creating each tree. According to Chen "Using column sub-sampling prevents over-fitting more so than traditional row sub-sampling".

A final aspect is the use of L1 or L2 regularization on the leaf weights to further avoid over-fitting.

Second-Order Approximation

One can define the Second-Order approximation of the loss function to be, following [Chen and Guestrin, 2016], we start with the loss function we want to minimize:

$$L = \sum_{i=1}^{n} L(y_i, F_{m-1}(x_i) + f_t(x_i)) + \Omega(f_t) \tag{4.8}$$

Where f_t is an independent tree and $\Omega(f_t)$ a penalization of the complexity of the model. Then, expanding by Taylor and simplifying, one can find an approximate to the Second-Order differential to be:

$$L \approx \sum_{i=1}^{n} \left[g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i) \right] + \Omega(f_t) \tag{4.9}$$

where $g_i = \partial_{\hat{y}^{(t-1)}} L(y_i, \hat{y}^{(t-1)})$ and $h_i = \partial^2_{\hat{y}^{(t-1)}} L(y_i, \hat{y}^{(t-1)})$.

Set $\Omega(f_t) = \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2$, with w_j the leaf weight $w_j = -\left(\sum g_i\right) / \left(\sum h_i + \lambda\right)$ and one can calculate the loss function we want as,

$$\tilde{L} = -\frac{1}{2} \sum_{j=1}^{T} \frac{\left(\sum g_i\right)^2}{\sum h_i + \lambda} + \gamma T \tag{4.10}$$

One can now approximate the loss reduction of the split by setting `Gain = Loss Before - Loss After` which is,

$$L_{split} = \frac{1}{2} \left[\frac{\left(\sum_{i \in I_L} g_i\right)^2}{\sum_{i \in I_L} h_i + \lambda} + \frac{\left(\sum_{i \in I_R} g_i\right)^2}{\sum_{i \in I_R} h_i + \lambda} - \frac{\left(\sum_{i \in I} g_i\right)^2}{\sum_{i \in I} h_i + \lambda} \right] - \gamma$$

One can now use 4.3.3 and implement it with the Gradient Boosting Algorithm above and achieve xgBoost, except now, one can reach that local minimum much more quickly and while still achieving similar performance.

4.4 Summary

One can observe in this chapter, how great Decision Trees are at being base learners for much more complex models. While one loses the ability to "interpret" these models, we do find that these models do reduce the issues of overfitting the model which is common in Decision Trees. We also note the differing approaches that these ensemble methods use to achieve better results, which show the advancements that have been achieved in computing to allow for efficient computation of these algorithms.

Chapter 5

Optimizing Trees

While both Random Forests and Boosting are very accurate predictors, they achieve this by sacrificing interpretability in favour of a "black-box" approach to prediction. This makes it harder to show how each decision is made and is the trade-off necessary in order to achieve higher accuracy. We now return to the idea of Decision Trees but under a more modern light using ideas from operations research as a basis for the algorithm.

5.1 Trees as an Optimization Problem

Improvements in technology have now allowed us to look at decision trees under a different perspective. Previously, the most efficient way to improve accuracy without inducing bias is to look at improving the metrics used for splitting and then when the limits have been pushed, use ensemble techniques to aggregate and minimizing the errors. Dimitris Bertsimas and his PhD student, Jack Dunn proposed reforming decision trees as an optimization problem [Bertsimas and Dunn, 2017]. This is actually a realisation of Leo Breiman's dream to solve the entire decision tree in one go. In fact, in CART, Breiman noted the following [Breiman et al., 1984, p.42] in his book:

"*Finally, another problem frequently mentioned (by others, not by us) is that the tree procedure is only one-step optimal and not overall optimal. …If one could search all possible partitions …the two results might be quite different. This issue is analogous to the familiar question in linear regression of how well the stepwise procedures do as compared with best subsets procedures. We do not address this problem. At this stage of computer technology, an overall optimal tree growing procedure does not appear feasible for any reasonably sized dataset.*"

Thus, really we should now look at the problem as a Mixed-Integer Optimization Problem (MIO). Revisiting CART, we can now look at this as a programming problem called the **Optimal Tree Problem**.

5.1.1 What is Mixed-Integer Optimization

Mixed-Integer Programming is a subset of Integer Programming where only some of the variables have integer constraints whereas others are more free and can be non-integers within the constraints. While the idea had been conceived by Kantorovich and, independently, Dantzig, it was not until 1981 [Arthanari and Dodge, 1981] that more attempts to tackle statistical problems took flight. Even then, it took until 2004 [Bixby et al., 2004] when optimizing the Mixed-Integer Programming problem became possible. Fortunately, in the past decade, a modern MIO solver in `Gurobi` [1] had been introduced which allow us to solve many Mixed-Integer problems more efficiently.

The method to solve these Mixed-Integer problems is call **Branch and Bound** first introduced in the *Travelling Salesman Problem* [Little et al., 1963]. In simple terms, one starts by "relaxing" the problem with no integer restrictions, and solving it using normal linear programming methods to see if the constraints we have relaxed end up being integer solutions anyways. Normally, this is not the case and so we pick one constraint and split the problem into two sub-problems with tighter constraints where, for example, if the value for variable x in linear programming "relaxation" is 4.6, then we impose constraints $x \leq 4$ and $x \geq 5$ the next time we run linear programming. The full algorithm can be found in [Little et al., 1963] and has been skipped on purpose here for conciseness.

In this instance, the software we use `Interpretable AI` has its own implementation of `Gurobi` already built in

5.1.2 CART as a Linear Problem

Given a complexity parameter α which controls the tradeoff between accuracy and complexity, a minimum number in each node N_{min} and a dataset $\{x_i, y_i\}_{i=1}^n$ we aim to solve the problem $\forall$ l leaf nodes $\in T$ tree:

$$\begin{aligned} \min \quad & R(T) + \alpha\,|T| \\ \text{s.t.} \quad & N(l) \geq N_{min} \end{aligned}$$

Where $N(l)$ is the number of points in leaf node l and $R(T)$ is the misclassification error from before.

5.2 The Optimal Classification Tree

We now proceed on to the Optimal Classification Tree [Bertsimas and Dunn, 2017]. Here, we attempt to solve the tree as a MIO problem which allows us to avoid having to choose whether to branch and what variable we would branch on. In this way, we are able to create the model in one go and avoid the decision making process which leads to the heuristically greedy nature of CART.

[1]https://www.gurobi.com/resource/mip-basics/

Note: We now follow the method presented by Jack Dunn in his thesis which, in turn, was published in [Bertsimas and Dunn, 2017] which you may refer to for further details.

5.2.1 Growing the Tree

From [Bertsimas and Dunn, 2017], to formulate this tree we note that the tree we can grow at depth D contains, at most, $T = 2^{(D+1)} - 1$ nodes if all splits are binary, which we will index as $t = 1, \ldots, T$. Now, we need the following notations. We denote for node t, $p(t)$ as the parent node and $A(t)$ the set of ancestor node of t. We then group all nodes into two sets as follows:

1. **Branch Nodes** - The nodes $t \in T_B = \{1, \ldots, \lfloor \frac{T}{2} \rfloor\}$ which apply the split in the form of a matrix $\mathbf{a}^\intercal \mathbf{x} < b$, where everything less than b goes into the left split and anything greater than b to the right.
2. **Leaf Nodes** - The nodes $t \in T_L = \{\lfloor \frac{T}{2} \rfloor + 1, \ldots, T\}$ which make predictions.

The split at node $t \in T_B$ corresponds to the variables $\mathbf{a}_t \in \mathbb{R}^n$ and $b_t \in \mathbb{R}$ and set both $\mathbf{a}_t = b_t = 0$ when we no longer want to split (Forcing all nodes down one branch) for computational reasons. To track which nodes were split on, an indicator variable $d_t = \mathbb{1}$ which gives an output of 1 when a node is split. The following constraints can be applied:

$$a_{jt} \in \{0,1\}, \text{ for each } j = 1, \ldots, p, \qquad \forall\, t \in T_B \tag{5.1}$$

$$\sum_{j=1}^{p} a_{jt} = d_t, \qquad \forall\, t \in T_B \tag{5.2}$$

$$0 \leq b_t \leq d_t, \qquad \forall\, t \in T_B \tag{5.3}$$

Since $0 \leq \mathbf{a}_t^\intercal \mathbf{x}_i \leq d_t$ for any i and t by construction of eq. 5.2. A final constraint $d_t \leq d_{p(t)}$, $\forall\, t \in T_B P \backslash \{1\}$ is added so that a branch node does not split if the parent node does not also split.

5.2.2 The Mixed-Integer Optimization Problem

Using the constraints listed above, we can now model the tree using Mixed-Integer Optimization. At each step we will keep track of what is being assigned to each leaf node. We have an indicator variable $z_{it} = \mathbb{1}\{\mathbf{x}_i \text{ in node } t\}$ with $\sum_{t \in T_L} z_{it} = 1$ tracking what is assigned to each node and $l_t = \mathbb{1}\{\text{leaf } t \text{ contains any points}\}$ to enforce N_{min} so that:

$$z_{it} \leq l_t,\ t \in T_B \tag{5.4}$$

$$\sum_{i=1}^{n} z_{it} \geq N_{min} l_t,\ t \in T_B \tag{5.5}$$

Which now allows us to formulate the tree splits, where $i = 1, \dots, n$, $\forall\ t \in T_B$ as thus:

$$\mathbf{a}_m^\intercal \mathbf{x}_i < b_t + M_1(1 - z_{it}),\ \forall\ m \in A_L(t), \tag{5.6}$$
$$\mathbf{a}_m^\intercal \mathbf{x}_i \geq b_t - M_2(1 - z_{it}),\ \forall\ m \in A_R(t), \tag{5.7}$$

Note: In practice, the strong inequality 5.6 is not computationally feasible and so often a small, but computationally significant, error term ϵ is used to combat this.

From the constraints, we can also solve the constants M_1 and M_2 which lead to the following inequalities which can be used by MIO when formulating the tree splits $\forall\ t \in T_L$:

$$\mathbf{a}_m^\intercal (\mathbf{x}_i + \epsilon) \leq b_t + (1 + \epsilon)(1 - z_{it}),\ \forall\ m \in A_L(t), \tag{5.8}$$
$$\mathbf{a}_m^\intercal \mathbf{x}_i \geq b_t - (1 - z_{it}),\ \forall\ m \in A_R(t), \tag{5.9}$$

Labelling Incorrect Predictions

To keep track of what each predictor gives out as a prediction, we need to find a way to keep track of what we have misclassified. Lets say we have a matrix Y, we note the following cost of an incorrect label prediction to the algorithm for $i = 1, \dots, n$ and $k = 1, \dots, K$:

$$Y_{ik} = \begin{cases} 1, & \text{if } y_i = k \\ -1, & \text{otherwise} \end{cases}$$

Setting N_t to be the total number of points in node t and N_{kt} the number of those labelled k in node t, we also get,

$$N_t = \sum_{i=1}^{n} z_{it},\ \ \forall\ t \in T_L \tag{5.10}$$

$$N_{kt} = \frac{1}{2} \sum_{i=1}^{n} (1 + Y_{ik}) z_{it},\ \ \forall\ t \in T_L \tag{5.11}$$

Thus, the optimal label c_t can be defined as:

$$c_t = \underset{k=1,\dots,K}{\operatorname{argmax}} N_{kt}$$

where we find that the predictor will contain these points at each leaf node,

$$\sum_{k=1}^{K} c_{kt} = l_t,\ \ \forall\ t \in T_L \tag{5.12}$$

Optimal Loss

Knowing the above, we can now find the optimal loss at each node L_t as,

$$L_t = N_t - \max_{k=1,\dots,K} N_{kt}$$

Which can be linearized $\forall\ t \in T_L$ as,

$$\begin{aligned}
&L_t \geq N_t - N_{kt} - M(1 - c_{kt}), && k = 1, \dots, K && (5.13)\\
&L_t \leq N_t - N_{kt} + Mc_{kt}, && k = 1, \dots, K && (5.14)\\
&L_t \geq 0, && && (5.15)
\end{aligned}$$

For any sufficiently large M.

5.2.3 The OCT Model

One can note from above that what we aim to do is to minimize the total misclassification cost as well as the complexity of the tree. We have a baseline model with accuracy $\hat{L}$ to normalize against and thus, for an independent α one has,

$$\min \frac{1}{\hat{L}} \sum_{t \in T_L} L_t + \alpha \sum_{t \in T_B} d_t \tag{5.16}$$

Which is the OCT Model with the restraining conditions, equations 5.1 - 5.5, and 5.8 - 5.15.

Warm Starts

Dunn noted in Optimal Classification Trees [Bertsimas and Dunn, 2017] that a "Warm Start" provides a benefit in the way of reducing run time of the algorithm. Please refer to section 2.3 of [Bertsimas and Dunn, 2017] for further details.

5.2.4 Optimal Classification Tree Algorithm

We can now implement the MIO problem as an algorithm:

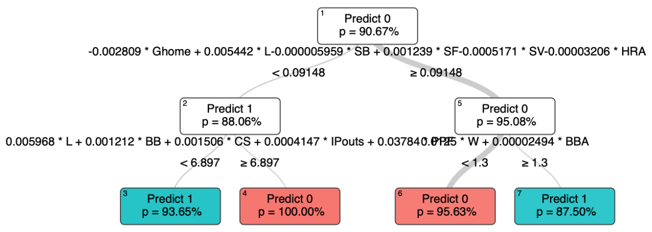
5.3 An Extension to Hyper-planes

The nature of CART makes it very difficult to try to introduce hyper-planes when splitting data sets. Previous attempts have been made by Wei-Yin Loh at the University of Wisconsin-Madison [Loh and Vanichsetakul, 1988], however, these have not been commonly used.

An interesting nature of using Mixed-Integer Programming is that you are no longer restricted to linear splits within categories. By adjusting a few of the constraints, such as redefining the cost-complexity parameter to work across splits, we have put on the

Algorithm 10: Optimal Classification Tree

```
1. Set a maximal depth $D_{max}$ and a minimum leaf size $N_{min}$
2. For $D = 1, \ldots, D_{max}$, for $C = 1, \ldots, 2^D - 1$:
    (a) Run CART with $N_{min}$ and $\alpha = 0$ to get an initial solution
    (b) Choose the warm start with the lowest error
    (c) Use selected warm start to solve the MIO problem
    (d) Add solution to warm start pool
3. For any value of $\alpha$, remove all non-optimal solutions
4. Select the best performing solution and the $\alpha$ for which this solution is optimal
```



1
Predict 0
p = 90.67%
-0.002809 * Ghome + 0.005442 * L-0.000005959 * SB + 0.001239 * SF-0.0005171 * SV-0.00003206 * HRA
< 0.09148
≥ 0.09148
2
Predict 1
p = 88.06%
0.005968 * L + 0.001212 * BB + 0.001506 * CS + 0.0004147 * IPouts + 0.03784
5
Predict 0
p = 95.08%
0.0125 * W + 0.00002494 * BBA
< 6.897
≥ 6.897
< 1.3
≥ 1.3
3
Predict 1
p = 93.65%
4
Predict 0
p = 100.00%
6
Predict 0
p = 95.63%
7
Predict 1
p = 87.50%

Figure 5.1: OCT with Hyper-Plane

OCT Model, one can apply the same method but on hyper-planes between variables. More can be found in [Bertsimas and Dunn, 2017].

An example of a Optimized Tree with Hyper-planes is seen in fig 5.1. Essentially, we form decision boundaries using lines which are similar to the ones formed in linear regression and split the group based on which side of this dividing line is on. Whilst this is an innovative advancement and allows for interpretable models using multiple predictors in one boundary which is useful in very large datasets, the mathematics behind this is beyond the scope of this project.

5.4 Comparing OCT with Bagging

While OCT solves many issues regarding accuracy while maintaining interpretability, while reading the literature, the author, myself, was curious to see how this model would perform if one were to create an ensemble model based off this model to see whether the gains achieved by Bertsimas and Dunn hold up in more real-world scenarios. I decided that the simplest and most interesting way (*albeit still very time consuming and computationally taxing*) to achieve this is by Boostrap Aggregating as we have seen in the previous chapter. In this case, we will be comparing it with the regular OCT model to see how well bagging improves the accuracy score of the test.

Model Type	Accuracy Score
Optimal Classification Tree	0.9140363
Bagged OCT	0.9120215
Cross Model Score	0.967092

Table 5.1: OCT vs Bagged OCT

Model Type	AUC Scores
Optimal Classification Tree	0.701825663192676
Bagged OCT	0.923786095034238

Table 5.2: AUC Scores for OCT models

5.4.1 Bagging OCT

To perform this bagging, we create a `for` loop to generate 100 bootstrapped samples from the learning dataset which are then fitted and a prediction is based on the same data is made for each run. We store the score of each run and average them all to create a final predictor similar to how `ipred` achieves this. The following results were achieved and shown in tables 5.1 and 5.2:

Table of results for comparison

Tables 5.1 and 5.2 show us how well our bagged model does in comparison with the base model.

ROC Curves and AUC Scores

One notices that the models are very similar to each other and only once one looks at the AUC score (Figure 5.2) can one see that an improvement occurs when we bag a model.

Note: The original OCT contains only 4 leaf nodes. With the nature of how OCT is calculated using MIO's, when predicting probabilities, this will result in "Return the probabilities of class membership predicted by a model for each point in the features" according to the IAI reference manual. Thus, one would only find 4 points on the leaf node, resulting in the interesting ROC curve that we observe. However, since Bagging averages our probabilities, we are able to find many more probabilities and thus the ROC curve there will seem more natural

5.5 Optimal Regression Trees

Using similar techniques to Optimal Classification Trees, one can easily extend this to the Regression Trees using a few tweaks to the model. In fact, most of the model remains the same as in the case for classification and it is only the labelling of incorrect

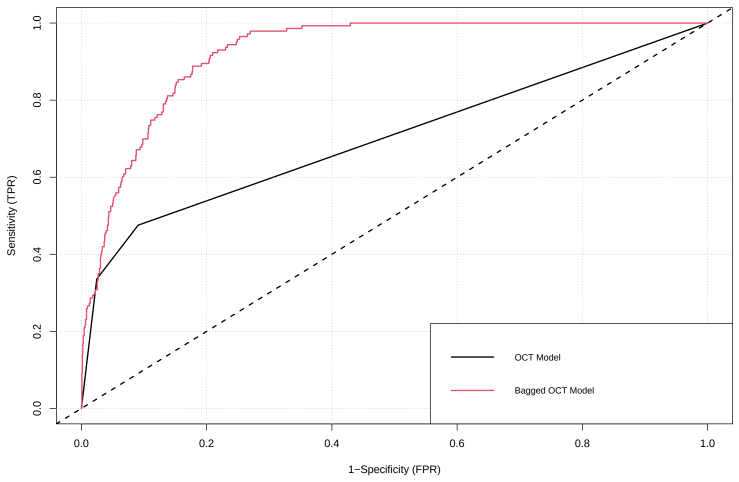


Figure 5.2: ROC Curves comparing OCT with Bagged OCT

predictions (not possible in regression) and loss function that needs changing. Here, one follows [Bertsimas and Dunn, 2019], and Jack Dunn's PhD thesis [Dunn, 2018].

5.5.1 The Optimal Regression Tree Problem

One can reuse most of what we have previously seen here. However, we must also look at how Regression Trees predict values to give us new restrictions. From CART, we have that at each node, one prediction is made. This can be modelled by a continuous variable β_{0t}. Thus one can predict the variables f_i using,

$$f_i = \sum_{t \in T_L} \beta_{0t} z_{it}, \ \forall\ i = 1, \ldots, n$$

Which can be expressed as the following, $\forall\ i \in n$ and $t \in T_L$:

$$f_i \geq \beta_{0t} - M_f(1 - z_{ik}) \tag{5.17}$$

$$f_i \leq \beta_{0t} + M_f(1 - z_{ik}) \tag{5.18}$$

Regression Loss Function

In the regression setting, our loss function is now the difference between the predicted value and the actual value. In our case, either the *Absolute Loss* or the *Squared Loss* can do just fine. Let us take the quadratic loss which linearly can be written as,

$$L_i \geq (f_i - y_i)^2, \ \forall\ i = 1, \ldots, n \tag{5.19}$$

5.5.2 ORT Model

Now, one wants to minimize the loss that we achieve when building the model. With a baseline loss $\hat{L}$, our new model is now:

$$\min \frac{1}{\hat{L}} \sum_{i=1}^{n} L_i + \alpha \sum_{t \in T_B} d_t \tag{5.20}$$

With the restraining conditions of equations 5.1 - 5.5, 5.8 - 5.11, from above, and 5.17 - 5.19.

5.5.3 Hyper-planes

Similar to the case with classification, using MIO's one can now introduce hyper-planes to create decision boundaries. Details can be found in [Bertsimas and Dunn, 2019] and [Dunn, 2018] as per the case for ORT's.

5.6 Summary

In this chapter, we have introduced the brand new concept of Optimized Trees, which I believe is potentially a very valuable approach to building Decision Trees. While some progress has been made, it still has not outperformed regular ensemble methods enough just yet to supplant them as the more advanced form of Tree-Based Models. Applying the method from Operations Research, Branch-and-Bound allows us new possibilities to create many more trees of varying shapes which can only be useful to the development of Tree-Based methods. As a curious side note, the author had also tested using known techniques on new models to test for improvements and it has shown that while close, the new technique still isn't quite as robust as Bagging just yet in regards to predicting more real world data.

Chapter 6

Application using Lahman Dataset

Using data that we have put together (see Appendix), we are able to apply this to a real world scenario. In this case, for Classification, we are trying to predict the likelihood that a baseball team would win their half of the league (Referred to as a Pennant and noted as `LgWin` in Lahman) at the end of the season given the statistics that they had achieved over the season. Here, we are not considering how playoffs affect a teams chance of winning the league due to the randomness of playoffs and that we often find that the best team wins more often than not.

By structure of Major League Baseball (MLB) we can efficiently create a training group based on the American League (AL) and test that with the National League (NL). As both leagues essentially are independent of each other (baring inter-league play which only became prominent in the last couple of years) one can assume a relatively clean separation between training and testing data.

Furthermore, we are able to collect data on players over an individual season as well as over an entire career. Each season's statistics is considered essentially independent of each other, however, it is possible for one to predict future performance based on past data.

On the other hand, we can also group the original Lahman data set by player and year to predict how much a player is paid based on certain statistics. Due to the structure of contracts, it can be observed from A.1 that the <u>mean</u> and <u>modal</u> salary is very low but the range is very high due to the very high salaries of the best and most senior players.

6.0.1 How Decision Trees are used

In modern baseball analytics, Decision Trees are one of the many methods used in evaluating performance for team construction. Many uses of Decision Trees are possible but we will be using only two usages in baseball. This is a small glimpse into what is now referred to as **Sabermetrics**, although more casually, this is called **Moneyball** as a reference to Michael Lewis' book which looked into this very thing.

Classification

The Classification we are using would be predicting the likelihood of a team winning "their" league. In this case, we are using team season data from around 110 seasons to project how well a team has to perform to be considered a team which likely wins, with a winner from each year (baring "1994" when the season was not finished) between 1901-2019. As the split league in the US historically have been small, only getting to 15 teams each since around 1997, we do not have a biased dataset with very low hit rates and therefore one should be comfortable with advancing to building the model based on this. Individually, this usage seems arbitrary but it is in fact the first method a team uses when constructing teams as knowing how well a team has to do is invaluable to informing which players to target based on other models. Our model here is a simplistic version used for a generic season, which unlike a realistic season, does not consider what other teams do as well. Further refinement of the model, such as the use of comparison between teams in each season separately, would be essential in this case, however, as a demonstration of application, it is not necessary for us to get into too much detail there.

Regression

An obvious example for a use of Regression Trees is predicting future salaries based on the previous seasons performance. In real life, salary raises and free agent contracts are often determined by last season's performance which are often used as a good predictor for next season's player output and value to the team. Often, this would involve including time into our data. As this project does not go into that detail, we will simplify the model such that that aspect can be ignored in our example.

Note: Often for regression scenarios, one would first use *linear regression* to see if the model can be easily fit as that is the simplest model we have. Only after doing that would one try other methods such as the tree-based methods we present here.

6.1 Playoff Prediction - Classification

As previously noted, we can split data into two (**AL** and **NL**) without worrying about confounding data (there are no instances where we have split the data such that we have that two teams can win in the same year in our data). The almost identical nature ("minor" rule and park differences notwithstanding) of the two allow for easy comparison. Our first approach is to build a regular Classification Tree using CART, and comparing this model's performance with that of Trees which have different control parameters. From here, we can see the natural progression of trying the alternative techniques shown in Chapters 4 and 5 to compare improvements in the model compared with the base learner.

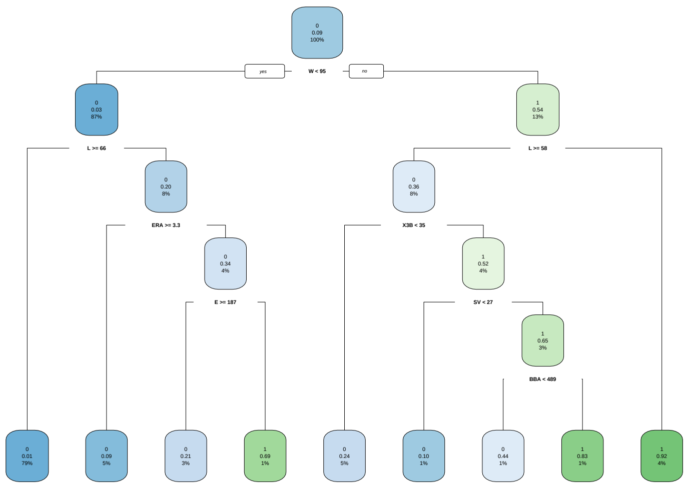


Figure 6.1: CART Decision Tree for Playoff Prediction

6.1.1 Classification Decision Tree

Basic Tree

Given our data is a binary classification task, to be considered better than random, one must expect this model to be more accurate than the known ratio of one prediction

```
[1] "Proportion where LgWin = 0, 0.9067194"
```

The basic Classification Tree for our model is the same one as in figure 3.2 and is built from: We can now make a prediction using NL data resulting in the following Confusion Matrix, where unnaturally, we denoted not winning the league, 0, as the True Positive and winning the league, 1, as our True Negative:

```
nlwinpred
            Model
             0    1
Actual   0 1312   34
         1   97   46

[1] "Accuracy for test 0.912021490933512"
```

Comparing Gini to Information Gain

By [Murthy and Salzberg, 1995], one should expect both Gini and Information Gain based Decision Trees to have similar performance related to accuracy.

Below is the Confusion Matrix for the Information Gain based Decision Tree using the

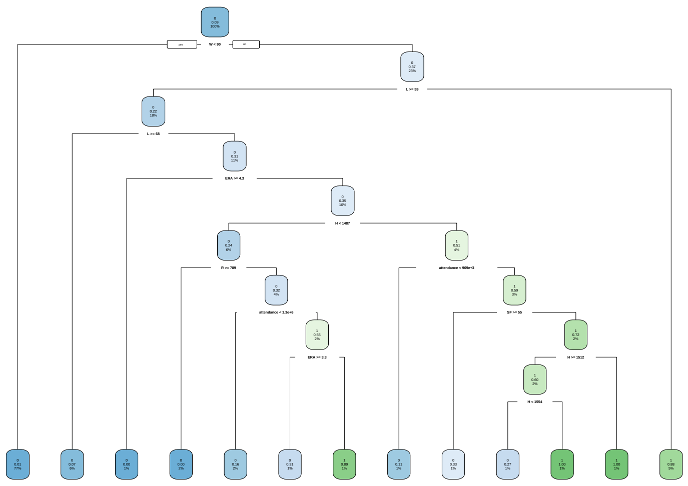


Figure 6.2: C4.5 Decision Tree using Information Gain

same notation as previously:

```
> infotable
   infonlpred
       0    1
  0 1303   43
  1   77   66

[1] "Accuracy for test 0.919408999328408"
```

Looking at the two trees, 6.1 - which uses Gini, and 6.2 - which uses Information Gain and the corresponding Confusion Matrices, one observes that both methods perform similarly as expected in [Murthy and Salzberg, 1995]. Indeed, creating a table showing the two methods, one can see the following:

```
          nlwinpred
infonlpred    0    1
         0 1358   22
         1   51   58

[1] "Accuracy for test 0.950973807924782"
```

One notices that the error seems to be in the edge cases and most of the errors are Type II errors which in this example is good because it is easily possible to have two teams perform very well in the same season even if only one could win.

Now, we can compare the performance of the two models into a table.

Split	**Accuracy**	**AUC**
`gini`	0.9120214909335	0.8329341535136
`information`	0.919408999328408	0.837550265484887

Table 6.1: Table Comparing Tree formed using Gini and Information Gain

Minimum Split	**Accuracy**	**AUC**
50	0.914036265950302	0.719765375783206
40	0.909335124244459	0.836137116969212
20	0.912021490933512	0.832934153513648
15	0.888515782404298	0.821764045761074
10	0.890530557421088	0.784855931586987

Table 6.2: Table Showing Performance of Different Sized Tree Models

As noted by [Murthy and Salzberg, 1995], we observe near identical values with both methods which suggest that we can easily use both splitting criterions without loss of performance.

Changing Tree Size

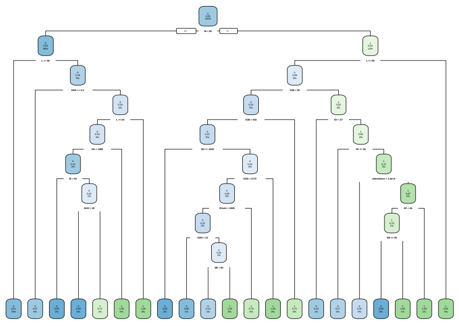
The first thing one can do to try and improve the Decision Tree is to adjust the parameters growing the tree which can change the size of the tree (Figure 6.3). We obtain the following ROC Curves (Fig 6.4) which have been compiled into one chart for comparison purposes.

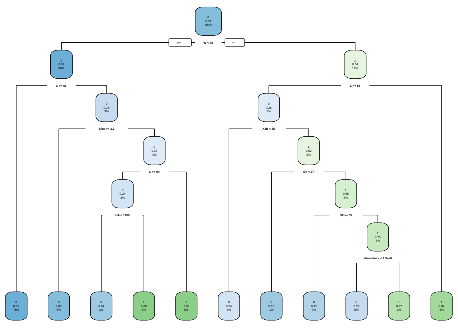
One can see (Table 6.2) how over-fitting a model (as seen when `minsplit = 10` (Fig 6.3a)), that actually, you can make the tree perform worse as one can see with the following AUC results. This is because the tree is very biased in this scenario and therefore one would select a smaller tree which is more flexible in this scenario. Conversely, making the tree too small (`minsplit = 50` (Fig 6.3e)) might mean you hide a very good split which improves the data due to the greedy nature of the algorithm. From Table 6.2, one expects the optimal Decision Tree to be one of figures 6.3b, 6.3c, and 6.3d. More analysis and trial and error should be able to yield the best size for Decision Trees from this point forward.

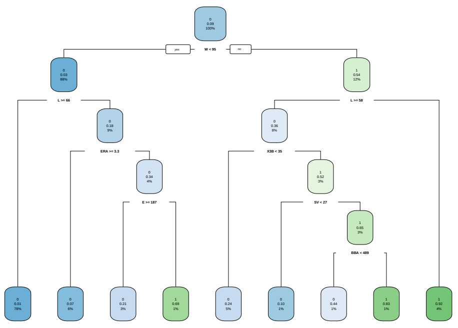
One notices that the accuracy of these models are very similar at around $\approx 90\%$, (likely due to the nature of the model being predicted) and thus in this case, the AUC, which varies significantly (Fig 6.4), is a much better predictor of model accuracy.

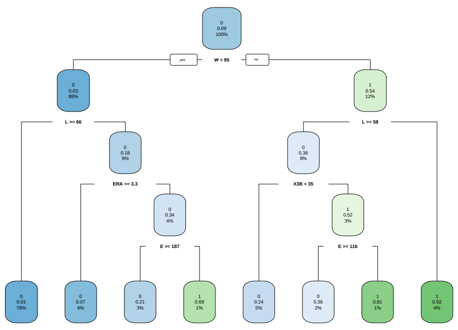
Pruning

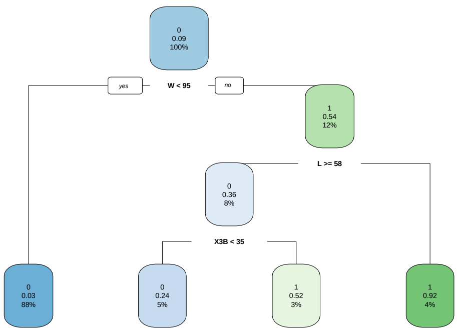
As shown in Chapter 3, pruning simplifies complicated trees. In reality, we would only start using pruning with much larger data sets and Decision Trees (Tree depth 20+ would often be needed to achieve a better result with pruning than growing a small tree


(a) Classification Tree `minsplit = 10`


(b) Classification Tree `minsplit = 15`


(c) Classification Tree `minsplit = 20`


(d) Classification Tree `minsplit = 40`



(e) Classification Tree `minsplit = 50`

Figure 6.3: Varying the size of Classification Trees

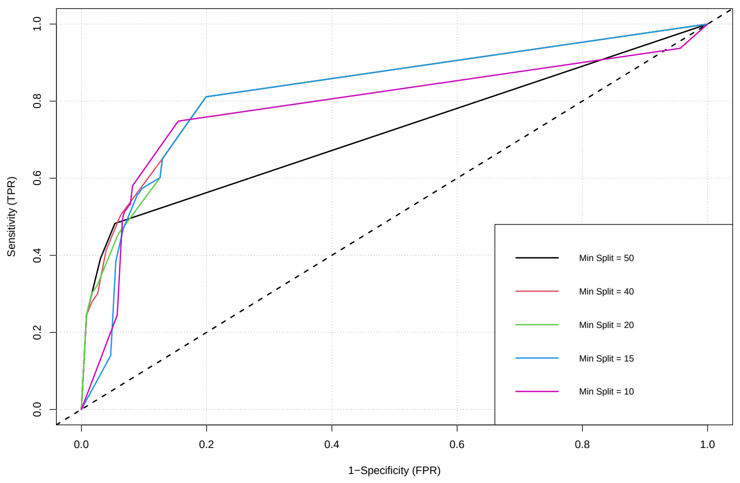


Figure 6.4: ROC Curves for different sized Decision Trees

Model	**Accuracy**	**AUC**
Pruned CART	0.920080591000672	0.71908737621962

Table 6.3: Results from Pruned Tree

and using Minimum Split as the stopping criterion).

Comparing Tables 6.1 and 6.3, we notice that while the accuracy score remains similar (or in our case improves) the AUC score decreased as fewer nodes leads to fewer probabilities for ROC curves to be mapped onto. Looking at the confusion matrix (See R Code in Appendix, we observe that most of the improvements in accuracy was achieved by having fewer “Type I” errors but having an increase in the number of “Type I” errors. This could be ideal in scenarios where one wants to be absolutely sure of something so much that only the best results give a sure result.

6.1.2 Additive Models - Bagging/Boosting

We now try to see if the “black box” methods in Chapter 4 provide improvements in accuracy of predictions to the model. By nature of these models, we lost the interpretability of the model, but hope to gain accuracy compared to a regular Decision Tree. We will be using Bagging, Random Forests, AdaBoost and xgBoost and comparing it with CART as we have seen previously.

Comparison of Models

Table 6.4 shows, additive ensemble models performed much better than normal CART as expected. An interesting aspect is how Bagging and Random Forests perform com-

Method	Implementation	Accuracy	AUC
CART	`rpart`	0.9120214909335	0.8329341535136
Pruning	`CART`	0.9200806	0.719087376220
Bagging	`ipred`	0.9106783075890	0.9225080268914
RF Bagging	`randomForest(mtry=38)`	0.9106783075890	0.923666600858
Random Forest	`randomForest`	0.9153794492948	0.9199363044088
AdaBoost	`adabag`	0.9066487575554	0.9155955485822
xgBoost	`xgbTree`	0.9180658159839	0.9265682311745
xgBoost	`xgboost`	0.9126930826058	0.9272540238364
OCT	`IAI`	0.9140363	0.7018256631927
Gini OCT	`IAI`	0.8932169	0.8238525961409
Bagged OCT	`IAI + caret`	0.9120215	0.9237860950342

Table 6.4: Table showing performance of different Tree-Based Models

paratively well to the Boosting methods. It is clear to see why losing “interpretability” was a necessary for those looking for models with perfect predictions.

6.1.3 Using Optimizing Methods

Using the above scores from Tables 5.1 and 5.2, one can see the following scores (Table 6.4). One can see that OCT actually performs more similarly to the Pruned Tree than the regular tree in terms of AUC score, which while not expected based on claims by Bertsimas and Dunn on their model, is not unrealistic due to the small tree size. While one expects the newer method to perform better than the status quo, it is likely that the nature of the data and the small tree size has led to this poor output. Dunn had noted in [Bertsimas and Dunn, 2017] that as the depth of the tree increase, so does the performance of the model. Maybe some curious reader with a very powerful computer can experiment with this. One expects playing around with the parameters would lead to some improvements to the model.

One also notices that using Gini (see [Bertsimas and Dunn, 2017] and Figure 6.6) as the criterion for labelling our incorrect predictions instead of just regular misclassification labelling improves our model. This supports what we said earlier about how Gini is a much better metric for tree-based models than misclassification error.

6.1.4 All ROC Curves

Summing up all of the above (Figure 6.8), we can compare all these ROC together to see how each model compares with each other. One can see that with our dataset, it may be better for us to sacrifice interpretability for accuracy if one wants the best predictive model. However, using Gini as a criterion, one can create an interpretable model which sacrifices a bit on accuracy but is still good enough for us to use as a flexible basis.

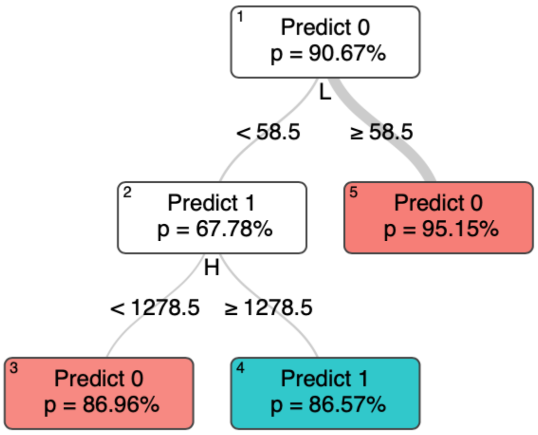


Figure 6.5: OCT

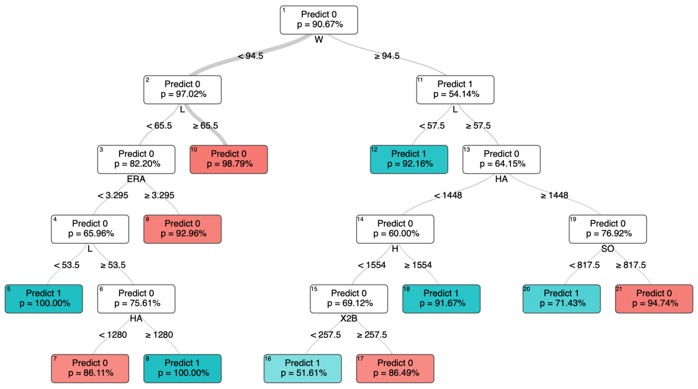


Figure 6.6: Gini OCT

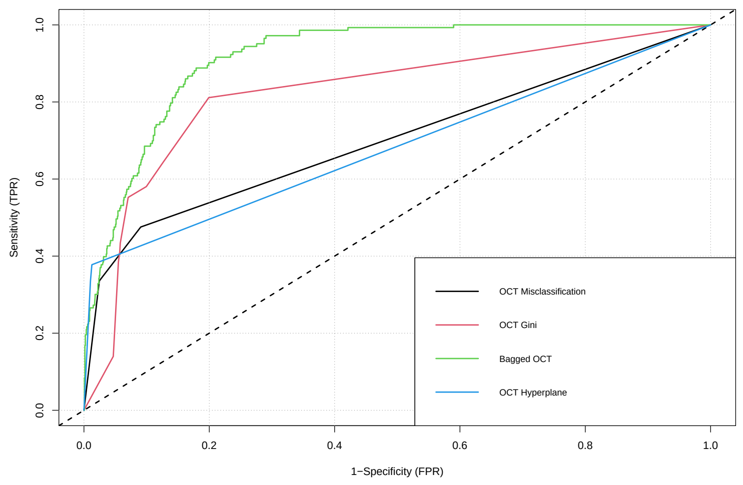


Figure 6.7: ROC curves for all OCT models

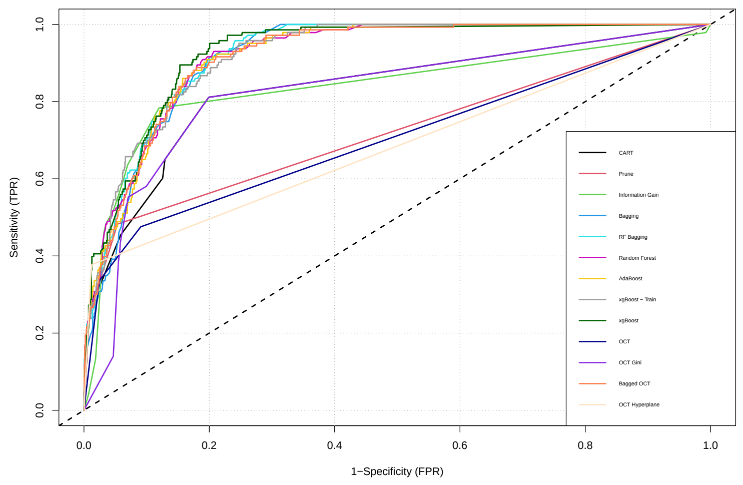


Figure 6.8: Summary of all ROC curves

6.2 Salary Prediction - Regression

Similarly to above, one can also see the natural progression of Regression Trees from the base tree to something much more complex. In this case, we will not go into as much detail as we did for classification, however, the code will be left in the Appendix for the reader to adjust if one were curious about how changing parameters would affect the model.

In this case, we will be looking at player salaries for a given year and predicting how much value a player gave the team given their outputs. This is useful for teams as a player's statistics do not vary as much each year and so a can predict salaries easily which is helpful for team construction.

6.2.1 Linear Regression

Before we proceed, one should always check to see if linear regression is already a suitable model. We find the following result provides the best linear fit.

Linear Model

After using step-wise methods to reduce the model from full size, one finds that our model can be fitted using,

```
reglmstep <- lm(formula = salary/1e+06 ~ stint + G + H + X3B + HR + RBI
                + BB + SO + IBB + SH, data = salarybatting)
```

Which results in an RMSE score of $\$4,302,863$. When plotting to check how well the model predicted against the actual salary however, we find that our model here performs very poorly as seen in Figure 6.9. This is in fact a very poor predictor. We notice that the correlation between the data is very low with $R^2 = 0.2036$, which suggests that a non-parametric method would result in a much better fit than any linear model would.

Given this information, we will now set this as a baseline to test whether our methods lead to improvements relative to linear regression. One now proceed onto Tree-Based methods in hopes of finding a better model which results in a small RMSE.

6.2.2 Regression Tree

Figure 6.10 shows a Regression Decision Tree for baseball players salaries using data from the `Lahman` Dataset (Lahman) for the year 2010. We note that the accuracy of this predictor can be found by calculating the Residual Mean Square Error (RMSE) value. In this case we used data from the next year (2011) as a test dataset and achieved the following RMSE scores (Table 6.5)

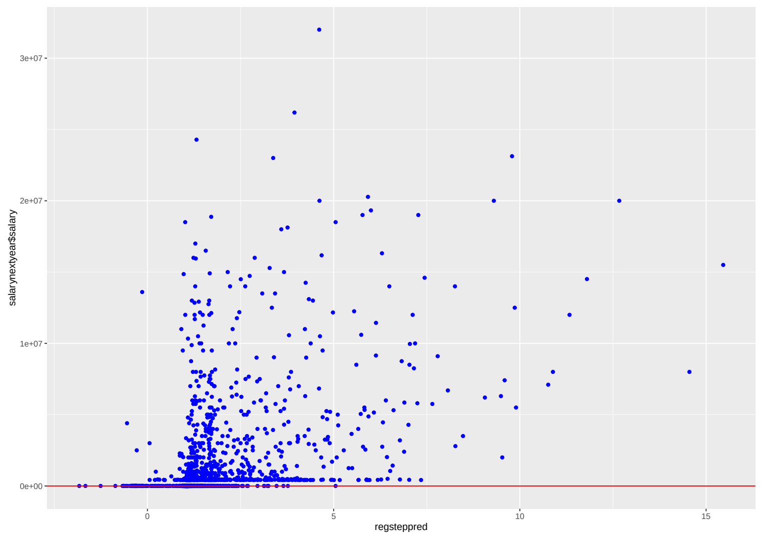


Figure 6.9: Plot of Predicted vs Actual Salaries

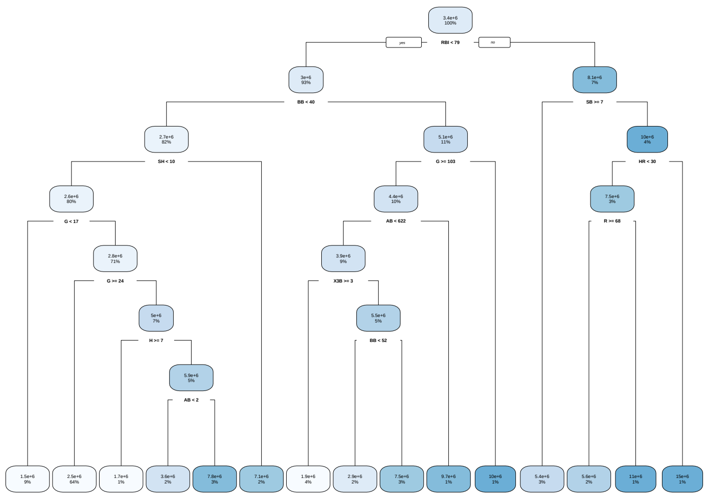


Figure 6.10: Regression Decision Tree

Predicted RMSE	\$3,479,857
Multiple R-squared	0.2036
Residual standard error	3.434 with 1345 df

Table 6.5: Scores from Regression Tree Model

Model	RMSE
Linear Regression	$4,302,863
CART	$3,479,857
Pruned CART	$3,492,685
Bagging	$3,298,992
Random Forest	$3,299,106
xgBoost	$3,386,000
Optimal Regression Tree	$3,561,030
ORT Hyperplane	$3,561,030

Table 6.6: Table showing RMSE scores of all models in our regression example for salary

6.2.3 Alternative Tree-Based Regression Methods

As we have done for classification, we are able to build multiple models using different techniques based off the same salary dataset. We are able to use the exact same libraries as before and thus it seems trivial to extend our application to other tree based methods. Table 6.6 provides us with a summary of all the Root Mean-Squared Error (RMSE) scores for all models for comparison.

The first thing one notices is how well CART already performs by itself without adding anything on top of it. We also see that the variance reducing methods of Bagging and Random Forests provide the best improvement to the model. As salaries are highly variable, this is to be expected, although its key to note that the improvement is not that great when compared with the improvement one gets from just moving from linear regression to tree methods.

An interesting factor is that the modern Optimal Regression Tree Methods (Figure 6.11 and 6.12) had the highest RMSE score out of all the tree-based models.

In a real world setting, one would also notice how all these methods are much better than the predictions we had with linear regression. Therefore, it is possible that one can use any of these methods to create prediction models.

6.3 Summary

We have now implemented Tree-Based methods with real world data and have been able to test the performance of these models with each other. One notices the similarities in performance scores with all the Ensemble Models from Chapter 4, whereas the newer method of Optimized Trees performed slightly worse than expected, being closer in performance to regular Decision Trees rather than the more complex algorithms that this new method is trying to out-compete. Nevertheless, we have shown than in general, Tree-Based Models are very good predictors and that they any of them are good enough for all of us to use.

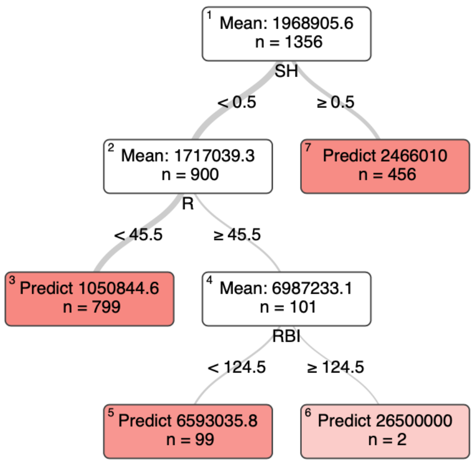

1 Mean: 1968905.6 n = 1356
SH
< 0.5
≥ 0.5
2 Mean: 1717039.3 n = 900
7 Predict 2466010 n = 456
R
< 45.5
≥ 45.5
3 Predict 1050844.6 n = 799
4 Mean: 6987233.1 n = 101
RBI
< 124.5
≥ 124.5
5 Predict 6593035.8 n = 99
6 Predict 26500000 n = 2

Figure 6.11: Optimal Regression Tree

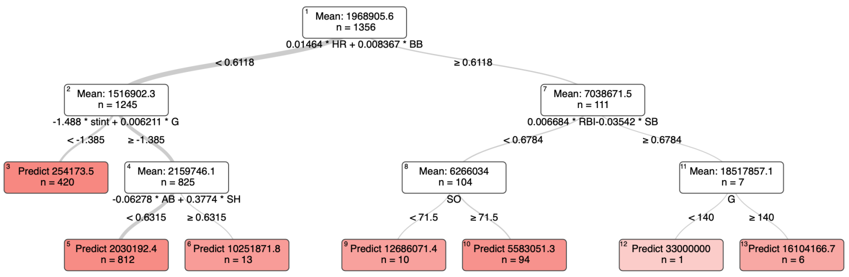

1 Mean: 1968905.6 n = 1356
0.01464 * HR + 0.008367 * BB
< 0.6118
≥ 0.6118
2 Mean: 1516902.3 n = 1245
-1.488 * stint + 0.006211 * G
< -1.385
≥ -1.385
7 Mean: 7038671.5 n = 111
0.006684 * RBI-0.03542 * SB
< 0.6784
≥ 0.6784
3 Predict 254173.5 n = 420
4 Mean: 2159746.1 n = 825
-0.06278 * AB + 0.3774 * SH
< 0.6315
≥ 0.6315
8 Mean: 6266034 n = 104
SO
< 71.5
≥ 71.5
11 Mean: 18517857.1 n = 7
G
< 140
≥ 140
5 Predict 2030192.4 n = 812
6 Predict 10251871.8 n = 13
9 Predict 12686071.4 n = 10
10 Predict 5583051.3 n = 94
12 Predict 33000000 n = 1
13 Predict 16104166.7 n = 6

Figure 6.12: Optimal Regression Tree with Hyperplane

Chapter 7

Conclusion

One can see that Tree-Based Methods, in general, are a very strong Supervised Statistical Machine Learning Method. They are very robust and predict reasonably well with regards to accuracy. They are also one of the best base learning methods as we can use trees as a base for all the methods we have seen.

We have also observed that the interpretability of models, while useful in many cases, often lend to loss of accuracy as they tend to not be as flexible on the overall predictor. While they still maintain their place in modern Machine Learning, it may be better for one to avoid using it unless you have a specific visual reason to use it. Another note is that the modern Optimal Tree method which was built as an attempt to optimize trees using Mixed-Integer Programming methods still do not outperform the more traditional "blackbox" methods that we have today, however, the potential is there and some minor changes to flexibility with regards to high bias and high variance models may be needed before more typical use.

Applying the methods we have seen in practical terms as we have done in the previous chapter, we can now use one of the models that we have fitted and apply it to new unseen data. This is what happens in the real world with propensity models in more than just baseball. In terms of baseball this is a great way to show how teams decide on where a team needs to improve to increase their chances of being successful and then onto how much a team should spend to achieve these goals they have set for themselves. In real life, much more depth is involved and many more models and methods would be used but this is already a great start in terms of how teams actually operate.

To conclude, we note that there have been many attempts to try and improve upon Decision Trees, many which have sacrificed interpretability in terms of accuracy, or have required more complex computing methods, yet CART and its equivalents have shown here why it is still one of the go-to methods Data Scientists use for prediction. I hope that this project has given an insight into Tree-Based methods and look forward to future advancements in technology allowing us to improve upon the work by Breiman and Friedman in creating accurate and interpretable predictors.

Acknowledgments

Firstly, I would like to that my supervisors Professor Peter Craig and Dr Louis Aslett for help and guidance with this project, without whom, I would not be writing this project. They were very much supportive and always on hand when I needed their help and without their time, this project would've been unfeasible

I would also like to thank Daisy Zhuo and the rest of the Interpretable AI `IAI` team for providing me access to the software needed for me to be able to try out the new method that we will be implementing in this project **The Optimal Classification/Regression Tree**.

Due to the computationally taxing nature of MIO's that were used in Optimized Trees, it was crucial to this project that I was able to access the software they have built for this purpose along with many others. Without access to this software, one would've had to use `Gurobi` or other programming optimization software which I was unable to access as easily as `IAI`.

Bibliography

[Ali et al., 1975] Ali, M. A., Hickman, P. J., and Clementson, A. T. (1975). The application of automatic interaction detection (aid) in operational research. *Operational Research Quarterly (1970-1977)*, 26(2):243–252.

[Arthanari and Dodge, 1981] Arthanari, T. and Dodge, Y. (1981). *Mathematical Programming in Statistics.* Wiley.

[Bertsimas and Dunn, 2017] Bertsimas, D. and Dunn, J. (2017). Optimal classification trees. *Machine Learning*, 106.

[Bertsimas and Dunn, 2019] Bertsimas, D. and Dunn, J. (2019). *Machine Learning Under a Modern Optimization Lens.* Dynamic Ideas LLC.

[Bixby et al., 2004] Bixby, R., Fenelon, M., Gu, Z., Rothberg, E., and Wunderling, R. (2004). *Mixed-Integer Programming: A Progress Report*, pages 309–324. SIAM.

[Breiman, 1996] Breiman, L. (1996). Bagging predictors. *Machine learning*, 24(2):123–140.

[Breiman, 2001] Breiman, L. (2001). Random forests. *Machine learning*, 45(1):5–32.

[Breiman et al., 1984] Breiman, L., Friedman, J. H., Olshen, R. A., and Stone, C. J. (1984). *Classification and Regression Trees.* Wadsworth and Brooks, Monterey, CA.

[Chen and Guestrin, 2016] Chen, T. and Guestrin, C. (2016). XGBoost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 785–794, New York, NY, USA. ACM.

[Dunn, 2018] Dunn, J. (2018). *Optimal trees for prediction and prescription.* PhD thesis, Massachusetts Institute of Technology Operations Research Center.

[Efron and Tibshirani, 1993] Efron, B. and Tibshirani, R. J. (1993). *An Introduction to the Bootstrap.* Number 57 in Monographs on Statistics and Applied Probability. Chapman & Hall/CRC, Boca Raton, Florida, USA.

[Freund and Schapire, 1997] Freund, Y. and Schapire, R. E. (1997). A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of computer and system sciences*, 55(1):119–139.

[Friedman, 1999a] Friedman, J. H. (1999a). Greedy function approximation: A gradient boosting machine. *The Annals of Statistics*, 29(5):1189–1232.

[Friedman, 1999b] Friedman, J. H. (1999b). Stochastic gradient boosting. *Computational statistics & data analysis*, 38.

[Friendly et al., 2020] Friendly, M., Dalzell, C., Monkman, M., and Murphy, D. (2020). *Lahman: Sean 'Lahman' Baseball Database*. R package version 8.0-0.

[Hastie et al., 2009] Hastie, T., Tibshirani, R., and Friedman, J. H. (2009). *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer, New York.

[Horst et al., 2020] Horst, A. M., Hill, A. P., and Gorman, K. B. (2020). *palmerpenguins: Palmer Archipelago (Antarctica) penguin data*. R package version 0.1.0.

[James et al., 2013] James, G., Witten, D., Hastie, T., and Tibshirani, R. (2013). *An Introduction to Statistical Learning : with Applications in R*. Springer, New York.

[Kass, 1980] Kass, G. V. (1980). An exploratory technique for investigating large quantities of categorical data. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 29(2):119–127.

[Khan and Brandenburger, 2020] Khan, M. R. A. and Brandenburger, T. (2020). *ROCit: Performance Assessment of Binary Classifier with Visualization*. R package version 2.1.1.

[Little et al., 1963] Little, J. D. C., Murty, K. G., Sweeney, D. W., and Karel, C. (1963). An algorithm for the traveling salesman problem. *Operations Research*, 11(6):972–989.

[Loh, 2014] Loh, W.-Y. (2014). Fifty years of classification and regression years. *International Statistical Review*, 82:329–348.

[Loh and Vanichsetakul, 1988] Loh, W.-Y. and Vanichsetakul, N. (1988). Tree-structured classification via generalized discriminant analysis. *Journal of the American Statistical Association*, 83:715–728.

[Messenger and Mandell, 1972] Messenger, R. and Mandell, L. (1972). A modal search technique for predictibe nominal scale multivariate analys. *Journal of the American Statistical Association*, 67(340):768–772.

[MLWiki, 2020] MLWiki (2020). Cost-complexity pruning.

[Morgan and Sonquist, 1963] Morgan, J. N. and Sonquist, J. A. (1963). Problems in the analysis of survey data, and a proposal. *Journal of the American Statistical Association*, 58(302):415–434.

[Murthy and Salzberg, 1995] Murthy, S. K. and Salzberg, S. (1995). Decision tree induction: How effective is the greedy heuristic? In *KDD*, pages 222–227.

[Quinlan, 1986] Quinlan, J. R. (1986). Induction of decision trees. *Machine Learning*, 1:81–106.

[R Core Team, 2020] R Core Team (2020). *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, Vienna, Austria.

[Ripley, 2019] Ripley, B. (2019). *tree: Classification and Regression Trees.* R package version 1.0-40.

[Ritschard, 2013] Ritschard, G. (2013). Chaid and earlier supervised tree methods. *Contemporary Issues in Exploratory Data Mining in Behavioral Sciences.*

[Ruopp et al., 2008] Ruopp, M. D., Perkins, N. J., Whitcomb, B. W., and Schisterman, E. F. (2008). Youden index and optimal cut-point estimated from observations affected by a lower limit of detection. *Biometrical journal. Biometrische Zeitschrift*, 50(3):419.

[Schwab, 2016] Schwab, K. (2016). *The Fourth Industrial Revolution.* World Economic Forum, Geneva.

[Shannon and Weaver, 1948] Shannon, C. E. and Weaver, W. (1948). The mathematical theory of communication. *Bell Syst. Tech. J*, 27(3):379–423.

[Therneau and Atkinson, 2019] Therneau, T. and Atkinson, B. (2019). *rpart: Recursive Partitioning and Regression Trees.* R package version 4.1-15.

[Wickham, 2016] Wickham, H. (2016). *ggplot2: Elegant Graphics for Data Analysis.* Springer-Verlag New York.

[Wickham et al., 2020] Wickham, H., Franois, R., Henry, L., and Mller, K. (2020). *dplyr: A Grammar of Data Manipulation.* R package version 1.0.2.

[Wilkinson, 1992] Wilkinson, L. (1992). Tree structured data analysis: Aid, chaid and cart. *Systat.*

Appendices

Appendix A

Extracting Training and Testing Datasets

We have selected four sets of data to work on this project, one each for classification and regression. Below is the code for the data and the variables we have selected in each.

Playoff - Classification

The following is to predict whether a team would win the league given how well a team did throughout the season. Here is the code to isolate the data:

Training Data

```
teamseason <- Lahman::Teams
head(teamseason)
alplayoff <- filter(teamseason, teamseason$lgID =="AL")
head(alplayoff)
is.factor(alplayoff)
alplayoff[alplayoff =="N"] = 0
alplayoff[alplayoff =="Y"] = 1
head(alplayoff)
alplayoff %>% filter(!is.na(LgWin))
alplayoff1 = subset(alplayoff,
                    select = -c(Rank, park, WSWin,
                                name, teamID, franchID, teamIDBR,
                                teamIDlahman45, teamIDretro))
alplayoff1$LgWin <- as.factor(alplayoff1$LgWin)
alplayoff1[is.na(alplayoff1)] <- 0
```

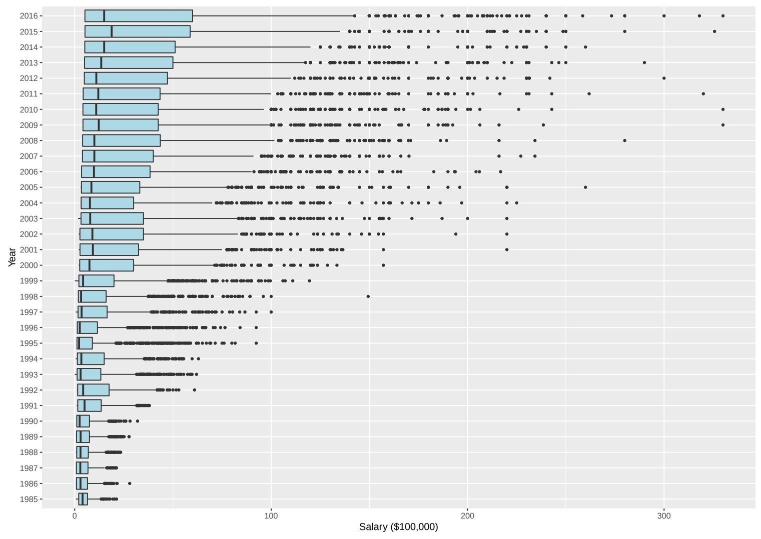


Figure A.1: Baseball Salaries over the years

Testing Data

```
nlplayoff <- filter(teamseason, teamseason$lgID =="NL")
head(nlplayoff)
is.factor(nlplayoff)
nlplayoff[nlplayoff =="N"] = 0
nlplayoff[nlplayoff =="Y"] = 1
head(nlplayoff)
nlplayoff %>% filter(!is.na(LgWin))
nlplayoff1 = subset(nlplayoff,
                    select = -c(Rank, park, WSWin,
                                name, teamID, franchID, teamIDBR,
                                teamIDlahman45, teamIDretro))
nlplayoff1$LgWin <- as.factor(nlplayoff1$LgWin)
nlplayoff1[is.na(nlplayoff1)] <- 0
```

Salary - Regression

We use data from 2010 to predict a players salary in 2011 given how well the player did. Here is the code to isolate the data:

Training Data

```
batting <- Batting %>%
  filter(yearID == 2010) %>%
  left_join(select(Salaries, playerID, yearID, teamID, salary),
            by=c("playerID", "yearID", "teamID"))
str(batting)

batting %>%
  filter(! is.na(salary))

names(batting)
salarybatting = subset(batting, select = -c(playerID, teamID, lgID))
salarybatting[is.na(salarybatting)] <- 0
```

Testing Data

```
nextyear <- Batting %>%
  filter(yearID == 2011) %>%
  left_join(select(Salaries, playerID, yearID, teamID, salary),
            by=c("playerID", "yearID", "teamID"))
str(nextyear)

nextyear %>%
  filter(! is.na(salary))

names(nextyear)
salarynextyear = subset(nextyear, select = -c(playerID, teamID, lgID))
salarynextyear[is.na(salarynextyear)] <- 0
```

Appendix B

Codes for Model Creation

The following is the README.md file from the GitHub repository where all the coding files have been hosted.

codeproject

Repository for Code for my 3rd Year Project on Statistical Machine Learning (4IR)

I have uploaded my code onto GitHub to reduce the file size of the overall pdf that I will have to submit. The repository for the codes I have used to create the models and plots for the tree-based methods can be found in the following repository:

https://github.com/luangamornlertp/codeproject/

The codes have been split into two similar to that in chapter 6 of this report, CLASSIFICATION and REGRESSION which makes each dataset easy to follow.

Furthermore, one can find the TeX file for the report in the following repository:

https://github.com/luangamornlertp/Project/maintreedraft.tex

The reader is free to download the codes and play around with it if they so choose.