Using Ranking Loss Functions to Prune Tree-Based Machine Learning Algorithms

Bradley Klassen BRADKLASSEN@OUTLOOK.COM
Department of Mathematics & Statistics
Brock University
St. Catharines, ON, L2S 3A1, CAN

Supervisor: Dr. William Marshall

Abstract

Most common decision tree programming packages use a mean squared error (MSE) loss function. However, MSE may not always be optimal, especially for a ranking problem. Some problems may benefit from custom loss functions. Is a ranking loss function optimal for a ranking problem? By building decision tree and AdaBoost models from scratch, it is possible to customize the pruning loss function used in the models. A simulation study found that Average Precision at k outperformed MSE for the decision tree model. While, MSE outperformed Average Precision at k for the AdaBoost model. Finally, the models were applied to professional golf data where the results aligned with the simulation study.
Keywords: Decision Tree, AdaBoost, Learning to Rank, Loss Function, Sports Analytics

Contents

1. Introduction

Ranking in machine learning has become increasingly popular with the amount of information growing exponentially. In 2020 alone, Google generated $104 billion in "search and other" revenues (Graham and Elias, 2021). The ordering of search results is crucial for providing relevant information to the user in the fastest way possible. In September 1998, Google filled a patent for the PageRank algorithm (Page et al., 1999), which was the first algorithm used by the company to sort results from a users search. Ranking algorithms are also very important for predicting performance in some individual sports such as Nascar and Golf. Many athletes compete in the same event in hopes of coming first. The awards are often given based on the ordering of the individual and not their score.

Some regression problems may benefit from various loss functions beyond the most popular function, Mean Squared Error (MSE). The regression loss function chosen for the problem may vary based on the type of errors the problem is trying to minimize. If the goal is to minimize the number of individual large errors made, MSE may be a good choice due to the errors being squared. However, if the goal is to minimize the average loss, Mean Absolute Error (MAE) may be a better choice as the errors are more equally weighted. If regression loss functions are chosen based on the error that is trying to be minimized, it makes sense to align the loss function with the problem. If we are solving a ranking problem, it is likely better to use a ranking loss function within the models.

Some existing programming packages allow a user to select the loss function used to build a decision tree or AdaBoost model. However, the user is typically given a small subset of loss functions to choose from. The Python library sklearn provides five regression loss functions as options, which include, MSE, MAE and others (Pedregosa et al., 2011). None of the given options are ranking loss functions. During my research, I did not come across any packages that allow a user to customize the loss function used to prune the tree after it is built. As a result, a decision tree and AdaBoost model needed to be built from scratch to allow for the customization of the pruning loss function. The Python code for these models can be found in Appendix C.

This project explores using MSE as a regression metric and Average Precision at k (AP@k) as a ranking metric for pruning loss functions in tree-based machine learning algorithms. A simulation study was first created to test and evaluate the decision tree and AdaBoost models. The models were then tested in an applied scenario for predicting rankings of golfers at professional golf tournaments.

1.1 Machine Learning

Famous computer scientist Arthur Samuel once defined machine learning as, "The field of study that gives computers the ability to learn without being explicitly programmed." There are three main types of machine learning algorithms, supervised learning, unsupervised learning and reinforcement learning. This project uses supervised learning.

The goal of supervised learning is to use the inputs to predict the values of the outputs (Hastie et al., 2017). Supervised learning is expressed in The Elements of Statistical Learning as the following, "Our goal is to find a useful approximation f(x) to the function f(x) that underlies the predictive relationship between the inputs and outputs" (Hastie et al., 2017). The two main types of supervised learning include, regression and classification. A

regression problem involves predicting a numerical output(s), whereas a classification problem involves predicting a class label (binary or multi-class). Some algorithms are designed to handle both regression and classification problems, including the decision tree and AdaBoost models used in this paper. Regression and classification models can be divided into two further sub-sections, linear and non-linear. Some of the most popular linear models include, Linear Regression, Logistic Regression and Linear Discriminant Analysis. Some of the most popular non-linear models include, Decision Trees, Support Vector Machines and Gradient Boosting models such as XGBoost (Chen and Guestrin, 2016), AdaBoost (Freund et al., 1999) and others.

1.2 Learning to Rank

Machine learning applications to ranking problems is commonly referred to as Learning to Rank (LTR). There are three main approaches to LTR, pointwise, pairwise and listwise. The pointwise approach trains a model to predict performance and then sorts and ranks the predicted outputs. Standard regression algorithms can be directly used. This project uses a pointwise approach. The pairwise approach analyzes a pair of objects at one time and finds the optimal ordering of the pair. Some example models include, RankBoost (Freund et al., 2003), RankNet (Burges et al., 2005) and LambdaMART (Burges, 2010). The listwise approach directly looks at the entire list of documents to decide on an optimal ordering. Example models include AdaRank (Xu and Li, 2007a), SoftRank (Taylor et al., 2008) and ListNet (Cao et al., 2007). The listwise model AdaRank has been shown to outperform pairwise approaches such as RankBoost (Xu and Li, 2007b).

1.3 Golf Analytics

In the age of big data, the field of sports analytics is becoming increasingly transdisciplinary, combining domain specific knowledge from sports management with the statistical and computational tools of data science. Golf is a sport that generates massive amounts of data with no shortage of opportunities for analysis. The sport presents a unique opportunity to use supervised learning for ranking.

The objective of golf is for an individual to hit a ball into a hole in the least number of strokes possible. A typical golf course consists of 18 holes and has a par of 72. Meaning, 72 strokes is the predetermined number of strokes that a proficient golfer should take to complete a round. It is estimated that less than 1% of golfers shoot par on average (usg). The PGA Tour is the organizer of the main professional golf tournaments and is recognized as the top professional league in the world (pga). In a typical PGA Tour tournament, approximately 150 players compete to shoot the lowest score. The number of strokes taken by each player throughout the four days determines the players ranking in the tournament.

In 2003 the PGA Tour introduced an advanced ball tracking system called ShotLink. The ShotLink System is a revolutionary platform for collecting and disseminating scoring and statistical data on every shot by every player in real-time (Bryant). The ShotLink program has provided the opportunity for the tour to gain insights into the specifics of how the players are hitting the ball.

Predicting upcoming performance can be done in two ways, either by predicting the number of strokes (regression), or the tournament rankings (ranking). In the game of golf,

the prize money is allocated based on the ranking of an individual and not their score. As a result, the application of the algorithms will explore predicting the players tournament rankings.

2. Methods

The following section will discuss the machine learning algorithms and loss functions that are used in this project.

2.1 Tree-Based Algorithms

Tree-based machine learning algorithms are some of the most popular supervised algorithms due to their predictive power, adaptability and interpretability. Many of these algorithms are capable of dealing with both regression and classification problems and are often regarded for their ability to map non-linear relationships quite well.

2.1.1 Decision Tree

Decision trees are a supervised machine learning algorithm that embodies an explicit representation of the structure in a data set. They explore the structure of the data by developing an easy to visualize decision rule for predicting outcome. Decision trees are constructed via an algorithmic approach that identifies a way to split a data set based on different conditions. The data used to build the tree is stored in the root node and gets divided through branches to create new nodes. This process continues until a stopping criteria is met. An example of a decision tree with a depth of two is highlighted in Figure 1. The data was collected by the U.S Census Service consisting of housing information in Boston, Massachusetts (Harrison and Rubinfeld, 1978). The target variable is price, which is a continuous value, so a regression decision tree is built. All combinations of the column and row value are tested as a potential split, and the combination that has the smallest error is chosen as the split. As shown below, in the root node there are 506 observations and the data is split on the fifth column and on the value of 6.941. Any observation where the fifth column has a value less than or equal to 6.941 will go to a new node on the left. Any observation where the fifth column has a value greater than 6.491 will go to a new node on the right. This process is repeated until a stopping criteria is met. The nodes that have a child node are referred to as an internal node. The nodes that do not have a child node are referred to as a leaf node. The larger the depth of the tree, the more leaf nodes, resulting in a larger number of unique predictions.

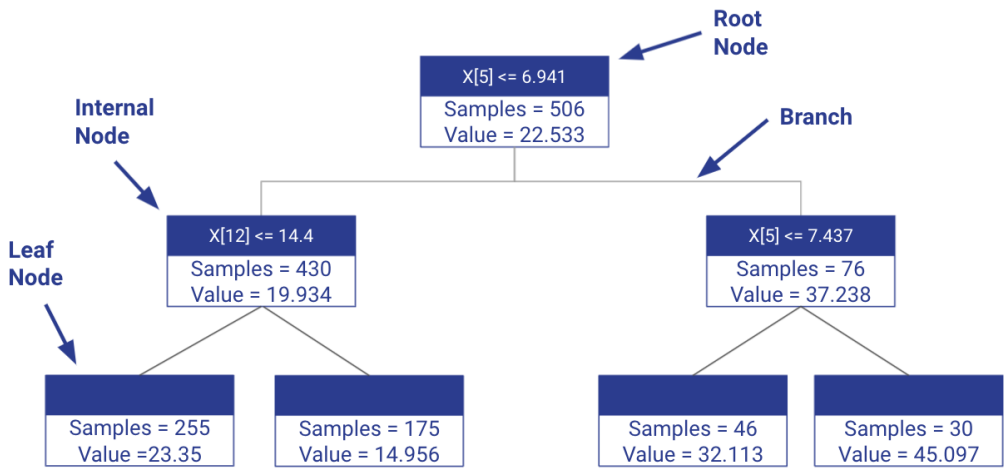

Figure 1: Regression decision tree built using the Boston Housing data set

There are multiple types of decision tree algorithms, the most popular include, CART (Classification and Regression Decision Trees) (Breiman et al., 2017), ID3 (Iterative Dichotomiser 3) (Quinlan, 1986) and C4.5 (successor of ID3) (Quinlan, 2014). These algorithms use different loss functions to decide on a split. ID3 and C4.5 can only be used for classification problems, whereas CART can be used for both classification and regression. A CART implementation is used in this project.

Pruning

Decision trees are a very popular algorithm, largely due to their interpretability. However, large decision trees tend to overfit. This means that the model is able to capture the detail and noise of the training data well, to the point of negatively impacting its performance on new data (Brownlee, 2019). The process of pruning a tree reduces the complexity of the model and improves the prediction accuracy by removing certain parts of a tree to reduce its size. Reducing the size of a decision tree may slightly hinder the performance on the training data but will likely increase the performance on unseen data. This is known as the bias-variance tradeoff. "Variance refers to the amount by which $\hat{f}$ would change if we estimated it using a different training data set" (James et al., 2021). "Bias refers to the error that is introduced by approximating a real-life problem, which may be extremely complicated, by a much simpler model." (James et al., 2021). There is a direct relationship between the bias and the variance. Decreasing the bias will increase the variance. Decreasing the variance will increase the bias.

Minimal Cost-Complexity (MCCP)

There are multiple ways to prune a regression decision tree, but the most popular method is Minimal Cost-Complexity Pruning (MCCP). First, lets define an overall cost-complexity measure of a tree T to be $R_\alpha(T)$.

$$R_\alpha(T) = \text{SSR} + \alpha \cdot T$$

Where, SSR is the Sum of Squared Residuals (training error) of the tree, α is a regularization parameter, and T is the number of leaf nodes in the tree.

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

Calculate the SSR of the full tree, then the SSR of the tree with one node pruned, and so on, until all nodes are pruned. Record the SSR values for each corresponding tree.

The tree complexity parameter $\alpha \cdot T$ adds a penalty for trees with a larger number of leaf nodes. When $\alpha = 0$, the full sized tree is chosen. As α increases, the penalty increases. As a result, the α value chosen directly affects the final pruned tree that is selected. Start with $\alpha = 0$, then increase the α value until pruning leaves of the tree will give a lower $R_\alpha(T)$. Repeat this process until all nodes are pruned. Record the α values each time a new tree is created. There then exists a subset of α values that gives a sequence of trees (Starmer, 2019).

Using k-fold cross validation, divide the original data set into k equal data sets. Split each of the k data sets into a training and testing data set. Calculate the SSR of each new tree using only the testing data. The value for α that on average gives the lowest SSR with the testing data is selected as the final α value. The final pruned tree is then the tree built using the entire data set that corresponds to the chosen α value (Starmer, 2019).

When compared to other pruning methods, MCCP tends to produce smaller trees (Quinlan, 1987). This suggests that the method typically results in a higher bias and lower variance. MCCP has become one of the most popular pruning methods when using SSR as the loss function. There is a direct path to finding an α value that corresponds to each pruned tree. However, this may not hold true for other loss functions. Finding a tree that minimizes a given loss function other than SSR can be non-trivial when the size of the tree is extremely large. It is possible that a direct relationship may not exist. There does not appear to be any widely available research on the optimality of various ranking loss functions in MCCP. As a result, it did not seem appropriate to alter the loss function in the method.

This project explores pruning the tree using backwards elimination. Once the tree is grown, the next step is to prune each internal node one by one and calculate the value of the evaluation metric for each sub-tree. At each step, prune the individual node that corresponds to the minimal tree error. An example of pruning a decision tree model through backwards elimination is discussed in the following paragraph.

Suppose a decision tree model was created with a depth of three, resulting in eight leaf nodes. Prune an internal node at the lowest depth and record the error for the entire model. Repeat this process until all internal nodes at the lowest depth have been tried. If the tree with all eight leaf nodes is optimal then the pruning process is finished and that

is the final tree. If not, remove the internal node that resulted in the minimal model error. Next, remove each internal node one at a time at the depth above. Repeat this process until reaching the root node, or until the remaining nodes are optimal.

2.1.2 AdaBoost

AdaBoost, which is short for Adaptive Boosting, is a supervised machine learning algorithm that is built off of decision trees (Freund et al., 1999). Boosting is an ensemble method that creates a strong classifier from weak learners (typically decision trees with a depth of one, also known as, an estimator or stump) (Brownlee, 2020). Estimators are built repetitively with each new estimator accounting for the errors of the previous estimator. This is possible by assigning a weight to each observation after building each estimator. If the first estimator incorrectly classifies an observation then the weight of the observation increases. The weight is used to calculate the probability that an observation will be chosen in the resampling of the data for the next estimator. The larger the weight, the higher the probability of the observation being chosen in the resampling for the next data set. It is likely that an observation that was previously incorrectly predicted in the previous estimators will be resampled multiple times in the next data set. Making it more important that the model is able to correctly predict the observation.

The AdaBoost regression model that was created for this project comes from a slight modification of the model in a paper written by Harris Drucker called "Improving Regressors Using Boosting Techniques" (Drucker, 1997) . The paper uses a weighted median technique for making predictions which requires solving a complex inequality. The weighted median technique is robust to outliers, so it reduces the importance of certain estimators. The AdaBoost model that was created for this project uses a weighted mean technique for making the predictions. The results are then slightly different than what one may receive from a standard programming package that fits an AdaBoost regression model. The weighted mean calculation considers the error from each estimator, assigning more weight to estimators with less error.

The first step for building an AdaBoost model is to create an estimator (decision tree with a depth of one). Next, calculate a loss for each observation in the training set. The loss is denoted $L_{i,t}$ with i referring to the i^{th} observation and t referring to the t^{th} estimator.

$$L_{i,t} = L\left[\left|y_i^{(t)}\left(x_i\right) - y_i\right|\right]$$

Where,

$$L = \left(y_i, y_i^{(t)}\left(x_i\right)\right) \quad L \in [0, 1].$$

The loss can be thought of as the difference between the predicted value and the true value. The linear loss function that is used is denoted as the following.

$$L_{i,t} = \frac{\left|y_i^{(t)}\left(x_i\right) - y_i\right|}{D}$$

Where N is the number of observations, and D is the largest error across all observations in the t^{th} estimator.

$$D = \sup \left| y_i^{(t)}(x_i) - y_i \right| \quad i = 1, \ldots, N$$

In simpler terms, the loss is the absolute value of the difference between the predicted value and the true value, divided by the absolute value of the largest error in the t^{th} estimator. The next step is to calculate an average loss $\overline{L}_t$ for the t^{th} estimator, across all the observations in the training data.

$$\overline{L}_t = \sum_{i=1}^{N} L_{i,t} \cdot p_i$$

Where p_i is the probability that an observation will be chosen in the next training set. It is possible that an individual observation can be selected more than once, or not at all.

$$p_i = \frac{w_i}{\sum w_i}$$

Where w_i is the weight assigned to the observation i. The initial weight for observation i is equal to $\frac{1}{N}$. Using the average loss $\overline{L}_t$, create a β_t value which is a measure of confidence in the predictor. A lower β_t value corresponds to a higher confidence in the prediction.

$$\beta_t = \frac{\overline{L}_t}{1 - \overline{L}_t}.$$

Using β_t, the weights that correspond to each observation are updated. Where $w_{i,t-1}$ is the weight of the observation used in the previous estimator. The following formula will update the weights.

$$w_{i,t} \to w_{i,t-1} \cdot \beta_t^{[1-L_{i,t}]}$$

The larger the loss, the larger the weight is increased, which increases the probability that the observation is chosen in the next training set. Next, an amount of say S_t is calculated for each observation by taking the inverse of the β_t value.

$$S_t = \frac{1}{\beta_t}$$

Where S_t denotes the amount of say for the t^{th} estimator, and β_t denotes the β value of the t^{th} estimator. A total amount of say is calculated by summing the amount of say for all estimators. The say for the t^{th} estimator is divided by the total say to obtain the corresponding normalized amount of say $\mathrm{NS_t}$ for the t^{th} estimator.

$$\mathrm{NS}_t = \frac{\mathrm{S}_t}{\sum_{t=1}^{T} \mathrm{S}_t}$$

The final predicted value for observation i is $\hat{y}_i$.

$$\hat{\mathrm{y}}_i = \sum_{t=1}^{T} y_i^{(t)}(x_i) \cdot \mathrm{NS}_t$$

The above formula multiplies the t^{th} estimators predicted value for the i^{th} observation, by the t^{th} estimators normalized say. It then adds all these values for all estimators to get the final predicted value for the i^{th} observation.

Pruning

The AdaBoost model is first built with a prespecified number of estimators. The process of pruning an AdaBoost model is similar to the process of pruning the decision tree, where the model is pruned through backwards elimination. After each estimator is removed, the value of the loss function (either MSE or AP@k) is calculated. The AdaBoost model with the optimal value for each loss function is chosen as the final model that will be used for predictions. An example of pruning an AdaBoost model through backwards elimination is discussed in the following paragraph.

Suppose an AdaBoost model was created with eight estimators. Remove an estimator and record the error for the entire model. Repeat this process until all estimators have been tried. If the model with all eight estimators is optimal then the pruning process is finished and that is the final model. If not, remove the estimator that resulted in the minimal model error. There would then be seven estimators left. Remove each of the remaining seven estimators one at time and record the error of the model. Repeat this process until one estimator is left, or until all of the remaining estimators is optimal.

2.1.3 Model Hyperparameters

The hyperparameters used for the decision tree model are, a maximum tree depth of ten, a minimum number of samples in a node to qualify for a split of two, and a k value of twenty which was used for the AP@k calculation. The hyperparameters used for the AdaBoost model include all of the hyperparameters used in the decision tree model, in addition to, fifty estimators.

2.2 Loss Functions

Loss functions are used to evaluate how well an algorithm models a data set.

2.2.1 Mean Squared Error (MSE)

Mean Squared Error (MSE) is one of the most common regression loss functions used. MSE is the average squared distance between the true and predicted values.

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

The smaller the MSE value the better the fit of the model. Since the errors are squared, MSE is good at avoiding large individual errors. Multiple smaller errors are less penalized than one very large error due to the values being squared.

2.2.2 Average Precision at k (AP@k)

The ranking loss function that will be used is Average Precision at k (AP@k). In order to understand AP@k better, one must first understand the functions it is built off of, Precision and Precision at k (P@k).

Precision

Precision attempts to answer the question, "What proportion of positive identifications was actually correct?" (Pre) Precision is calculated using the following formula.

$$\text{Precision} = \frac{\text{True Positive}}{\text{True Positive + False Positive}}$$

Suppose there are 150 golfers in a tournament and we would like to calculate the precision of the golfers that will come in the top 75. Precision evaluates the proportion of golfers that were predicted to make the top 75 that actually did make the top 75. If 18 of the 75 golfers predicted to make the top 75 actually made the top 75, then the precision is 0.24.

Precision at k (P@k)

P@k is the precision up to the specified k^{th} observation. It refers to the number of relevant results among the top k recommendations. Every observation after k is not used in the calculation.

Suppose there are 150 golfers in a tournament and we would like to calculate the P@k for the top 10. In this case, k is set to 10. If 3 of the 10 golfers predicted to make the top 10 actually made the top 10, then the P@k is 0.3.

Average Precision at k (AP@k)

AP@k is the mean of the precision at i, for i = 1 to k. For example, if we let k = 10, the P@k for one through ten would be calculated, and the average of all of the P@k values is the AP@k. With a ranking loss function like AP@k, there is less importance on the predicted value, and more importance on the ordering of the values. The AP@k algorithm is shown in Figure 2 (Felipe, 2020).

$CorrectPredictions \leftarrow 0$
$RunningSum \leftarrow 0$
for $i \leftarrow 1$ to k **do**
  if document at rank i is relevant **then**
    $CorrectPredictions \leftarrow CorrectPredictions + 1$
    $RunningSum \leftarrow RunningSum + \frac{CorrectPredictions}{i}$
  end if
end for
$AP\ @k = \frac{RunningSum}{TotalNumberOfRelevantDocuments\ @k}$

Figure 2: Pseudocode for AP@k

Suppose there are 150 golfers in a tournament and we would like to calculate the AP@k for the top three. In this case, k is set to three. First, calculate the P@k of the top one, then calculate the P@k of the top two, then calculate the P@k of the top three. Take the average of the the three P@k values, to get the AP@k.

3. Simulation Study

Simulation studies are a way to obtain empirical results about the performance of statistical methods in certain scenarios through creating data using pseudo-random sampling (Morris et al., 2019). The steps of the simulation study are as follows. Step one, simulate the data. Step two, fit continuous models for each tree based algorithm with one using MSE and one using AP@k as pruning loss functions. Step three, rank the predictions outputted by the model. Step four, evaluate the model using AP@k.

3.1 Distributions of Simulated Data

The models will be trained and tested on data that is simulated from a Gaussian distribution and a log-normal distribution. The Gaussian distribution was chosen because it aligns with the distribution of many real world applications. The log-normal distribution was chosen as it is a heavy-tailed distribution with many extreme observations.

3.1.1 Gaussian Distribution

The first group of data sets is simulated from a Gaussian distribution.

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$$

Where μ is the mean and σ is the standard deviation. Since the distribution is Gaussian, $\mu = 0$ and $\sigma = 1$. The Gaussian distribution is a continuous probability distribution for a real-valued random variable.

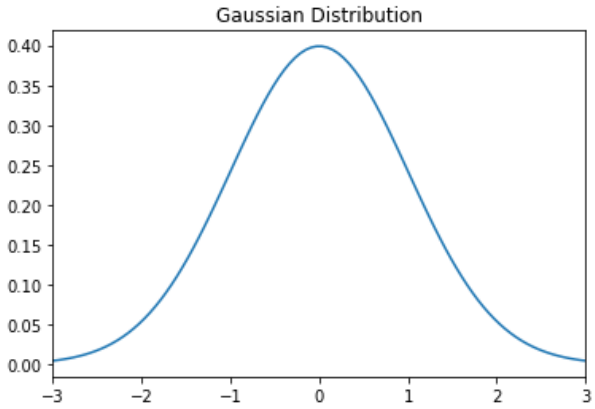

Figure 3: Gaussian distribution

3.1.2 Log-Normal Distribution

The second group of data sets is simulated from a log-normal distribution.

$$f(x) = \frac{1}{x\sigma\sqrt{2\pi}} \exp\left(-\frac{(\ln x - \mu)^2}{2\sigma^2}\right)$$

Where μ is the mean and σ is the standard deviation. The values of $\mu = 0$ and $\sigma = 1$ have been chosen. The log-normal distribution is a continuous distribution in which the logarithm of a variable has a normal distribution.

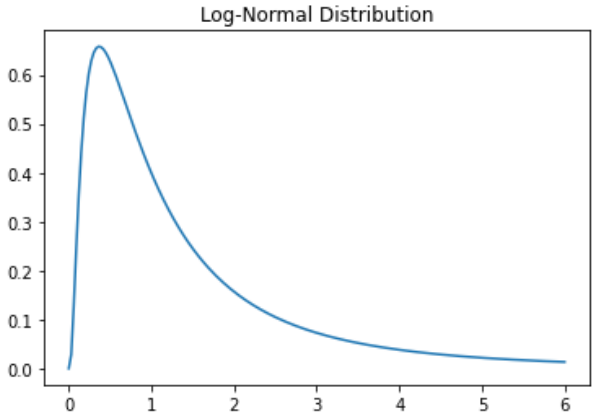

Figure 4: Log-Normal distribution

3.2 Variations of Simulated Data

This project explores many different variations in the simulated data sets to observe how the models perform in each scenario. Each data set is simulated with 150 observations which was chosen as it closely aligns with the number of observations in the data sets used in the application. The target variable is created using the following multiple linear regression formula.

$$Y_i = \beta_1 X_{1i} + \beta_2 X_{2i} + \beta_3 X_{3i} + \beta_4 X_{4i} + \beta_5 X_{5i} + \varepsilon_i$$

The β values were randomly generated rational numbers with one as the lower bound and the value of the number of inputs as the upper bound. For simplicity, β_0 was set to zero. X is simulated from one of the two chosen distributions. ε is the random component of the linear relationship between x and y, and is independently and identically distributed from the Gaussian or log-normal distribution.

3.2.1 Number of Inputs

The first variation has no irregularities and is simulated with two differing number of inputs. One data set with five inputs and one with fifty inputs.

3.2.2 Missing Data in Inputs

The second variation explores missing values in the input variables. The data set is simulated with a chosen percentage of the observations missing completely at random. An observation chosen, has a missing value in one of the inputs. The missing values are imputed with the mean value of the column. Five inputs were used for the data sets in this variation. The β values were not changed from the five β values used in the previous variation. Multiple data sets were simulated, with 10%, 25% and 50% of the observations having a missing value in one of the inputs.

3.3 Results

Five data sets were simulated for each combination of variation, distribution and loss function. The results shown in Figure 5 and Figure 6 are the average and standard deviation of the five data sets.

3.3.1 Decision Tree

As shown in Figure 5, AP@k outperforms MSE for all scenarios except for the Gaussian simulated data with 50% missing values. These results indicate that AP@k is generally the optimal pruning metric when using a decision tree model. The individual results for each of the five simulated data sets can found in Appendix A.2.

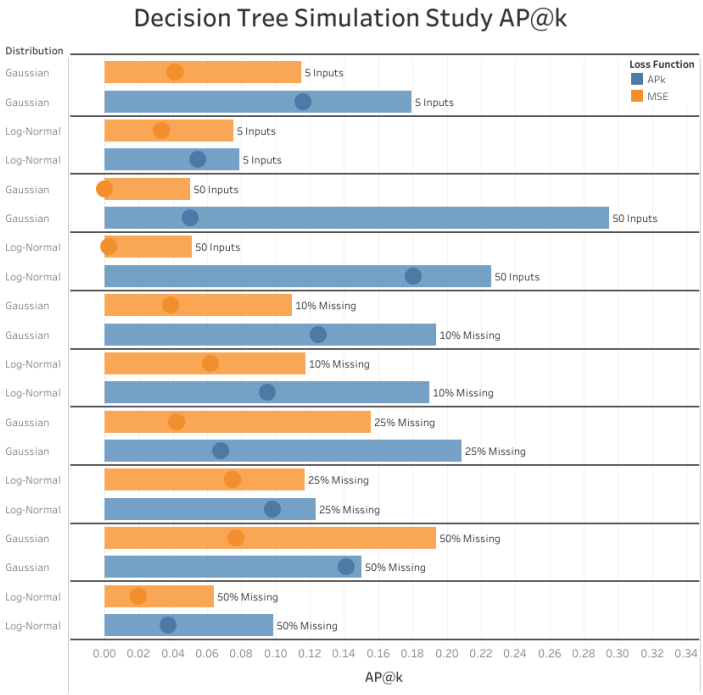

Figure 5: The mean value is represented by the bar, and the standard deviation is represented by the circle

3.3.2 AdaBoost

As shown in Figure 6, MSE performs better than AP@k in all scenarios except for the log-normal data with fifty inputs and the Gaussian data with 50% missing values. These results indicate that MSE is generally the optimal pruning metric when using an AdaBoost model. The individual results for each of the five simulated data sets can found in Appendix A.4.

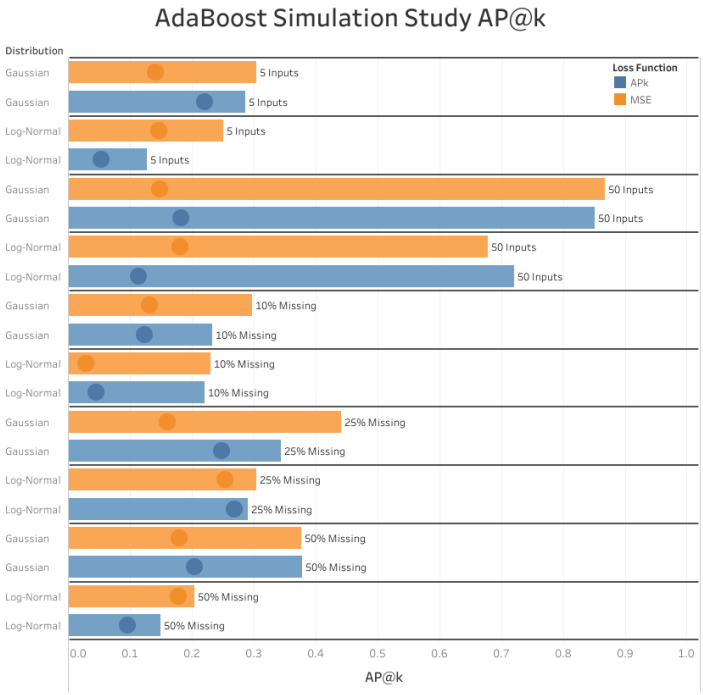

Figure 6: The mean value is represented by the bar, and the standard deviation is represented by the circle

3.3.3 Decision Tree vs. AdaBoost

Comparing the decision tree and AdaBoost algorithms tells us which algorithm is generally a better choice, regardless of the distribution or variation of the data. The results are highlighted in Table 1.

Table 1: Decision Tree vs. AdaBoost Simulation Results

Model	Average	Std Dev
Decision Tree	0.1396	0.06353
AdaBoost	0.3779	0.2168

AdaBoost outperformed the decision tree models when looking at the average AP@k, but also had a higher standard deviation. These results are expected, since boosting methods

such as AdaBoost have been proven to outperform a single estimator such as a decision tree (Drucker, 1997). The AdaBoost model benefits from using a combination of many learners, whereas, a decision tree model is built from one learner.

3.3.4 Gaussian vs. Log-Normal

Comparing the Gaussian and log-normal distribution tells us which distribution is easier for the models to predict.

Table 2: Gaussian vs. Log-Normal Simulation Results

Distribution	Average	Std Dev
Gaussian	0.3016	0.2105
Log-Normal	0.2160	0.1773

As seen in Table 2, the Gaussian distribution performs better as it is not a heavy-tailed distribution so there are not many extreme observations which are often tough for models to predict.

3.3.5 AP@k vs. MSE

Comparing AP@k with MSE overall for each model gives a good indication of which metric is generally better to use.

Table 3: AP@k vs. MSE Simulation Results

Loss Function	Model	Average	Std Dev
AP@k	Decision Tree	0.1744	0.06097
MSE	Decision Tree	0.1049	0.04400
AP@k	AdaBoost	0.3602	0.2273
MSE	AdaBoost	0.3957	0.2043

For the decision tree model, AP@k outperforms MSE. Since the model is being evaluated using AP@k, it is logical that the AP@k loss function is optimal.

For the AdaBoost model, MSE slightly outperforms AP@k. An AdaBoost model is built off of many weak decision tree estimators with a depth of one, which is a root node and two leaf nodes. The data gets classified into one of the two groups. Using AP@k gives two unique ranks for each estimator. The predicted value is the average rank for the observations in the node. Using MSE there are still only two unique predictions, but they are able to account for the numerical differences within each group to give a more appropriate average. As a result, MSE is able to outperform AP@k.

4. Application

This section explores the application of the models to professional golf data. The steps of the application are as follows. Step one, using the golf data, fit continuous models for each tree based algorithm with one using MSE and one using AP@k as pruning loss functions.

Step two, rank the predictions outputted by the model. Step three, evaluate the model using AP@k.

4.1 Data

Using Python, a script was created to acquire data from the PGA Tour website (pga). There are several hundreds of statistics recorded by the PGA Tour that are used to analyze the performance of each athlete. The statistics can be divided into the following subcategories, Off the Tee, Approach the Green, Around the Green, Putting, Scoring, Streaks, Money/Finishes and Points/Rankings. The created data set consists of tournament statistics from January 15, 2017 to November 8, 2020. Each observation is the statistics of one player in a given tournament. There are 17,985 observations and 38 columns of data from 141 tournaments. A subset of eight columns shown in Table 4 were selected based on golf knowledge.

Table 4: Subset of columns selected from golf data

Column Name
One-Putt Percentage
3-Putt Avoidance Percentage
Average Distance of Putts Made
Driving Accuracy Percentage
Green in Regulation Percentage
Driving Distance Average
Sand Save Percentage
Scrambling Percentage

Major championships are played on more challenging courses when compared to the average week-to-week tournaments. If a player only plays at easier courses, their scoring average will be better than someone that plays more challenging courses. Therefore, a non-adjusted scoring average is not a great performance statistic on its own. When predicting rankings, the strength of the field is very important. If a strong player is playing in a weaker field they have a higher probability of ranking high. These are all things to consider when predicting professional golf performance.

The target variable used for the application is the players tournament ranking. Nearly every PGA Tour event has a cut line which is typically the score of the 70^{th} ranked player in the tournament (Preston, 2020). Players that have a score worse than the cut line after the first two days do not compete in the last two days of the tournament. The players that miss the cut are not included in the tournament rankings and are grouped together as a "missed cut". It is not possible to know what the true tournament score over the four days would have been for players that missed the cut. Using a multiple linear regression model, the scores for the last two days are imputed based on the individuals score from the first two days. For every player in the field there is then either a true tournament score, or an imputed tournament score. The tournament score is then ranked so there is a ranking for every individual in the tournament, and not only those that made the cut.

Tournaments of Interest

A total of five tournaments have been chosen to be predicted for the application. The tournaments chosen are some of the biggest events that happen each year, two of which are a major championship, while the third is recognized as the best non-major tournament. In order to predict an upcoming tournament, the five previous tournaments leading up to it are used as the training data. The five tournaments that will be predicted are highlighted in Table 5

Table 5: Professional golf tournaments of interest

Event ID	End Date	Tournament Name
1	2018-08-12	PGA Championship
2	2019-03-17	THE PLAYERS Championship
3	2019-06-16	U.S. Open
4	2020-08-09	PGA Championship
5	2020-09-20	U.S. Open

4.2 Results

Golf is a very tough sport to predict as there are a large number of golfers competing in each tournament. Golfers performance can be quite volatile when compared to other sports, so modelling the sport is trickier than most. There are a lot of factors that affect the performance of a player, including the weather, the length and layout of the course, the altitude for the location, the type of grass and many others.

4.2.1 Decision Tree

As shown in Figure 7, AP@k outperformed MSE for three out of the five tournaments for the decision tree model. These results align with the results from the simulation study that suggest that AP@k is the optimal loss function when using a decision tree model. The individual results for each of the five simulated data sets can found in Appendix A.6.

Figure 7: Decision Tree application AP@k results

4.2.2 AdaBoost

As shown in Figure 8, MSE outperformed AP@k for two out of the five tournaments for the AdaBoost model. These results align with the results from the simulation study that suggest that MSE is the optimal loss function when using an AdaBoost model. The individual results for each of the five simulated data sets can found in Appendix A.8.

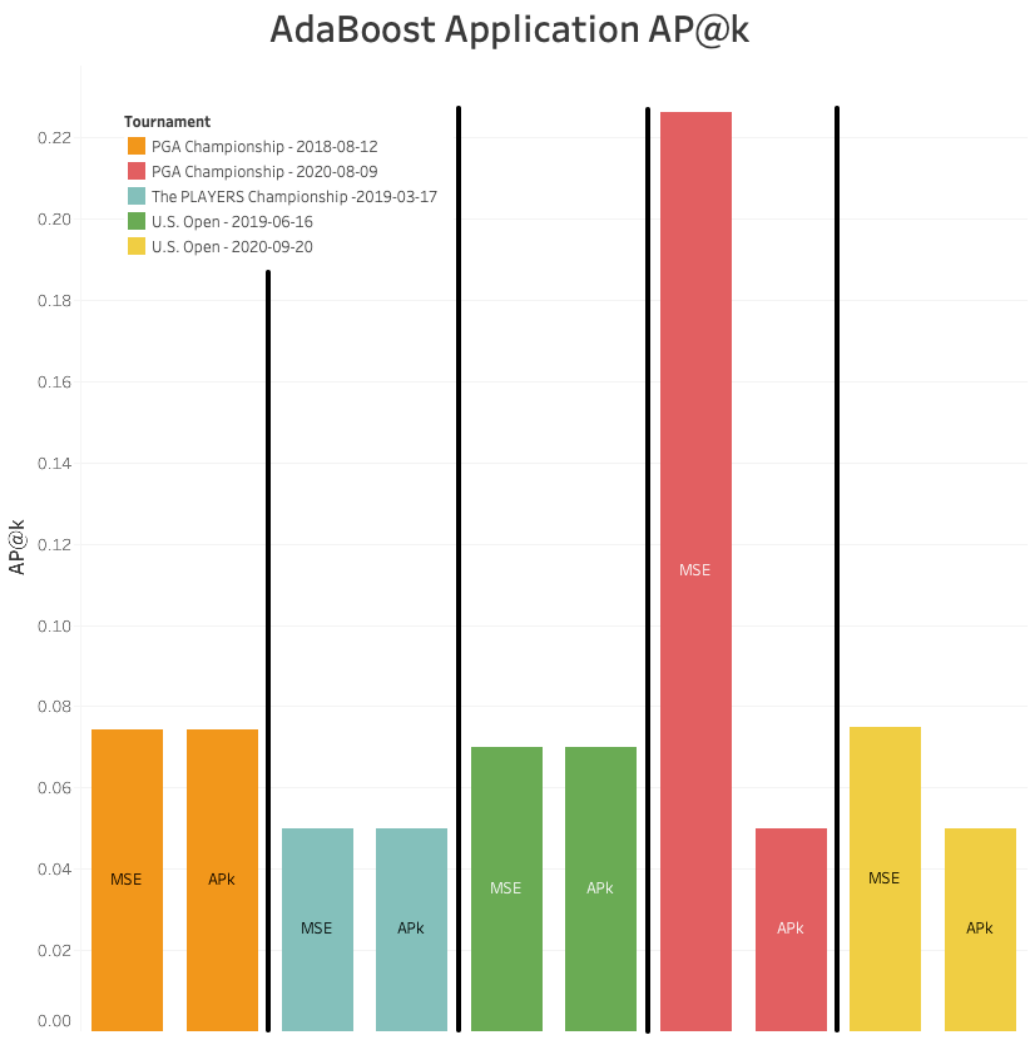

Figure 8: AdaBoost application AP@k results

4.2.3 Decision Tree vs. AdaBoost

In the simulation study, the AdaBoost models outperformed the decision tree models. However, in the application, the decision tree models slightly outperformed the AdaBoost models. This is likely due to the noise that comes with real world data. The simulation study is a simple linear model with added noise which makes it easier for the models to predict. This is also why the results for the loss functions are a lot closer in the application than in the simulation study.

5. Conclusion

In addition to the AP@k values, the P@k of the models for both the simulation study and the application were recorded as well. P@k is a classification metric and does not evaluate the performance of rankings. The simulation results (highlighted in Appendix A.3, Appendix A.5, Appendix B.1 and Appendix B.2) showed that MSE is the optimal loss function in nearly all scenarios for both the decision tree and AdaBoost models when using P@k as the evaluation metric. The application results (highlighted in Appendix A.7, Appendix A.9, Appendix B.3 and Appendix B.4) show that for a decision tree model, AP@k is the optimal loss function. However, for an AdaBoost model, MSE and AP@k have the same average, but MSE has a slightly lower standard deviation. Overall, if the problem is dealing with classification and not ranking, then MSE is generally the optimal metric.

When working with a large number of observations in a data set, the depth of the tree must be large to ensure there are enough unique predictions. As the depth of the tree grows, the number of leaf nodes increases exponentially. A large number of leaf nodes reduces the possibility of ties. This project showed that AP@k was able to outperform MSE when the depth of the tree is large enough. Otherwise, when the depth of the tree was not large, MSE was the optimal metric.

AdaBoost models were shown in Table 1 to outperform the decision tree models in the simulation study. As a result, they should be relied upon more than decision trees for real world applications.

The Gaussian distribution was shown in Table 2 to outperform the log-normal distribution which shows that heavy-tailed distributions are tougher for the models to predict.

5.1 Future Applications

Creating the models from scratch required deep technical knowledge and understanding that led to ideas for potential future applications. Adding a second regression and ranking loss function would reduce the potential variance involved with only considering one function of each. Evaluating performance using Mean Absolute Error (MAE) and Normalized Discounted Cumulative Gain (NDCG) in addition to the existing loss functions used would increase the credibility of the results.

There are two opportunities to use a loss function in a decision tree model, when building the tree, and when pruning the tree. An AdaBoost model introduces a third opportunity when calculating a loss for an estimator. This paper uses a linear loss function, however, a ranking loss function could have been implemented as well.

There are many things that could be done to improve the predictive power of the models in the application. The paper does not consider the time series component of the application. The training data consists of information from the past five tournaments. It is very likely that the most recent tournaments should have a higher say in the model. This can be achieved through lagging the inputs and target variable. This allows the model to use the previous weeks values (including the target variable) as inputs for predicting the target variable of the upcoming week.

Acknowledgments

I would like to thank Dr. William Marshall for his support and guidance on this project. I would also like to acknowledge Brock University for the funding and research grants that were given for this project.

Appendix A. Tables

A.1 Simulated Data Sets

Table 6: Simulated Data Sets

#	Type	Variation	Distribution	Loss Function
1	No Irregularities	5 Inputs	Gaussian	MSE
2	No Irregularities	5 Inputs	Gaussian	AP@k
3	No Irregularities	5 Inputs	Log-Normal	MSE
4	No Irregularities	5 Inputs	Log-Normal	AP@k
5	No Irregularities	50 Inputs	Gaussian	MSE
6	No Irregularities	50 Inputs	Gaussian	AP@k
7	No Irregularities	50 Inputs	Log-Normal	MSE
8	No Irregularities	50 Inputs	Log-Normal	AP@k
9	Missing Values	10% Missing	Gaussian	MSE
10	Missing Values	10% Missing	Gaussian	AP@k
11	Missing Values	10% Missing	Log-Normal	MSE
12	Missing Values	10% Missing	Log-Normal	AP@k
13	Missing Values	25% Missing	Gaussian	MSE
14	Missing Values	25% Missing	Gaussian	AP@k
15	Missing Values	25% Missing	Log-Normal	MSE
16	Missing Values	25% Missing	Log-Normal	AP@k
17	Missing Values	50% Missing	Gaussian	MSE
18	Missing Values	50% Missing	Gaussian	AP@k
19	Missing Values	50% Missing	Log-Normal	MSE
20	Missing Values	50% Missing	Log-Normal	AP@k

A.2 Simulation Study - Decision Tree AP@k Individual Results

Table 7: Simulation Study - Decision Tree AP@k Individual Results

#	Seed 0	Seed 1	Seed 2	Seed 3	Seed 4	Average	Std Dev
1	0.1226	0.0577	0.1674	0.1473	0.0794	0.1149	0.0410
2	0.1781	0.3072	0.05	0.3109	0.05	0.1792	0.1159
3	0.0553	0.05	0.0762	0.0553	0.1392	0.0752	0.0332
4	0.05	0.05	0.05	0.0553	0.1878	0.0786	0.0546
5	0.05	0.05	0.05	0.05	0.05	0.05	0
6	0.3821	0.3181	0.2648	0.2701	0.2402	0.2951	0.0503
7	0.05	0.0559	0.05	0.05	0.05	0.0512	0.0024
8	0.3610	0.1640	0.05	0.05	0.5061	0.2262	0.1803
9	0.1554	0.0577	0.1528	0.0825	0.0993	0.1095	0.0387
10	0.3650	0.2403	0.05	0.2647	0.05	0.194	0.1248
11	0.0553	0.1329	0.2308	0.0779	0.0911	0.1176	0.0620
12	0.05	0.2117	0.2861	0.2896	0.1123	0.1899	0.0951
13	0.2044	0.1074	0.1741	0.1887	0.1039	0.1557	0.0420
14	0.1143	0.3097	0.2564	0.1887	0.1749	0.2088	0.0677
15	0.0861	0.1522	0.05	0.05	0.2463	0.1169	0.0747
16	0.05	0.1701	0.05	0.05	0.2965	0.1233	0.0983
17	0.1673	0.1562	0.1301	0.3451	0.1698	0.1937	0.0770
18	0.05	0.1889	0.05	0.4115	0.05	0.1501	0.1413
19	0.05	0.05	0.05	0.1010	0.0679	0.0638	0.0199
20	0.05	0.05	0.05	0.1353	0.2087	0.0988	0.0372

A.3 Simulation Study - Decision Tree P@k Individual Results

Table 8: Simulation Study - Decision Tree P@k Individual Results

#	Seed 0	Seed 1	Seed 2	Seed 3	Seed 4	Average	Std Dev
1	0.5	0.6	0.5	0.55	0.65	0.56	0.0510
2	0.6	0.55	0.65	0.55	0.5	0.57	0.0510
3	0.65	0.55	0.6	0.5	0.45	0.55	0.0707
4	0.35	0.5	0.55	0.5	0.45	0.47	0.0678
5	0.5	0.7	0.4	0.45	0.65	0.54	0.1158
6	0.55	0.4	0.7	0.6	0.35	0.52	0.1288
7	0.55	0.55	0.6	0.4	0.5	0.52	0.0678
8	0.45	0.45	0.6	0.4	0.45	0.47	0.0678
9	0.65	0.6	0.5	0.5	0.55	0.56	0.0583
10	0.6	0.6	0.65	0.5	0.5	0.57	0.06
11	0.65	0.6	0.65	0.6	0.55	0.61	0.0374
12	0.4	0.6	0.65	0.5	0.55	0.54	0.0860
13	0.6	0.6	0.5	0.45	0.55	0.54	0.0583
14	0.6	0.55	0.5	0.45	0.7	0.56	0.0860
15	0.65	0.6	0.55	0.5	0.45	0.55	0.0707
16	0.35	0.6	0.55	0.5	0.5	0.5	0.0837
17	0.55	0.5	0.5	0.6	0.65	0.56	0.0583
18	0.5	0.5	0.6	0.6	0.45	0.53	0.0600
19	0.4	0.55	0.55	0.6	0.5	0.52	0.0678
20	0.4	0.5	0.55	0.6	0.5	0.51	0.0663

A.4 Simulation Study - AdaBoost AP@k Individual Results

Table 9: Simulation Study - AdaBoost AP@k Individual Results

#	Seed 0	Seed 1	Seed 2	Seed 3	Seed 4	Average	Std Dev
1	0.2099	0.5306	0.2219	0.2025	0.3562	0.3042	0.1414
2	0.0567	0.4328	0.1657	0.1797	0.5949	0.286	0.221
3	0.4156	0.1229	0.1602	0.1523	0.4065	0.2515	0.1463
4	0.1268	0.1229	0.0556	0.1245	0.2054	0.127	0.0531
5	0.6427	0.8811	1.0	1.0	0.8172	0.8682	0.1486
6	0.5553	0.8811	1.0	1.0	0.8172	0.8507	0.183
7	0.7845	0.4093	0.7499	0.8618	0.5890	0.6789	0.1806
8	0.7845	0.6228	0.7499	0.8618	0.5890	0.7216	0.1138
9	0.4368	0.1445	0.2657	0.4322	0.2083	0.2975	0.1322
10	0.1277	0.1445	0.1997	0.4322	0.2609	0.233	0.123
11	0.2633	0.1855	0.2384	0.2196	0.2434	0.23	0.0294
12	0.2002	0.1613	0.2143	0.2811	0.2434	0.2201	0.0451
13	0.3740	0.5380	0.1923	0.5097	0.5916	0.4411	0.1606
14	0.05	0.6440	0.1923	0.2801	0.5530	0.3439	0.2486
15	0.1672	0.1648	0.0577	0.4611	0.6703	0.3042	0.2538
16	0.075	0.1794	0.0635	0.4611	0.6752	0.2908	0.2681
17	0.3834	0.4294	0.1182	0.6169	0.3384	0.3773	0.1794
18	0.1166	0.4535	0.2214	0.6169	0.4866	0.379	0.2046
19	0.0903	0.2524	0.1147	0.4958	0.0667	0.204	0.1784
20	0.0903	0.2524	0.1017	0.2545	0.05	0.1498	0.0966

A.5 Simulation Study - AdaBoost P@k Individual Results

Table 10: Simulation Study - AdaBoost P@k Individual Results

#	Seed 0	Seed 1	Seed 2	Seed 3	Seed 4	Average	Std Dev
1	0.6	0.55	0.5	0.5	0.65	0.56	0.0652
2	0.55	0.55	0.5	0.5	0.65	0.55	0.0612
3	0.65	0.55	0.65	0.5	0.55	0.58	0.0671
4	0.5	0.55	0.6	0.5	0.55	0.54	0.0418
5	0.55	0.4	0.7	0.65	0.35	0.53	0.1525
6	0.55	0.4	0.7	0.65	0.35	0.53	0.1525
7	0.5	0.45	0.55	0.7	0.45	0.53	0.1037
8	0.5	0.45	0.55	0.7	0.45	0.53	0.1037
9	0.7	0.55	0.6	0.5	0.6	0.59	0.0742
10	0.55	0.55	0.5	0.5	0.6	0.54	0.0418
11	0.6	0.55	0.65	0.55	0.55	0.58	0.0447
12	0.6	0.55	0.65	0.5	0.55	0.57	0.057
13	0.6	0.55	0.5	0.5	0.65	0.56	0.0652
14	0.4	0.55	0.5	0.55	0.65	0.53	0.0908
15	0.65	0.6	0.65	0.55	0.5	0.59	0.0652
16	0.7	0.65	0.6	0.55	0.5	0.6	0.0791
17	0.55	0.5	0.55	0.6	0.65	0.57	0.057
18	0.55	0.5	0.55	0.6	0.65	0.57	0.057
19	0.6	0.55	0.6	0.45	0.5	0.54	0.0652
20	0.6	0.55	0.6	0.45	0.5	0.54	0.0652

A.6 Application - Decision Tree AP@k Individual Results

Table 11: Application - Decision Tree AP@k Individual Results

Eval Metric	Event 1	Event 2	Event 3	Event 4	Event 5	Average	Std Dev
MSE	0.075	0.1115	0.05	0.05	0.05	0.0673	0.02413
AP@k	0.125	0.1404	0.05	0.09487	0.05	0.09205	0.03733

A.7 Application - Decision Tree P@k Individual Results

Table 12: Application - Decision Tree P@k Individual Results

Eval Metric	Event 1	Event 2	Event 3	Event 4	Event 5	Average	Std Dev
MSE	0.0	0.2	0.2	0.0	0.05	0.09	0.09165
AP@k	0.2	0.25	0.2	0.0	0.05	0.14	0.09695

A.8 Application - AdaBoost AP@k Individual Results

Table 13: Application - AdaBoost AP@k Individual Results

Eval Metric	Event 1	Event 2	Event 3	Event 4	Event 5	Average	Std Dev
MSE	0.07417	0.05	0.07	0.2263	0.075	0.09909	0.06325
AP@k	0.07417	0.05	0.07	0.05	0.05	0.05883	0.01090

A.9 Application - AdaBoost P@k Individual Results

Table 14: Application - AdaBoost P@k Individual Results

Eval Metric	Event 1	Event 2	Event 3	Event 4	Event 5	Average	Std Dev
MSE	0.15	0.1	0.15	0.05	0.2	0.13	0.05099
AP@k	0.15	0	0.15	0.25	0.1	0.13	0.08124

Appendix B. Figures

B.1 Simulation - Decision Tree P@k Results

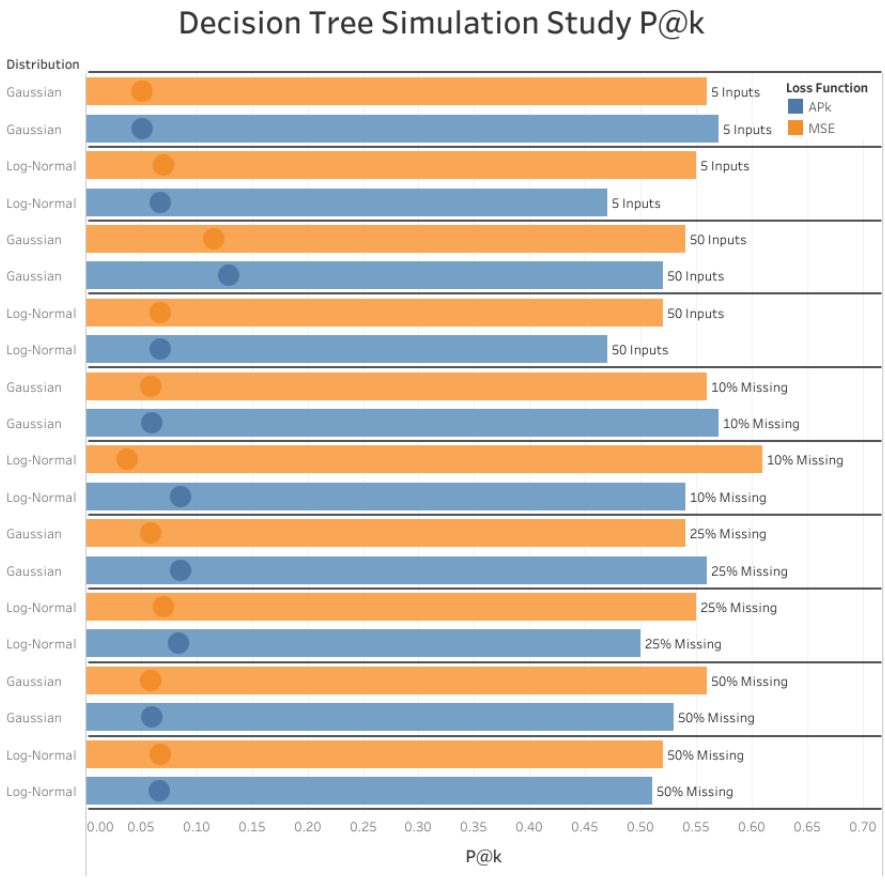

Figure 9: The mean value is represented by the bar, and the standard deviation is represented by the circle

B.2 Simulation - AdaBoost P@k Results

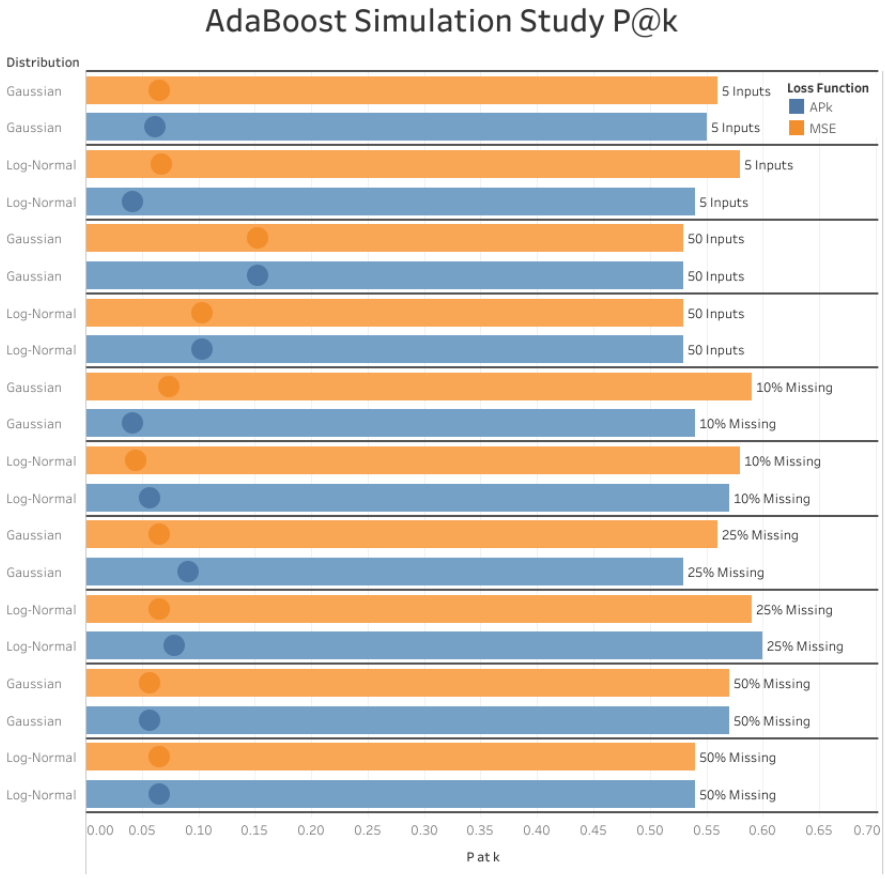

Figure 10: The mean value is represented by the bar, and the standard deviation is represented by the circle

B.3 Application - Decision Tree P@k Results

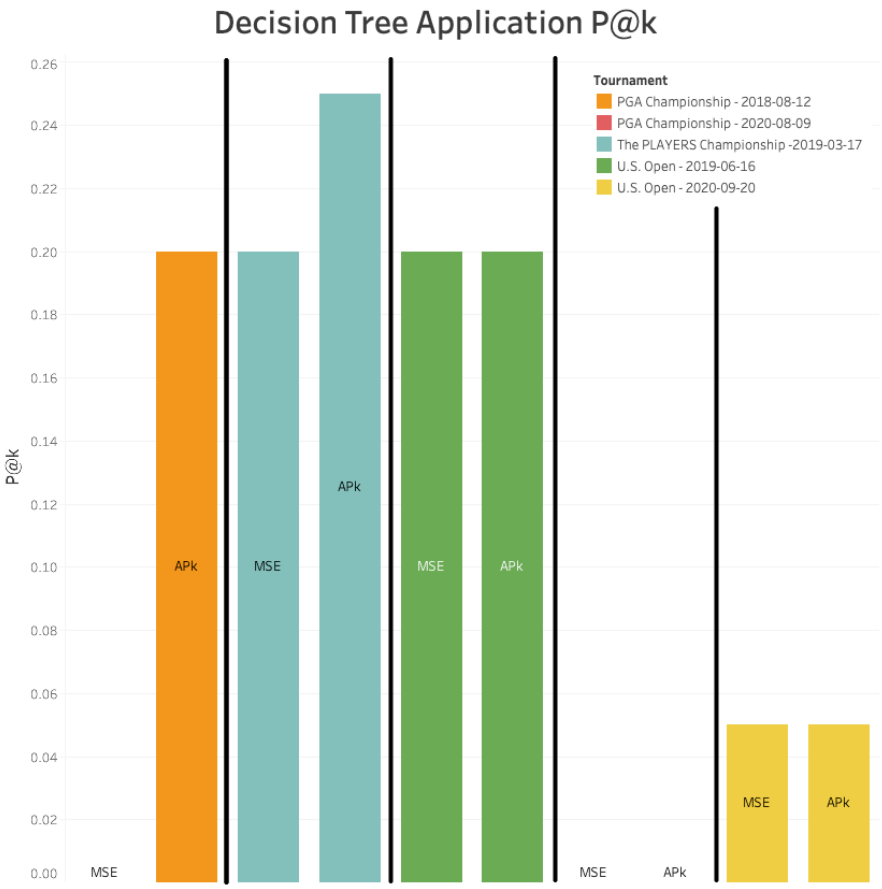

Figure 11: Decision Tree Simulation P@k Results

B.4 Application - AdaBoost P@k Results

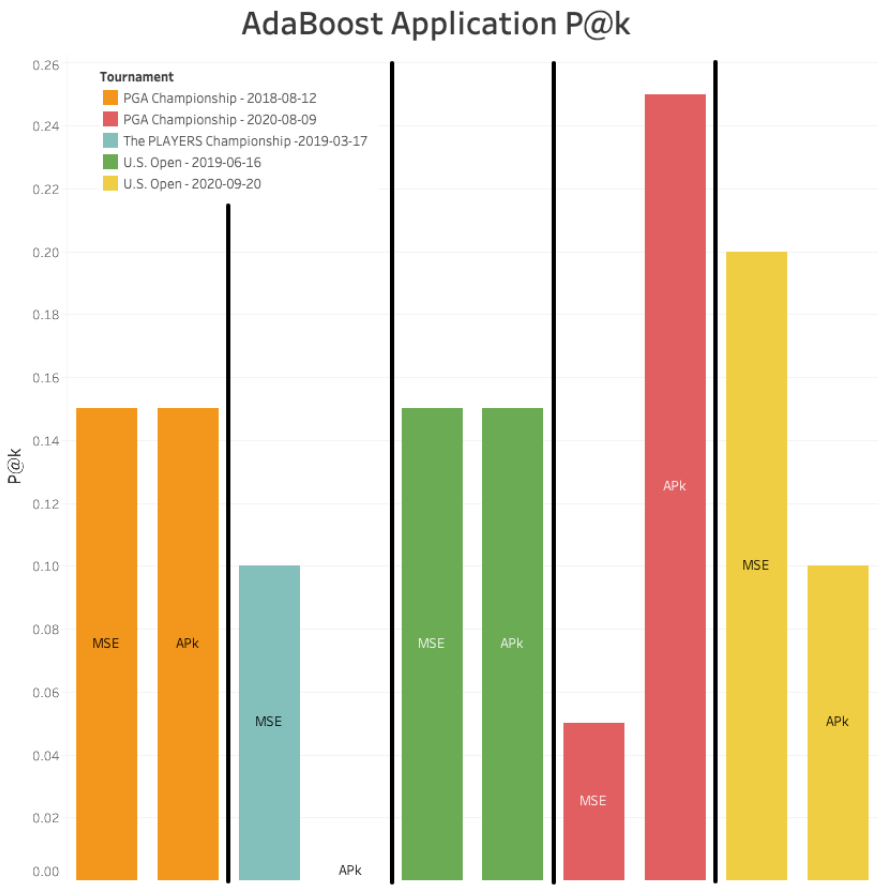

Figure 12: AdaBoost Simulation P@k Results

Appendix C. Code

C.1 Decision Tree Code

Listing 1: Python code for building a regression decision tree algorithm with the ability to use a ranking pruning loss function

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
@author: bradklassen
"""

# Import libraries
import pandas as pd
import numpy as np
from itertools import chain
import pickle
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split
import scipy.stats as ss
import ml_metrics as metrics


def get_potential_splits(data):
    """
    Gets potential splits to test, combinations are each column
       and their unique values.

        Parameters:
            data (pd.DataFrame): Data used to get the splits.

        Returns:
            potential_splits (dictionary): Dictionary mapping
               each column with it's unique values.
    """

    # Create empty dictionary
    potential_splits = {}

    # Number of columns
    n_columns = len(data.columns)

    # Excludes last column which is the target variable
    for column_index in range(n_columns - 1):
```

```
            # Values in column
            values = data.iloc[:, column_index]

            # Remove duplicate values
            unique_values = np.unique(values)

            # Add the unique values to the dictionary for potential
                splits
            potential_splits[column_index] = unique_values

    return potential_splits


def mse(data):
    """
    Calculate Mean Squared Error (MSE).

        Parameters:
            data (pd.DataFrame): Data used to calculate MSE.

        Returns:
            mse_val (float): Mean squared error value.
    """

    # Target variable
    target_var = data.iloc[:, -1]

    # If there is data in target variable
    if len(target_var) != 0:

        # Mean of observations in target variable
        prediction = np.mean(target_var)

        # Calculate MSE
        mse_val = np.mean((target_var - prediction) ** 2)

    # If there is no data in target variable
    else:

        # MSE equals 0
        mse_val = 0

    return mse_val
```

```
def split_data(data, split_column, split_value):
    """
    Get the data for less than or equal to the split value and
       the data for greater than the split value.

        Parameters:
            data (pd.DataFrame): Data that will be split.
            split_column (integer): Column number to split the
               data on.
            split_value (float): Value to split the data on.

        Returns:
            data_below (pd.DataFrame): Data from the chosen split
                column that is less than or equal to the split
               value.
            data_above (pd.DataFrame): Data from the chosen split
                column that is greater than the split value.
    """

    # Split data on split column
    split_column_values = data.iloc[:, split_column]

    # Data below split value
    data_below = data[split_column_values <= split_value]

    # Data above split value
    data_above = data[split_column_values > split_value]

    return data_below, data_above


def calculate_overall_metric(data_below, data_above):
    """
    Weighs the MSE values based on the amount of data below and
       above the split value.

        Parameters:
            data_below (pd.DataFrame): Data from the chosen split
                column that is less than or equal to the split
               value.
            data_above (pd.DataFrame): Data from the chosen split
                column that is greater than the split value.

        Returns:
            overall_metric (float): Weighted MSE value.
```

```
    """

    # Number of observations
    n = len(data_below) + len(data_above)

    # Proportion of data below split value
    p_data_below = len(data_below) / n

    # Proportion of data above split value
    p_data_above = len(data_above) / n

    # Calculate the overall metric with proportion taken into
        consideration
    overall_metric = (p_data_below * mse(data_below) +
        p_data_above * mse(data_above))

    return overall_metric


def get_split(data):
    """
    Get the column and value to split on.

        Parameters:
            data (pd.DataFrame): Data used to get the split.

        Returns:
            split_column (integer): Column number to split the
                data on.
            split_value (float): Value to split the data on.
    """

    # Get potential splits
    potential_splits = get_potential_splits(data)

    # Set first iteration to true
    first_iteration = True

    # Empty list for current metrics
    current_met = []

    # For each column
    for column in potential_splits:

        # For each value in the column
```

```
        for value in potential_splits[column]:

            # Get data above and below split value
            data_below, data_above = split_data(data,
                split_column=column, split_value=value)

            # If either data set is empty
            if len(data_below) == 0 or len(data_above) == 0:
                pass

            # Get the current overall metric
            current_overall_metric = calculate_overall_metric(
                data_below, data_above)

            # Append current metric
            current_met.append(current_overall_metric)

            # If it is the first iteration or current overall
                metric is <= best metric
            if first_iteration or current_overall_metric <=
                best_overall_metric:

                # Set first iteration to false
                first_iteration = False

                # Set best metric to current
                best_overall_metric = current_overall_metric

                # Save column
                best_split_column = column

                # Save value
                best_split_value = value

    return best_split_column, best_split_value


def splitting_var(nodes, sub_trees, index):
    """
    Evaluate which input is the best splitting variable at the
        given depth.

        Parameters:
            nodes (pd.DataFrame): Structure of the tree, stored
                as a DataFrame.
```

```
            sub_trees (list): List of pd.DataFrames for each sub-
                tree.
            index (int): Index corresponding to the sub-tree of
                choice.

        Returns:
            nodes (pd.DataFrame): Structure of the tree, stored
                as a DataFrame.
            sub_trees (list): List of pd.DataFrames for each sub-
                tree.
    """

    # Create DataFrame without target variable
    no_target_variable = sub_trees[index].iloc[:, :-1].copy()

    # For non-leaf nodes
    if len(no_target_variable.columns) > 0:

        # Get the split
        best_split_column, best_split_value = get_split(sub_trees
            [index])

        # Add the splitting variable to the DataFrame
        nodes.at[index, 'split_var'] = no_target_variable.iloc[:,
            best_split_column].name

        # Add the splitting value to the DataFrame
        nodes.at[index, 'split'] = best_split_value

    # For leaf nodes
    else:

        # Add the splitting variable to the DataFrame
        nodes.at[index, 'split_var'] = np.NaN

        # Add the splitting value to the DataFrame
        nodes.at[index, 'split'] = np.NaN

    return nodes, sub_trees


def root_node(data):
    """
    Create the root node of the decision tree.
```

```
        Parameters:
            data (pd.DataFrame): Data used to create the root
                node.

        Returns:
            nodes (pd.DataFrame): Structure of the tree with only
                the root node, stored as a DataFrame.
    """

    # Create DataFrame from column names
    nodes = pd.DataFrame(columns=['parent', 'split', 'split_var',
        'leaf', 'prediction', 'children'])

    # Create root node which has no parent, is currently a leaf
        node, has a prediction, has no children
    nodes = nodes.append({'parent': np.NaN, 'leaf': True, '
        prediction': data[data.columns[-1]].mean(), 'children':
        [],
                          'num_obs': len(data)}, ignore_index=
                              True)

    # Convert split variable to string
    nodes['split_var'] = nodes['split_var'].astype('str')

    return nodes


def create_nodes(nodes, node, sub_trees):
    """
    Create the internal nodes & leaf nodes of the decision tree.

        Parameters:
            nodes (pd.DataFrame): Structure of the tree, stored
                as a DataFrame.
            node (int): Parent node of the newly created nodes.
            sub_trees (list): List of pd.DataFrames for each sub-
                tree.

        Returns:
            nodes (pd.DataFrame): Structure of the tree, stored
                as a DataFrame.
            sub_trees (list): List of pd.DataFrames for each sub-
                tree.
    """
```

```
# Set index
index = nodes.shape[0]

# Get the split for the first data set
best_split_column, best_split_value = get_split(sub_trees[
    node])

# Get the best split column and value
data_below, data_above = split_data(sub_trees[node],
    best_split_column, best_split_value)

# Append the new nodes
nodes = nodes.append({'parent': node, 'split':
    best_split_value, 'leaf': True, 'prediction': np.NaN, '
    children': [],
                      'num_obs': len(data_below)},
                         ignore_index=True)

nodes = nodes.append({'parent': node, 'split':
    best_split_value, 'leaf': True, 'prediction': np.NaN, '
    children': [],
                      'num_obs': len(data_above)},
                         ignore_index=True)

# Set 'leaf' to False because it now has children and is no
    longer a leaf
nodes.at[node, 'leaf'] = False

# Add the nodes of the children as a list to the observation
nodes.at[node, 'children'].append(index)
nodes.at[node, 'children'].append(index + 1)

# Append data
sub_trees.append(data_below)
sub_trees.append(data_above)

# Set splitting variable
nodes.at[node, 'split_var'] = sub_trees[node].iloc[:,
    best_split_column].name

# Make prediction (mode of the target variable of the
    observations in the leaf)
# Set prediction as value in DataFrame for given node
nodes.at[index, 'prediction'] = sub_trees[index][sub_trees[
    index].columns[-1]].mean()
```

```
    nodes.at[index + 1, 'prediction'] = sub_trees[index + 1][
        sub_trees[index + 1].columns[-1]].mean()

    # Make prediction (mode of the target variable of the
        observations in the leaf)
    # Set prediction as value in DataFrame for given node
    nodes.at[index, 'prediction'] = sub_trees[index][sub_trees[
        index].columns[-1]].mean()
    nodes.at[index + 1, 'prediction'] = sub_trees[index + 1][
        sub_trees[index + 1].columns[-1]].mean()

    return nodes, sub_trees


def empty(seq):
    """
    Check if a list of lists is empty.

        Parameters:
            seq (list): Nested list.

        Returns:
            True/False (Boolean): Returns boolean statement
                stating whether the sequence given to the function
                 is empty.
    """

    try:
        return all(map(empty, seq))

    except TypeError:
        return False


def leaves(nodes):
    """
    Find all of the children and the leaves of the internal nodes
        . The children and leaf of the leaf nodes are
    themselves.

        Parameters:
            nodes (pd.DataFrame): Structure of the tree, stored
                as a DataFrame.

        Returns:
```

```
        nodes (pd.DataFrame): DataFrame in parameters with
            two added columns, all_children and leaves.
"""

# Add leaves as a column to the DataFrame
nodes['leaves'] = ''

# Add all children as a column to the DataFrame
nodes['all_children'] = ''

# For all nodes
for node in range(len(nodes)):

    # For internal nodes
    if nodes.at[node, 'children'] != []:

        # First set of children of node
        all_children = nodes.at[node, 'children']
        new_children = nodes.at[node, 'children']

        # While not all of the children are NaN
        while not empty([nodes.at[x, 'children'] for x in
            new_children]):

            # Find the children of the children found prior
            new_children = [nodes.at[x, 'children'] for x in
                new_children]

            # If there is NaN value in the new_children list
            if np.NaN in new_children:

                # While there is a NaN value in the
                    new_children list
                while np.NaN in new_children:

                    # Remove NaN values from new_children
                        list
                    new_children.remove(np.NaN)

            # Remove the nested lists, so it is just one
                depth
            new_children = list(chain.from_iterable(
                new_children))
```

```
                # Append the new children to the list of all of
                    it's children
                all_children = all_children + new_children

            # Empty list to append leaves to
            leaves_list = []

            # For each node
            for obs_num in all_children:

                # If it is a leaf append to list
                if nodes.loc[obs_num]['leaf']:

                    # Append to list
                    leaves_list.append(obs_num)

                # If it is not a leaf then pass
                else:

                    pass

            # Add the new children and all children values to the
                DataFrame
            nodes.at[node, 'leaves'] = leaves_list
            nodes.at[node, 'all_children'] = all_children

        # For leaf nodes
        else:

            # Add the new children and all children values to the
                DataFrame
            nodes.at[node, 'leaves'] = int(node)
            nodes.at[node, 'all_children'] = int(node)

    return nodes


def build_tree(training_data, min_samples_split, max_depth):
    """
    Build the Decision Tree.

        Parameters:
            training_data (pd.DataFrame): Data used to build the
                tree.
```

```
        min_samples_split (int): Minimum number of
            observations in a node to qualify for a split.
        max_depth (int): Maximum depth of tree.

    Returns:
        nodes (pd.DataFrame): Structure of the tree, stored
            as a DataFrame.
        sub_trees (list): List of pd.DataFrames for each sub-
            tree.
"""

# Root node
nodes = root_node(training_data)

# Sets default max depth value
if max_depth is None:

    # Max depth is the number of inputs in the DataFrame
    max_depth = len(training_data.columns) - 1

# Append DataFrame from first node to list
sub_trees = [training_data]

# Loop for internal nodes
for node in range(2 ** max_depth - 1):

    try:

        if nodes.loc[node, 'num_obs'] < min_samples_split or
           nodes.loc[node, 'num_obs'] < 2:

            pass

        else:

            # Set index
            index = nodes.shape[0]

            # Get the split for the first data set
            best_split_column, best_split_value = get_split(
                sub_trees[node]) # , best_data_below,
                best_data_above

            # Get the best split column and value
```

```
data_below, data_above = split_data(sub_trees[
    node], best_split_column, best_split_value)

if len(data_below) > 1 and len(data_above) > 1:

    # Append the new nodes
    nodes = nodes.append({'parent': node, 'split'
        : best_split_value, 'leaf': True, '
        prediction': np.NaN,
                          'children': [], '
                            num_obs': len(
                            data_below)},
                            ignore_index=True)

    nodes = nodes.append({'parent': node, 'split'
        : best_split_value, 'leaf': True, '
        prediction': np.NaN,
                          'children': [], '
                            num_obs': len(
                            data_above)},
                            ignore_index=True)

    # Set 'leaf' to False because it now has
        children and is no longer a leaf
    nodes.at[node, 'leaf'] = False

    # Add the nodes of the children as a list to
        the observation
    nodes.at[node, 'children'].append(index)
    nodes.at[node, 'children'].append(index + 1)

    # Append data
    sub_trees.append(data_below)
    sub_trees.append(data_above)

    # Set splitting variable
    nodes.at[node, 'split_var'] = sub_trees[node
        ].iloc[:, best_split_column].name

    # Make prediction (mode of the target
        variable of the observations in the leaf)
    # Set prediction as value in DataFrame for
        given node
```

```
                    nodes.at[index, 'prediction'] = sub_trees[
                        index][sub_trees[index].columns[-1]].mean
                        ()
                    nodes.at[index + 1, 'prediction'] = sub_trees
                        [index + 1][sub_trees[index + 1].columns
                        [-1]].mean()

                    # Make prediction (mode of the target
                        variable of the observations in the leaf)
                    # Set prediction as value in DataFrame for
                        given node
                    nodes.at[index, 'prediction'] = sub_trees[
                        index][sub_trees[index].columns[-1]].mean
                        ()
                    nodes.at[index + 1, 'prediction'] = sub_trees
                        [index + 1][sub_trees[index + 1].columns
                        [-1]].mean()

                else:

                    pass

            except:

                break

        # Convert sub_trees to a series
        sub_trees = pd.Series(sub_trees)

        # If leaf equals true then splitting variable is nan
        nodes.loc[nodes['leaf'] == True, 'split_var'] = np.nan

        # Add column for the all children and the leaves of each node
        nodes = leaves(nodes)

        # Replace empty list value in nodes DataFrame with nan
        nodes = nodes.mask(nodes.applymap(str).eq('[]'))

        return nodes, sub_trees


def predict_tree(nodes, testing_data):
    """
    Make predictions using built decision tree.
```

```
    Parameters:
        nodes (pd.DataFrame): Structure of the tree, stored
            as a DataFrame.
        testing_data (pd.DataFrame): Data used to make
            predictions.

    Returns:
        predictions (list): Predicted values.
"""

# Reset index of testing_data
testing_data.reset_index(drop=True, inplace=True)

# List of predictions
predictions = []

# For each observations to predict
for i in range(len(testing_data)):

    # Set node = 0
    node = 0

    # While node is not a leaf node
    while nodes.at[node, 'leaf'] == False:

        # If testing data value <= corresponding split value
        if testing_data.at[i, nodes.at[node, 'split_var']] <=
            nodes.at[nodes.at[node, 'children'][0], 'split']:

            # Set node as the below split
            node = nodes.at[node, 'children'][0]

        else:

            # Set node as the above split
            node = nodes.at[node, 'children'][1]

    # Since leaf is true output the prediction
    predictions.append(nodes.at[node, 'prediction'])

predictions = pd.Series(predictions)

return predictions
```

```
def precision_at_k(actual, predictions, k):
    """
    Calculates the Precision at k (P@k).

        Parameters:
            actual (list): Actual values from the testing data.
            predictions (list): Predicted values from the
                decision tree.
            k (int): Cut-off value for P@k.

        Returns:
            pre_at_k (float): P@k value.
    """

    # Now rank predictions and testing_data
    ranked_predictions = pd.Series(ss.rankdata(predictions,
        method='min'))
    ranked_target_var = pd.Series(ss.rankdata(actual, method='min
        '))

    # Sort the ranked values
    ranked_predictions_sorted = ranked_predictions.sort_values()
    ranked_target_var_sorted = ranked_target_var.sort_values()

    # Ranked and sorted until k
    ranked_predictions_at_k = ranked_predictions_sorted[:k]
    ranked_target_var_at_k = ranked_target_var_sorted[:k]

    # Indices in top k
    top_k_pred_indices = ranked_predictions_at_k.index.values
    top_k_target_var_indices = ranked_target_var_at_k.index.
        values

    # True Positives
    tp = len(set(top_k_pred_indices) & set(
        top_k_target_var_indices))

    # False Positives
    fp = len(np.setdiff1d(top_k_pred_indices,
        top_k_target_var_indices))

    # Number of correct observations until k
    pre_at_k = tp / (tp + fp)

    return pre_at_k
```

```
def trim_tree_rank(nodes_copy, i):
    """
    Prune the given node in the decision tree.

        Parameters:
            nodes_copy (pd.DataFrame): Copy of the structure of
                the tree, stored as a DataFrame.
            i (int): Node number to be pruned.

        Returns:
            nodes (pd.DataFrame): Structure of the tree, stored
                as a DataFrame.
            trimmed_df (pd.DataFrame): Structure of the pruned
                tree, stored as a DataFrame.
    """

    # Make a copy of the nodes data frame
    nodes = pickle.loads(pickle.dumps(nodes_copy))

    # Change node to a leaf
    nodes.at[i, 'leaf'] = True

    # Nodes to drop
    nodes_to_drop = nodes.at[i, 'all_children']

    # Delete it's children nodes
    [nodes.drop(node, inplace=True) for node in nodes_to_drop]

    # Drop certain nodes
    droppable_nodes = nodes_to_drop.copy()

    for ind in nodes.index.values:

        for child in droppable_nodes:

            try:
                nodes.at[ind, 'all_children'].remove(child)

            except:
                pass

    # Set children & split_var columns to np.NaN
    nodes.at[i, 'children'] = np.NaN
```

```
nodes.at[i, 'split_var'] = np.NaN

# Set leaves & all_children columns to itself
nodes.at[i, 'leaves'] = i
nodes.at[i, 'all_children'] = i

try:

    # First parent of node
    new_parent = int(nodes.at[i, 'parent'])

except:

    new_parent = nodes.at[i, 'parent']

# While parent is not null
while np.isfinite(new_parent):

    # Add the newly created leaf to its parents leaves &
       all_children column
    nodes.at[new_parent, 'leaves'].append(i)
    nodes.at[new_parent, 'all_children'].append(i)

    # Delete potential duplicates
    nodes.at[new_parent, 'all_children'] = list(set(nodes.at[
       new_parent, 'all_children']))

    # Sort values in the list of leaves & all_children
    nodes.at[new_parent, 'leaves'].sort()
    nodes.at[new_parent, 'all_children'].sort()

    try:

        # Find parent of the parent found prior
        new_parent = int(nodes.at[new_parent, 'parent'])

    except:

        # Find parent of the parent found prior
        new_parent = nodes.at[new_parent, 'parent']

# Append the trimmed DataFrame to the list
trimmed_df = pickle.loads(pickle.dumps(nodes))

return nodes, trimmed_df
```

```
def backwards_trim_mse(data, min_samples_split, max_depth):
    """
    Backwards elimination pruning of the decision tree using MSE.

        Parameters:
            data (pd.DataFrame): Data used to build, prune and
                evaluate the decision trees.
            min_samples_split (int): Minimum number of samples in
                 a node to qualify for a split.
            max_depth (int): Maximum tree depth.

        Returns:
            df (pd.DataFrame): Information for pruning iteration
                number, node trimmed, number of leaves and MSE.
            nodes_list (list): List of pd.DataFrames that were
                prior to pruning.
            trimmed_dfs (list): List of pd.DataFrames that were
                pruned.
    """

    # Empty list to append values to
    mse_list = []
    node_trimmed = []
    iteration = []
    trimmed_dfs = []
    nodes_list = []
    num_leaves_list = []

    # Create DataFrame
    df = pd.DataFrame()

    # Split into train test sets
    training_data, testing_data = train_test_split(data,
        random_state=1)

    # Build fully grown tree
    full_tree, sub_trees = build_tree(training_data,
        min_samples_split, max_depth)

    # Append tree to list
    nodes_list.append(full_tree)

    # Make predictions using testing data
```

```
predictions = predict_tree(full_tree, testing_data)

# Calculate mse
mse_list.append(mean_squared_error(testing_data.iloc[:, -1:],
    predictions))
node_trimmed.append('Full_Tree')
iteration.append(1)

# Save the number of leaves in each tree
num_leaves_list.append(full_tree['leaf'].value_counts()[True
    ])

# Append leaf nodes
node_trimmed.extend(list(full_tree[full_tree['leaf'] == False
    ].index.values))

# Trim each leaf node
for i in node_trimmed[1:]:

    # Trim node i in tree
    nodes, trimmed_df = trim_tree_rank(full_tree, i)

    # Append to list
    nodes_list.append(nodes)
    trimmed_dfs.append(trimmed_df)
    iteration.append(1)

    # Save the number of leaves in each tree
    num_leaves_list.append(trimmed_df['leaf'].value_counts()[
        True])

    # Make predictions using testing data
    predictions = predict_tree(nodes, testing_data)

    # Append mse values
    mse_list.append(mean_squared_error(testing_data.iloc[:,
        -1:], predictions))

# Create DataFrame using lists
mse_df = pd.DataFrame(data=list(zip(iteration, node_trimmed,
    num_leaves_list, mse_list)),
                      columns=['Iteration', 'Node_Trimmed', '
                          Num_Leaves', 'MSE'])

# Append to DataFrame that will be used for all iterations
```

```
df = df.append(mse_df)

# If there is only 1 tree
if len(mse_df) == 1:

    mse_df_subset = mse_df.copy(deep=True)

else:

    # Exclude full tree
    mse_df_subset = mse_df.iloc[1:]

# Reset index
mse_df_subset.reset_index(drop=True, inplace=True)

# Find the node at the current iteration with the maximum
    precision @ k value, take the deepest node if there is
# a tie
optimal_node = mse_df_subset[mse_df_subset['MSE'] ==
    mse_df_subset['MSE'].min()].iloc[[-1]].index.values[0]

# Convert to integer
try:

    optimal_node = int(optimal_node)

except:

    optimal_node

if optimal_node == 0:

    optimal_nodes = pickle.loads(pickle.dumps(full_tree))

else:

    # Save the dataframe to be used for next trim
    optimal_nodes = pickle.loads(pickle.dumps(trimmed_dfs[
        optimal_node]))

while len(mse_df) > 1:

    # Test precision at each depth of tree
    # Prune at each leaf node
    trimmed_dfs = []
```

```
node_trimmed = []
iteration = []
mse_list = []
num_leaves_list = []

node_trimmed.extend(list(optimal_nodes[optimal_nodes['
    leaf'] == False].index.values))

if node_trimmed != []:

    for i in node_trimmed:

        nodes, trimmed_df = trim_tree_rank(optimal_nodes,
            i)

        nodes_list.append(nodes)

        trimmed_dfs.append(trimmed_df)
        iteration.append(df.iloc[[-1]].values[0][0] + 1)

        # Save the number of leaves in each tree
        num_leaves_list.append(trimmed_df['leaf'].
            value_counts()[True])

        # Make predictions using testing data
        predictions = predict_tree(nodes, testing_data)

        # Append mse values
        mse_list.append(mean_squared_error(testing_data.
            iloc[:, -1:], predictions))

    mse_df = pd.DataFrame(data=list(zip(iteration,
        node_trimmed, num_leaves_list, mse_list)),
                          columns=['Iteration', '
                              Node_Trimmed', 'Num_Leaves',
                               'MSE'])

    # Reset index
    mse_df.reset_index(drop=True, inplace=True)

    df = df.append(mse_df)

    optimal_node = mse_df[mse_df['MSE'] == mse_df['MSE'].
        min()].iloc[[-1]].index.values[0]
```

```
            try:
                optimal_node = int(optimal_node)

            except:

                optimal_node

            optimal_nodes = pickle.loads(pickle.dumps(trimmed_dfs
                [optimal_node]))

        else:

            break

    df.reset_index(drop=True, inplace=True)

    return df, nodes_list, trimmed_dfs


def backwards_trim_precision(data, min_samples_split, max_depth,
    k):
    """
    Backwards elimination pruning of the decision tree using MSE.

        Parameters:
            data (pd.DataFrame): Data used to build, prune and
                evaluate the decision trees.
            min_samples_split (int): Minimum number of samples in
                a node to qualify for a split.
            max_depth (int): Maximum tree depth.
            k (int): Cut-off for AP@k calculation.

        Returns:
            df (pd.DataFrame): Information for pruning iteration
                number, node trimmed, number of leaves and MSE.
            nodes_list (list): List of pd.DataFrames that were
                prior to pruning.
            trimmed_dfs (list): List of pd.DataFrames that were
                pruned.
    """

    # Empty list to append values to
    precision_at_k_values = []
    node_trimmed = []
    iteration = []
```

```
trimmed_dfs = []
nodes_list = []
num_leaves_list = []

# Create DataFrame
df = pd.DataFrame()

# Split into train test sets
training_data, testing_data = train_test_split(data,
    random_state=1)

# Build fully grown tree
full_tree, sub_trees = build_tree(training_data,
    min_samples_split, max_depth)

# Append tree to list
nodes_list.append(full_tree)

# Make predictions using testing data
predictions = predict_tree(full_tree, testing_data)

# Calculate precision at k
precision_at_k_values.append(precision_at_k(testing_data,
    predictions, k))
node_trimmed.append('Full_Tree')
iteration.append(1)

# Save the number of leaves in each tree
num_leaves_list.append(full_tree['leaf'].value_counts()[True
    ])

# Append leaf nodes
node_trimmed.extend(list(full_tree[full_tree['leaf'] == False
    ].index.values))

# Trim each leaf node
for i in node_trimmed[1:]:

    # Trim node i in tree
    nodes, trimmed_df = trim_tree_rank(full_tree, i)
    nodes_list.append(nodes)
    trimmed_dfs.append(trimmed_df)
    iteration.append(1)

    # Save the number of leaves in each tree
```

```
        num_leaves_list.append(trimmed_df['leaf'].value_counts()[
            True])

        # Make predictions using testing data
        predictions = predict_tree(nodes, testing_data)

        # Append precision at k values
        precision_at_k_values.append(precision_at_k(testing_data,
            predictions, k))

    # Create DataFrame using lists
    precision_df = pd.DataFrame(data=list(zip(iteration,
        node_trimmed, num_leaves_list, precision_at_k_values)),
                                columns=['Iteration', '
                                    Node_Trimmed', 'Num_Leaves', '
                                    Precision_at_K'])

    # Append to DataFrame that will be used for all iterations
    df = df.append(precision_df)

    # If there is only 1 tree
    if len(precision_df) == 1:

        precision_df_subset = precision_df.copy(deep=True)

    else:

        # Exclude full tree
        precision_df_subset = precision_df.iloc[1:]

    # Reset index
    precision_df_subset.reset_index(drop=True, inplace=True)

    # Find the node at the current iteration with the maximum
        precision @ k value, take the deepest node if there
    # is a tie
    optimal_node = precision_df_subset[precision_df_subset['
        Precision_at_K'] == precision_df_subset['Precision_at_K'].
        max()].iloc[[-1]].index.values[0]

    # Convert to integer
    try:

        optimal_node = int(optimal_node)
```

```
except:

    optimal_node

if optimal_node == 0:

    optimal_nodes = pickle.loads(pickle.dumps(full_tree))

else:

    # Save the dataframe to be used for next trim
    optimal_nodes = pickle.loads(pickle.dumps(trimmed_dfs[
       optimal_node]))

while len(precision_df) > 1:

    # Test precision at each depth of tree
    # Prune at each leaf node
    trimmed_dfs = []
    node_trimmed = []
    iteration = []
    precision_at_k_values = []
    num_leaves_list = []

    node_trimmed.extend(list(optimal_nodes[optimal_nodes['
       leaf'] == False].index.values))

    if node_trimmed != []:

        for i in node_trimmed:

            nodes, trimmed_df = trim_tree_rank(optimal_nodes,
                i)
            nodes_list.append(nodes)
            trimmed_dfs.append(trimmed_df)
            iteration.append(df.iloc[[-1]].values[0][0] + 1)

            # Save the number of leaves in each tree
            num_leaves_list.append(trimmed_df['leaf'].
               value_counts()[True])

            # Make predictions using testing data
            predictions = predict_tree(nodes, testing_data)
```

```
                precision_at_k_values.append(precision_at_k(
                    testing_data, predictions, k))

            precision_df = pd.DataFrame(data=list(zip(iteration,
                node_trimmed, num_leaves_list,
                precision_at_k_values)),
                                        columns=['Iteration', '
                                            Node_Trimmed', '
                                            Num_Leaves', '
                                            Precision_at_K'])

            # Reset index
            precision_df.reset_index(drop=True, inplace=True)

            df = df.append(precision_df)

            optimal_node = precision_df[precision_df['
                Precision_at_K'] == precision_df['Precision_at_K'
                ].max()].iloc[
                 [-1]].index.values[0]

            try:

                optimal_node = int(optimal_node)

            except:

                optimal_node

            optimal_nodes = pickle.loads(pickle.dumps(trimmed_dfs
                [optimal_node]))

        else:

            break

    df.reset_index(drop=True, inplace=True)

    return df, nodes_list, trimmed_dfs


def optimal_nodes_function(data, min_samples_split, max_depth, k)
    :
    """
```

```
    Get the built and pruned decision tree using MSE and AP@k as
        pruning loss functions.

        Parameters:
            data (pd.DataFrame): Data used to build, prune and
                evaluate the decision trees.
            min_samples_split (int): Minimum number of samples in
                 a node to qualify for a split.
            max_depth (int): Maximum tree depth.
            k (int): Cut-off for AP@k calculation.

        Returns:
            optimal_nodes_mse (pd.DataFrame): Structure of the
                built and pruned decision tree using MSE.
            optimal_nodes_precision (pd.DataFrame): Structure of
                the built and pruned decision tree using AP@k.
    """

    mse_results, mse_nodes, mse_trimmed_dfs = backwards_trim_mse(
        data, min_samples_split, max_depth)

    precision_results, precision_nodes, precision_trimmed_dfs =
        backwards_trim_precision(data, min_samples_split,









    optimal_nodes_mse = mse_nodes[mse_results.iloc[mse_results.
        iloc[:, -1:].idxmin()].index.values[0]]
    optimal_nodes_precision = precision_nodes[precision_results.
        iloc[precision_results.iloc[:, -1:].idxmax()].index.values
        [0]]

    return optimal_nodes_mse, optimal_nodes_precision


def rank_obs(actual, predicted):
    """
    Rank the predicted values and the actual values.

        Parameters:
```

```
            actual (pd.Series): Actual results from the testing
                data.
            predicted (list): Predicted results using the model.

        Returns:
            actual (list): Sorted list of actual results from the
                testing data.
            predicted (list): Sorted list of predicted results
    """

    # Sort observations
    predicted.sort()

    # Actual testing_data
    actual = actual.tolist()
    actual.sort()

    # Now rank predictions and testing_data
    predicted = ss.rankdata(predicted, method='min')
    actual = ss.rankdata(actual, method='min')

    # Convert to list
    predicted = predicted.tolist()
    actual = actual.tolist()

    return actual, predicted


def obtain_dt_results(data, min_samples_split=2, max_depth=10, k
    =20, testing_data=None):
    """
    Obtain AP@k and precision results for MSE and AP@k built and
        pruned decision trees.

        Parameters:
            data (pd.DataFrame): Data used to build, prune and
                evaluate the decision trees.
            min_samples_split (int): Minimum number of samples in
                a node to qualify for a split.
            max_depth (int): Maximum tree depth.
            k (int): Cut-off for AP@k calculation.
            testing_data (pd.DataFrame): If None, split data into
                training and testing. If not None, use the
                pd.DataFrame as testing data and use data as the
                training data.
```

```
    Returns:
        mse_apk (float): AP@k value for decision tree built
            and pruned using MSE.
        mse_pk (float): Precision value for decision tree
            built and pruned using MSE.
        precision_apk (float): AP@k value for decision tree
            built and pruned using AP@k.
        precision_pk (float): Precision value for decision
            tree built and pruned using AP@k.
"""

if testing_data is None:

    # Split into training and testing data
    training_data, testing_data = train_test_split(data,
        random_state=1)

else:

    training_data = data.copy(deep=True)

# Get optimal nodes
mse_nodes, precision_nodes = optimal_nodes_function(
    training_data, min_samples_split, max_depth, k)

# Get predictions
mse_predictions = predict_tree(mse_nodes, testing_data).
    tolist()
precision_predictions = predict_tree(precision_nodes,
    testing_data).tolist()

# Rank observations
actual, mse_ranked_predictions = rank_obs(testing_data.iloc
    [:, -1], mse_predictions)
actual, precision_ranked_predictions = rank_obs(testing_data.
    iloc[:, -1], precision_predictions)

# Get apk values
mse_apk = metrics.apk(actual, mse_ranked_predictions, k)
precision_apk = metrics.apk(actual,
    precision_ranked_predictions, k)

# Get precision at k
```

```
mse_pk = precision_at_k(testing_data.iloc[:, -1],
    mse_ranked_predictions, k)
precision_pk = precision_at_k(testing_data.iloc[:, -1],
    precision_ranked_predictions, k)

return mse_apk, mse_pk, precision_apk, precision_pk
```

C.2 AdaBoost Code

Listing 2: Python code for building a regression AdaBoost algorithm with the ability to use a ranking pruning loss function

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
@author: bradklassen
"""

# Import libraries
import pandas as pd
import numpy as np
import pickle
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split
import ml_metrics as metrics
import DT_Rank as dt_rank


def create_weights(training_data):
    """
    Create the initial weights.

        Parameters:
            training_data (pd.DataFrame): Training data used to
                build the AdaBoost model.

        Returns:
            weight (pd.Series): Weight assigned to each
                observation to determine probability of being
                resampled.
    """

    # Initial weight is equal for all observations
    weight = [1 / len(training_data)] * len(training_data)

    # Convert to Series
    weight = pd.Series(weight)

    return weight


def adjust_weight(stump, new_data, weight):
```

```
    """
    Change the weights which are used to place more importance on
        previously incorrectly predicted observations,
    and less importance on previously correctly predicted
       observations.

        Parameters:
            stump (pd.DataFrame): Estimator used to make the
                AdaBoost model.
            new_data (pd.DataFrame): Re-sampled data.
            weight (pd.Series): Weight assigned to each
                observation to determine probability of being
                resampled.

        Returns:
            new_weight (pd.Series): Normalized adjusted weights.
            beta (float): Beta is a measure of how well the
                estimators predicts the observations.
    """

    # Calculate difference between predicted and actual (MSE for
        each observation)
    err = np.abs(new_data.iloc[:, -1] - dt_rank.predict_tree(
       stump, new_data))

    # Normalize losses by dividing each by the maximum loss
    max_err = err.max()
    normalized_err = err / max_err

    # Take average of all losses
    avg_loss = normalized_err.mean()

    # Compute beta which is average loss divided by 1 - average
        loss
    beta = avg_loss / (1 - avg_loss)

    # Update weights w[i] = w[i] * beta ^ 1 - loss[i] (where loss
        [i] is the normalized loss for the given observation)
    new_weight = weight * beta ** (1 - normalized_err)

    # Replace null values with 0
    new_weight = new_weight.fillna(0)

    # Small error term that is added in case total error is 0 or
        1
```

```
    new_weight = new_weight + (10 ** -10)

    # Re-normalize (weight/sum(weight))
    new_weight = new_weight / sum(new_weight)

    return new_weight, beta


def resample_data(training_data, weight):
    """
    Resample the data based on the weights.

        Parameters
            training_data (pd.DataFrame): Training data used to
                build the AdaBoost model.
            weight (pd.Series): Weight assigned to each
                observation to determine probability of being
                resampled.

        Returns:
            new_data (pd.DataFrame): Re-sampled data.
    """

    # Set seed
    np.random.seed(1)

    # Generate indices to be used in data to create next stump
    resampled_indices = np.random.choice(len(training_data), len(
        training_data), p=weight)

    # Create DataFrame using generated indices
    new_data = pd.DataFrame(training_data.iloc[resampled_indices
        ])

    # Reset index
    new_data.reset_index(drop=True, inplace=True)

    return new_data


def adaboost(training_data, num_estimators, min_samples_split):
    """
    Create the AdaBoost model using above functions.

        Parameters:
```

```
        training_data (pd.DataFrame): Data used to build the
            AdaBoost model
        num_estimators (int): Number of estimators used to
            build the AdaBoost model.
        min_samples_split (int): Minimum number of samples in
             a node to qualify for a split.

    Returns:
        stumps (list): List of pd.DataFrames that are the
            structure of the estimators used to build the
            model.
        beta_vals (list): Beta values corresponding to a
            measure of how well the estimators predict the
             observations.
"""

# Empty list for appending stumps, amount of say & DataFrames
stumps = []
beta_vals = []

# Create new_data as a copy of training_data
new_data = pickle.loads(pickle.dumps(training_data))

# For each estimator
for i in range(num_estimators):

    # Create stump
    stump, sub_trees = dt_rank.build_tree(new_data,
        min_samples_split, max_depth=1)

    # Create the stump, list of sub trees and the weight for
        each observation
    weight = create_weights(new_data)

    # Append the stumps DataFrame to a list
    stumps.append(pickle.loads(pickle.dumps(stump)))

    # Adjust weight based on incorrect observations from
        previous node
    weight, beta = adjust_weight(stump, new_data, weight)

    # Append the amount of say to the created list
    beta_vals.append(beta)

    # Resample the data using the new weights
```

```
            new_data = resample_data(new_data, weight)

            # Reset the index of the new data set
            new_data.reset_index(drop=True, inplace=True)

        # Beta values equal to zero
        null_beta = [i for i, item in enumerate(beta_vals) if item ==
            0]

        # Remove beta values and stumps where beta is zero
        for index in sorted(null_beta, reverse=True):
            del beta_vals[index]
            del stumps[index]

        return stumps, beta_vals


def ada_predict(testing_data, stumps, beta_vals):
    """
    Makes predictions by using the stumps created from AdaBoost.

        Parameters:
            testing_data (pd.DataFrame): Data used to make
                predictions.
            stumps (list): List of pd.DataFrames that are the
                structure of the estimators used to build the
                model
            beta_vals (list): Beta values corresponding to a
                measure of how well the estimators predict the
                 observations.

        Returns:
            predictions (pd.Series): Predictions made by AdaBoost
                 model.
    """

    # List to append values to
    ada_predictions = []
    beta = []
    observations = []

    # Prediction for each stump
    for stump in range(len(stumps)):

        # Obtain predictions
```

```
        predictions = dt_rank.predict_tree(stumps[stump],
            testing_data)

        # Append the predictions
        ada_predictions.append(predictions)

        # Append beta values
        beta.append(list(np.repeat(beta_vals[stump], len(
            testing_data))))

        # Append the observation number
        observations.append(testing_data.index.values)

    # Flatten lists
    ada_predictions = [item for sublist in ada_predictions for
        item in sublist]
    beta = [item for sublist in beta for item in sublist]
    say = [1 / x for x in beta]
    observations = [item for sublist in observations for item in
        sublist]

    # Use list to create DataFrame
    predictions_df = pd.DataFrame({'observation': observations, '
        predictions': ada_predictions, 'say': say})

    # Sum of say
    total_say = predictions_df[predictions_df['observation'] ==
        0]['say'].sum()

    # Normalize the say
    predictions_df['normalized_say'] = predictions_df['say'] /
        total_say

    # Multiply predictions by the normalized say
    predictions_df['pred_say'] = predictions_df['predictions'] *
        predictions_df['normalized_say']

    # Get predictions
    predictions = predictions_df.groupby(['observation'])['
        pred_say'].sum()

    return predictions
```

```
def ada_backwards_trim_mse(data, num_estimators,
    min_samples_split):
    """
    Backwards elimination pruning of the AdaBoost model using MSE
        .

        Parameters:
            data (pd.DataFrame): Data used to build, prune and
                evaluate the AdaBoost model.
            num_estimators (int): Number of estimators used to
                build the AdaBoost model.
            min_samples_split (int): Minimum number of samples in
                 a node to qualify for a split.

        Returns:
            ada_stumps (list): Estimators in the AdaBoost model.
            ada_mse (list): MSE value for each model.
            ada_beta_vals (list): Beta values corresponding to a
                measure of how well the estimators predict the
                 observations.
    """

    # List to append values to
    ada_stumps = []
    ada_mse = []
    ada_beta_vals = []

    # Create DataFrame
    df = pd.DataFrame()

    # Split into train test sets
    training_data, testing_data = train_test_split(data,
        random_state=1)

    # Reset indices
    training_data.reset_index(inplace=True, drop=True)
    testing_data.reset_index(inplace=True, drop=True)

    # Build fully grown tree
    stumps, beta_vals = adaboost(training_data, num_estimators,
        min_samples_split)

    # Iteration counter
    iteration_num = 0
```

```
# Empty list to append values to
min_index_list = []

# While there are stumps
while len(stumps) > 1:

    # Append mse values
    mse = []
    indices_list = []
    iteration = []

    # Add 1 to the iteration number
    iteration_num += 1

    # Indices in stumps list
    indices = [i for i, x in enumerate(stumps)]

    for ind in indices:

        # Append index value
        indices_list.append(ind)

        # Make predictions using testing data
        predictions = dt_rank.predict_tree(stumps[ind],
            testing_data)

        # Append mse values
        mse.append(mean_squared_error(testing_data.iloc[:,
            -1:], predictions))

        # Append iteration number
        iteration.append(iteration_num)

    # Create DataFrame using lists
    mse_df = pd.DataFrame(data=list(zip(iteration,
        indices_list, mse)), columns=['Iteration', 'Index', '
        MSE'])

    # Append to DataFrame that will be used for all
        iterations
    df = df.append(mse_df)

    # Minimum index
    min_index = mse_df[mse_df['MSE'] == mse_df['MSE'].min()][
        'Index'].values[0]
```

```
            # Delete stump
            del stumps[min_index]

            # Append the index values
            min_index_list.append(min_index)

            # Remove beta value that corresponds to minimum index
            del beta_vals[min_index]

            # Get predictions using AdaBoost results
            predictions = ada_predict(testing_data, stumps, beta_vals
                )

            # Append MSE values
            ada_mse.append(mean_squared_error(testing_data.iloc[:,
                -1:], predictions))

            # Append stumps
            ada_stumps.append(stumps.copy())

            # Append beta values
            ada_beta_vals.append(beta_vals.copy())

    return ada_stumps, ada_mse, ada_beta_vals


def ada_backwards_trim_precision(data, num_estimators,
    min_samples_split, k):
    """
    Backwards elimination pruning of the AdaBoost model using
        AP@k.

        Parameters:
            data (pd.DataFrame): Data used to build, prune and
                evaluate the AdaBoost model.
            num_estimators (int): Number of estimators used to
                build the AdaBoost model.
            min_samples_split (int): Minimum number of samples in
                 a node to qualify for a split.
            k (int): Cut-off for AP@k calculation.

        Returns:
            ada_stumps (list): Estimators in the AdaBoost model.
            ada_pak (list): MSE value for each model.
```

```
        ada_beta_vals (list): Beta values corresponding to a
            measure of how well the estimators predict the
            observations.
"""

# List to append values to
ada_pak = []
ada_stumps = []
ada_beta_vals = []

# Create DataFrame
df = pd.DataFrame()

# Split into train test sets
training_data, testing_data = train_test_split(data,
    random_state=1)

# Reset indices
training_data.reset_index(inplace=True, drop=True)
testing_data.reset_index(inplace=True, drop=True)

# Build fully grown tree
stumps, beta_vals = adaboost(training_data, num_estimators,
    min_samples_split)

# Iteration counter
iteration_num = 0

# Empty list to append values to
min_index_list = []

# While there are stumps
while len(stumps) > 1:

    # Append mse values
    pak = []
    indices_list = []
    iteration = []

    # Add 1 to the iteration number
    iteration_num += 1

    # Indices in stumps list
    indices = [i for i, x in enumerate(stumps)]
```

```
for ind in indices:

    # Append index value
    indices_list.append(ind)

    # Make predictions using testing data
    predictions = dt_rank.predict_tree(stumps[ind],
        testing_data)

    # Append pak values
    pak.append(dt_rank.precision_at_k(testing_data,
        predictions, k))

    # Append iteration number
    iteration.append(iteration_num)

# Create DataFrame using lists
pak_df = pd.DataFrame(data=list(zip(iteration,
    indices_list, pak)),
                      columns=['Iteration', 'Index', '
                          Precision_at_K'])

# Append to DataFrame that will be used for all
    iterations
df = df.append(pak_df)

# Minimum index
min_index = pak_df[pak_df['Precision_at_K'] == pak_df['
    Precision_at_K'].min()]['Index'].values[0]

# Delete stump
del stumps[min_index]

# Append the index values
min_index_list.append(min_index)

# Remove beta value that corresponds to minimum index
del beta_vals[min_index]

# Get predictions using AdaBoost results
predictions = ada_predict(testing_data, stumps, beta_vals
    )

# Append MSE values
```

```
        ada_pak.append(dt_rank.precision_at_k(testing_data,
            predictions, k))

        # Append stumps
        ada_stumps.append(stumps.copy())

        # Append beta values
        ada_beta_vals.append(beta_vals.copy())

    return ada_stumps, ada_pak, ada_beta_vals


def optimal_stumps_function(data, num_estimators,
    min_samples_split, k):
    """
    Get the built and pruned AdaBoost model using MSE and AP@k as
        pruning loss functions.

        Parameters:
            data (pd.DataFrame): Data used to build, prune and
                evaluate the Adaboost model.
            num_estimators (int): Number of estimators used to
                build the AdaBoost model.
            min_samples_split (int): Minimum number of samples in
                a node to qualify for a split.
            k (int): Cut-off for AP@k calculation.

        Returns:
            optimal_stumps_mse (pd.DataFrame): Structure of the
                built and pruned AdaBoost model using MSE.
            optimal_beta_mse (float): Beta value corresponding to
                the optimal AdaBoost model using MSE.
            optimal_stumps_precision (pd.DataFrame): Structure of
                the built and pruned AdaBoost model using AP@k.
            optimal_beta_precision (float): Beta value
                corresponding to the optimal AdaBoost model using
                AP@k.
    """

    ada_mse_stumps, ada_mse_results, ada_mse_beta =
        ada_backwards_trim_mse(data, num_estimators,
        min_samples_split)
    ada_precision_stumps, ada_precision_results,
        ada_precision_beta = ada_backwards_trim_precision(data,
        num_estimators,
```

```
    optimal_stumps_mse = ada_mse_stumps[np.argmin(ada_mse_results
        )]
    optimal_beta_mse = ada_mse_beta[np.argmin(ada_mse_results)]

    optimal_stumps_precision = ada_precision_stumps[np.argmax(
        ada_precision_results)]
    optimal_beta_precision = ada_precision_beta[np.argmax(
        ada_precision_results)]

    return optimal_stumps_mse, optimal_beta_mse,
        optimal_stumps_precision, optimal_beta_precision


def obtain_ada_results(data, num_estimators=50, min_samples_split
    =2, k=20, testing_data=None):
    """
    Obtain AP@k and precision results for MSE and AP@k built and
        pruned AdaBoost model.

        Parameters:
            data (pd.DataFrame): Data used to build, prune and
                evaluate the AdaBoost model.
            num_estimators (int): Number of estimators used to
                build the AdaBoost model.
            min_samples_split (int): Minimum number of samples in
                 a node to qualify for a split.
            k (int): Cut-off for AP@k calculation.
            testing_data (pd.DataFrame): If None, split data into
                 training and testing. If not None, use the
                pd.DataFrame as testing data and use data as the
                    training data.

        Returns:
            mse_apk (float): AP@k value for the AdaBoost model
                built and pruned using MSE.
            mse_pk (float): Precision value for the AdaBoost
                model built and pruned using MSE.
```

```
        precision_apk (float): AP@k value for the AdaBoost
            model built and pruned using AP@k.
        precision_pk (float): Precision value for the
            AdaBoost model built and pruned using AP@k.
"""

if testing_data is None:

    # Split into training and testing data
    training_data, testing_data = train_test_split(data,
        random_state=1)

else:

    training_data = data.copy(deep=True)

# Get optimal stumps
mse_stumps, mse_beta_values, precision_stumps,
    precision_beta_values = optimal_stumps_function(
    training_data,









# Get predictions
mse_predictions = ada_predict(testing_data, mse_stumps,
    mse_beta_values).to_list()
precision_predictions = ada_predict(testing_data,
    precision_stumps, precision_beta_values).to_list()

# Rank observations
actual, mse_ranked_predictions = dt_rank.rank_obs(
    testing_data.iloc[:, -1], mse_predictions)
actual, precision_ranked_predictions = dt_rank.rank_obs(
    testing_data.iloc[:, -1], precision_predictions)

# Get apk values
mse_apk = metrics.apk(actual, mse_ranked_predictions, k)
```

```
precision_apk = metrics.apk(actual,
    precision_ranked_predictions, k)

# Get precision at k
mse_pk = dt_rank.precision_at_k(testing_data.iloc[:, -1],
    mse_ranked_predictions, k)
precision_pk = dt_rank.precision_at_k(testing_data.iloc[:,
    -1], precision_ranked_predictions, k)

return mse_apk, mse_pk, precision_apk, precision_pk
```

References

Classification: Precision and recall — machine learning crash course. URL https://developers.google.com/machine-learning/crash-course/classification/precision-and-recall.

Official home of golf and the fedexcup. URL https://www.pgatour.com/.

Handicap index statistics. URL https://www.usga.org/content/usga/home-page/handicapping/handicapping-stats.html.

Leo Breiman, Jerome H Friedman, Richard A Olshen, and Charles J Stone. *Classification and regression trees*. Routledge, 2017.

Jason Brownlee. Overfitting and underfitting with machine learning algorithms, Aug 2019. URL https://machinelearningmastery.com/overfitting-and-underfitting-with-machine-learning-algorithms/.

Jason Brownlee. Boosting and adaboost for machine learning, Aug 2020. URL https://machinelearningmastery.com/boosting-and-adaboost-for-machine-learning/.

Ron Bryant. URL http://www.shotlink.com/.

Chris Burges, Tal Shaked, Erin Renshaw, Ari Lazier, Matt Deeds, Nicole Hamilton, and Greg Hullender. Learning to rank using gradient descent. In *Proceedings of the 22nd international conference on Machine learning*, pages 89–96, 2005.

Christopher JC Burges. From ranknet to lambdarank to lambdamart: An overview. *Learning*, 11(23-581):81, 2010.

Zhe Cao, Tao Qin, Tie-Yan Liu, Ming-Feng Tsai, and Hang Li. Learning to rank: from pairwise approach to listwise approach. In *Proceedings of the 24th international conference on Machine learning*, pages 129–136, 2007.

Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794, 2016.

Harris Drucker. Improving regressors using boosting techniques. *Proceedings of the 14th International Conference on Machine Learning*, 08 1997.

Felipe. Evaluation metrics for ranking problems: Introduction and examples, Apr 2020. URL https://queirozf.com/entries/evaluation-metrics-for-ranking-problems-introduction-and-examples.

Yoav Freund, Robert Schapire, and Naoki Abe. A short introduction to boosting. *Journal-Japanese Society For Artificial Intelligence*, 14(771-780):1612, 1999.

Yoav Freund, Raj Iyer, Robert E Schapire, and Yoram Singer. An efficient boosting algorithm for combining preferences. *Journal of machine learning research*, 4(Nov):933–969, 2003.

Megan Graham and Jennifer Elias. How google's $150 billion advertising business works, May 2021. URL https://www.cnbc.com/2021/05/18/how-does-google-make-money-advertising-business-breakdown-.html.

D. Harrison and D.L. Rubinfeld. The boston housing dataset. 1978.

Trevor Hastie, Jerome Friedman, and Robert Tisbshirani. *The Elements of statistical learning: data mining, inference, and prediction*. Springer, 2017.

Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An introduction to statistical learning: with applications in R*. Springer, 2021.

Tim P. Morris, Ian R. White, and Michael J. Crowther. Using simulation studies to evaluate statistical methods. *Statistics in Medicine*, 38(11):2074–2102, 2019. doi: 10.1002/sim.8086.

Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

Robert Preston. How is the cut determined in golf tournaments?, Sep 2020. URL https://golftips.golfweek.usatoday.com/cut-determined-golf-tournaments-1857.html.

J. Ross Quinlan. Induction of decision trees. *Machine learning*, 1(1):81–106, 1986.

J Ross Quinlan. *C4.5: programs for machine learning*. Elsevier, 2014.

J.r. Quinlan. Simplifying decision trees. *International Journal of Man-Machine Studies*, 27 (3):221–234, 1987. doi: 10.1016/s0020-7373(87)80053-6.

Josh Starmer, Nov 2019. URL https://www.youtube.com/watch?v=D0efHEJsfHo&t=730s.

Michael Taylor, John Guiver, Stephen Robertson, and Tom Minka. Softrank: optimizing non-smooth rank metrics. In *Proceedings of the 2008 International Conference on Web Search and Data Mining*, pages 77–86, 2008.

Jun Xu and Hang Li. Adarank: a boosting algorithm for information retrieval. In *Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 391–398, 2007a.

Jun Xu and Hang Li. Adarank. *Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval - SIGIR 07*, 2007b. doi: 10.1145/1277741.1277809.