November 2019

Fractional Random Weighted Bootstrapping for Classification on Imbalanced Data with Ensemble Decision Tree Methods

Sean Charles Carter
University of South Florida

Fractional Random Weighted Bootstrapping for Classification on Imbalanced Data with Ensemble Decision Tree Methods

by

Sean Charles Carter

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Master of Arts
Department of Mathematics and Statistics
College of Art and Sciences
University of South Florida

Major Professor: Lu Lu, Ph.D.
Mingyang Li, Ph.D.
Seung-Yeop Lee, Ph.D.

Date of Approval:
October 30th, 2019

Keywords:Statistics, Classification, Bootstrapping, Machine Learning, Decision Trees

Table of Contents

List of Figures

List of Tables

Abstract

Ensemble methods are commonly used for building predictive models for classification. Models that are unstable to perturbations in the training set, such as the decision tree, often see considerable reductions in error when grouped, using bootstrapped resamples of the training data to train many models.

The non-parametric bootstrap, however, has limited efficacy when used on severely imbalanced data, especially when the number of observations of one or more classes is exceptionally small. We explore the fractional random weighted bootstrap, which randomly assigns fractional weights to observations, as an alternative resampling procedure in training machine learning ensembles, particularly decision tree ensembles.

We carry out a methodological study comparing the standard bagging and random forest ensemble models for decision trees against their fractionally random weighted alternatives, finding some evidence supporting their use on data with severe imbalance.

1 Introduction

Among the most common tasks faced by statisticians and data scientists are those of classification and decision making. Much statistical literature has been written on how to build models that can sort new data into one of two or more disparate classes.

The problem of classification is one of still very active development, especially in light of advancements in computational power and the increasing availability of massive quantities of labeled data. Many algorithms have been developed for classification tasks, utilizing both linear and nonlinear techniques.

As a result of the availability of more powerful computers, it has become increasingly common to train not just one model, but many models that are then grouped into an *ensemble*. Ensemble models are usually built by training a collection of models, either in parallel or sequence, on *bootstrapped* resamples of the original training dataset, allowing for the resulting ensemble to consist of many different models.

The bootstrap procedure, however, is not without its limitations. While the procedure has allowed analysts to produce considerably better estimates under limited data, it can often fail in the face of labeled data where one or more classes are exceptionally rare. When there are few observations of the rare events in the observed data, the regular bootstrap resamples could result in even fewer rare events and hence lead to biased predictions.

In this paper, we examine an alternative technique known as the *fractional-random-weighted (FRW) bootstrap* for training ensembles of classification models. Specifically, we focus on a subset of classification models known as *decision trees* and their ensemble counterparts *bagged trees* and *random forests*.

We begin by reviewing classification problems and related methods, evaluating classification models, bootstrapping, ensemble methods, and issues with classification of imbalanced data. We then proceed to introduce the FRW bootstrap, and we illustrate how to modify ensemble methods to adopt the FRW bootstrap, and finally we conduct a study comparing the FRW methods to their traditional counterparts on several datasets.

2 Literature Review

2.1 Overview of Classification Problems

Classification models are distinguished from other models by both their end result - a single discrete class, or a vector of probabilities representing the likelihood of a datapoint being each of a set of classes - and the data that they're trained on.

Classification algorithms are a subset of machine learning algorithms referred to as "supervised learning algorithms". That is, classification is done on *unlabeled* data using models trained on *labeled* data. An example of this might be a medical analyst with data for patients showing symptoms of one of two diseases with similar presentations. The analyst may train a model on a dataset where the patients were confirmed to have one disease or the other, then use that model to predict which disease incoming patients are presenting with.

Many techniques exist to carry out classification and decision making on new data given a set of labeled training data, such as logistic regression [21], naïve Bayes classifiers [31], decision trees [23], support vector machines [9], and neural networks [30]. The focus of this paper is on decision tree models, but we begin by introducing linear models for classification for comparison and to introduce some of the problems associated with classification.

2.2 General Linear Models

Especially in the binary case, an immediately appealing possibility for solving the classification problem is to use a regression technique to predict the probability of a datapoint belonging to a given class or not. However, linear regression models are not suitable as the

classification data are categorical. General linear regression models are more appropriate for modeling categorical responses. Since the probabilities are bounded between 0 and 1 and the simple linear link function (a function linking the mean response with the predicting covariates) could be problematic when the true probability is close to 0 or 1.

The most common GLM used for classification is the logistic regression model. This model uses the logistic link function which links the log odds with the predictors. The predicted probability using the logistic link function is naturally bounded within the probability range between 0 and 1.

The logistic regression model takes the form:

$$\begin{aligned} Y_i &\sim Binomial(p_i) \\ log(\frac{p_i}{1-p_i}) &= \beta_0 + \beta_1 x_{i1} + ... + \beta_k x_{ik} \end{aligned} \tag{1}$$

Henceforth we can predict the probability of a data point beling to a certain class as its covariate vector $\boldsymbol{x}_i$ and the vector of parameters β. The odds predicted by this model then takes the form:

$$OR_i = (\frac{p_i}{1-p_i}) = e^{\boldsymbol{\beta}^T \mathbf{x}_i}, \tag{2}$$

Which can be transformed to yield the predicted probability

$$p(\boldsymbol{x}_i) = \frac{e^{\boldsymbol{\beta}^T \mathbf{x}_i}}{1 - e^{\boldsymbol{\beta}^T \mathbf{x}_i}} \tag{3}$$

The parameters of the model can be estimated using maximum likelihood estimation (MLE) by finding the parameters that maximize the log-likelihood function.

Suppose that we have a set of n training points $(y_i, \boldsymbol{x}_i)$. In the binary case, the probability that for an individual datapoint $y_i = 1$ is given by $p(\boldsymbol{x}_i)$, with $P(y_i = 0) = 1 - p(\boldsymbol{x}_i)$. The likelihood function is thus:

$$L(\beta_0, \beta) = \prod_{i=1}^{n} p(\boldsymbol{x}_i)^{y_i} \left[1 - p(\boldsymbol{x}_i)\right]^{1-y_i} \tag{4}$$

By utilizing the log-likelihood to break the likelihood apart, then substituting $p(\boldsymbol{x}_i)$ by Eqn. (3), we have

$$\begin{aligned} l(\beta_0, \beta) &= \sum_{i=1}^{n} y_i p(\boldsymbol{x}_i) + (1 - y_i) log\left[1 - p(\boldsymbol{x}_i)\right] \\ = \sum_{i=1}^{n} log\left[1 - p(\boldsymbol{x}_i)\right] + \sum_{i=1}^{n} y_i log \frac{p(\boldsymbol{x}_i)}{1 - p(\boldsymbol{x}_i)} &= \sum_{i=1}^{n} log\left[1 - p(\boldsymbol{x}_i)\right] + \sum_{i=1}^{n} y_i (\boldsymbol{\beta}^T \mathbf{x}_i) \\ &= \sum_{i=1}^{n} -log\left[1 + e^{\boldsymbol{\beta}^T \mathbf{x}_i}\right] + \sum_{i=1}^{n} y_i (\boldsymbol{\beta}^T \mathbf{x}_i) \end{aligned}$$

Then by taking the derivative with respect to the individual parameters and set the derivatives to equal zero, we arrive at:

$$\begin{aligned} \frac{\partial l}{\partial \beta_j} &= -\sum_{i=1}^{n} \frac{1}{1 + e^{\boldsymbol{\beta}^T \mathbf{x}_i}} e^{\boldsymbol{\beta}^T \mathbf{x}_i} x_{ij} + \sum_{i=1}^{n} y_i x_{ij} \\ &= \sum_{i=1}^{n} \left[y_i - p(\mathbf{x}_i; \boldsymbol{\beta})\right] x_{ij} \end{aligned} \tag{5}$$

The above equation normally has no closed form solution, but can we can use numerical methods, such as Newton's method and iterative scaling [22], to yield parameter estimates.

Logistic regression as a classification model has one final requirement: a decision boundary – a line that separates the space of observable data into two regions denoting the two classes. This decision boundary is effectively a choice as to what level of p is acceptable for determining that a data point belongs to one class over another. For some situations, it may be acceptable to shift this boundary away from parity in order to improve the accuracy of

prediction on one class (or minimize the costs associated with misclassifying into one of the two classes).

The logistic regression model can be expanded to response variables of more than two classes in the form of the *multi-class logistic regression model*. Under this model, instead of a single set of parameters $\boldsymbol{\beta}$, each class $c_i = k \in 1, \ldots, K$ will have its own parameter set $\boldsymbol{\beta}^{(k)}$. The predicted conditional probabilities are given by the expression:

$$Pr(Y = c_i | \boldsymbol{x}_i) = \frac{e^{\boldsymbol{\beta}^{(c_i)^T} \mathbf{x}_i}}{\sum_{c_i=1}^{K} e^{\boldsymbol{\beta}^{(c_i)^T} \mathbf{x}_i}} \tag{6}$$

Note that the above expression is exactly equivalent to the equation (3) in the binary case. The parameters of this model can similarly be estimated by numerical methods in conjunction with MLE.

2.3 Decision Trees

One of the most common, widely used, easily implemented, and easily interpreted alternatives to logistic regression is that of the decision tree.

Decision trees stand out among classification methods for being uniquely interpretable. Their representation of the structure of their internal knowledge is easy even for laymen to read: proceeding from the root node, each subsequent node represents a new decision layer, with the final nodes or "leaves" representing a final decision or classification.

Given a training sample of n observations on a class variable Y with k possible outcomes, and a set of p predictor variables $X_1, \ldots, X_p$, a classification tree produces a set of "rectangular" sets $A_1, \ldots, A_R$ that are partitions of the original dataset corresponding to some "split" on one or more of the p predictor variables.

Classification and regression tree methods yield rectangular sets by a series of recursive partitions, which can be represented as a tree structure. An initial split on one of the predictor variables is determined using some optimal splitting criterion, such as the Gini

criterion, the twoing criterion [19], and the entropy criterion [23] (which we will elaborate upon later). Within each of the two rectangular sets resulting from this split, a new split is found on one of the predictor variables by the same method. This is carried out recursively until some stopping criterion is met.

The initial split would be represented in a tree structure as the root node. Each subsequent split branches off from the split (node) before it. When no more splits could be done based on some chosen termination criterion, terminal nodes specifying one of the classes are created.

Each branch of the resulting decision tree is itself a decision tree (or *subtree*) on the remaining dataset. That is, the remaining splits can operate as an independent decision tree conditioned on the set of data at the root node of the subtree.

Splits can also be thought of as statements of conditional probability. Each side of a split contains a subset of the original dataset, and each subset has its own class proportions. The proportion of members of the class j on side A, $p_{A,j}$, represents the conditional probability of a datapoint sorted into side A being a member of class j. Terminal nodes work similarly: conditioned on all prior splits, data arriving at terminal node m has probability $p_{m,j}$ of being a member of class j, where $p_{m,j}$ is the proportion of data at that node of class j.

Given a subset A_i of the data after some set of splits $\boldsymbol{S} = (s_1, \ldots, s_k)$, the probability that a datapoint meeting the set of conditions S being a member of class C is given by:

$$P(a \in C|S) = \frac{n_{c,i}}{n_{A_i}}, \tag{7}$$

where the numerator denotes the number of members of class C in A_i and the denominator denotes the size of A_i. Iterating over all possible classes would yield the conditional probability distribution of classes for data meeting the condition set S.

Given a decision tree and a test datapoint, the datapoint can be classified by descending the decision tree. Beginning at the root node, the datapoint is directed along an edge of

the tree descending from the current node by whichever condition it meets. This process is repeated recursively until a terminal node is reached, giving a classification for the datapoint. [23]. Alternatively, the terminal node can itself represent a conditional probability: the class proportions within the subregion of the training data captured by that terminal node give probabilities for a new datapoint being a member of each class conditioned on the prior splitting conditions being met.

We give an example of splitting regions in Figure 2.1. For a given tree such as the following, we might have a set of splitting regions such as the following:

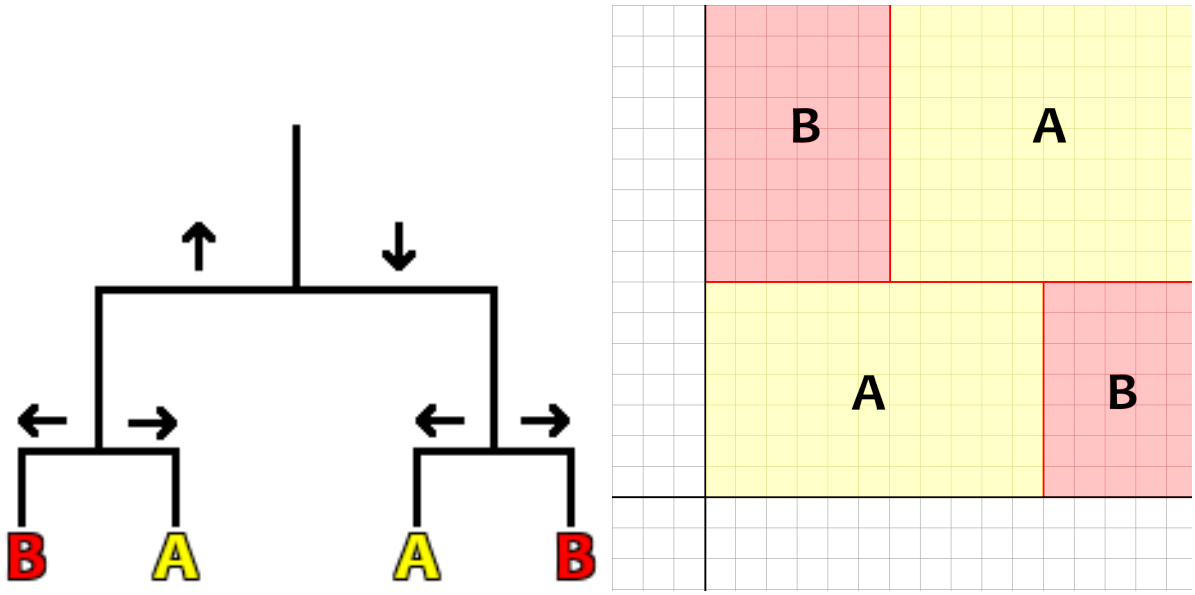

Figure 2.1: An example decision tree (left) with splitting regions (right) in Euclidean space

The key insight here is that each "split" on a decision tree corresponds to a "straight" decision threshold in Euclidean space. Where a logistic regression yields a single curve splitting the decision space into two sides, the decision tree model can split a decision space into many regions bounded by a condition on a single variable. The general algorithm for a decision tree works is summarized in the flow diagram shown in Figure 2.2:

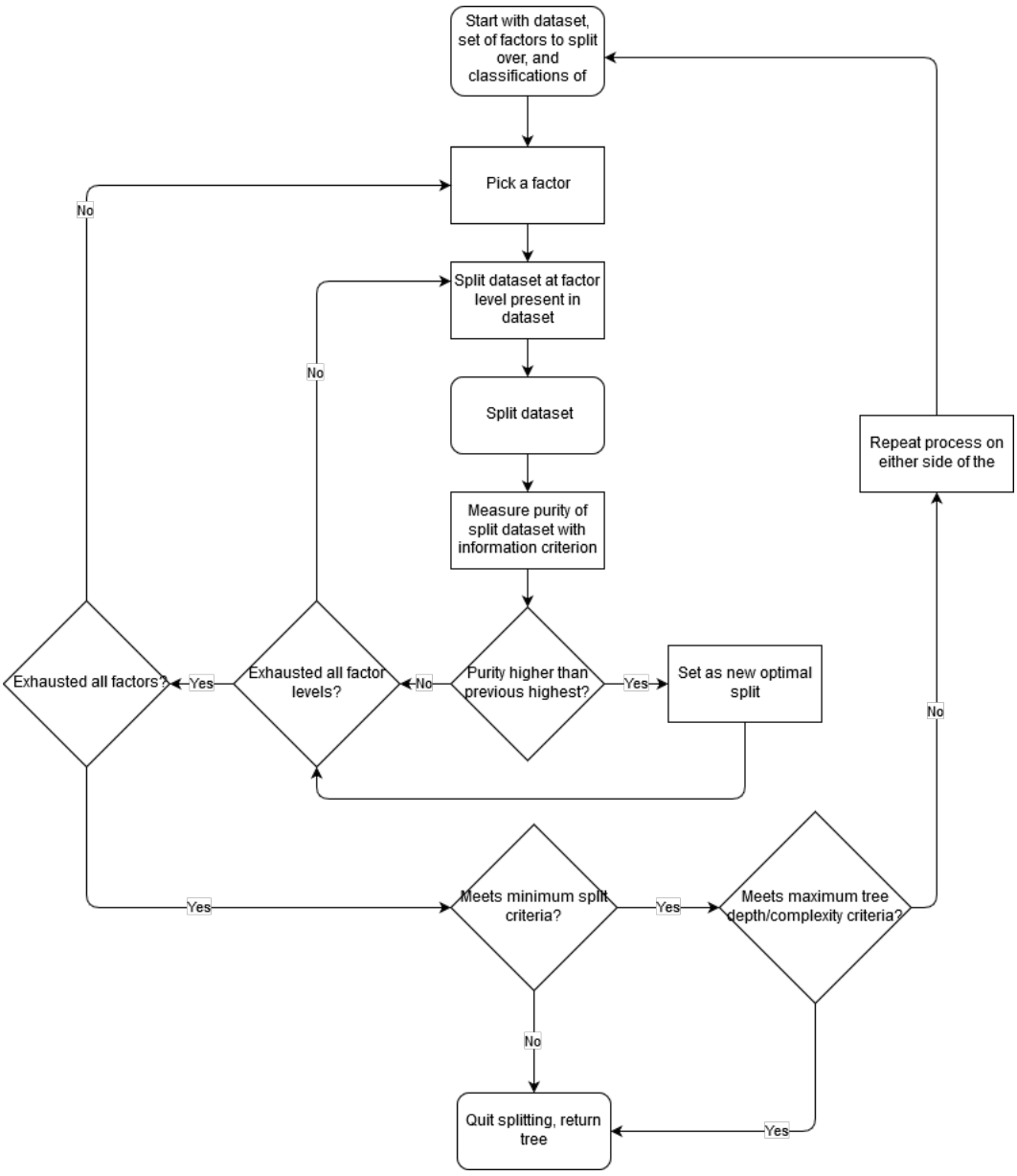

Figure 2.2: A flow diagram demonstrating the algorithmic steps for constructing a decision tree

2.3.1 Constructing a Decision Tree

Decision trees can be constructed in many ways and several different algorithms are commonly used for training a new decision tree on a set of training data. In general, a decision tree is trained on a training set $\boldsymbol{X}$ by successively splitting the training dataset into smaller and smaller chunks, each "split" chosen from among the space of all possible conditionals over

the set of all available parameters. Each potential split is evaluated by a *goodness-of-split* function $G(s, \boldsymbol{X})$, with the best scoring split used at that junction.

The general algorithm for decision tree construction is as follows [5]:

1. For each model parameter β_i, iterate over every potential split in the training set for that parameter X_{β_i}. Let s by the split that minimizes $G(s, \boldsymbol{X})$
    - For numerical variables, the set of splits is a set of real numbers $s_i : s_i <= x_i, x_i \in \boldsymbol{X_{\beta_i}}$. Here, each s_i is simply a numerical point that splits the dataset into a set that is greater than that point and a set that is less than or equal to that point.
    - For categorical variables, the set of potential splits is the powerset - or, set of all possible subsets of a set - of the set of classes, $P(X_{\beta_i})$. Each $s_i \in PX_{\beta_i}$ splits the dataset into those points that are members of the split's subset of classes and those which are not.
2. Create two training subsets, $\boldsymbol{X}_{left}$ and $\boldsymbol{X}_{right}$, such that one subset meets the criteria given by s and the one subset fails to meet the criteria given by s
3. For each training subset, repeat steps 1 and 2 until some stopping criterion is met, retaining each splitting criterion and its relative position to other splits.
4. Optionally, prune the resulting tree according to a given set of pruning criteria.
5. To each terminal node, assign a class

In the first step, we begin with a training set and a pre-chosen goodness-of-split function. Each level of each factor that is present in our dataset represents a possible split, separating the dataset into two sets, where one meets some condition on that level of that factor and the other does not meet the condition. For numeric variables, the condition is typically a "less than or equal to" type relation, whereas for categorical variables, the splitting condition would separate the data into one set that are in a given subset of classes and another set that are not.

We iterate through every possible split and test that splitting condition with our goodness-of-split function. The set of possible splitting conditions is given by our training dataset: for a numeric variable, every quantity present in the data is a potential splitting point, and for a categoric variable, every possible unordered set of categories present in the data represents a potential splitting point.

The split that minimizes the value of the function is the optimal split at this depth. In our next step, we create two training subsets based on the split. We then carry out the same procedure iteratively, until a stopping criterion is met. Finally, the resulting tree can be pruned – reduced to a tree of less depth - according to some given pruning criterion. During the pruning step, utilizing one of several different rules, splits are iteratively removed from the tree, replaced instead by terminal nodes combining the nodes on either side of the removed split.

This generalized algorithm suggests several places where specific decision tree algorithms can differ from one another given the same dataset: in the goodness-of-split function, the stopping criteria, and the pruning criteria. Many different decision tree algorithms have been proposed and are seen in practice today, including ID3, C4.5, and CHAID.

The ID3 (Iterative Dichotomiser 3) algorithm utilizes the information gain to create splits, growing a tree to its maximum size and then utilizing a pruning step. This was Quinlan's original 1986 algorithm and utilized only discrete attributes. [23]

The C4.5 algorithm works similarly, but introduced partitioning of continuous attribute values into discrete intervals. [24]

The CHAID algorithm, proposed in 1980 by Kass [18], builds classification trees using non-binary splitting. This algorithm uses Bonferroni adjusted p-values to determine the most significant split.

Breiman introduced a joint classification and regression tree model, *CART*, or Classification and Regression Trees. [5] CART trees utilize the Gini impurity rather than information gain to determine goodness of split.

The goodness-of-split function is the primary determinant of what splits appear in a decision tree. At each node of a decision tree, the test data is partitioned into two regions (or more, if the tree is not binary). The best partition is chosen by evaluating many potential partitions using a metric called an impurity function.

Suppose $p = (p_1, \ldots, p_k)$ are proportions of k classes present in a given set of data $\mathbf{X}$. $f(p)$ is an impurity function if it meets the following criteria: $f(p)$ is convex, has a maximum when all p_j are equal, and is a minimum when one of the $p_i = 1$. [5]

A given split s produces two sets of data, a "left" set containing a proportion of the data P_L and a "right" set containing a proportion of the data P_R. Given an impurity function and a set of splits s on a dataset $\mathbf{X}$, the goodness-of-split function can be defined as [28]:

$$G(s, X) = f(p)–P_L f(p_L) - P_R f(p_R) \tag{8}$$

The most common impurity functions (and, therefore, goodness of split functions) are information gain and the Gini impurity coefficient. In general, these and other splitting criteria arrive at decision trees with similar rules and similar rates of error [3]. However, the Gini coefficient prefers to put the largest class into one pure node and all other classes into the other node while splitting, whereas the entropy metric prefers to balance the sizes of the two children nodes. [6]

The Gini impurity measure can be interpreted as the training error rate of a splitting rule. Within a node i, each class j occurs with proportion p_{ij}. If an observation was classified by the splitting rule at this node as class j with probability p_{ij}, the error rate would be exactly the Gini impurity of that node:

$$\sum_{j'} p_{ij}(1 - p_{ij}) \tag{9}$$

That is, the impurity of any node is the set of products $p_c(1 - p_c)$ where p_c is the proportion of some class c within that node. Thus we can calculate the Gini impurity of either side of

a given split: G_L and G_R. The total Gini impurity of a given split is equal to the sum of the impurity of each node, weighted by the proportion of the original dataset present in that node. That is:

$$G_{split} = p_R G_R + (1 - p_R) G_L \tag{10}$$

Where p_R is the proportion of the original dataset assigned to the right side of the split. This expression can be interpreted, as well, as a summation over the intraclass variance over the training observations captured by the node. [15] (Elements of Statistical Learning II, page 310)

The information gain metric evaluates the decrease in entropy from a given split on a dataset. The entropy, or expected information, of a dataset is given by the following summation over classes:

$$H = \sum_i -p_i log_2 p_i \tag{11}$$

The information gain is defined in terms of H by subtracting the entropy of the left and right nodes from the entropy of the dataset pre-split, weighted by the proportion of the original dataset present in each side of the split:

$$\Delta H = H - p_L H_L - p_R H_R \tag{12}$$

When utilizing the information gain metric, the feature chosen to split over is the one that yields the highest information gain by the above equation.

Other metrics include the variance reduction metric, Chi-Square, and information gain ratio. The Gini impurity measure is the primary metric that we test in our simulations.

In order to grow a decision tree, there must be a stopping criterion. A stopping criterion prevents overfitting – without one, a decision tree could grow to provide a terminal node for every single training observation. Stopping criteria also prevent a tree from growing overly complex, which can be detrimental in cases where the researcher desires an interpretable

decision tree. Common stopping rules include:

- A maximum tree depth
- A minimum number of training observations on each side of a split
- A minimum (or maximum) value from the information metric required to create a split
- All instances in the remaining training set belong to a single class

Each of the above stopping rules can yield subtly different results. A maximum tree depth sets a hard limit on the number of splits that can exist, especially in a binary decision tree. For instance, a maximum depth of 3 in a binary decision tree would yield, at most, 7 splitting criteria in the tree. A minimum number of training observations or minimum information value can yield very imbalanced trees: for instance, an early split that captures most of the variation in one level of a class can yield one side with no further splits, but an opposite side with potentially many splits. Loose stopping rules can lead to large trees and overfitting, but exceedingly strict stopping rules can lead to small and underfit trees.

An additional method of controlling overfitting and complexity is pruning of the resulting decision tree. It is often prudent to grow a tree beyond optimal size and prune afterwards, allowing the tree to explore more of the sample space before being cut back. [26] There are numerous pruning methods for decision trees. For the sake of brevity, listed below are a few such methods with minimal detail, including (Mingers, 1989):

- Critical Value Pruning: (Mingers 1987)
  - A critical value of the goodness-of-split measure is specified. Values of the goodness-of-split measure are retained during tree creation, and those nodes which do not meet or exceed the critical value are pruned post-creation unless a node in the subtree meets or exceeds the critical value.
- Reduced Error Pruning: (Quinlan, 1987)

- For each node in the decision tree, remove the subtree and leave the node as a leaf that classifies as the most common class at that node. The change is kept if the performance on a validation set is no worse than the original tree. Which nodes are ultimately removed is decided in order of accuracy – the changes that result in the greatest improvement in accuracy are retained first, and pruning continues until further pruning is harmful to accuracy on the validation set.
- Requires a training, test, and validation set, so is most useful when large amounts of data are available.

- Pessimistic Error Pruning: (Quinlan, 1986)
  - For each node of the decision tree represents a split as well as a subtree in itself. The splitting rule of the node yields one misclassification rate, while the entire subtree yields a different misclassification rate:
    * Node misclassification with continuity correction = number of misclassifications + 1/2
    * Subtree misclassification with continuity correction = sum of the number of misclassifications at each leaf + the number of leaves/2
  - The subtree is only retained if its corrected error rate is more than one standard error better than the node error rate, where standard error is given by:
    * $SE(T) = \sqrt{\frac{n(T)(N(T)-n(T))}{N(T)}}$
    * Where n(T) is the number of misclassifications in the subtree or node and N(T) is the number of training set examples at that node (or in that subtree).

In general, when growing decision trees, the researcher must make decisions on what sort of metric they feel is relevant to their problem, what stopping criteria to use, and how to prune the resulting decision trees. In some cases, it may be good to have very strong stopping criteria and minimal if any pruning – e.g. to generate many decision trees in a short period

of time. In other cases, it may be preferable to have weak stopping criteria that produce large, complex trees, and use a more elaborate pruning scheme in order to arrive at more robust decision trees (at the cost of computational time).

2.4 Evaluating Classification Models

As with any statistical model, a primary concern when building a classifier is evaluating its performance. In general, we want to evaluate our models on out of sample data - usually, data that was held back during the training process specifically for this purpose. There is often little relationship between the error on the training set and the error on the testing set. Testing on out of sample data provides better feedback on how the model can be expected to perform "in the wild", and restrains the risk of overfitting to our training data.

In a machine learning context, carefully chosen metrics and well constructed test datasets can be especially important, as these metrics are also used to guide the selection of hyper-parameters for training a final model.

2.4.1 True Positives, False Positives, Recall and Precision

Suppose we have a classifier $\hat{f}(\mathbf{x}, T)$ trained on a training set T with a set of labels c_j. For any observation x_i that is a member of the test set $x_i, i = 1, \ldots, n$, the estimated response $\hat{f}(\mathbf{x}, T)$ has four potential outcomes relative to some given class c_j: true positive, false positive, true negative, false negative. Let us denote the true class of x_i as $\gamma(x_i)$. Then we have the following definitions

- True Positive (TP): $\hat{f}(x_i, T) = c_j$ AND $\gamma(x_i) = c_j$
- False Positive (FP): $\hat{f}(x_i, T) = c_j$ AND $\gamma(x_i) \neq c_j$
- True Negative (TN): $\hat{f}(x_i, T) \neq c_j$ AND $\gamma(x_i) \neq c_j$
- False Negative (FN): $\hat{f}(x_i, T) \neq c_j$ AND $\gamma(x_i) = c_j$

In the binary case, the number of true positives for one class is equal to the number of true negatives for the other class, and the number of false positives for one class is equal to the false negatives for the other.

These metrics are usually examined in terms of rates. For instance, the true positive rate (TPR) would be the proportion of true positives to total positives in the test set (true positives + false negatives). This is often referred to as the recall or sensitivity of a model (for that class). Another rate of interest is the precision, which is the proportion of true positives to total predicted positives (true positives + false positives).

A useful tool in analyzing the performance of a model is what's known as the confusion matrix. This presents the TP, FP, TN, and FN in an easily digestible format. For each class, the confusion matrix displays the number of members of that class that were predicted as each class. This can be used for any number of classes, and we demonstrate an example for three classes below.

Suppose a model has been trained to distinguish among three classes and we wish to evaluate its performance on a set of 300 testing observations, where each class is represented equally. A potential confusion matrix may look like the following:

	Actual A	Actual B	Actual C
Predicted A	70	80	30
Predicted B	20	20	20
Predicted C	10	0	50

Table 2.1: An example confusion matrix for a classification model on three classes.

Note that each column sums to the total number of observations of that class in the training set (in this case, 100). We can make a quick assessment of the classifier's performance. We might observe that the classifier most effectively predicts A and least effectively predicts B

and that the classifier most often predicts A, among other observations. Here, the recall for class A is 70/100. The precision for class A, however, is 70/180. The recall for class B is 20/100, with a precision of 20/60. Finally, the recall for class C is 50/100, with a precision of 50/60.

2.4.2 Receiver-Operator Curves, Precision-Recall Curves, and Area Under the Curve

An alternative method of quantifying the predictive power of a model is the Receiver-Operator Characteristic curve (ROC curve) and the area under the ROC curve (AUC). Given a binary classifier that outputs probabilities, the analyst must decide at what probability cutoff to sort a datapoint into one or the other class – it may be that the analyst obtains better results by sorting a datapoint into the 1-class when the trained model predicts a probability of, for instance, 0.6 of the datapoint being the 1-class.

The ROC curve plots the true-positive rate for a class on the y-axis versus the false-positive rate on the x-axis: in general, the more false-positives we're willing to accept, the more true-positive's we'll capture. The greater the area under this curve, the lower of a false-positive rate we need accept in order to capture more true-positives. Changing the false-positive rate is equivalent to simply adjusting the probability boundary for sorting into one class or the other. [7]

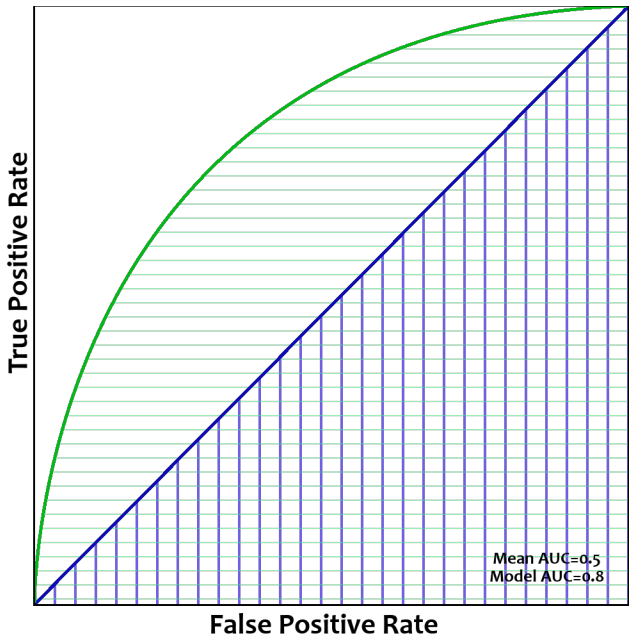

Figure 2.3: Example of an ROC curve: Blue highlight is the area under a "mean" curve with an AUC of 0.5, green highlight is the area under an ROC curve for a fitted model having an AUC of 0.8

A related measure is the Precision-Recall Curve and it's AUC. More commonly used in imbalanced dataset contexts [10], the PRAUC quantifies the tradeoff between the precision of a model and it's recall. In general, the greater precision of a model, the lower the recall, and vice versa – however, for a well fitted model, the tradeoff is more generous, allowing for both greater recall and precision without having to trade off one for the other. This is captured by the AUC – the greater the AUC, the higher the joint precision and recall.

Measures based on the ROC and PR curves are most useful in contexts where optimizing for recall and precision alone may yield poor results, which can often occur in contexts of dataset imbalance.

2.5 Bootstrapping

Often, a statistician or data scientist is faced with limited data or wants to make more robust inferences on the population. More generally, the analyst is often faced with situations where the ideal would be to take multiple independent samples from the population.

A key example of this issue is in assessing the statistical accuracy of an estimate. Because the population is unknown, the error between the sample statistics and the true population parameter the statistic is measuring cannot itself be directly measured. In this case, it would be ideal to take multiple samples and measure the variance in the sample statistic among multiple samples. This is often not possible, however.

The non-parametric bootstrap simulates the process of taking multiple samples from the population by repeatedly sampling m observations, with replacement, from a dataset of size n, where $m \leq n$. We provide a formal definition below [12]:

Definition 2.1. The Bootstrap Let $\boldsymbol{X} = (X_1, \ldots, X_n), X_i \sim F$ represent a random sample of size n drawn from an unspecified distribution F, with $\boldsymbol{x} = (x_1, \ldots, x_n)$ representing an observed realization of the random sample. Let $\hat{F}$ represent a sample probability distribution, where a mass of $\frac{1}{n}$ is located at each point $x_i \in \boldsymbol{x}$.

With $\hat{F}$ fixed, a **bootstrap sample** is collected by drawing a random sample of size n from $\hat{F}$:

$$\boldsymbol{X^*} = (X_1^*, \ldots, X_n^*), \text{ and } X_i^* \sim \hat{F} \tag{13}$$

Let $\boldsymbol{x^*} = (x_1^*, \ldots, x_n^*)$ represent an observed bootstrap resample. The distribution of a sample statistic $S(\boldsymbol{X}, F)$ can be approximated by the **bootstrap distribution** of the statistic, $S^*(\boldsymbol{X^*}, \hat{F})$.

On each set of resampled data, the test statistic of interest is calculated or the model of interest is retrained. The variance in the test statistic or the performance of the model can then be measured using the set of results from this procedure.

Suppose we have a set of data $\boldsymbol{X} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)$ and let $S(\boldsymbol{X})$ represent a sample statistic measured on that data. Under the bootstrapping procedure, we draw k resamples from the dataset and generate a sample statistic $S(\boldsymbol{X}_i), i = 1, ..., k$ for each resample, yielding a set of sample statistics $\boldsymbol{S} = S(\boldsymbol{X}_1), ..., S(\boldsymbol{X}_k)$. Using $\boldsymbol{S}$, we can generate statistics on the distribution of $S(\boldsymbol{X})$, such as the variance:

$$\bar{\boldsymbol{S}} = \frac{1}{k}\sum_{i=1}^{k} S(\boldsymbol{X}_i)$$

$$\hat{Var}[S(\boldsymbol{X})] = \frac{1}{k-1}\sum_{i=1}^{k}(S(\boldsymbol{X}_i) - \bar{\boldsymbol{S}})^2 \tag{14}$$

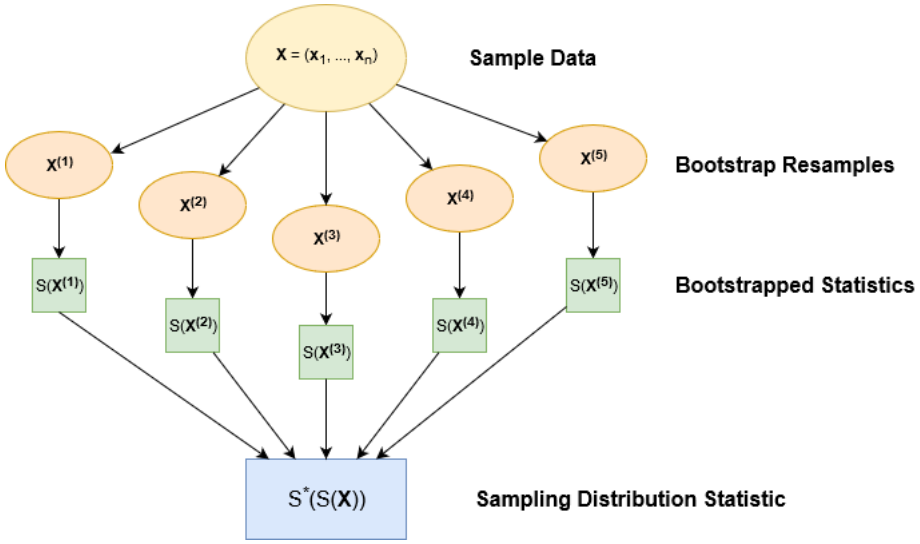

Figure 2.4: Diagram of generating sampling distribution statistics using bootstrapped statistics on resamples from the original dataset

By similar means, any other measure of interest on the sampling distribution can be derived using the bootstrap.

2.6 Bootstrap Aggregating (Bagging)

A common usage of the bootstrap procedure is to improve an estimate or prediction. Given a set T of training data $(y_n, x_n), n = 1, \ldots, N$ where each y_i is a response, either numerical or a label, suppose we have a procedure to form a predictor $f(x, T)$ that yields a predicted response $\hat{y}$

With a sequence of training sets T_k, each drawn from the same underlying distribution, it is possible to construct a better predictor than the single predictor from our single training

set. If our response is numerical, the obvious method would be simply to replace $f(x, T)$ with the average response $f(x, T_k)$ over k.

If our response is instead a class label, one method (though not the only method, as this would be dependent on the underlying predictor constructed) of aggregating over the $f(x, T_k)$ is to carry out a voting procedure, where all k labels $\hat{y}_k = f(x, T_k)$ are collected and the most common label is treated as the aggregate response.

In most cases, the analyst will not have many sets of training data available. In such cases, the bootstrapping procedure conducted n times over a single training set T yields our set of training sets T_k.

Consider, for instance, a regression model. For each of the n bootstrap samples T_k, we fit the regression model anew, yielding an estimator f_k. For a given observation x, the bagged estimate of the response is given by averaging over the k estimated responses:

$$(f_{bag})(x) = \frac{1}{n} \sum_{k=1}^{n} f_k(x) \tag{15}$$

In (Breiman, 1994), Breiman states: *"if perturbing the learning set can cause significant changes in the predictor constructed, then bagging can improve accuracy"*. In general if a predictive procedure is unstable, bootstrap aggregation (bagging) can lead to considerable improvement in accuracy.

2.7 Ensemble Decision Tree Methods

In the machine learning literature, ensemble models have a long history. A variety of ensemble methods exist for usage across many different model types. Among the earliest theoretical examinations of ensemble methods were Kearn's 1988 paper and Schapire's 1990 paper, wherein the notion of weak and strong learners were introduced.

A weak learner is an algorithm that produces classifiers that are, at minimum, consistently better than random guessing.

The error rate for a classifier produced by this algorithm is given by [20]:

$$e <= \frac{1}{2} - \gamma, \gamma >= 0 \tag{16}$$

By contrast, a strong learner is one whose error rate can be arbitrarily close to zero given an arbitrarily high sample size. A core question in early machine learning papers was whether there exists an algorithm that, given a weak learner, can produce an arbitrarily strong learning. Schapire (1990) [27] answered this in the affirmative, showing that weak learnability and strong learnability are equivalent. The hypothesis boosting algorithm was introduced therein, which was the basis of boosting algorithms in ML models that have been introduced since.

In this paper, our focus is on the *bagged trees* and *random forest* ensemble methods for decision trees, which we introduce below.

2.7.1 Bagging Decision Trees

The simplest form of ensemble method for decision trees is the bagged model. As previously discussed, bagging allows us to use bootstrapping to generate a series of training sets, each of which can be used to generate a new predictor. Our aggregate predictor simply averages the predictions of the individual models in some way to yield a prediction.

The decision tree model is an unstable model (Breiman, 1994). That is, slight perturbations to the training set T can yield large differences in the resulting model f(x, T). As a result, it can be expected that trees trained under resamples of the original training set will be noticeably different, and that the aggregate method will be closer to optimality as a result than a single decision tree will be on its own. We're concerned with two kinds of output: class probabilities and classifications. There exist multiple possible ways for aggregating over our models. For calculating an aggregate probability, we could:

- Substitute the proportion of votes for a class in the ensemble as the probability of that class, or

- Average over the vector of class probabilities predicted by each tree in the ensemble

For predicting a label, we could:

- Label the data according to which had the greatest number of votes
- Label the data according to which had the highest average probability
- Label the data according to a threshold probability

Breiman (1996) utilized the C4.5 decision tree algorithm (utilizing normalized information gain as a splitting criterion) to demonstrate the efficacy of bagging decision trees on a set of seven datasets, producing classifiers with an error rate ranging from 0.57-0.94 of the error of a single classifier trained on the original dataset.

2.7.2 Random Forests

The random forest algorithm is an algorithm that produces an ensemble of learners trained in parallel, similar to the bagging algorithm. However, in addition to selecting random subsets of the data, the random forest algorithm also selects random subsets of the predicting variables. This reduces correlation among the trees in the ensemble.
Correlation among decision trees is defined in terms of the *raw margin function*. The *margin function* is given by:

$$mr(x, y) = P_{\Theta}(h(x, \Theta) = y) - \max_{j=/=y} P_{\Theta}(h(x, \Theta) = j) \tag{17}$$

Where x is an input datapoint, y is the true class of that datapoint, Θ_k is a family of iid random vectors, and $h(x, \Theta_k), k = 1, \ldots, L$ is an ensemble of classifiers. Thus, the margin function is the difference between the probability that the ensemble assigns to the true class and the probability that the ensemble assigns to the class it deems most probable beside the true class. [4]
The raw margin function is then given as

$$rm(\Theta, x, y) = I(h(x, \Theta) = y) - I(h(x, \Theta) = j(x, y)) \tag{18}$$

Here, $I(\cdot)$ is the indicator function and $j(x, y)$ stands for the most probable class (as deemed by the ensemble) beside the true class. The raw margin function represents the difference in votes within the ensemble between the true class and the most probable class beside the true class.

The correlation among decision trees is simply the correlation of $rm(\Theta, x, y)$ and $rm(\Theta', x, y)$ among all pairs of (Θ, Θ'). This is what we seek to minimize when using random forest techniques.

2.8 Classification on Imbalanced Datasets

The models discussed thus far are all useful in the classification context to varying degrees. One common obstacle faced by analysts in training classification models, however, is that of the imbalanced dataset.

Most classification problems encountered in the wild are not perfectly balanced. When the data in question is especially imbalanced, the analyst is usually most interested in the low probability class or classes, and often the greatest cost of misclassification is with the low probability class(es) as well.

Imbalanced data can present issues for evaluating the performance of a model. [2] Many evaluation metrics become unreliable as the datasets in question become more imbalanced. As discussed previously, the recall and precision alone are often not enough to tell the whole story - ROC and PR curves are especially useful in imbalanced datasets.

Training algorithms themselves often respond poorly to severely imbalanced data. [8] As a result of their reliance on the nonparametric bootstrap, ensemble machine learning methods in particular face issues when training on imbalanced data. As the bootstrap procedure samples from the original training set, if the proportion of a class is extremely low, there's

a high probability of some resamples that contain very few or no instances of that class. Models trained on these resamples will never predict the low probability class.

Several data level methods exist for handling imbalanced training datasets to improve classification on the low probability class. [16] Among these techniques, the most common are *oversampling* and *undersampling*. [29]

In undersampling, a training dataset with a more balanced class distribution is constructed by sampling (or resampling) from the original training dataset as ordinary, then randomly remove members of the majority class.

In oversampling, a training dataset with a more balanced class distribution is constructed by sampling from the original training dataset as ordinary, and randomly resampling additional datapoints from the subset of data representing the minority class. Oversampling can also be carried out by fabricating datapoints using the empirical distributions of the minority class data subset's parameters.

Alternatively, samples of pre-specified sizes can be randomly drawn from each class' subset of the original training dataset, allowing the analyst to build a training set with an exactly specified class distribution. In addition, more advanced modifications of under- and oversampling exist whereby the removed (or added) datapoints are chosen according to some analytical criteria.

3 Chapter 3: Fractional Random Weighting for Classification on Imbalanced Data

3.1 Fractional Random Weighted Bootstrapping

In this paper, we examine the utility of the fractional randomly weighted (FRW) bootstrap as an alternative to the non-parametric bootstrap for training ensemble decision tree models. The FRW bootstrap offers an alternative resampling method that never fails to capture every single class in a multi-class dataset, regardless of the underlying probability distribution of the classes. [25]

Under the fractional random weighted bootstrap, every observation in the training dataset is paired with a fractional weight drawn from a probability distribution, such that the sum of the weights is equal to the number of discrete observations. For each "resample", a new set of weights is drawn and assigned to the training data. Statistics can then be computed, and models trained, on the dataset with modification for weighting the observations.

The most examined form of the FRW bootstrap utilizes the uniform Dirichlet distribution to generate weights. However, the FRW bootstrap can utilize any continuous distribution to draw weights that has a mean and standard deviation of 1, as long as the weights drawn are positive and independently, identically distributed.

The probability density function of the Dirichlet distribution of order n (where n is the sample size of the dataset we're bootstrapping) is given by:

$$f(w_1, \ldots, w_n; \alpha_1, \ldots, \alpha_n) = \frac{1}{B(\alpha_1, \ldots, \alpha_n)} \prod_{i=1}^{n} w_i^{\alpha_i - 1} \tag{19}$$

Where $B(\alpha_1, \ldots, \alpha_n)$ is a normalization factor equal to the multivariate beta function, the vector $(w_1, \ldots, w_n)$ is our weight vector and $\sum_i w_i = 1$, and $(\alpha_1, \ldots, \alpha_n)$ is a vector of positive real parameters. [14] When using the FRW bootstrap, the alpha parameter is always set to 1.

The FRW bootstrap, like the non-parametric bootstrap, is provably consistent. [13] That is:

FRW ML Estimation 1. *Let $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$ be a set of random iid observations. Given a set of unknown parameters $\boldsymbol{\theta}$, the FRW maximum likelihood estimator $\hat{\boldsymbol{\theta}}^* \to \boldsymbol{\theta}$ if and only if the maximum likelihood estimator $\hat{\boldsymbol{\theta}} \to \boldsymbol{\theta}$ as $n \to \infty$.*

Proof. Let $\boldsymbol{X} = \boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$ be a set of n random iid observations, and let $\boldsymbol{\theta}$ be a vector of unknown parameters. The log-likelihood function is written:

$$\bar{l}(\boldsymbol{\theta}) = \frac{1}{n} \sum_{i=1}^{n} l_i(\boldsymbol{\theta}; \boldsymbol{x}_i) \tag{20}$$

where $l_i(\boldsymbol{\theta}; \boldsymbol{x}_i)$ is the contribution to the likelihood made by observation i. The maximum likelihood estimate $\hat{\theta}$ is the solution to the first derivative:

$$\bar{l}'(\boldsymbol{\theta}) = \frac{\partial \bar{l}(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} = 0 \tag{21}$$

Let $W = w_1, \ldots, w_n$ be a set of iid weights drawn from the exponential distribution with mean 1. The fractional random weighted log-likelihood is given by

$$\bar{l}^*(\boldsymbol{\theta}) = \frac{1}{n} \sum_{i=1}^{n} w_i l_i(\boldsymbol{\theta}; \boldsymbol{x}_i) \tag{22}$$

and the MLE is obtained by similar means as above. Let $h(\boldsymbol{\theta})$ be the expectation of the log-likelihood function and note that $\boldsymbol{E}[\hat{l}(\boldsymbol{\theta})] = h(\boldsymbol{\theta})$. Let us assume that the MLE $\hat{\boldsymbol{\theta}}$ is

consistent, and thus $Var[\bar{l}(\boldsymbol{\theta})] \to 0$ as $n \to \text{inf}$. We show consistency of the FRW MLE by demonstrating that the expectation of the FRW log-likelihood expression is equivalent to the expectation of the standard log-likelihood, and that the variance of the same also converges to 0. The expectation of the FRW log-likelihood is given as:

$$\begin{aligned}
\boldsymbol{E}[\hat{l} * (\boldsymbol{\theta})] &= \frac{1}{n}\sum_{i=1}^{n} \boldsymbol{E}[w_i l_i(\boldsymbol{\theta}; \boldsymbol{x}_i)] \\
&= \frac{1}{n}\sum_{i=1}^{n} \boldsymbol{E}_{\boldsymbol{x}_i}[\boldsymbol{E}_{w_i}[w_i l_i(\boldsymbol{\theta}; \boldsymbol{x}_i)]|\boldsymbol{x}_i] \\
&= \frac{1}{n}\sum_{i=1}^{n} \boldsymbol{E}[l_i(\boldsymbol{\theta}; \boldsymbol{x}_i)] \\
&= h(\boldsymbol{\theta})
\end{aligned} \tag{23}$$

We note here that the third expression is equivalent to the first due to separability and the fact that under the FRW model, $\sum_{i=1}^{n} w_i = 1$. The variance of the FRW log-likelihood is given by:

$$\begin{aligned}
Var[\bar{l} * (\boldsymbol{\theta})] &= Var[\boldsymbol{E}[\bar{l} * (\boldsymbol{\theta})|\boldsymbol{X}]] + \boldsymbol{E}[Var[\bar{l} * (\boldsymbol{\theta})|\boldsymbol{X}]] \\
&= \frac{1}{n}[2Var[l_i(\boldsymbol{\theta}, \boldsymbol{x}_i)] + h(\boldsymbol{\theta})^2] \to 0 \text{ as } n \to \text{inf}
\end{aligned} \tag{24}$$

□

It has been shown that FRW bootstrap estimators (such as bootstrapped confidence intervals) have good properties under other distributions as well. Jin (2001) provides examples using the Gamma(1, 1) and Beta(sqrt(2) – 1, 1) distributions. [17] In general, as long as the weights can be represented as iid random variables from a continuous distribution with a mean and standard deviation of 1, such that the sum of the weights is a random variable with expectation n, the FRW bootstrap is well behaved.

The FRW bootstrap has proven utility for statistical estimators on censored and heavily imbalanced datasets. We hypothesized that ensemble decision tree methods utilizing the

FRW bootstrap would produce more accurate decision tree models on datasets with few observations of one or more of the classes.

3.2 FRW Bootstrapping for Ensemble Methods

Bagging, random forests and other common ensemble machine learning methods utilize the non-parametric bootstrap to resample from the dataset of interest in order to train additional models to add to the ensemble. However, there exist many contexts in which the non-parametric bootstrap cannot improve accuracy, or worse, can reduce it. In particular, when working with censored datasets, datasets with very few observations of one or more types of event in the data, or small datasets with many classes or one or more low probability classes, the non-parametric bootstrap can fail to adequately represent the training data when training ensemble models. For such datasets, the non-parametric bootstrap may fail to sample any of the low probability class(es), or may only sample very few of the low probability class(es), resulting in one or more classes being neglected entirely from one model in the ensemble.

While a dataset of interest would never 0 observations of a class of interest, a large dataset of size n with a small number of observations m of a class may have a surprisingly large chance of resampling with no observations of the low probability class. In r resamples with replacement, the number of resamples we'd expect to contain no observations of class A is given by $(\frac{n-m}{n})^n r$.

For instance, in a dataset of 5000 observations, where only 5 are of class A and the rest are class B, should we take 1,000 resamples, the number of resamples we'd expect to contain no observations of class A is: $(\frac{5000-5}{5000})^{5000}1000 = 6.721$. Across 10,000 resamples, that number increases to 67.21.

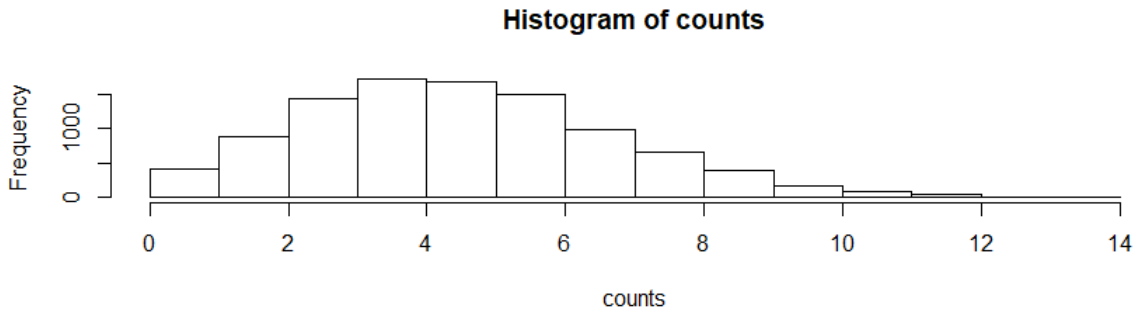

Figure 3.1: Distribution of number of observations of a low probability class (5 observations in an n=5000 sample)

While many ensemble methods, provided some tuning would be able to deal with this level of imbalance, in some instances we may have as few as 1, 2, or 3 training observations altogether, even in a massive dataset.

Should a resample in an ensemble be drawn containing none of the low probability class, the resulting model would be of negligible or negative value in the ensemble for correctly predicting the low probability class. Furthermore, due to the constraints of many common classification models, if only one or two observations are drawn, a model may not be trainable or may be equally as bad as if there were no observations of the low probability class.

In order to utilize the FRW bootstrap in ensemble models, the metric for evaluating splits on the data has to be modified to account for weights. This is relatively simple for the Gini splitting criterion.

Definition 3.1. Weighted Gini Criterion Let $\boldsymbol{T} = (\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_n, y_n)$ be a set of training data, where y_k is a class label. Let w_k represent the weight assigned to $\boldsymbol{x}_k \in \boldsymbol{X}$. Then the weight of all members of a single class or set of classes j in a set of data can be denoted $W_j = \sum_{k=1}^{n} w_k I_j(k)$, where $I_j(k)$ is an indicator function for the class or set of classes j. The weight of all members of a single class within a single node i may be denoted W_{ij}.

At node i, the proportion of observations in a class j is thus $p_{ij} = \frac{W_{ij}}{W_i}$, where W_i denotes the sum of the weights at node i. Reusing our previous expression for node impurity under

the Gini splitting criterion, the impurity at node i is now expressed:

$$G_i = \sum_j p_{ij}(1 - p_{ij}) = \sum_j [\frac{W_{ij}}{W_i}(1 - \frac{W_{ij}}{W_i})]$$
$$= \sum_j [\frac{\sum_{k \in \boldsymbol{n}(i)} w_k I_j(k)}{\sum_{k \in \boldsymbol{n}(i)} w_k}(1 - \frac{\sum_{k \in \boldsymbol{n}(i)} w_k I_j(k)}{\sum_{k \in \boldsymbol{n}(i)} w_k})]$$

Furthermore, the Gini split impurity under fractional random weighting is now:

$$G_{split} = p_L G_L + (1 - p_L) G_R = \frac{\sum_{k \in L} w_k}{n} G_L + \frac{\sum_{k \in R} w_k}{n} G_R$$
$$= \frac{\sum_{k \in L} w_k}{n} \sum_j [\frac{W_{Lj}}{W_L}(1 - \frac{W_{Lj}}{W_L})] + \frac{\sum_{k \in R} w_k}{n} \sum_j [\frac{W_{Rj}}{W_R}(1 - \frac{W_{Rj}}{W_R})]$$

Similar modifications can be made to other splitting metrics. For instance, the information gain criterion can be rewritten using the same notation as above:

Definition 3.2. Weighted Information Gain Let $\boldsymbol{T} = (\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_n, y_n)$ be a set of training data, where y_k is a class label. Let w_k represent the weight assigned to $\boldsymbol{x}_k \in \boldsymbol{X}$.
The weighted entropy of a node i is given by:

$$H_i = -\sum_j p_{ij} log_2 p_{ij} = -\sum_j [\frac{W_{ij}}{W_i} log_2 (\frac{W_{ij}}{W_i})]$$

The information gain from splitting the data at node O into two nodes L and R is then:

$$\Delta H = H_O - p_L H_L - p_R H_R$$
$$= \sum_j [\frac{W_{Oj}}{W_O} log_2 (\frac{W_{Oj}}{W_O})]$$
$$- \frac{\sum_{k \in L} w_k}{n} \sum_j [\frac{W_{Lj}}{W_L} log_2 (\frac{W_{Lj}}{W_L})]$$
$$- \frac{\sum_{k \in R} w_k}{n} \sum_j [\frac{W_{Rj}}{W_R} log_2 (\frac{W_{Rj}}{W_R})]$$

All other aspects of the decision tree algorithm should work similarly as before. These

modifications are not unique to fractional random weighting, as they are also necessary to utilize case weighting (e.g. for denoting classes whose misclassification is more costly). In practice, most standard decision tree implementations utilize case weighting, to which a vector of fractional weights can be passed.

4 Chapter 4: Methodology for Algorithm Study

4.1 Datasets

Four datasets were selected to build a comparison of standard ensemble models against models built using FRW bootstrapping: the Heart [11] dataset (https://archive.ics.uci.edu/ml/datasets/Heart+Disease), Iris [11] dataset (https://archive.ics.uci.edu/ml/datasets/Iris), Glass [11] dataset (https://archive.ics.uci.edu/ml/datasets/glass+identification), and Credit Card Fraud [1] dataset (https://www.kaggle.com/mlg-ulb/creditcardfraud). The bulk of our analysis focused on the Heart and Credit Card Fraud datasets, which are presented below - the Iris and Glass results can be found in Appendix 1.

The Heart dataset is a multiclass dataset, but can be treated as binary, with positive observations having heart disease and negative observations not having heart disease. The Heart dataset is not on its own severely imbalanced. To conduct simulations on imbalanced data, we randomly select 1, 2, 3, 4, 5, and 10 training observations of the positive class and coupled them with 100 observations of the negative class. This dataset had 303 observations total, 13 predictor variables and 1 label.

A severely imbalanced credit card fraud dataset was selected from Kaggle as an example of naturally imbalanced data. This dataset has 285,299 observations, 492 of which are fraud, with 30 predictor columns and a single column labeling each row as fraud or not fraud. We trained models on this dataset at varying sample sizes. Reducing the sample size reduced the number of training observations of the low probability class while retaining the same class

proportions, allowing us to compare performance of the models when only the number (and not the proportion) of training observations of the low probability class changed. Models were trained at sample sizes of 1500, 2000, and 3000.

4.2 Grid Searching

In order to control for the effect of hyperparameters, a grid search was conducted over sets of hyperparameters. In each simulation, models were trained with the same hyperparameter tunings for comparison against each other.

Furthermore, repeated simulations were conducted on a smaller subset of hyperparameter tunings and averaged to estimate an average effect from the FRW model.

5 Chapter 5: Results and Analysis

5.1 Heart Data

The Heart dataset is not a naturally imbalanced dataset, but it represented a good opportunity to test at various levels of imbalance. The Heart dataset has 302 observations near evenly split between patients having heart disease (139 observations) and patients without heart disease (163 observations), and 13 predictor variables.

We constructed imbalanced datasets by randomly selecting 1, 2, 3, 4, 5, and 10 observations of the positive class and 100 observations of the negative class.

At all but the 5 and 10 observation levels, performance was minimal from either the standard or the FRW models, achieving virtually no recall on the low probability class. At 5 and 10 observations, we saw noticeable improvement from the FRW model at the naive threshold.

A much more significant difference in performance is observable under the AUC metrics. The FRW bagging model saw consistently higher ROC-AUC and PR-AUC than the standard model when broken down by the number of low probability training observations and the number of trees in the model. The improvement in performance varied from modest to relatively great, with the greatest improvement seen at the lowest numbers of trees and the lowest numbers of training observation.

This pattern could be observed under both the bagging and the random forest models, with apparently diminishing effect of the FRW procedure as the number of training observations and the number of trees in the ensemble increased.

5.1.1 Performance at the Naive Threshold

Bagging Performance

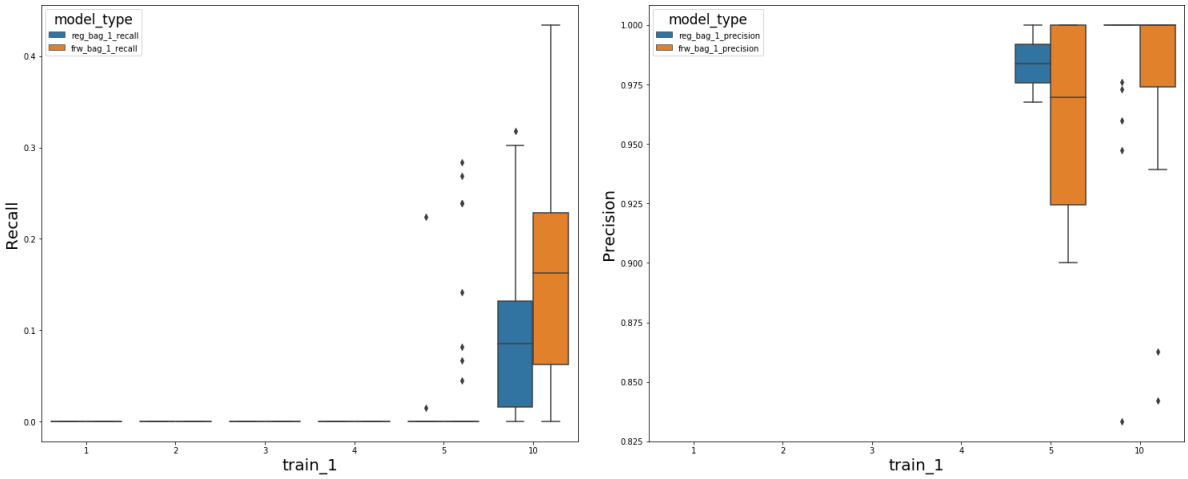

Figure 5.1: Recall (left) and Precision (right) for bagging models on the Heart dataset

Random Forest Performance

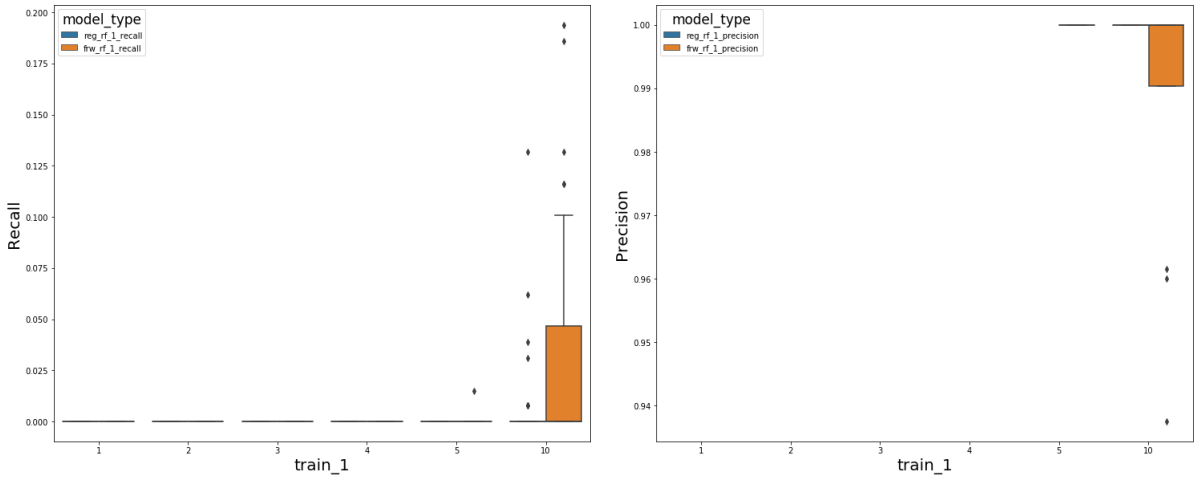

Figure 5.2: Recall (left) and Precision (right) for random forest models on the Heart dataset

5.1.2 AUC Metrics

Bagging performance

Tree Count	LP Count	1	2	3	4	5	6+
20	**ROCAUC Diff**	0.0149	0.0324	-0.0003	-0.0071	-0.0401	
20	**PRAUC Diff**	0.0743	0.1315	-0.0169	-0.0156	-0.0393	
50	**ROCAUC Diff**	0.0271	0.0257	0.0168	0.0091	-0.0102	-0.0180
50	**PRAUC Diff**	0.1621	-0.0832	0.0143	-0.0041	0.0059	-0.0079
100	**ROCAUC Diff**	0.01140	0.0348	0.0239	0.0392	-0.0020	0.0062
10	**PRAUC Diff**	-0.0257	0.1421	0.0225	0.0142	-0.0052	0.0019

Table 5.1: Mean difference in ROC-AUC and PR-AUC between standard and FRW bootstrapping models at each level of low probability training observations under the bagging model. There was an overall mean increase in ROC-AUC of 0.0062, with variability 0.0067. There was an overall mean increase in PR-AUC of 0.0222, with variability 0.0880.

Breaking down the ROC-AUC and PR-AUC differences across different numbers of low probability training observations demonstrates a trend towards higher performance by the FRW procedure at lower counts. As the number of low probability training observations increases, the FRW and standard procedures tend to converge towards similar levels of performance.

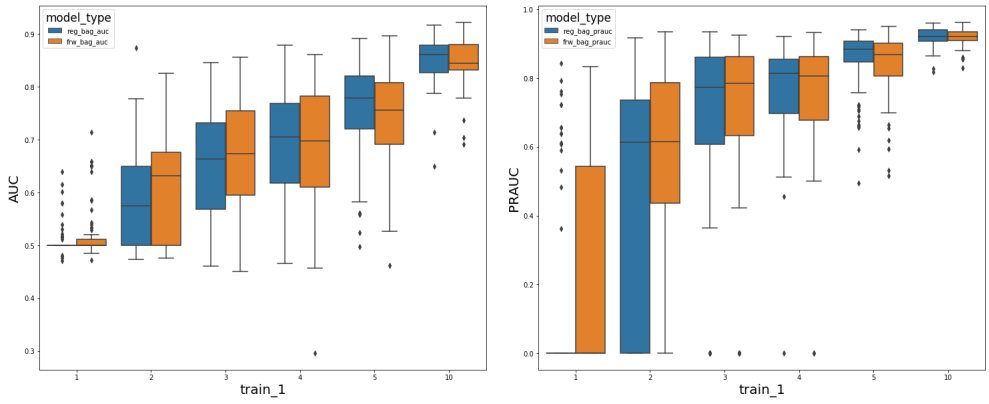

Figure 5.3: AUC (left) and PRAUC (right) for bagging models on the Heart dataset, broken down by low probability training observation count

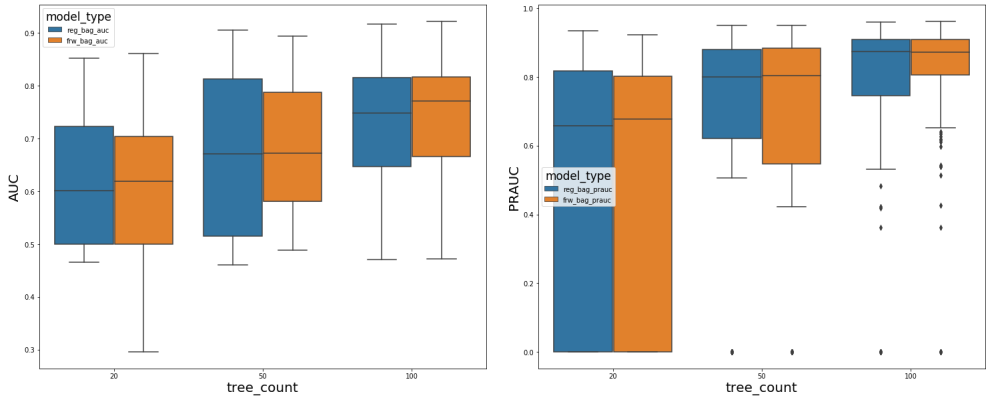

Figure 5.4: AUC (left) and PRAUC (right) for bagging models on the Heart dataset, broken down by tree count

Visually breaking down the performance by number of low probability training observations makes the trend more obvious. In addition to having a higher mean performance, the FRW model tended to have a slightly wider range of performance, with the best performing FRW models tending to perform better than the best performing models under the standard procedure.

Breaking down the performances by tree count reveals a more modest trend towards higher mean performances by the FRW model, but no clear trend in performance variability.

We can examine these trends closer by separating out performance by both tree count and LP training observation count.

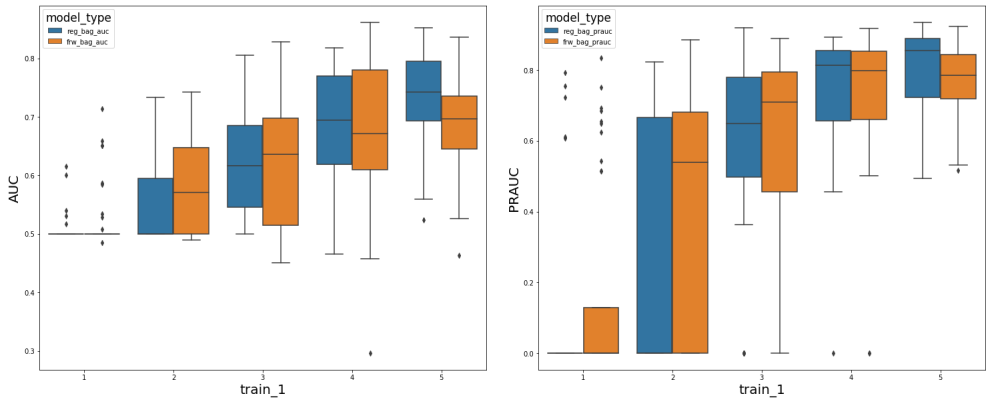

Figure 5.5: Plots of average spread of ROC-AUC (left) and PR-AUC (right) for bagging models with 20 trees on the Heart dataset

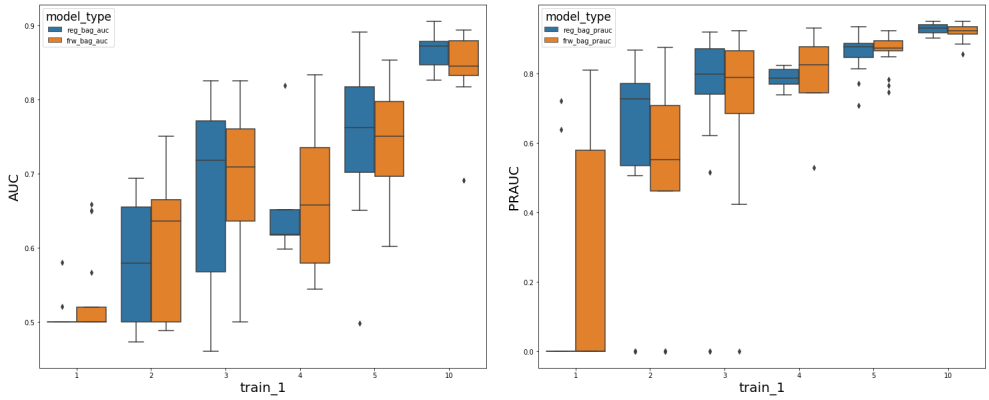

Figure 5.6: Plots of average spread of ROC-AUC (left) and PR-AUC (right) for bagging models with 50 trees on the Heart dataset

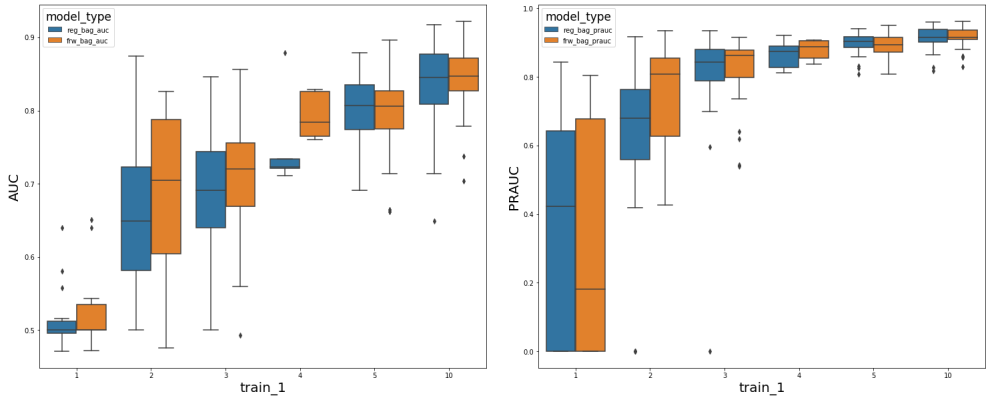

Figure 5.7: Plots of average spread of ROC-AUC (left) and PR-AUC (right) for bagging models with 100 trees on the Heart dataset

Breaking down performance by both tree count and number of low probability training observations makes the difference in performance more clear. While there were some cases of reduced performance under AUC measures by the FRW model, in general the FRW model achieved greater mean performances, with the strongest models performing better than the strongest models under the standard procedure. The difference in performance was most pronounced at very low observation counts for the low probability class, with the difference moderately diminished at higher tree counts.

Random Forest performance

Tree Count	**LP Count**	1	2	3	4	5	6+
20	**ROCAUC Diff**	0.0042	0.0325	0.0055	-0.0192	0.0178	
20	**PRAUC Diff**	-0.0085	0.0793	0.0322	-0.0092	-0.0039	
50	**ROCAUC Diff**	-0.0077	0.0204	0.0143	0.0270	0.029	-0.0067
50	**PRAUC Diff**	-0.2673	-0.1004	-0.0016	-0.0215	-0.0085	-0.0021
100	**ROCAUC Diff**	-0.0028	-0.0354	0.0444	0.0269	-0.0034	0.0045
100	**PRAUC Diff**	0.0162	-0.0685	0.0303	0.0151	-0.0069	-0.0012

Table 5.2: Mean difference in ROC-AUC and PR-AUC between standard and FRW bootstrapping models at each level of low probability training observations under the random forest model. There was an overall mean increase in ROC-AUC of 0.0077, with variance 0.0103. There was an overall mean decrease in PR-AUC of -0.0088, with variance 0.1060.

Breaking down the ROC-AUC and PR-AUC differences across different numbers of low probability training observations fails to reveal any particular trend under the random forest model. The FRW procedure doesn't appear to consistently hurt or harm ROC-AUC, while consistently attaining slightly lower PR-AUC numbers.

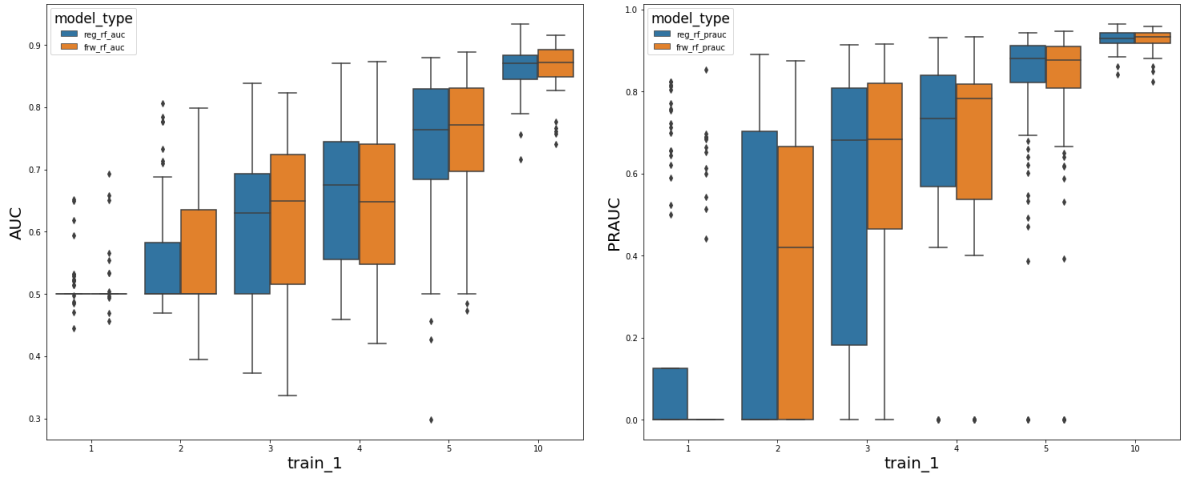

Figure 5.8: AUC (left) and PRAUC (right) for random forest models on the Heart dataset, broken down by low probability training observation count

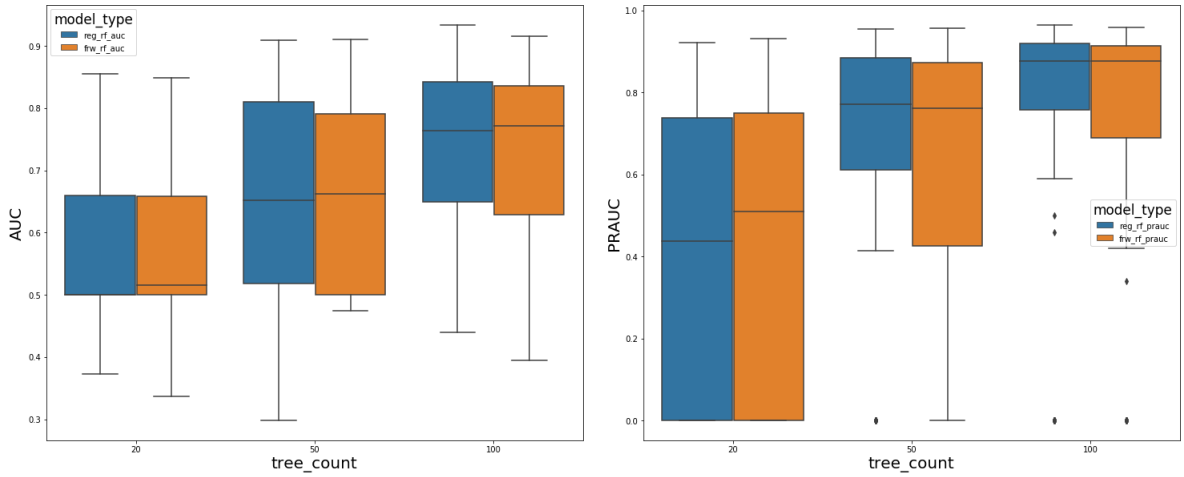

Figure 5.9: AUC (left) and PRAUC (right) for random forest models on the Heart dataset, broken down by tree count

The trend observed previously under the bagging model is sustained under the random forest model as well, with more pronounced differences between the model performances at lower training counts and lower numbers of trees. Both sets of models converge to similar ROC-AUCs and PR-AUCs at higher tree and observation counts.

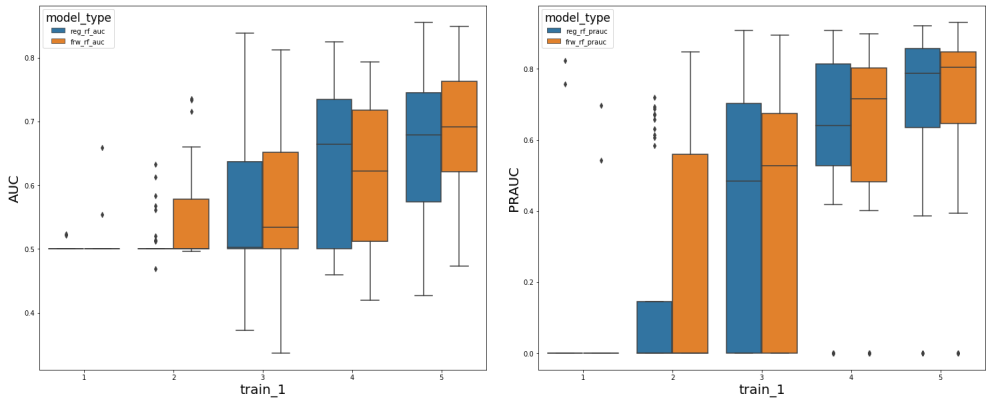

Figure 5.10: Plots of average spread of ROC-AUC (left) and PR-AUC (right) for random forest models with 20 trees on the Heart dataset

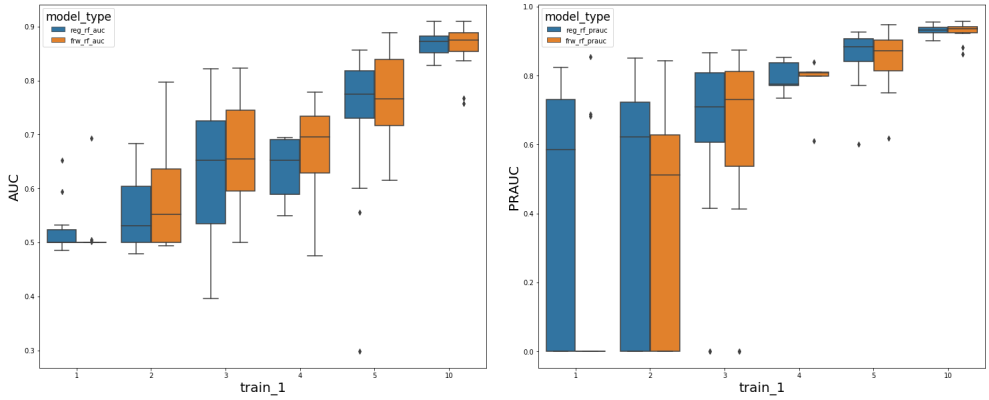

Figure 5.11: Plots of average spread of ROC-AUC (left) and PR-AUC (right) for random forest models with 50 trees on the Heart dataset

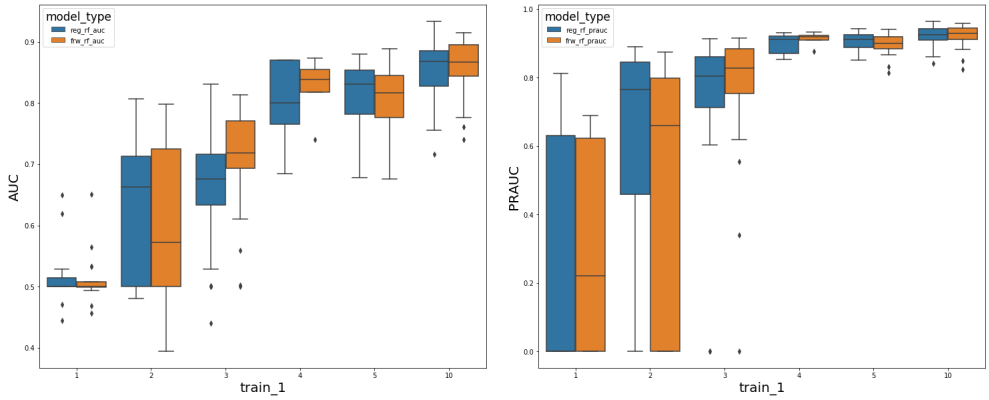

Figure 5.12: Plots of average spread of ROC-AUC (left) and PR-AUC (right) for random forest models with 100 trees on the Heart dataset

Further breaking down the performance by both tree count and low probability training observations upholds the trend, with greater average performance by the FRW model in most cases, with a few exceptions. We continue to see a general trend towards greater average performance by the FRW model, especially at lower training counts of the low probability observation and tree counts, with the strongest models under the FRW procedure often outperforming the strongest models under the standard procedure and performance converging at higher training and tree counts.

5.2 Credit Data

The credit card dataset was a large (284,807 observations), extremely imbalanced (284,215:492 ratio of negative to positive dataset collected from Kaggle. With 30 predictor variables, this dataset represented a real world example of a challenging and imbalanced classification problem to test our methods against.

At the naive threshold, we saw considerable improvement in recall from the FRW model. In particular, while no models attained high performance at the naive threshold at 1 and 2 observations of the low probability class, the FRW model began correctly classifying members of the low porbability class as early as 3 training observations, and maintained consistently higher recall over the regular model at other levels of training observations.

AUC metrics showed similar gains in performance, with the FRW models consistently outperforming the regular models - both bagging and random forests - in ROC-AUC and PRAUC. The effect was most pronounced at very low training observations, but AUC metrics remained consistently higher for the FRW models at almost all training levels.

Simulations were conducted at training sizes of 1500, 2000, and 3000. However, results are split on the number of training observations of the low probability class. In most simulations, we observed very high or perfect recall and precision even at the naive threshold for the high probability class. Performance, as such, was more a function of the number of low probability observations than of the number of high probability observations.

5.2.1 Performance at the Naive Threshold

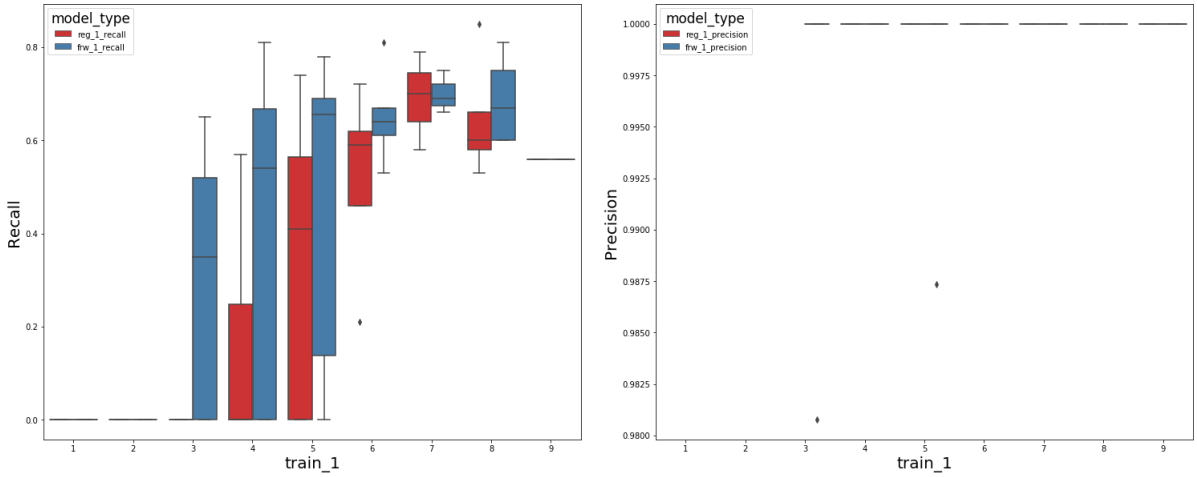

Figure 5.13: Recall (left) and precision (right) at the naive threshold for bagging models, under both standard bootstrapping and FRW bootstrapping, on credit card fraud data

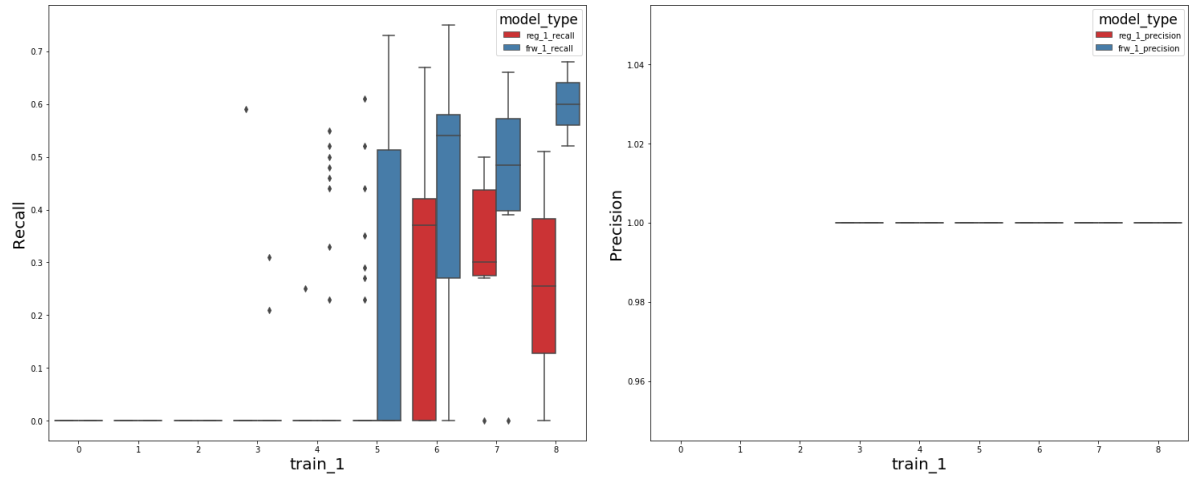

Figure 5.14: Recall (left) and precision (right) at the naive threshold for random forest models, under both standard bootstrapping and FRW bootstrapping, on credit card fraud data

At the naive threshold, there appears to be a marked difference in recall by models generated with the FRW procedure. While neither procedure yields models that perform exceptionally at the naive threshold when the number of low probability training observations is low, the FRW models achieve modest performance at lower counts and retain higher mean performances throughout.

5.2.2 AUC Metrics

Bagging Performance

Tree Count	LP Count	1	2	3	4	5	6+
20	**ROCAUC Diff**	0.1286	0.1649	0.0611	0.0300	0.5312	-0.0490
20	**PRAUC Diff**	0.1384	0.1320	-0.0536	0.0642	-0.1574	-0.192
50	**ROCAUC Diff**	0.1483	0.0877	0.0236	-0.0023	-0.0115	-0.0050
50	**PRAUC Diff**	0.3530	0.1723	-0.0516	-0.1404	-0.2569	-0.053
100	**ROCAUC Diff**	0.1654	0.0314	0.0165	-0.0024	-0.0063	-0.0179
100	**PRAUC Diff**	0.2398	0.0280	0.0181	-0.0092	-0.0620	-0.1500

Table 5.3: Mean difference in ROC-AUC and PR-AUC between standard and FRW bootstrapping models at each level of low probability training observations under the bagging model, with positive values favoring the FRW model. Overall mean difference in ROC-AUC was 0.0489, with a variance of 0.0139. Overall mean difference in PR-AUC was 0.0291, with a variance of 0.0484.

The same trend we observed in the Heart data under the bagging model appears here, with a very pronounced positive effect on the ROC-AUC and PR-AUC from the FRW procedure. The effect diminishes at greater numbers of low probability training observations, and is most profound at the smaller ensemble sizes.

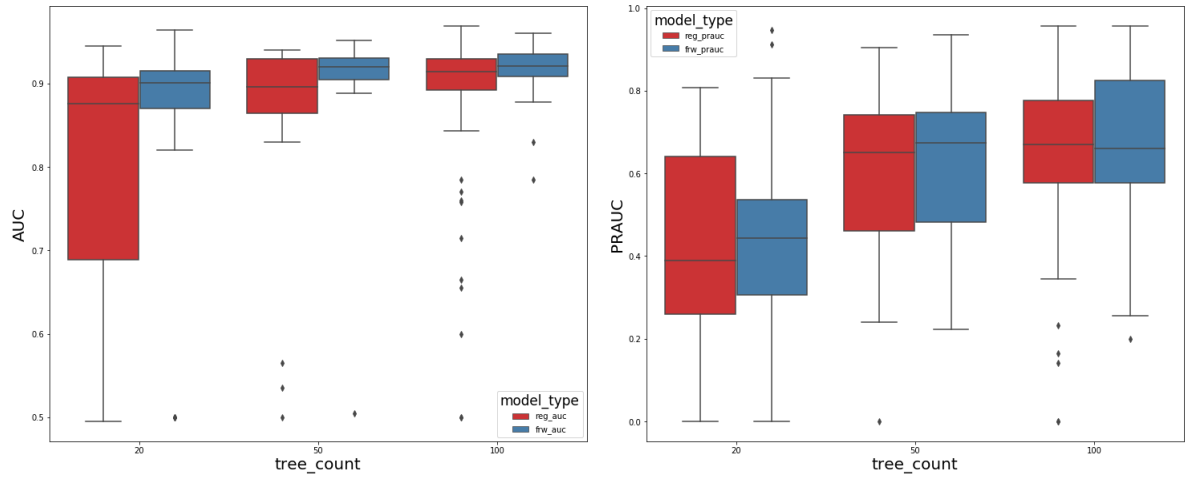

Figure 5.15: Plots of average spread of ROC-AUC (left) and PR-AUC (right) for bagging models on the Credit Card Fraud dataset, split by number of training observations

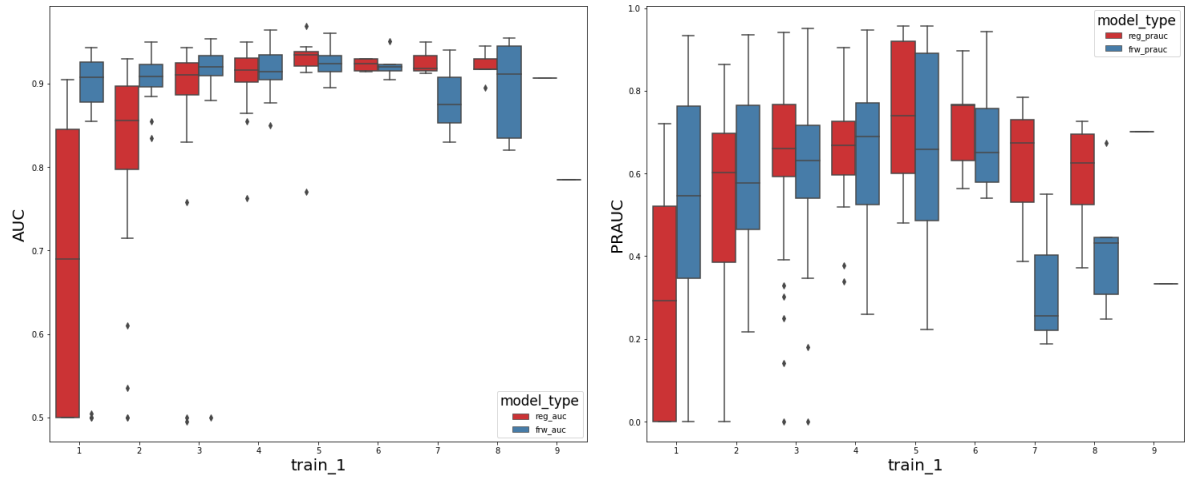

Figure 5.16: Plots of average spread of ROC-AUC (left) and PR-AUC (right) for bagging models on the Credit Card Fraud dataset, split by number of trees

We examined the average difference in ROC-AUC and PR-AUC over various counts of training observations above, then examined the average difference in performance across tree counts. In addition to greater mean ROC-AUC and PR-AUC values, the ROC-AUC and PR-AUC were much more consistent, with reduced variability compared to the standard procedures at lower training observations and lower tree counts.

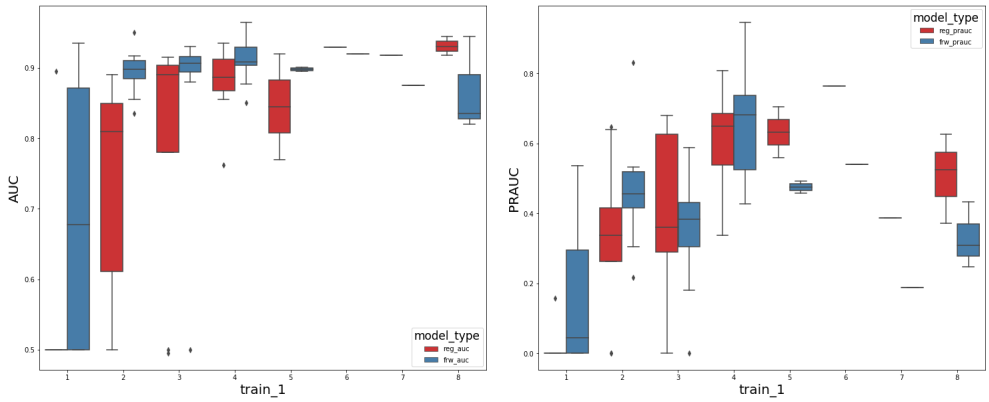

Figure 5.17: Plots of average spread of ROC-AUC (left) and PR-AUC (right) for bagging models with 20 trees on the Credit Card Fraud dataset

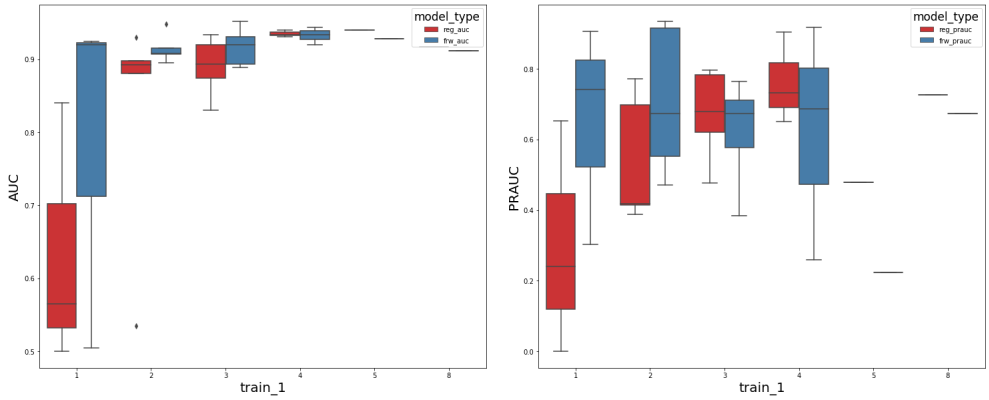

Figure 5.18: Plots of average spread of ROC-AUC (left) and PR-AUC (right) for bagging models with 50 trees on the Credit Card Fraud dataset

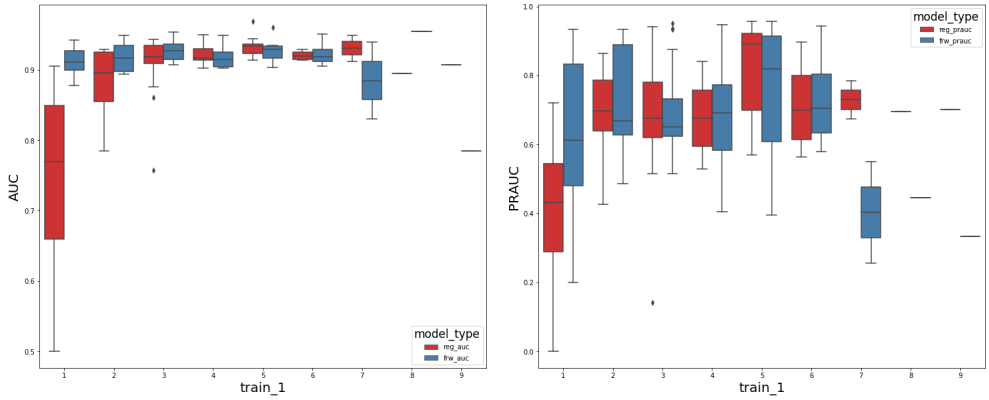

Figure 5.19: Plots of average spread of ROC-AUC (left) and PR-AUC (right) for bagging models with 100 trees on the Credit Card Fraud dataset

Breaking down by both tree count and low probability class training observation count made the trend more clear: the FRW model had the greatest positive effect on performance at the lowest tree and training observation counts. We conducted simulations at 20, 50, and 100 trees composing the ensemble. The most pronounced differences appeared at the 20 and 50 tree counts, with the performance gap between standard ensemble and FRW ensembles closing at lower observation counts under higher tree counts.

As in the Heart dataset, we see that the strongest models produced under the FRW procedure tend to perform better than the strongest models produced under the standard bootstrapping procedure. Furthermore, the FRW procedure results in lower variability in ROC-AUC performance than the standard model in many cases where the performance is improved. FRW models had greater performance under the PRAUC metric on average as well, but the difference tended to be more modest, especially at high tree counts.

Random Forest Performance

Tree Count	**LP Count**	1	2	3	4	5	6+
20	**ROCAUC Diff**	0.0813	0.0422	0.0123	-0.0233	0.0809	0.0077
20	**PRAUC Diff**	0.0621	0.0918	0.1241	0.1296	0.0072	0.0656
50	**ROCAUC Diff**	0.0922	0.0226	0.0150	-0.0383	0.0043	0.0035
50	**PRAUC Diff**	0.0873	0.1970	0.0716	-0.1657	0.0121	-0.0081
100	**ROCAUC Diff**	0.0976	0.0242	0.0004	-0.0018	-0.0031	-0.0027
100	**PRAUC Diff**	0.0855	0.0079	-0.0009	0.0316	-0.0002	0.0095

Table 5.4: Mean difference in ROC-AUC and PR-AUC between standard and FRW bootstrapping models at each level of low probability training observations under the random forest model, with positive values favoring the FRW model. Overall mean difference in ROC-AUC was 0.0235, with a variance of 0.0167. Overall mean difference in PRAUC was 0.0515, with a variance of 0.0790.

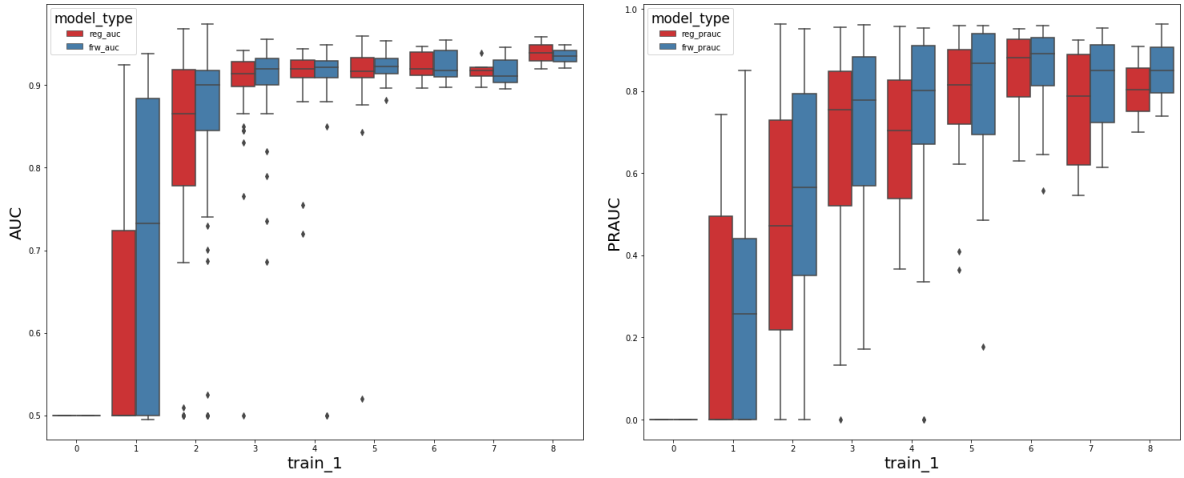

Figure 5.20: Plots of spread of ROC-AUC (left) and PR-AUC (right) for random forest models on the Credit Card Fraud dataset, split across count of low probability training observations

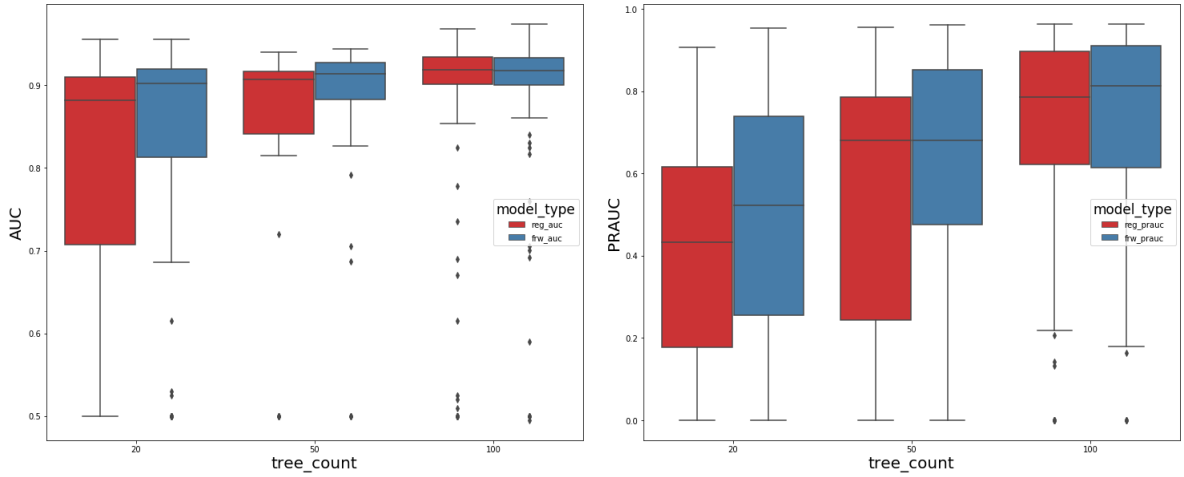

Figure 5.21: Plots of spread of ROC-AUC (left) and PR-AUC (right) for random forest models on the Credit Card Fraud dataset, split across tree count

Breaking down the performance of the RF models at different counts of low probability observations reveals a similar trend as under the bagging model. As before, we observe the greatest difference in performance at low tree counts and low numbers of observations of the low probability training class. Under the random forest models, there's also a trend towards reduced variability by the FRW models, especially among models with the same ensemble size.

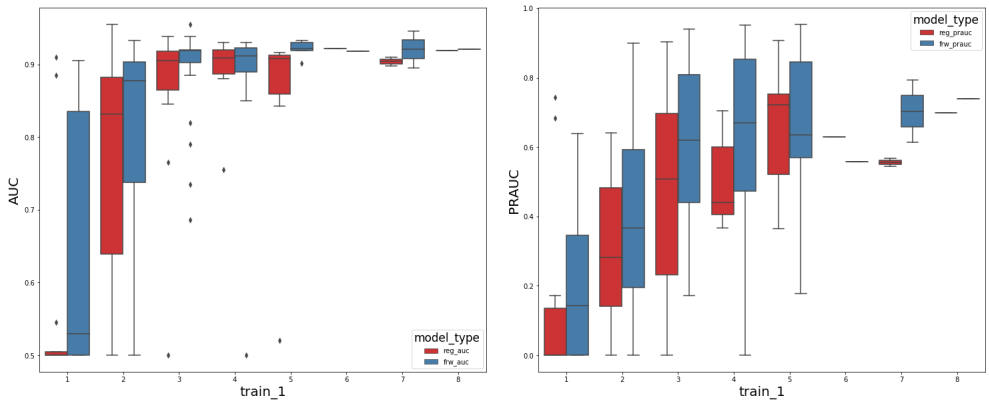

Figure 5.22: Plots of average spread of ROC-AUC (left) and PR-AUC (right) for random forest models with 20 trees on the Credit Card Fraud dataset

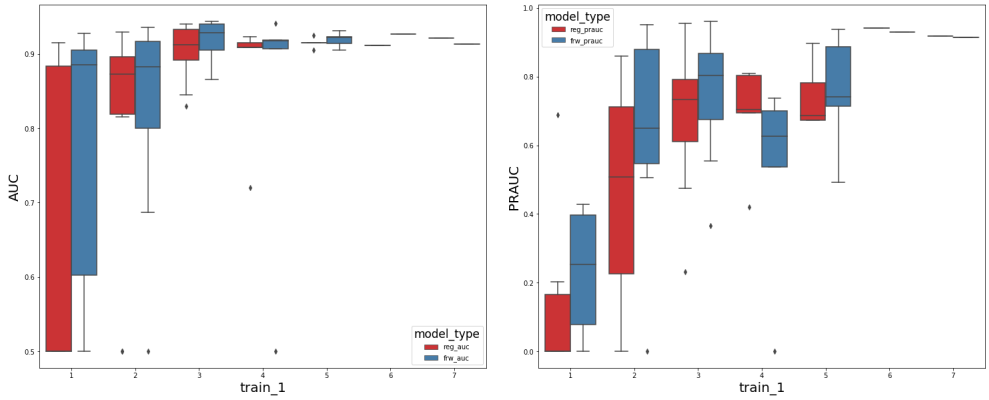

Figure 5.23: Plots of average spread of ROC-AUC (left) and PR-AUC (right) for random forest with 50 trees on the Credit Card Fraud dataset

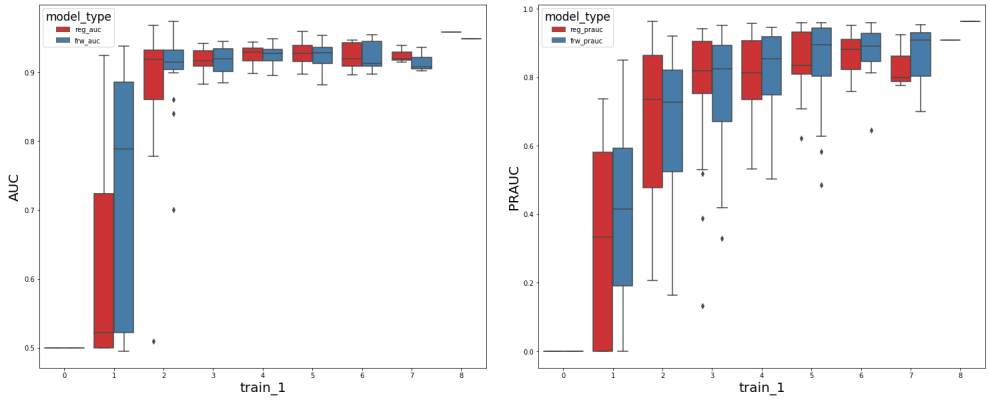

Figure 5.24: Plots of average spread of ROC-AUC (left) and PR-AUC (right) for random forest with 100 trees on the Credit Card Fraud dataset

Breaking down model performance by both tree count and low probability training observations upholds the previous trends, with the most pronounced effects at low numbers of training observations and low tree counts. As before, the FRW model appeared to have a more modest effect on PR-AUC, and performance under both metrics appeared to converge as ensemble size and the count of low probability training observations increased.

6 Concluding Remarks

Performance measures of our models across multiple datasets suggest a positive effect when training in limited contexts - very low numbers of training observations and low numbers of trees.

In general, the FRW model appeared to improve performance at the naive threshold modestly, especially in contexts of more limited data. However, the greatest apparent difference in performance was in the ROC-AUC and PR-AUC metrics.

Under the ROC-AUC and PR-AUC metrics, the FRW model consistently outperformed the standard bootstrapping ensembles at low tree counts and at low observation counts for the low probability class. The effect diminished as tree count and observation counts increased. The FRW model showed consistently greater mean performance over the equivalent standard model in both AUC metrics, but most prominently in the ROC-AUC metric. Furthermore, the strongest models trained under the FRW procedure tended to have better performance than the strongest models trained under the standard bootstrapping procedure. In some cases, we also observed reduced variability in model performance under the FRW procedure, resulting in greater consistency of model performance.

The FRW methods appeared to have the most profound effect on the credit card dataset when the number of low probability training observations was very low. As this was the dataset with the greatest number of columns and the greatest imbalance, it is likely that the FRW bootstrap proves most useful when the parameter space is large, the number of observations of a low probability class is extremely limited, and when the number of samples to explore the parameter space over is small.

At minimum, the FRW method appears no worse on average than using the standard bootstrap. The FRW bootstrap offers an option for confronting data imbalance without misrepresenting the underlying data structure, introducing less risk of overfitting than alternative sampling strategies such as oversampling. Given the minimal difference in computational time, this method may prove useful as a first option prior to implementing more elaborate sampling schemes when dealing with highly imbalanced data.

Furthermore, the improved performance at fewer resamples and lower training sizes suggests the FRW model may serve a role in producing quicker turnaround in model production, allowing ensembles to be trained more quickly, requiring smaller data and fewer resamples.

Future work might focus on other resampling methods utilizing continuous weighting. In addition, as the fractional random weighted bootstrap has seen much use for statistical inference on censored data, future experimentation might explore FRW ensemble methods on labeling for censored data.

In addition, ensemble methods exist for many other machine learning algorithms, such as the support vector machine and logistic regression. The evidence of performance differences attained on decision tree ensembles indicate it may be fruitful to explore FRW alternatives for other ensemble methods.

References

[1] Credit card fraud data.

[2] Bekkar, M., Djemaa, H. K., and Alitouche, T. A. Evaluation measures for models assessment over imbalanced data sets. *Journal of Information Engineering and Applications 3*, 10 (2013).

[3] Ben-Bassat, M. Myopic policies in sequential classification. *IEEE Transactions on Computers 27*, 2 (1978), 170–174.

[4] Bernard, S., Heutte, L., and Adam, S. A study of strength and correlation in random forests. International Conference on Intelligent Computing, pp. 186–191.

[5] Breiman, L. *Classification and Regression Trees*. Taylor and Francis, 1984.

[6] Breiman, L. Bagging predictors. *Machine Learning 24* (1996), 123–140.

[7] Brown, C. D., and Davis, H. T. Receiver operator characteristics curves and related decision measures: A tutorial. *Chemometrics and Intelligent Laboratory Systems 80*, 1 (2006), 24–38.

[8] Chawla, N. V. Data mining for imbalanced datasets: An overview. In *Data Mining and Knowledge Discovery Handbook*. 2010, pp. 875–886.

[9] Cortes, C., and Vapnik, V. Support-vector networks. *Machine Learning 20* (1995), 273–297.

[10] Davis, J., and Goadrich, M. The relationship between precision-recall and roc curves. Tech. rep., Computer Sciences Department, University of Wisconsin-Madison, 2006.

[11] Dua, D., and Graff, C. UCI machine learning repository, 2017.

[12] Efron, B. Bootstrap methods: Another look at the jackknife. *The Annals of Statistics 7*, 1 (1979).

[13] Gotwalt, C., Xu, L., Hong, Y., and Meeker, W. Applications of the fractional-random-weight bootstrap. Preprint available at https://arxiv.org/pdf/1808.08199.pdf.

[14] Gupta, R. D., and Richards, D. S. P. The history of the dirichlet and liouville distributions. *International Statistical Review 69*, 3 (2001), 433–446.

[15] Hastie, T., Tibshirani, R., and Friedman, J. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*, 2 ed. Springer, 2009.

[16] Hulse, J. D. V., Khoshgoftaar, T. M., and Napolitano, A. Experimental perspectives on learning from imbalanced data. International Conference on Machine Learning.

[17] Jin, Z., Ying, Z., and Wei, L. J. A simple resampling method by perturbing the minimand. *Biometrika 88*, 2 (2001), 381–390.

[18] Kass, G. V. An exploratory technique for investigating large quantities of categorical data. *Applied Statistics 29*, 2 (1980), 119–127.

[19] Kayri, M., and Kayri, I. The comparison of gini and twoing algorithms in terms of predictive ability and misclassification cost in data mining: An empirical study. *International Journal of Computer Trends and Technology 27*, 1 (2015).

[20] Kearns, M. Thoughts on hypothesis boostings. Unpublished Manuscript, 1988.

[21] Kleinbaum, D. G., and Klein, M. *Logistic Regression: A Self-Learning Text*, 3 ed. Springer, 2010.

[22] Minka, T. P. A comparison of numerical optimizers for logistic regression. Preprint available at https://arxiv.org/pdf/1808.08199.pdf.

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

[24] Quinlan, J. R. C4.5: Programs for machine learning. *Machine Learning 16*, 3 (1994), 225–240.

[25] Rubin, D. B. The bayesian bootstrap. *The Annals of Statistics 9*, 1 (1981), 130–134.

[26] Safavian, S. R., and Landgrebe, D. A survey of decision tree classifier methodology. *IEEE Transactions on Systems 21* (1991), 660–674.

[27] Schapire, R. E. The strength of weak learnability. *Machine Learning 5* (1990), 197–227.

[28] the Gini Index, T. C. B., and Criteria, I. G. *Annals of Mathematics and Artificial Intelligence 41* (2004), 77–93.

[29] Yap, B. W., Rani, K. A., Rahman, H. A. A., Fong, S., Khairudin, Z., and Abdullah, N. N. An application of oversampling, undersampling, bagging and boosting in handling imbalanced datasets. First International Conference on Advanced Data and Information Engineering, pp. 13–22.

[30] Zhang, G. P. Neural networks for classification: A survey. *IEEE Transactions on Systems, Man, and Cybernetics 30*, 4 (2000).

[31] Zhang, H. The optimality of naïve bayes. In *In FLAIRS2004 conference* (2004).

A Additional Datasets

In addition to the Heart and Credit Card Fraud datasets, we conducted simulations over the Iris and Glass datasets to test the FRW method on multiclass classification problems. On each of these datasets, we trained at varying numbers of training observations in order to observe the effect of increasingly low numbers of training counts to select on. The Iris dataset is perfectly balanced across three classes with a total of 150 observations, while the Glass dataset has six classes unevenly distributed across 214 observations.

A.1 Iris Data

The Iris dataset has an even class distribution (with 50 observations of each of 3 classes). While not highly imbalanced, at small sample sizes, this dataset would suffer from the same issue posed by highly imbalanced data: potentially excluding classes in bootstrapped resamples. The Iris data provides a starting point for comparison of our models. As there were only five predictor variables, only bagging models were tested on the Iris dataset.

We saw mixed effects of the FRW resampling method on this dataset. Recall was modestly improved in some cases (e.g. on Setosa at 50 observations, Virginica at all observation counts, and Versicolor at 15 and 30 observations), but moderately worse in others (e.g. on Setosa at 15 and 30 observations, Bersicolor at 50 observations). Precisions was also modestly affected, with similar mixed results

Regular Mean Recalls

	Setosa	Virginica	Versicolor
15 Observations	0.45	0.2	0.35
30 Observations	0.9271	0.7698	0.7346
50 Observations	0.9946	0.9247	0.9035

Table A.1: Recall by class of the standard bagging model. Each row represents an average over all simulations conducted at a set sample size.

FRW Mean Recalls

	Setosa	Virginica	Versicolor
15 Observations	0.375	0.225	0.4
30 Observations	0.9220	0.8064	0.7381
50 Observations	1.0	0.9394	0.8975

Table A.2: Recall by class of the FRW bagging model. Each row represents an average over all simulations conducted at a set sample size.

Regular Mean Precisions

	Setosa	Virginica	Versicolor
15 Observations	0.3235	0.3222	0.3206
30 Observations	0.9253	0.7627	0.8672
50 Observations	0.9963	0.9036	0.9279

Table A.3: Precision by class of the regular bagging model. Each row represents an average over all simulations conducted at a set sample size. Note that these numbers are not completely accurate - simulations where a method made no predictions of a class are not counted, as precision would be NaN.

FRW Mean Precisions

	Setosa	Virginica	Versicolor
15 Observations	0.3230	0.3218	0.3208
30 Observations	0.9400	0.7602	0.9041
50 Observations	1.0	0.9053	0.9376

Table A.4: Precision by class of the FRW bagging model. Each row represents an average over all simulations conducted at a set sample size. Note that these numbers are not completely accurate - simulations where a method made no predictions of a class are not counted, as precision would be NaN.

Recall Visualizations

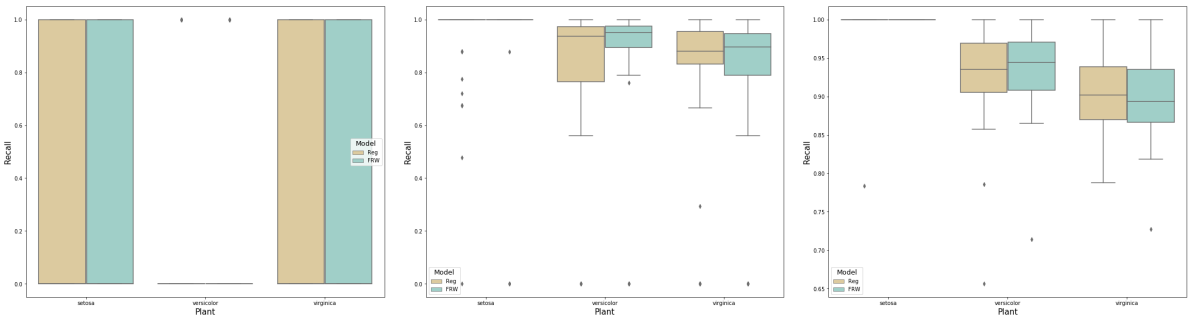

Table A.5: Recall on Iris dataset at 15 (left), 30 (middle), and 50 (right) training observations

Precision Visualizations

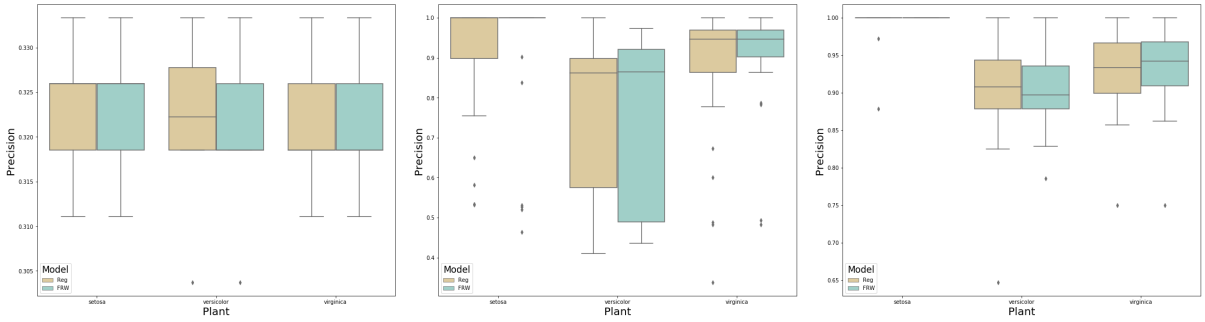

Table A.6: Precision on Iris dataset at 15 (left), 30 (middle), and 50 (right) observations

At 30 and 50 training observations, we see moderately improved performance over the standard model, especially on the versicolor class. Precision was only modestly improved at 30 and 50 observations over the standard model.

A.2 Glass Data

The Glass dataset posed a particular challenge, having six classes, several of them with few observations and several with many observations. Across both random forest and bagging models, we saw little difference in either recall or precision between the standard and FRW models.

Class	1	2	3	5	6	7
Count	69	76	17	13	9	29

Table A.7: Class counts for Glass dataset

A.2.1 Bagging on Glass Data

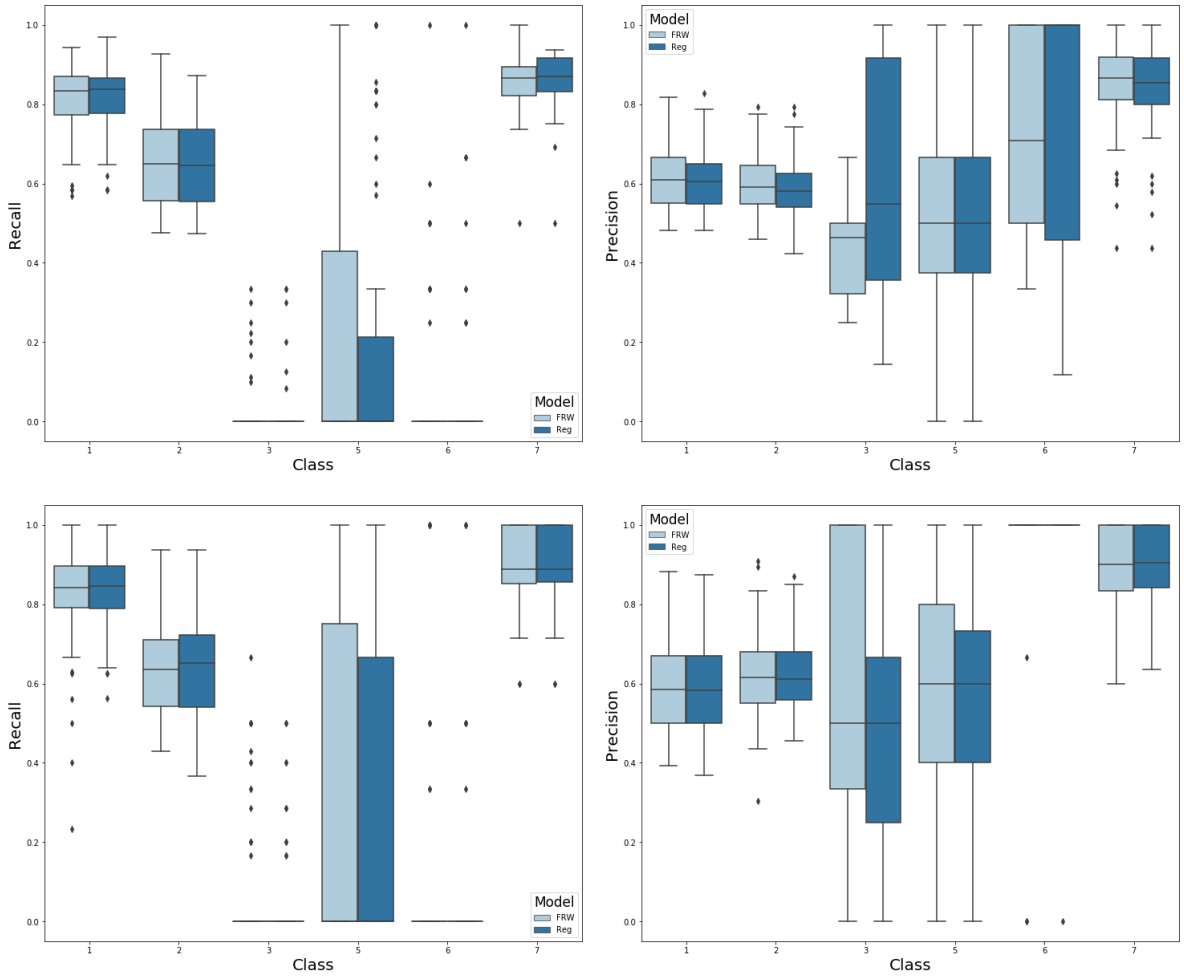

Table A.8: Recall (left) and Precision (right) for bagging models on the Glass dataset at 100 (top) and 150 (bottom) training observations. Across both random and FRW models, performance was relatively similar.

A.2.2 Random Forests on Glass Data

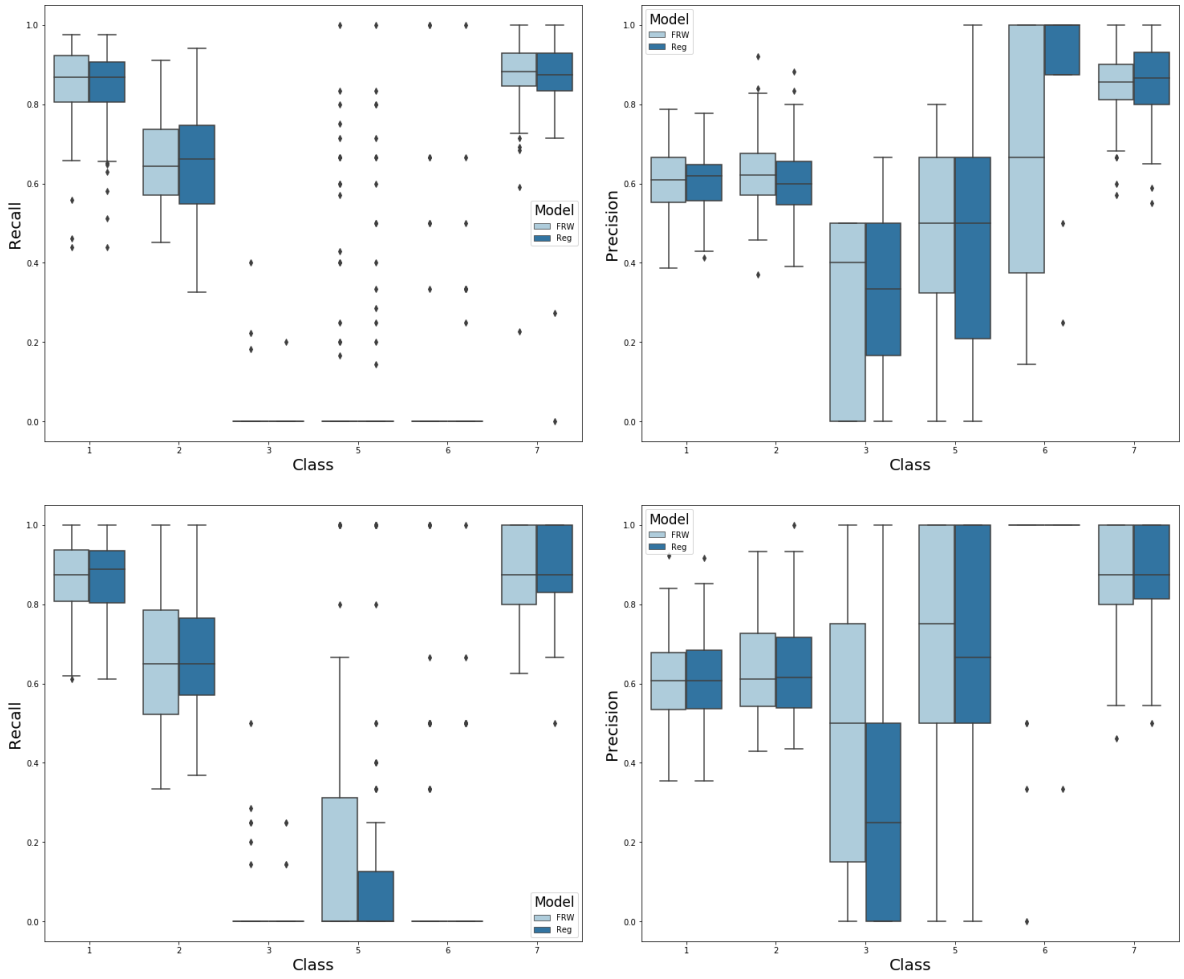

Figure A.1: Recall (left) and Precision (right) for random forest models on the Glass dataset at 100 (top) and 150 (bottom) training observations. Similar to the bagging model, performance was largely unaffected by the FRW procedure under the random forest model.

B Decision Tree and Ensemble Code

```
library(MCMCpack)
library(rpart)
library(data.table)

# Builds a tree using a bootstrap sample
bootstrap_tree <- function(dataset, factors, group, train_size,tree_
   control){
  samp <- bootstrap(dataset, train_size, TRUE)
  if(length(unique(samp[,group])) < 2){
    #make root tree if fewer than 2 groups are present
    root_tree <- as.character(samp[1,group])
    return(root_tree)
  }
  else{
    if(is.na(tree_control)){
      # Tree without control
      Tree <- rpart(reformulate(termlabels=factors, response=group),
          data=samp, method="class")
      return(Tree)
    }
    else{
```

```
      # Tree with control
      Tree <- rpart(reformulate(termlabels=factors, response=group),
          data=samp, method="class", control=tree_control)
      return(Tree)
    }
  }
}

# Builds a tree using an FRW bootstrap sample
frw_tree <- function(dataset, factors, group, tree_control, weight){
  if(length(unique(dataset[,group])) <2){
    root_tree <- as.character(dataset[1,group])
    return(root_tree)
  }
  else{
    if(is.na(tree_control)){
      Tree <- rpart(reformulate(termlabels=factors, response=group),
          data=dataset,weights=weight, method="class")
      return(Tree)
    }
    else{
      Tree <- rpart(reformulate(termlabels=factors, response=group),
          data=dataset,weights=weight, method="class", control=tree_
         control)
      return(Tree)
    }
  }
}
```

```
# Builds a bag of regular decision trees
Reg_Bag <- function(dataset, tree_count, factors, group, train_size
   =0, tree_control=NA){
  if (train_size == 0){
    train_size = nrow(dataset)
  }
  rbag <- replicate(tree_count, bootstrap_tree(dataset, factors,
     group, train_size,tree_control), simplify=FALSE)
  return(rbag)
}

# Builds an FRW bag of decision trees
FRW_Bag <- function(dataset, tree_count, factors, group,tree_control
   =NA){
  weights <- rdirichlet(tree_count, alpha=rep(1, nrow(dataset)))
  frwbag <- lapply(as.list(seq(1, tree_count)), function(x) frw_tree
     (dataset, factors, group, tree_control, weights[x,]))
  return(frwbag)
}

# Constructs a standard random forest
Reg_RF <- function(dataset, tree_count, factors, group, train_size
   =0, max_fac=5, tree_control=NA){
  if(train_size==0){
    train_size = nrow(dataset)
  }
  subfac <- function(){
    facs <- sample(factors, max_fac)
    return(facs)
```

```
  }
  rfbag <- replicate(tree_count, bootstrap_tree(dataset, subfac(),
     group, train_size, tree_control), simplify=FALSE)
  return(rfbag)
}

# Constructs an FRW random forest
FRW_RF <- function(dataset, tree_count, factors, group, max_fac=5,
   tree_control=NA){
  subfac <- function(){
    facs <- sample(factors, max_fac)
    return(facs)
  }
  weights <- rdirichlet(tree_count, alpha=rep(1, nrow(dataset)))
  frw_rfbag <- lapply(as.list(seq(1, tree_count)), function(x) frw_
     tree(dataset, subfac(), group, tree_control, weights[x,]))
  return(frw_rfbag)
}

#
# The section below contains many different prediction methods,
   encompassing several different prediction types and
# averaging mechanisms for generating ensemble predictions
#


# Returns the top most voted for class
bag_prediction <- function(bag, datapoint){
  predictions <- unlist(lapply(seq(1, length(bag)), function(x) tree
```

```
    _prediction(bag[[x]], datapoint)))
  return(names(sort(table(predictions), decreasing=TRUE)[1]))
}

# Utilizes the above method to make predictions on a dataframe of
   new observations
bag_prediction_df <- function(bag, test_data){
  predictions <- lapply(seq(1, nrow(test_data)), function(x) bag_
     prediction(bag, test_data[x,]))
  predictions_df <- do.call(rbind, predictions)
  return(predictions_df)
}

# Returns the class vote that each tree made
bag_prediction_long <- function(bag, datapoint){
  predictions <- unlist(lapply(seq(1, length(bag)), function(x) tree
     _prediction(bag[[x]], datapoint)))
  return(predictions)
}

# Probability prediction using proportion of tree votes
bag_prediction_prop <- function(bag, datapoint, classes){
  predictions <- bag_prediction_long(bag, datapoint)
  counts <- as.data.frame(table(predictions))
  counts$Freq <- counts$Freq/length(predictions)
  for(class in classes){
    if(!(is.element(class, counts$predictions))){
      new_row <- list(class, 0)
      counts <- rbindlist(list(counts, new_row))
```

```
    }
  }
  return(counts)
}

# Uses the above method to make predictions on a dataframe of new
   observations
bag_prediction_prop_df <- function(bag, test_data, classes){
  prop_preds <- lapply(seq(1, nrow(test_data)), function(x) bag_
     prediction_prop(bag, test_data[x,], classes))
  out_df <- do.call(rbind, prop_preds)
  colnames(out_df) <- c("Class", "Probability")
  out_df$pred_ID <- ceiling(as.numeric(rownames(out_df))/length(
     classes))
  out_df <- dcast(data=out_df, formula=pred_ID~Class, value.var="
     Probability")
  return(out_df)
}

# Probability prediction using response function - i.e. averages
   over the probabilities returned by rpart
bag_prediction_prob <- function(bag, datapoint, classes){
  predictions <- lapply(seq(1, length(bag)), function(x) tree_
     prediction_prob(bag[[x]], datapoint, classes))
  prediction_df <- do.call(rbind, predictions)
  return(colMeans(prediction_df))
}

# Uses the above method to make predictions on a dataframe of new
```

```
   observations
bag_prediction_prob_df <- function(bag, test_data, classes){
  prob_preds <- lapply(seq(1, nrow(test_data)), function(x) bag_
     prediction_prob(bag, test_data[x,], classes))
  out_df <- do.call(rbind, prob_preds)
  return(out_df)
}


# Returns all probabilities predicted by trees in the bag
bag_prediction_probs_long <- function(bag, datapoint, classes){
  predictions <- lapply(seq(1, length(bag)), function(x) tree_
     prediction_prob(bag[[x]], datapoint, classes))
  prediction_df <- do.call(rbind, predictions)
  return(prediction_df)
}

# Wrapper for rpart's prediction method to account for root trees
# Returns a class prediction
tree_prediction <- function(tree, datapoint){
  if(is.character(tree)){
    return(tree)
  }
  else{
    prediction <- predict(tree, datapoint, type="class")
    return(levels(prediction)[prediction])
  }
}
```

```
# Wrapper for rpart's prediction method to account for root trees
# Returns a probability prediction
tree_prediction_prob <- function(tree, datapoint, classes){
  pvec <- rep(0, length(classes))
  prediction_df <- as.data.frame(matrix(nrow=0, ncol=length(classes)
     ))
  prediction_df <- rbind(prediction_df, pvec)
  colnames(prediction_df) <- classes
  if(is.character(tree)){
    prediction_df[1,tree] <- 1
  }
  else{
    prediction <- predict(tree, datapoint, type="prob")
    pclasses <- dimnames(prediction)[[2]]
    for(i in seq(1, length(pclasses))){
      prediction_df[1,pclasses[i]] <- prediction[i]
    }
  }
  return(prediction_df)
}

# Standard bootstrapping method, sampling with replacement
bootstrap <- function(dataset, count, replacement){
  resample <- dataset[sample(nrow(dataset), count, replace=
     replacement),]
  return(resample)
}
```