Tree Aggregation in Random Forest

Aman Murba (20573620), Jasper Zhu (20652630), Chuqiao (Veronica) Li (20551964)

Contents

1.0 Introduction

This report explores different methods to aggregate trees in a Random Forest as an alternative of taking the average of all trees in an effort to improve the predictive power of a Random Forest. We are proposing two different weight assignment methods, OOB (out-of-bag) error weighted and glmnet weighted approaches. We will explain the motivation, illustrate how the algorithms are set up, and test the algorithms on both simulated and real-world data. We wil interpret the results, present and explain any interesting findings before we draw a conclusion.

1.1 Motivation

A random forest uses a subset of explanatory variables at each split in addition to bootstrap resampling to build trees. Through limiting the set of explanatory variables, we decorrelate trees within the random forest, improving predictive performance. However, since each tree samples a subset of explanatory variables when considering each node split, it is only able to explain some aspect of the response. Therefore trees in a random forest are overall weaker learners compared to bagging, and some trees capture significant variables better than others. Even though a large number of trees are used in a random forest to compensate for the fact that the trees are weak learners, the predictitve power of the random forest is still compromised since all trees are assigned equal weights.

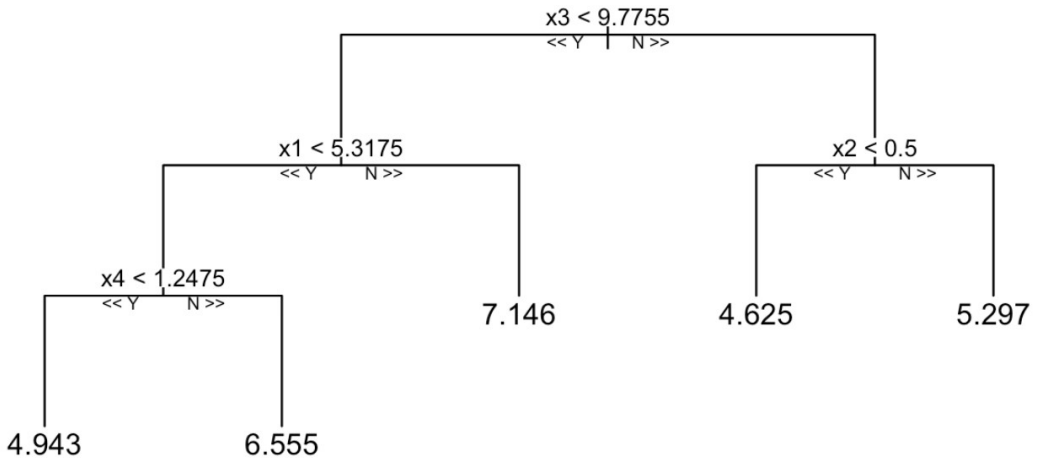
To show that trees in a random forest does not always split at the significant explanatory variables, we simulate a dataset as follows.

$$X_1 \sim N(5, 1)$$
$$X_2 \sim Bern(0.5)$$
$$X_3 \sim Unif(0, 10)$$
$$X_4 \sim Exp(1)$$

Note that X_1 through X_4 are independent from each other. We can build a responce Y using only X_1 and X_4.

$$Y = X_1 + X_4 + \epsilon, \quad \epsilon \sim Norm(0, 1)$$

We build a random forest using the similated dataset to predict Y, with the following as a tree sampled within the random forest.



We limit the maximum number of nodes to 5 for simplicity. In the random sample tree we have above, we see a split at X_2 and X_3 even though they are not used in the empirical model for Y. To further investigate how many trees in the random forest split at the unused explanatory varialbes, we can calculate the frequency of variables used in the random forest of 500 trees.

```
##    x1    x2    x3    x4
## 0.730 0.690 0.738 0.706
```

The table above summarized portion of trees in the random forest that split at each of the 4 explanatory variables. Note that X_2, X_3 are included in a non-negligible portion of the random forest, even though they are not used in the simulated model. Through this simple example, we see that a portion of the trees in a random forest do not select the correct variables. Assigning equal weights to all trees may not be the optimal algorithm.

1.2 Proposal

Since equal weight assignment in a random forest compromises its predictive power, we propose two algorithms of weight assignment in an effort to improve prediction of random forest.

One approach is to use out-of-bag error to assign weights. We transform the out-of-bag error of all trees in the random forest so that the trees with stronger predictive power are assigned higher weights, and vice versa. We apply three different transformations and compare results.

The second algorithm we are proposing is to apply shrinkage to determine weights assignment through the `glmnet` function. We treat each tree of a random forest as an explanatory variable, and apply either LASSO or ridge regression to determine the optimal weight assignment.

In Appendix A, we also explore an alternative algorithm in which we treat each tree as a linear combination of indicator functions. We apply ridge regression to shrink the parameters at each node before taking the average of all ridge-shrunken trees.

2.0 Analysis

In this section, we will further explain in detail how we set up the two algorithms. Implementations of the algorithms in R can be found in Appendix B.

2.1 Algorithms

2.1.1 OOB Weighted Random Forest

Instead of taking the average of trees in a random forest, we explore using out-of-bag error to assign weights. We examined weight assignments that invert out-of-bag error such that trees with a lower out-of-bag error get assigned a larger weight. Since lower out-of-bag error generally indicate stronger predictive power, the OOB weighted random forest is expected to provide improved results. That is, if we have M trees in a random forest:

$$rf = \{T_1, T_2, ..., T_M\}$$

We take the weighted average of the M tree outputs:

$$\hat{Y} = \sum_{i=1}^{M} w_i^* T_i(X_i)$$

Note that in a random forest, $w_i^* = \frac{1}{M}$, whereas we apply three transformations to out-of-bag error to determine w_i. The transformations we selected are inverse, log and linear.

Inverse:

$$w_{inverse_i} = \frac{1}{MSPE_{OOB_i}} \quad \text{for } i = 1, 2, \ldots, M$$
$$w^*_{inverse_i} = \frac{w_{inverse_i}}{\sum_{i=1}^{M} w_{log_i}}$$

Logarithm:

$$w_{log_i} = log(MSPE_{OOB_i}) \quad \text{for } i = 1, 2, \ldots, M$$
$$w^*_{log_i} = \frac{w_{log_i}}{\sum_{i=1}^{M} w_{sqrt_i}}$$

Linear:

$$w_{linear_i} = \max_i(MSPE_{OOB_i}) - MSPE_{OOB_i} \quad \text{for } i = 1, 2, \ldots, M$$
$$w^*_{linear_i} = \frac{w_{linear_i}}{\sum_{i=1}^{M} w_{linear_i}}$$

We will implement this algorithm with the three out-of-bag transformations and compare results.

2.1.2 glmnet Weighted Random Forest

The second algorithm we are proposing is to apply shrinkage through the `glmnet` function to determine weights assignment. In particular, we will focus on two `glmnet` implementations. Firstly, ridge regression is a form of regularization introduced to reduce multicollinearity. The second is LASSO, a type of regularization that combines the least squares method with a penalty term. It allows for parameters to equal zero, therefore acting as a variable selection procedure.

In a random forest, we can treat the weights as parameters and use `glmnet` to determine the optimal weight assignment. When fitting with LASSO, we allow weights of 0 to be assigned to trees with little predictive power, selecting significant trees to take the average of. We expect prediction results to further improve using this algorithm, compared to the previous algorithm.

Suppose we build a random forest with M trees:

$$rf = \{T_1, T_2, ..., T_M\}$$

We have n responses, y_1 to y_n:

$$y = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}$$

Each of the p explanatory variables has n observations. Thus we get a n by p explanatory matrix X, based on which we will make our prediction of Y.

$$x = \begin{pmatrix} x_{11} & x_{12} & \dots & x_{1p} \\ x_{21} & x_{22} & \dots & x_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \dots & x_{np} \end{pmatrix} = \begin{pmatrix} \vec{x}_1^T \\ \vec{x}_2^T \\ \vdots \\ \vec{x}_n^T \end{pmatrix}$$

We run each of the M trees on all n observations to predict Y. Each tree is assigned a weight $\beta_i, i = 1, 2, \dots, M$. Note that in a random forest, $\beta_i = \frac{1}{M}, i = 1, 2, \dots, M$.

$$\hat{Y} = \begin{pmatrix} T_1(\vec{x}_1) & T_2(\vec{x}_1) & \dots & T_M(\vec{x}_1) \\ T_1(\vec{x}_2) & T_2(\vec{x}_2) & \dots & T_M(\vec{x}_2) \\ \vdots & \vdots & \ddots & \vdots \\ T_1(\vec{x}_n) & T_2(\vec{x}_n) & \dots & T_M(\vec{x}_n) \end{pmatrix} \begin{pmatrix} \beta_1 \\ \beta_2 \\ \vdots \\ \beta_m \end{pmatrix} = x^* \beta$$

We can apply the `glmnet` function to estimate the optimal weight, $\hat{\beta}$, to assign to each tree. Since this function generally works better with an intercept, we use the following in our model.

$$X^* = \begin{pmatrix} 1 & T_1(\vec{x}_1) & T_2(\vec{x}_1) & \dots & T_M(\vec{x}_1) \\ 1 & T_1(\vec{x}_2) & T_2(\vec{x}_2) & \dots & T_M(\vec{x}_2) \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & T_1(\vec{x}_n) & T_2(\vec{x}_n) & \dots & T_M(\vec{x}_n) \end{pmatrix} = \begin{pmatrix} \vec{x}_1^{T*} \\ \vec{x}_2^{T*} \\ \vdots \\ \vec{x}_n^{T*} \end{pmatrix}$$

2.2 Applications

2.2.1 Simulated Examples

In this section, we explore the results of Algorithm 1, also referred to as the OOB weighted random forest and Algorithm 2, also referred to as the glmnet weighted random forest, on several numerical simulations. A brief explanation of each simulation will be provided below, but exact `R` implementations can be found within Appendix C. Here, we will focus on primarily four results:

1) For Algorithm 1, we are curious whether the choice of linear, logarithmic or inverse scaling has a significant impact on the predictive power. For Algorithm 2, we are curious whether the choice of LASSO or ridge regression has a significant impact on the predictive power.
2) How Algorithm 1 and Algorithm 2 affects predictions for various values of `ntree`, the number of trees used in a random forest
3) How Algorithm 1 and Algorithm 2 is affected by the choice of `mtry`, the number of candidate variables sampled for each split in a random forest tree.
4) For Algorithm 2, we are curious if we can use the frequency of variables in tree split points as a measure of variable importance.

To explore this fourth consideration, we generate variable frequency tables for each simulation. Random forest and LASSO weighted random forest are built on the simulated data. The `LASSO.Splits.Rel` column counts the number of splits a variable is used within trees selected by the LASSO weighted random forest and divides it by the total number of splits. The `LASSO.Trees.Used` column counts the number of trees in the LASSO weighted random forest with splits on each variable. The `RF.Splits.Rel` and `RF.Trees.Used` columns are defined similarly for the original random forest. Lastly, the `X..Chg.in.Rel.Splits` calculates the percentage change in the relative frequency of each variable, that is, `X..Chg.in.Rel.Splits = (LASSO.Splits.Rel / RF.Splits.Rel - 1) * 100`. If the frequency of variable is a useful measure of variable importance, we expect features which generate the response to have positive values in this last column.

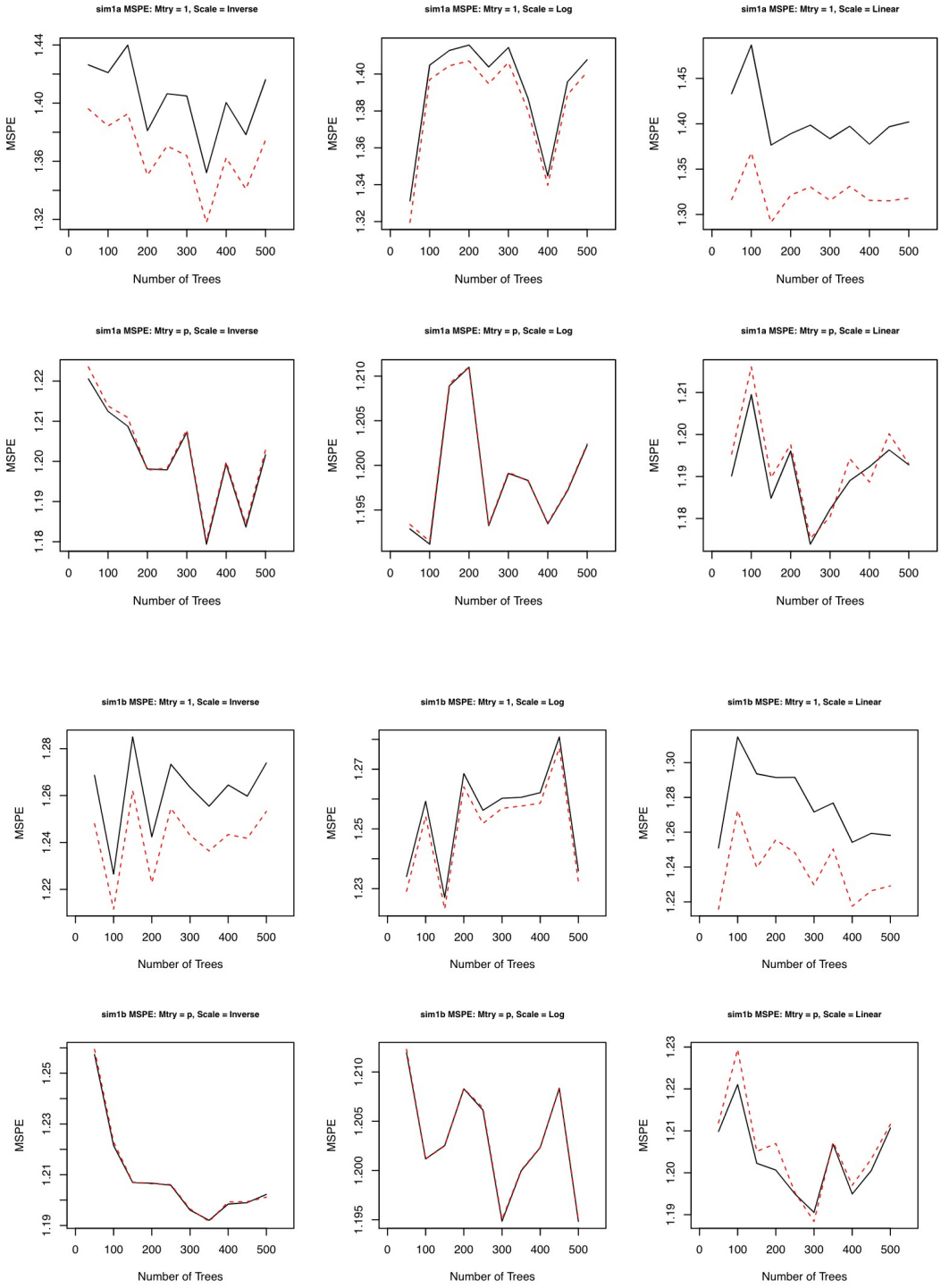
Again, we use p to denote the number of explanatory variables in the data-set. Moreover, we compare the results separately for 1, p, and the default `mtry` value in the `randomForest` function. However, note that the default `mtry` value for the simulated examples below is equal to 1, so we will be exploring two different `mtry` values, with `mtry` $= p$ being equivalent to bagging. The data set of features consists of variables X_1, X_2, X_3, and X_4 as previously defined in section 1. We also introduce a correlated variable X_5 to test the performance of our models in the presence of multicollinearity. We generate X_5 as $X_5 = X_1 + 5X_4$.

Below, all MSPE numbers for a random forest are drawn in solid black lines, and MSPE numbers for our LASSO weighted random forests are drawn in dashed red lines. Results begin on the following page.

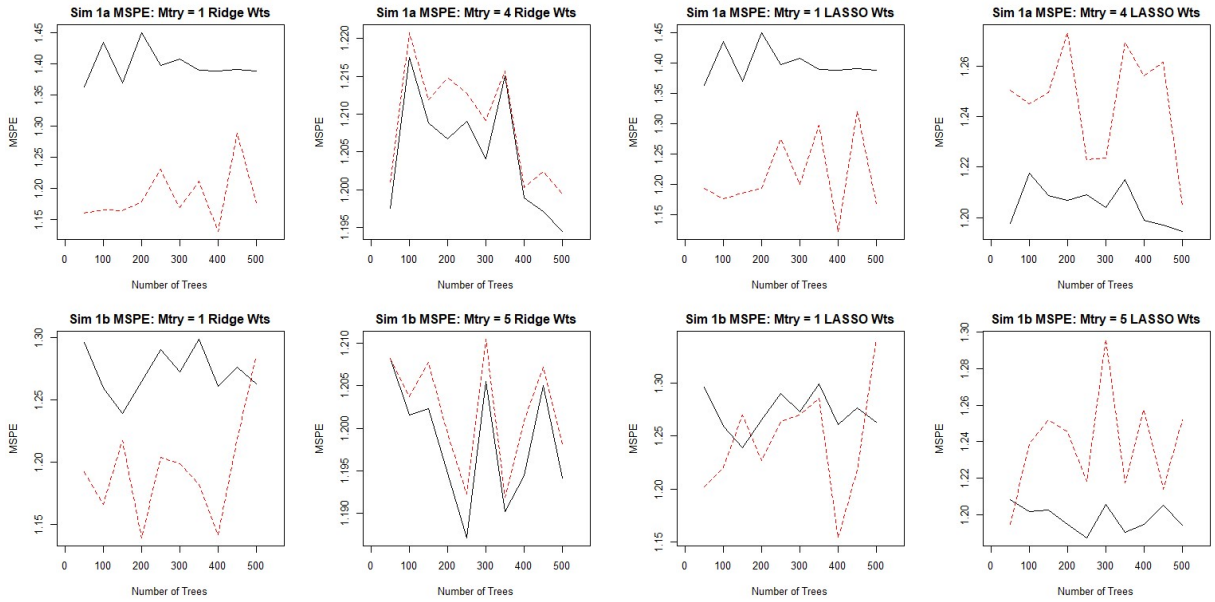
Simulation 1: Additive Model with Linear Basis Functions

The reponse variable is generated by $Y = X_1 + X_4 + \epsilon$, where $\epsilon \sim N(0, 1)$. Our MSPE results are shown below separately where Sim 1a excludes the correlated X_5 in the data set, and Sim 1b includes X_5:

Algorithm 1: OOB Weighted Random Forest



Algorithm 2: LASSO Weighted Random Forest



Variable Frequency in Simulation 1a

	LASSO.Splits.Rel	LASSO.Trees.Used	RF.Splits.Rel	RF.Trees.Used	X..Chg.in.Rel.Splits
x1	0.2983174	87	0.3048271	487	-2.1355207
x2	0.1091405	88	0.0969213	500	12.6073349
x3	0.2907685	88	0.2962480	491	-1.8496327
x4	0.3017735	87	0.3020036	485	-0.0761747

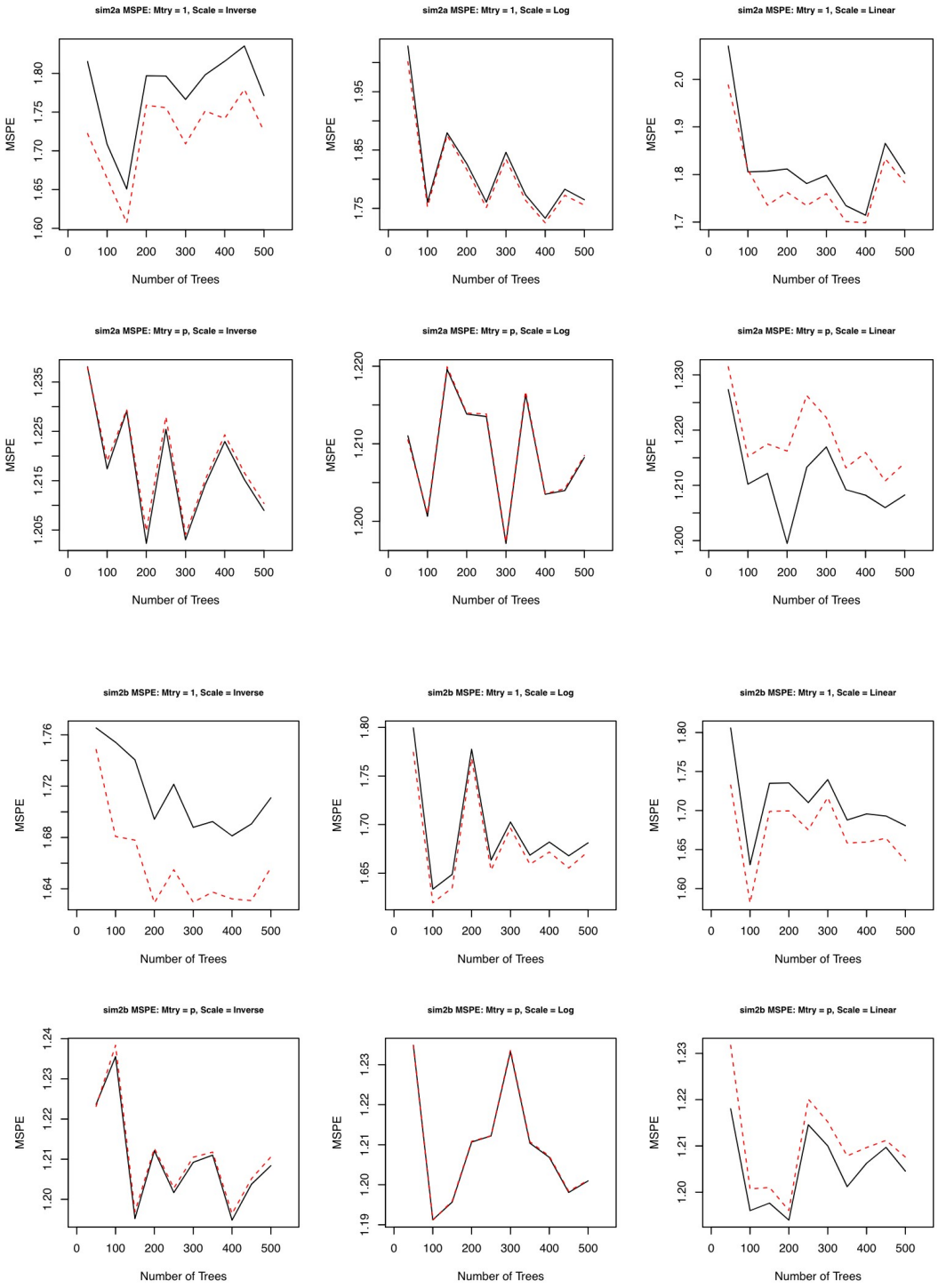
Variable Frequency in Simulation 1b

	LASSO.Splits.Rel	LASSO.Trees.Used	RF.Splits.Rel	RF.Trees.Used	X..Chg.in.Rel.Splits
x1	0.2251594	102	0.2331991	491	-3.4475531
x2	0.0936049	107	0.0837436	500	11.7756345
x3	0.2343218	102	0.2278287	493	2.8499703
x4	0.2231784	102	0.2240807	488	-0.4026885
x5	0.2237355	102	0.2311479	491	-3.2067687

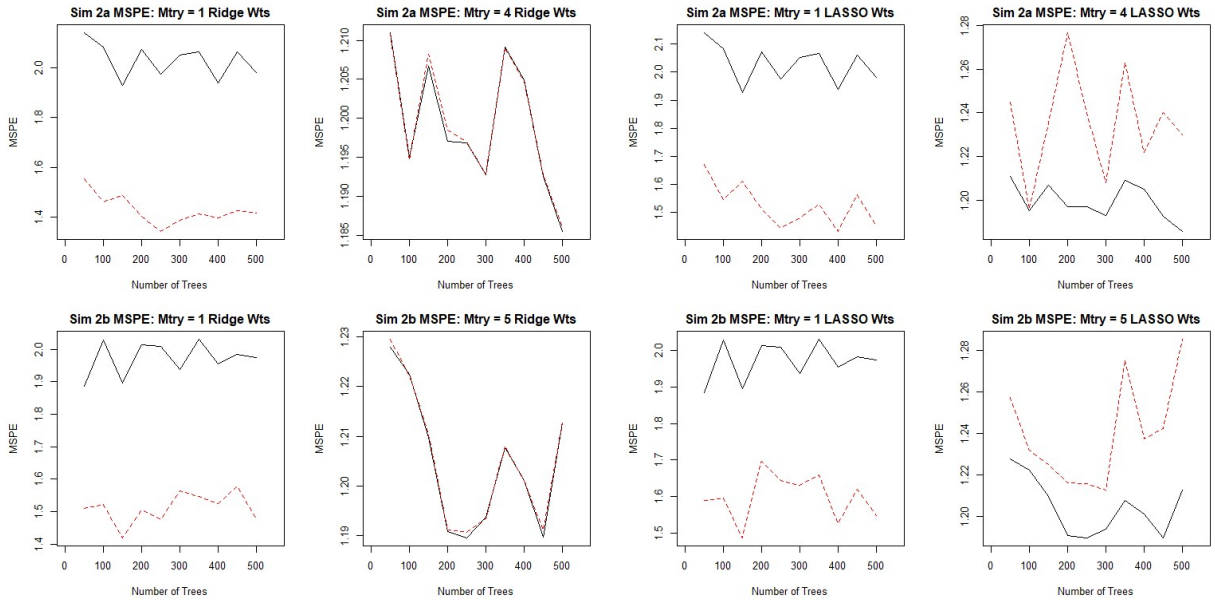
Simulation 2: Additive Model with Linear Basis Functions and Interaction

The response variable is generated by $Y = X_1 + X_4 + X_1 \times X_2 + \epsilon$ where $\epsilon \sim N(0, 1)$. Our MSPE results are shown below separately where Sim 2a excludes the correlated X_5 in the data set, and Sim 2b includes X_5:

Algorithm 1: OOB Weighted Random Forest



Algorithm 2: LASSO Weighted Random Forest



Variable Frequency in Simulation 2a

	LASSO.Splits.Rel	LASSO.Trees.Used	RF.Splits.Rel	RF.Trees.Used	X..Chg.in.Rel.Splits
x1	0.3071804	72	0.3060652	491	0.3643721
x2	0.0959536	72	0.0846607	500	13.3390315
x3	0.2999893	72	0.3028486	490	-0.9441507
x4	0.2968767	72	0.3064255	491	-3.1161862

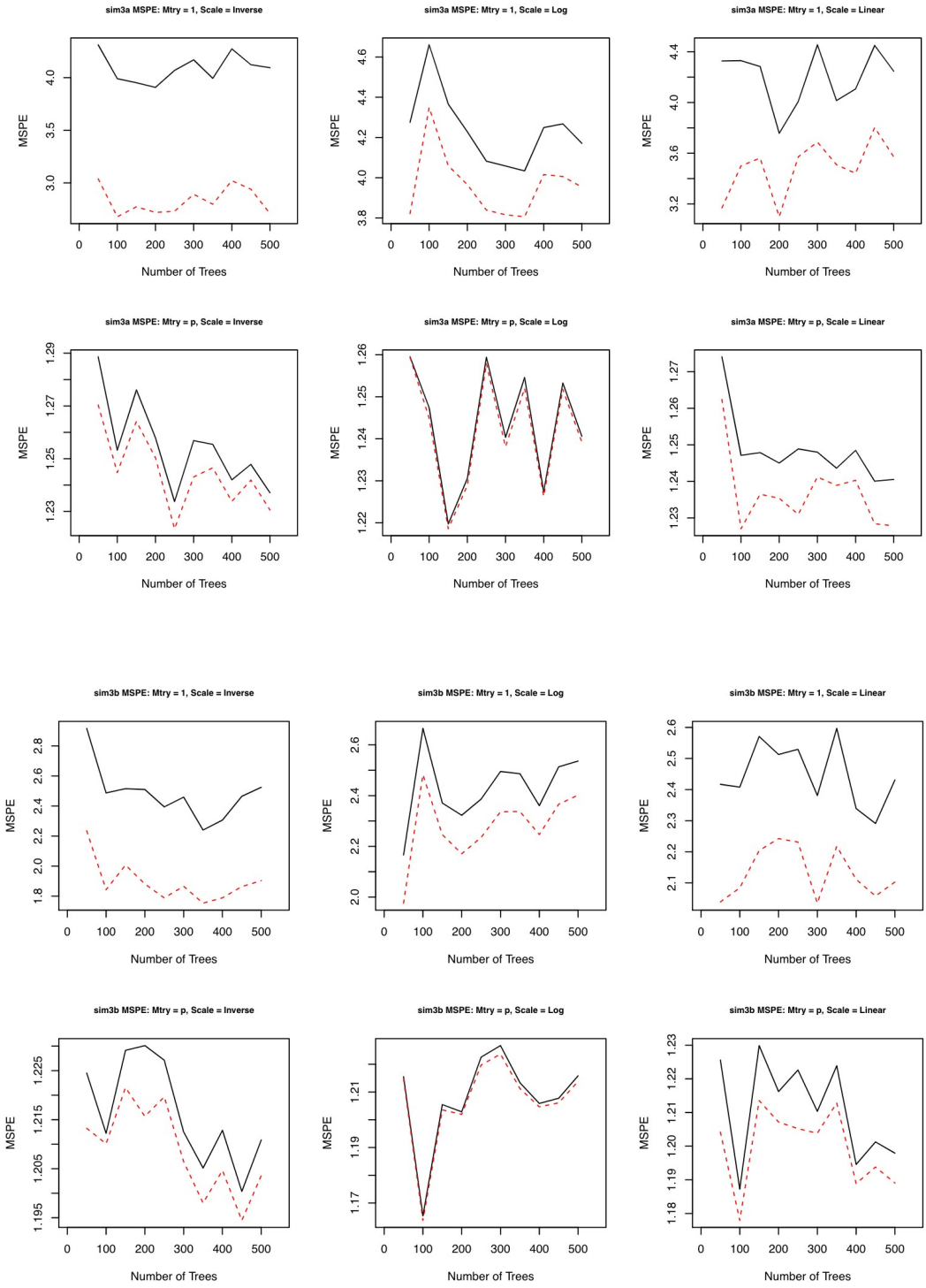
Variable Frequency in Simulation 2b

	LASSO.Splits.Rel	LASSO.Trees.Used	RF.Splits.Rel	RF.Trees.Used	X..Chg.in.Rel.Splits
x1	0.2337833	90	0.2320042	492	0.7668517
x2	0.0766873	90	0.0705764	500	8.6585481
x3	0.2278730	90	0.2289976	491	-0.4910926
x4	0.2361182	90	0.2366103	494	-0.2079881
x5	0.2255381	90	0.2318114	494	-2.7062132

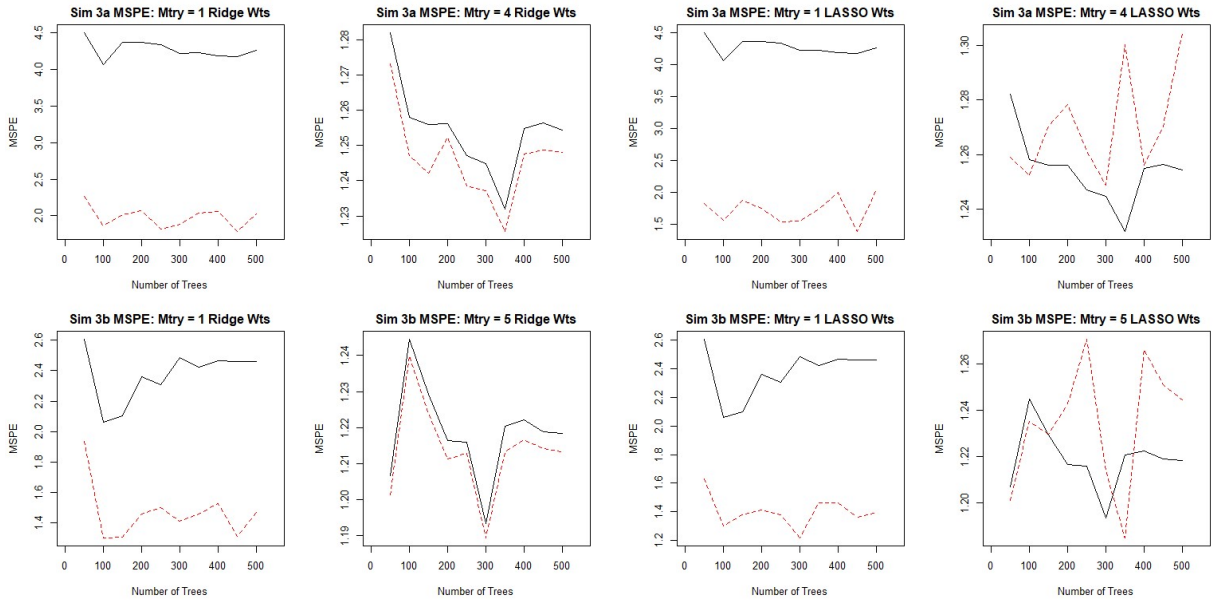
Simulation 3: Additive Model with Non-Linear Basis Functions

The response variable is generated by $Y = \sin(X_1) + X_4^2 + \epsilon$ where $\epsilon \sim N(0, 1)$. Our MSPE results are shown below separately where Sim 3a excludes the correlated X_5 in the data set, and Sim 3b includes X_5:

Algorithm 1: OOB Weighted Random Forest



Algorithm 2: LASSO Weighted Random Forest



Variable Frequency in Simulation 3a

	LASSO.Splits.Rel	LASSO.Trees.Used	RF.Splits.Rel	RF.Trees.Used	X..Chg.in.Rel.Splits
x1	0.2987098	53	0.3053795	486	-2.1840764
x2	0.1053405	53	0.0919445	500	14.5696538
x3	0.2969133	53	0.3017666	488	-1.6083141
x4	0.2990364	53	0.3009093	484	-0.6224205

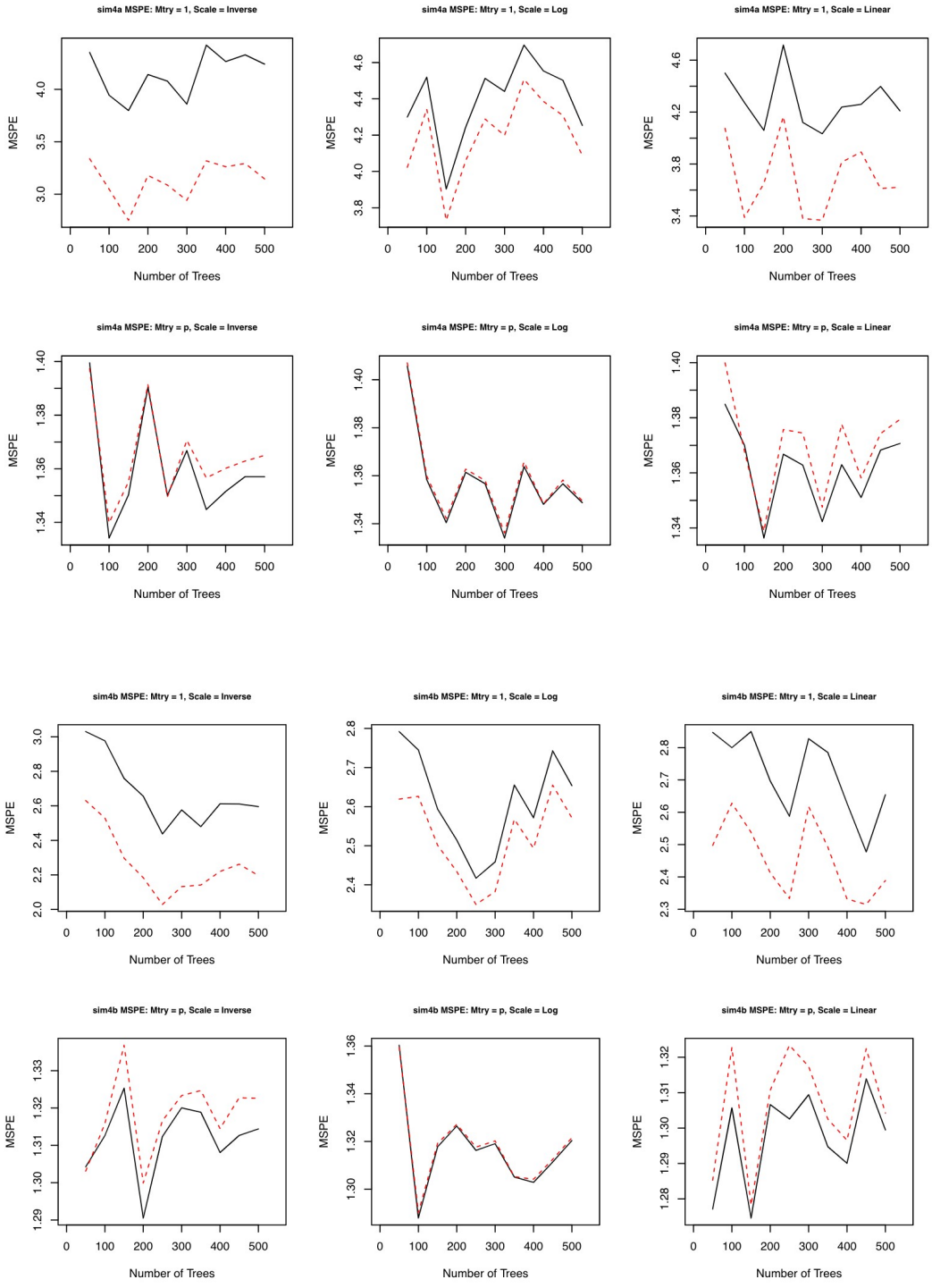
Variable Frequency in Simulation 3b

	LASSO.Splits.Rel	LASSO.Trees.Used	RF.Splits.Rel	RF.Trees.Used	X..Chg.in.Rel.Splits
x1	0.2300154	72	0.2336290	491	-1.5467148
x2	0.0900100	72	0.0757569	500	18.8142502
x3	0.2307413	72	0.2290809	490	0.7248276
x4	0.2246620	72	0.2300991	491	-2.3629381
x5	0.2245713	72	0.2314341	490	-2.9653614

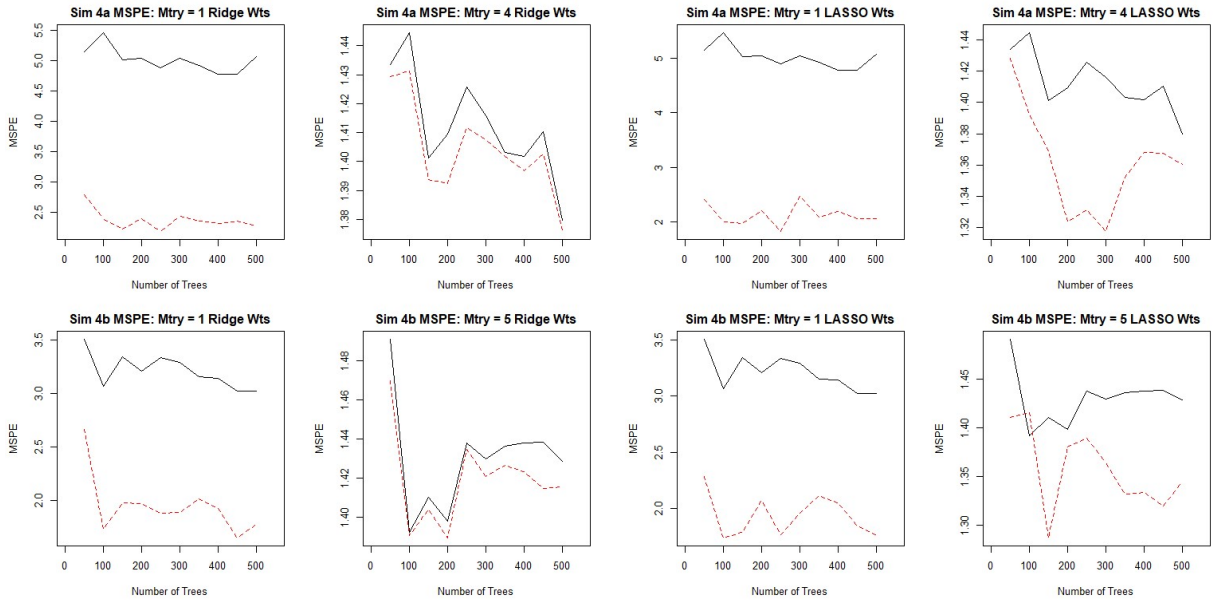
Simulation 4: Additive Model with Non-Linear Basis Functions and Interaction

The response variable is generated by $Y = \sin(X_1) + X_4^2 + X_1 \times X_2 + \epsilon$ where $\epsilon \sim N(0,1)$. Our MSPE results are shown below separately where Sim 4a excludes the correlated X_5 in the data set, and Sim 4b includes X_5:

Algorithm 1: OOB Weighted Random Forest



Algorithm 2: LASSO Weighted Random Forest



Variable Frequency in Simulation 4a

	LASSO.Splits.Rel	LASSO.Trees.Used	RF.Splits.Rel	RF.Trees.Used	X..Chg.in.Rel.Splits
x1	0.3024007	118	0.3052206	487	-0.9239155
x2	0.0860927	118	0.0848168	500	1.5043741
x3	0.3025662	118	0.3018399	485	0.2406193
x4	0.3089404	118	0.3081227	488	0.2653926

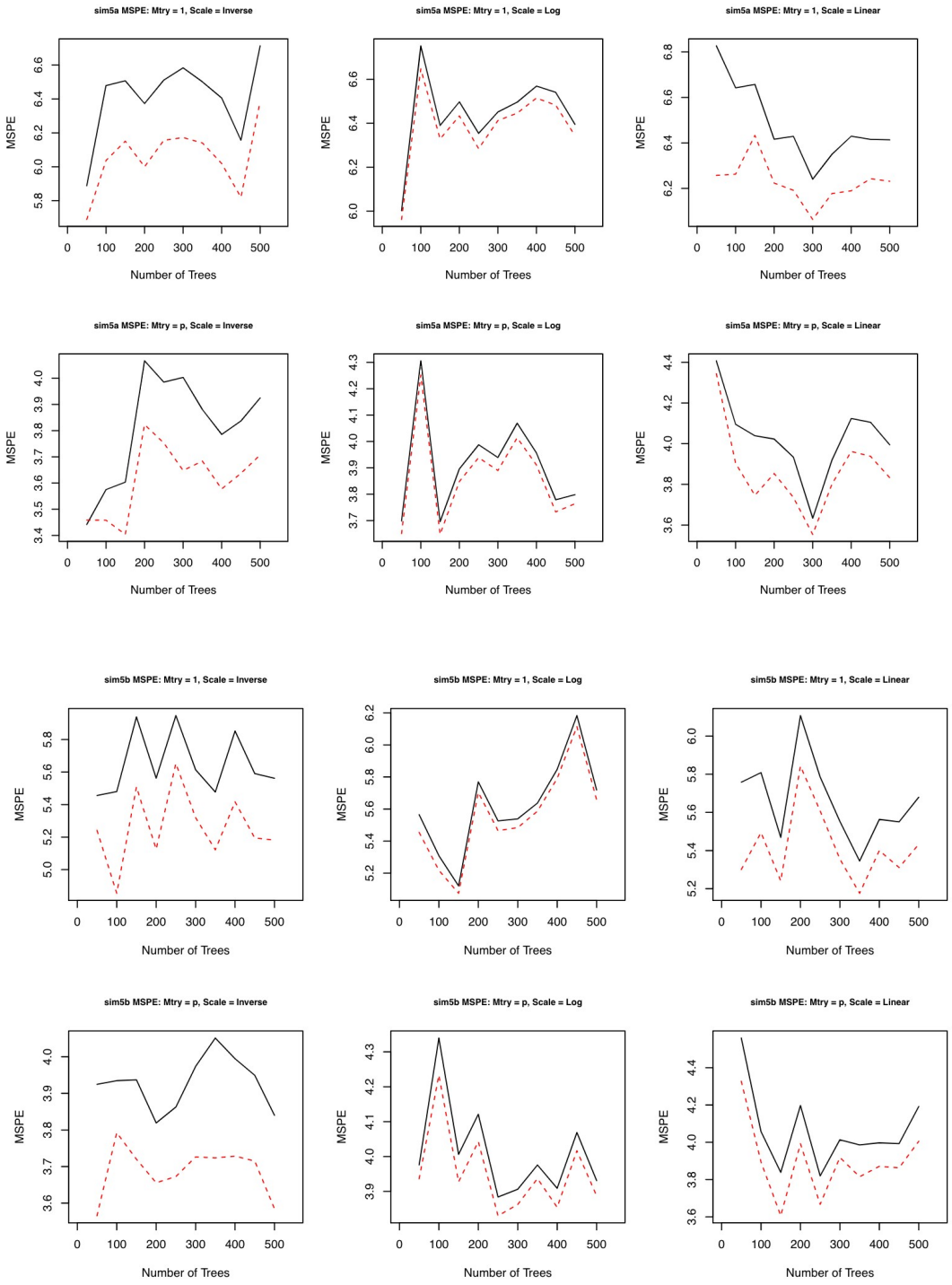
Variable Frequency in Simulation 4b

	LASSO.Splits.Rel	LASSO.Trees.Used	RF.Splits.Rel	RF.Trees.Used	X..Chg.in.Rel.Splits
x1	0.2290490	100	0.2330268	495	-1.7070123
x2	0.0719737	101	0.0687948	500	4.6208692
x3	0.2304992	100	0.2337748	496	-1.4011967
x4	0.2307281	100	0.2325428	492	-0.7803542
x5	0.2377500	99	0.2318608	493	2.5399636

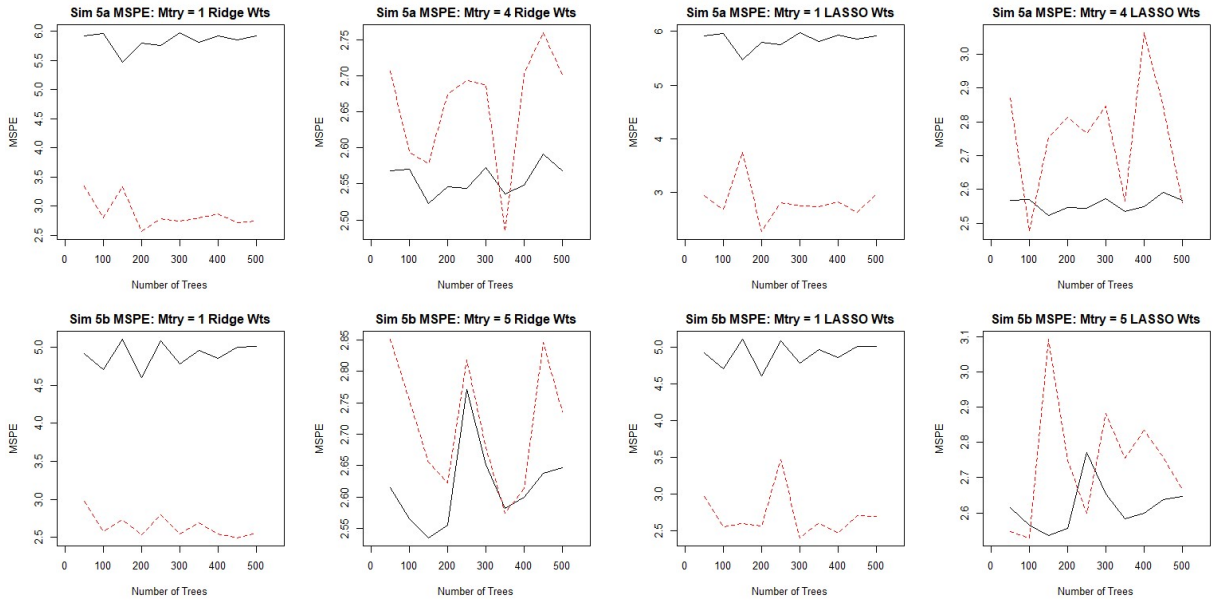
Simulation 5: Non-additive Model

The response variable is generated by $Y = \sin(X_1) \times X_4^2 + X_1 \times X_2 + \epsilon$ where $\epsilon \sim N(0, 1)$. Our MSPE results are shown below separately where Sim 5a excludes the correlated X_5 in the data set, and Sim 5b includes X_5:

Algorithm 1: OOB Weighted Random Forest



Algorithm 2: LASSO Weighted Random Forest



Variable Frequency in Simulation 5a

	LASSO.Splits.Rel	LASSO.Trees.Used	RF.Splits.Rel	RF.Trees.Used	X..Chg.in.Rel.Splits
x1	0.3078631	45	0.3089796	483	-0.3613514
x2	0.0969473	45	0.0836107	500	15.9508292
x3	0.2934320	45	0.3032653	486	-3.2424740
x4	0.3017576	45	0.3041444	487	-0.7847572

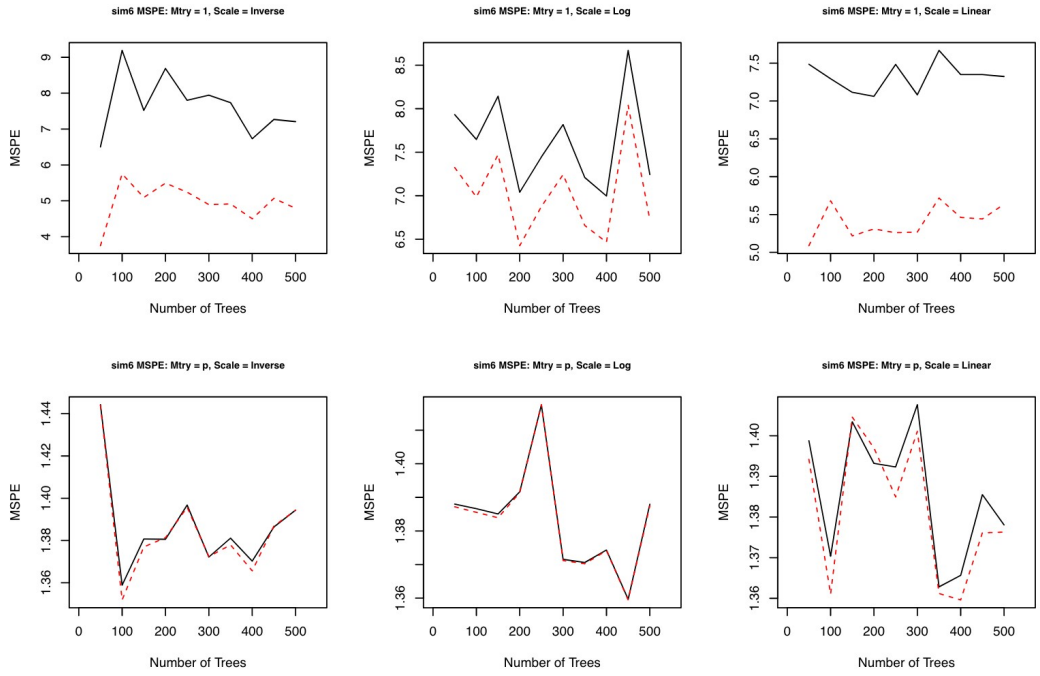
Variable Frequency in Simulation 5b

	LASSO.Splits.Rel	LASSO.Trees.Used	RF.Splits.Rel	RF.Trees.Used	X..Chg.in.Rel.Splits
x1	0.2337808	56	0.2313751	493	1.0397162
x2	0.0790455	56	0.0692415	500	14.1591385
x3	0.2257022	56	0.2329166	492	-3.0974255
x4	0.2377579	56	0.2357674	497	0.8442699
x5	0.2237136	56	0.2306994	495	-3.0280692

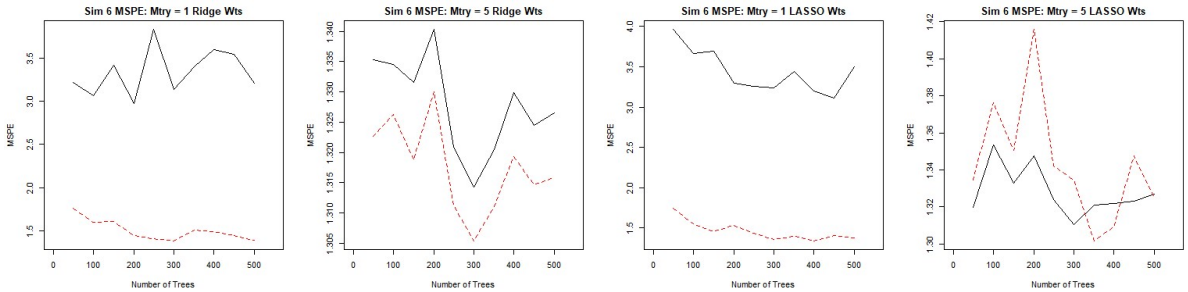
Simulation 6: Linear Model with Correlated Variable

The response variable is generated by $Y = X_1 + X_4 + X_5 + \epsilon$ where $\epsilon \sim N(0, 1)$. Note the distinguishing factor between this simulation and simulation 1 is that the response is now generated by highly-correlated variables. As a result, we do not split this simulation into two different experiments, and our results are shown below:

Algorithm 1: OOB Weighted Random Forest



Algorithm 2: LASSO Weighted Random Forest



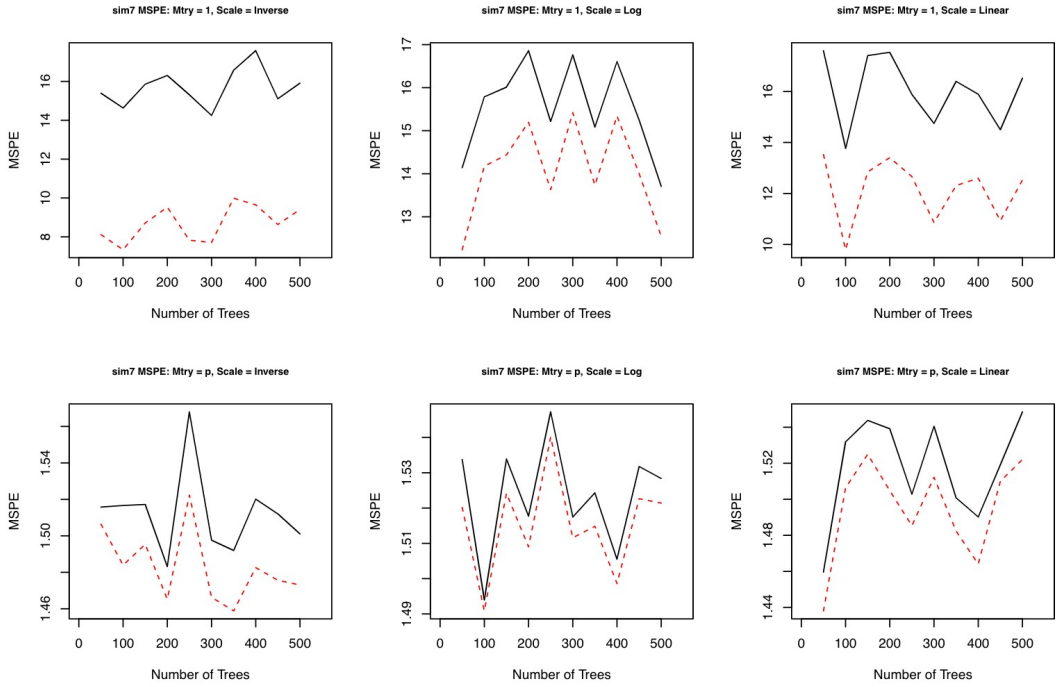
Variable Frequency in Simulation 6

	LASSO.Splits.Rel	LASSO.Trees.Used	RF.Splits.Rel	RF.Trees.Used	X..Chg.in.Rel.Splits
x1	0.2205732	83	0.2313689	496	-4.6659913
x2	0.0925795	83	0.0824613	500	12.2702631
x3	0.2259128	83	0.2287174	496	-1.2262068
x4	0.2295249	83	0.2288234	495	0.3065617
x5	0.2314095	83	0.2286290	495	1.2161614

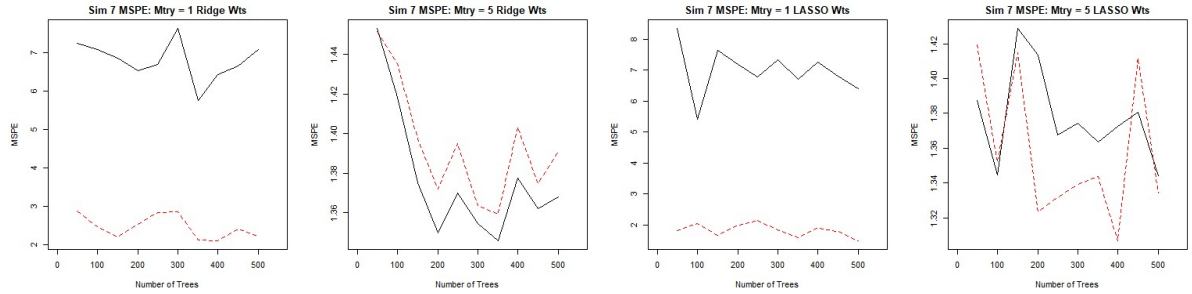
Simulation 7: Non-Linear Basis Functions with Correlated Variable

The response variable is generated by $Y = \sin(X_1) + X_4^2 + X_5 + \epsilon$ where $\epsilon \sim N(0, 1)$. Like simulation 6, this is distinguishable from simulation 3 by the highly-correlated variables used to generate the response. Our MSPE results are shown below:

Algorithm 1: OOB Weighted Random Forest



Algorithm 2: LASSO Weighted Random Forest



Variable Frequency in Simulation 7

	LASSO.Splits.Rel	LASSO.Trees.Used	RF.Splits.Rel	RF.Trees.Used	X..Chg.in.Rel.Splits
x1	0.2233467	54	0.2302091	489	-2.9809358
x2	0.0950812	54	0.0802206	500	18.5246884
x3	0.2219346	54	0.2290562	487	-3.1091107
x4	0.2331137	54	0.2315925	487	0.6568327
x5	0.2265239	54	0.2289217	488	-1.0474297

Do note that the MSPE graphs are not strictly decreasing with respect to the number of trees in the random forest. This is usually expected, for example, when we plot the OOB error of a random forest because of the nested nature of the models. However, we have chosen to construct these plots for new random forests each time, to obtain a better representation of the true variability of these models.

2.2.2 Interpretation of Results for Simulated Examples

Overall, it appears that Algorithm 1 and Algorithm 2 do an effective job in improving MSPE when `mtry` is low, but does poorly when `mtry` is high.

We believe this can be attributed to the bias-variance trade-off of our method. Our first algorithm aims to reduce bias by using optimal reweighting through by measuring proportion individual tree impact on OOB Error. Our second algorithm aims to reduce bias by using LASSO or ridge regression to reduce the contribution of poorly-performing/biased trees on future predictions. However, the variability of the model may increase with the introduction of these non-deterministic weights. In the case when `mtry` $= p$, or equivalently, when bagging is used, each tree in the random forest is likely to be unbiased. This is because all variables available in the data set are candidate splitting variables, and as a result, splits chosen within each tree will be optimal.

For Algorithm 1, in cases where `mtry` is lower than p, ie. either the default value or 1, the inverse scaling transform performs the best in most cases, althought there are instances when linear transformations also perform well. The reason for the improvement is proportional bias reduction for poorly-performing/biased trees. The log scale transform tends to perform the worst, primarily due to the fact this is the penalty of the highest magnitude. We believe that this penalty may be too extreme, and tends to reward trees which got lucky in minimizing OOB error. Therefore, the magnitude of the log penalty may incorporate bias in the weighted random forest structure by disrupting the ensemble learning advantages that random forests already possess and not completely eliminating poorly performing trees.

We also note that there is no clear difference in the performance in Algorithm 2 when LASSO or ridge regression is used to select the weights for each tree. Moreover, it appears that the number of trees used in the random forest does not significantly impact the relative improvement of our method to random forest. The relative improvement does not appear to change with the number of trees, as evidenced by the fact that most MSPE curves are parallel for the random forest and our proposed methodology.

One last item to note is the result of variable split frequency. In general, the changes in variable frequency when applying Algorithm 2 with LASSO did not match the variables which were used to generate our simulated data. This is especially present in cases where we included a multicollinear set of features. It is more likely that these tables captured the overall shape of the relationship between each feature and the response. In order to improve this heuristic, percentage contributions to the incremental changes in RSS or OOB error may be of interest. However, it is interesting to see that our weighted random forests under Algorithm 2 with LASSO tend to select a small subset ($\approx 20\%$) of the trees in the random forest each time. Despite the small number of trees employed in generating predictions, this method was able to improve MSPE relative to random forest in most cases, and it may be of future interest to use these results when examining sampling protocols within random forest.

2.2.3 Real-World Examples

In this section, we explore the results of Algorithm 1, also referred to as the OOB weighted random forest and Algorithm 2, also referred to as the LASSO weighted random forest, on several real-world data sets. A brief explanation of each data set will be provided below, but details can be found within Appendix D. Here, we will focus on primarily four results:

1) For Algorithm 1, we are curious whether the choice of linear, logarithmic or inverse scaling has a significant impact on the predictive power. For Algorithm 2, we are curious whether the choice of LASSO or ridge regression has a significant impact on the predictive power.

2) How Algorithm 1 and Algorithm 2 affects predictions for various values of `ntree`, the number of trees used in a random forest
3) How Algorithm 1 and Algorithm 2 is affected by the choice of `mtry`, the number of candidate variables sampled for each tree in a random forest.
4) For Algorithm 2, we are curious whether the choice of `maxnodes` affects the prediction error against the baseline random forest model.

Note that these explorations are similar to the results we examined with the simulated data in section 2.3. However, in the case of Algorithm 2, we are no longer interested in using relative frequency as an heuristic for variable importance, as this was shown to be ineffective with the simulated data. Instead, we examine the choice of `maxnodes` on predictions for detailed insights on tuning our methods.

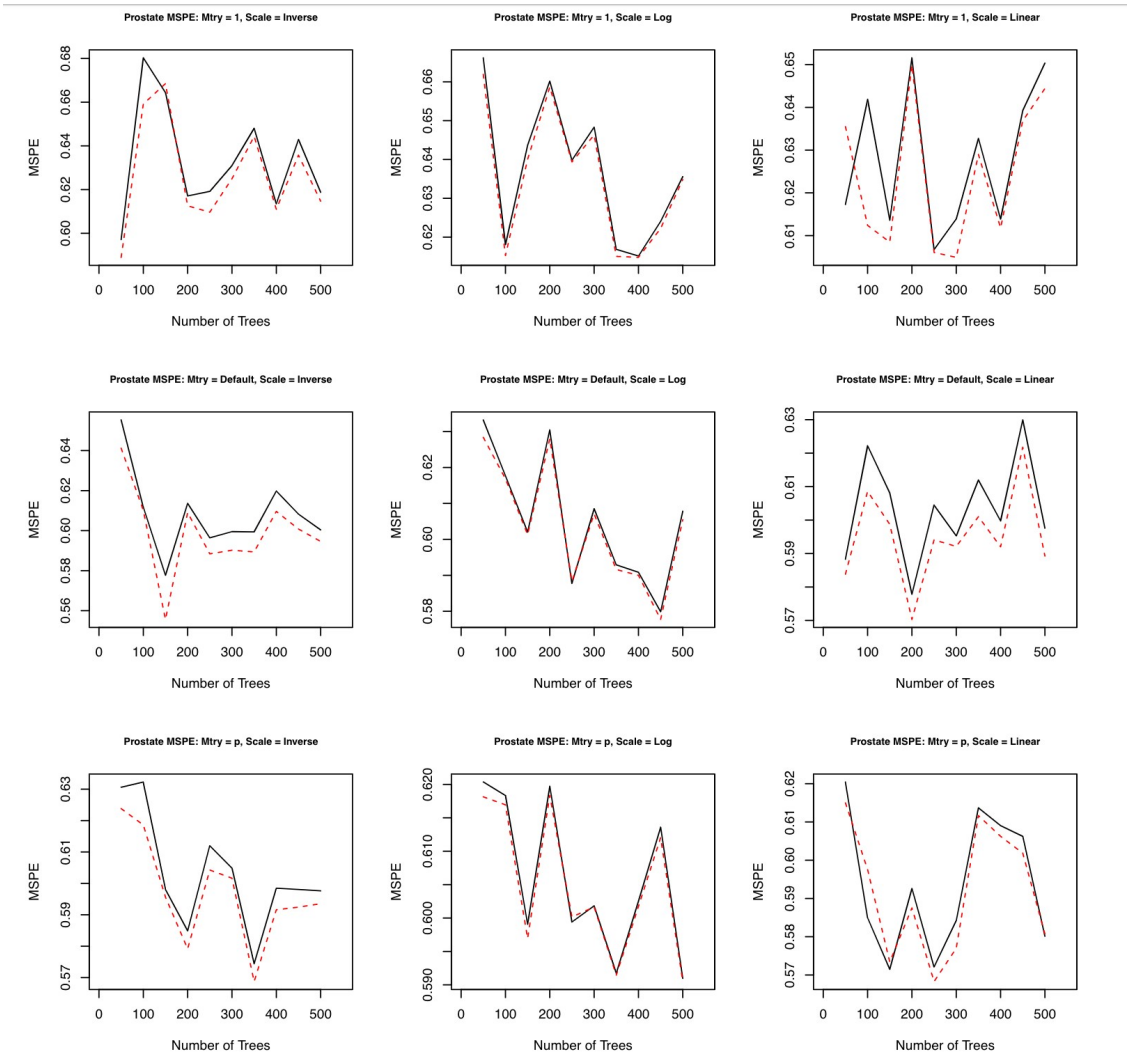
Again, we will use p to denote the number of explanatory variables in the data-set. Moreover, we will continue to compare the results separately for 1, p, and the default `mtry` value chosen by the `randomForest` function.

Below, all MSPE numbers for a random forest are drawn in solid black lines, and MSPE numbers for our LASSO weighted random forests are drawn in dashed red lines. Results begin on the following page

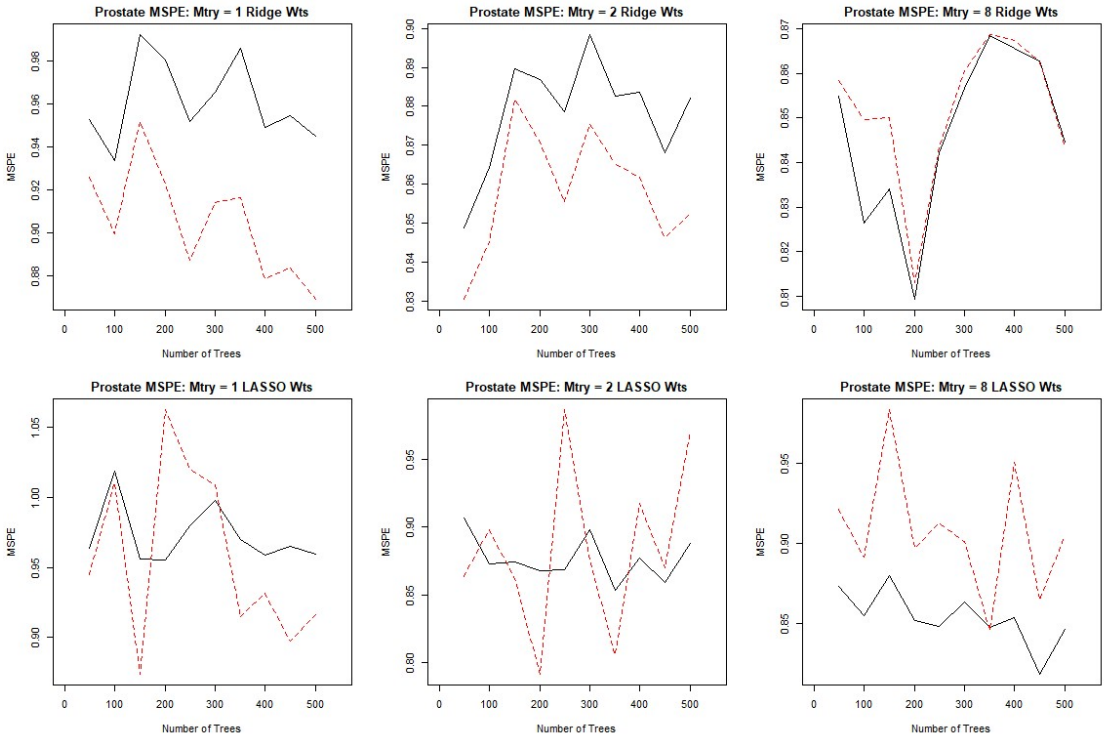
Dataset 1: Prostate

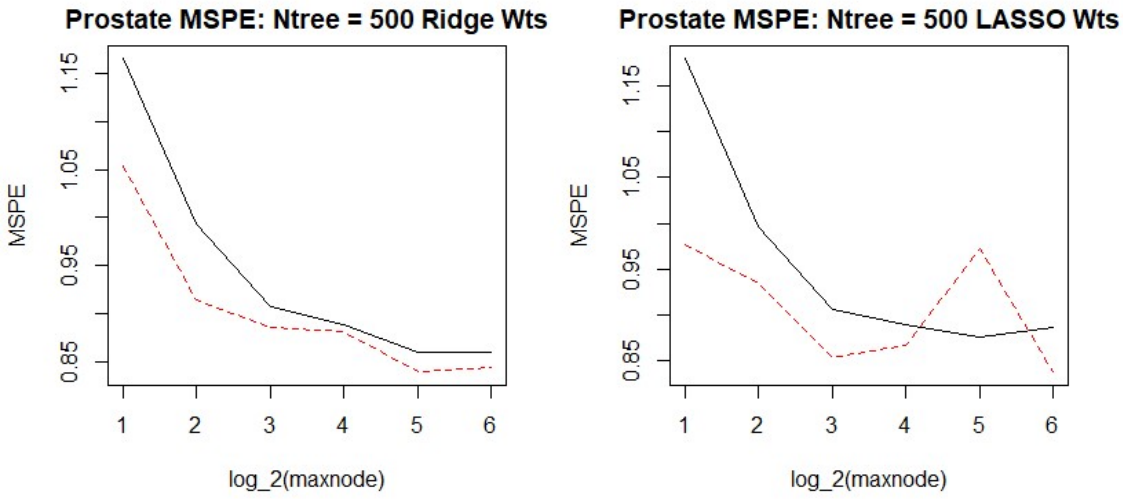
From `R`, the prostate cancer data come from a study that examined the correlation between the level of prostate specific antigen and a number of clinical measures in men who were about to receive a radical prostatectomy. It is a data frame with 97 rows and 9 columns, and the goal is to predict the variable `lpsa`. From previous investigations throughout STAT 444, the variable `lcavol` was deemed to be of importance.

Algorithm 1: OOB Weighted Random Forest



Algorithm 2: LASSO Weighted Random Forest

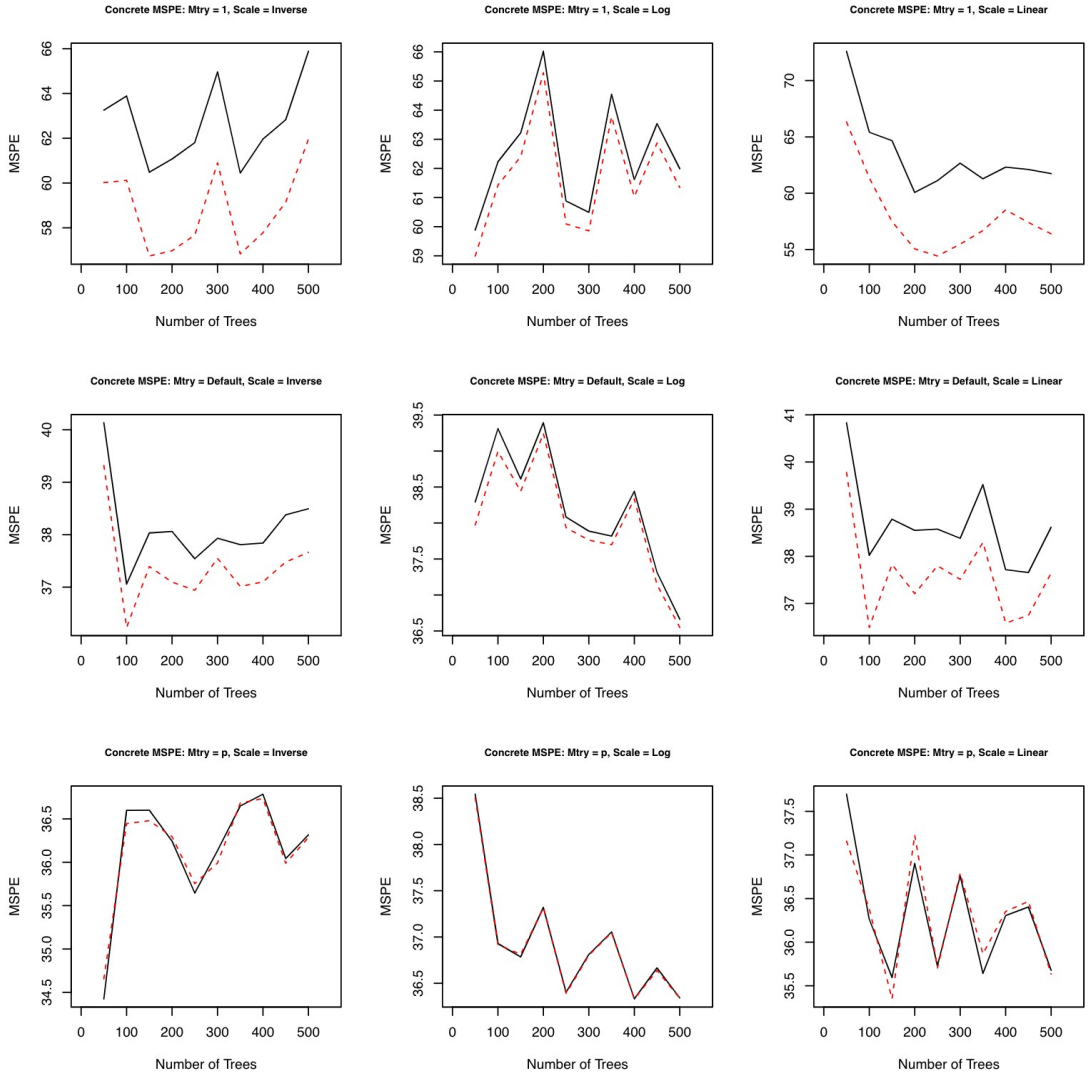




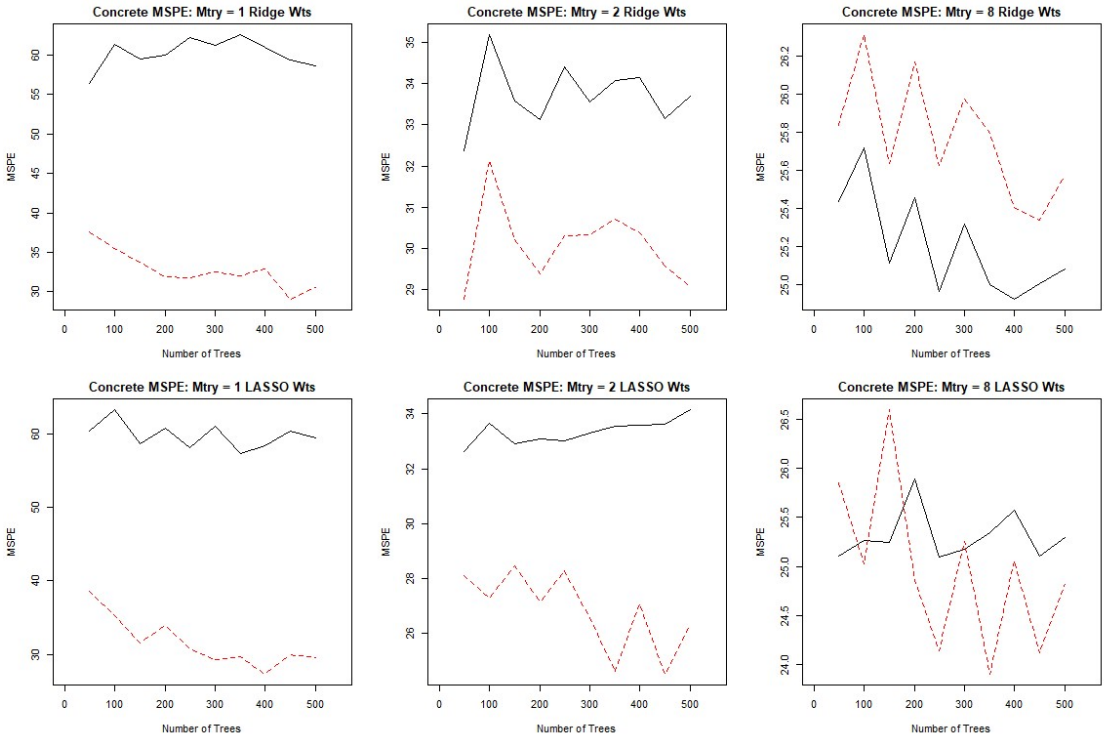
Dataset 2: Concrete

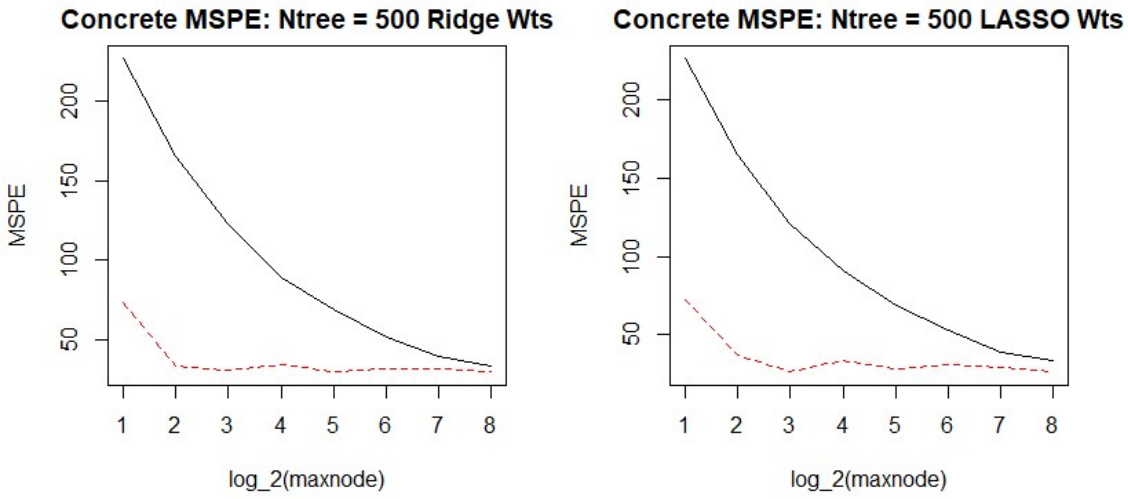
From UCI Machine Learning repository, concrete is the most important material in civil engineering. The goal is to predict concrete compressive strength, which is a highly nonlinear function of age and ingredients. This is a dataframe of 1030 rows and 9 columns.

Algorithm 1: OOB Weighted Random Forest



Algorithm 2: LASSO Weighted Random Forest

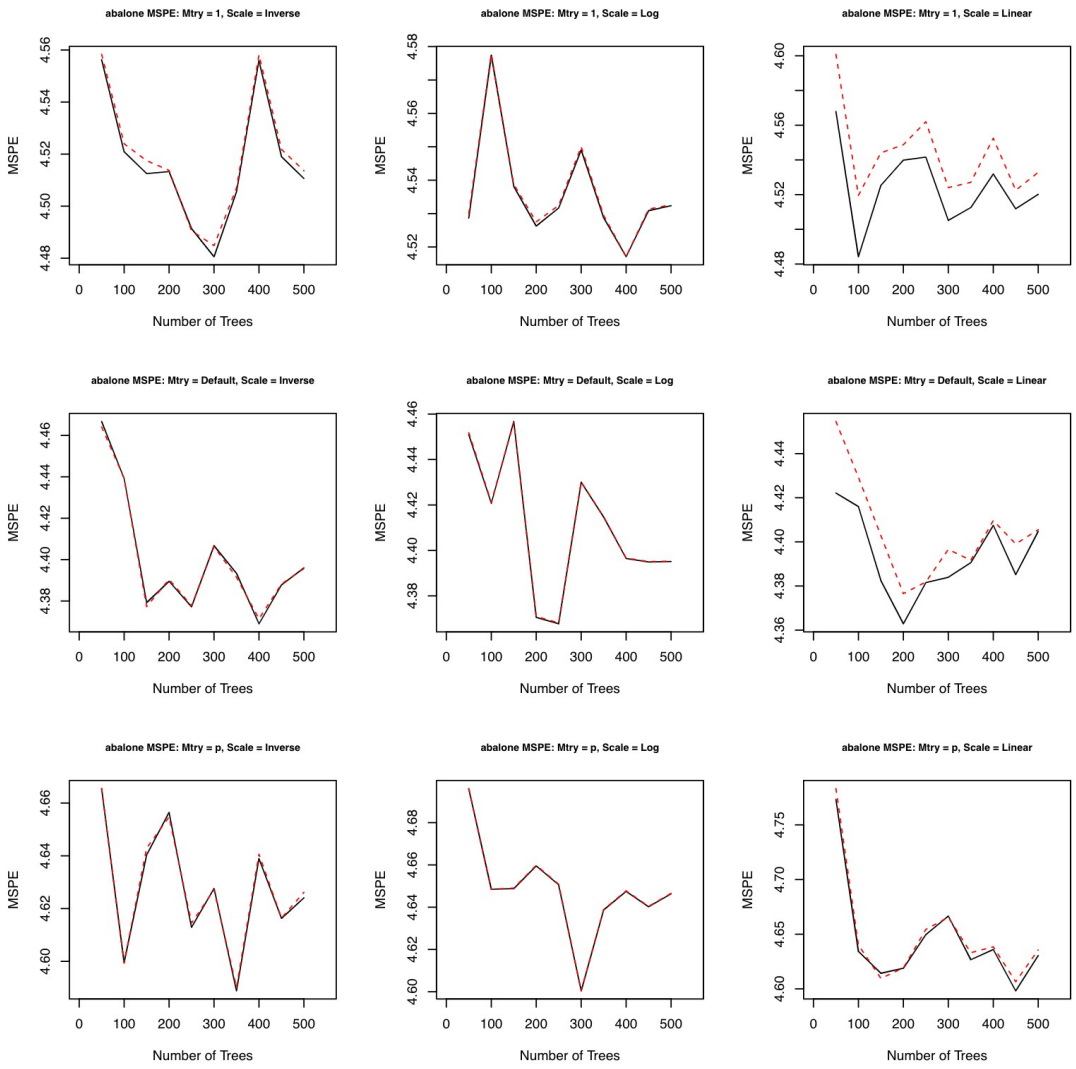




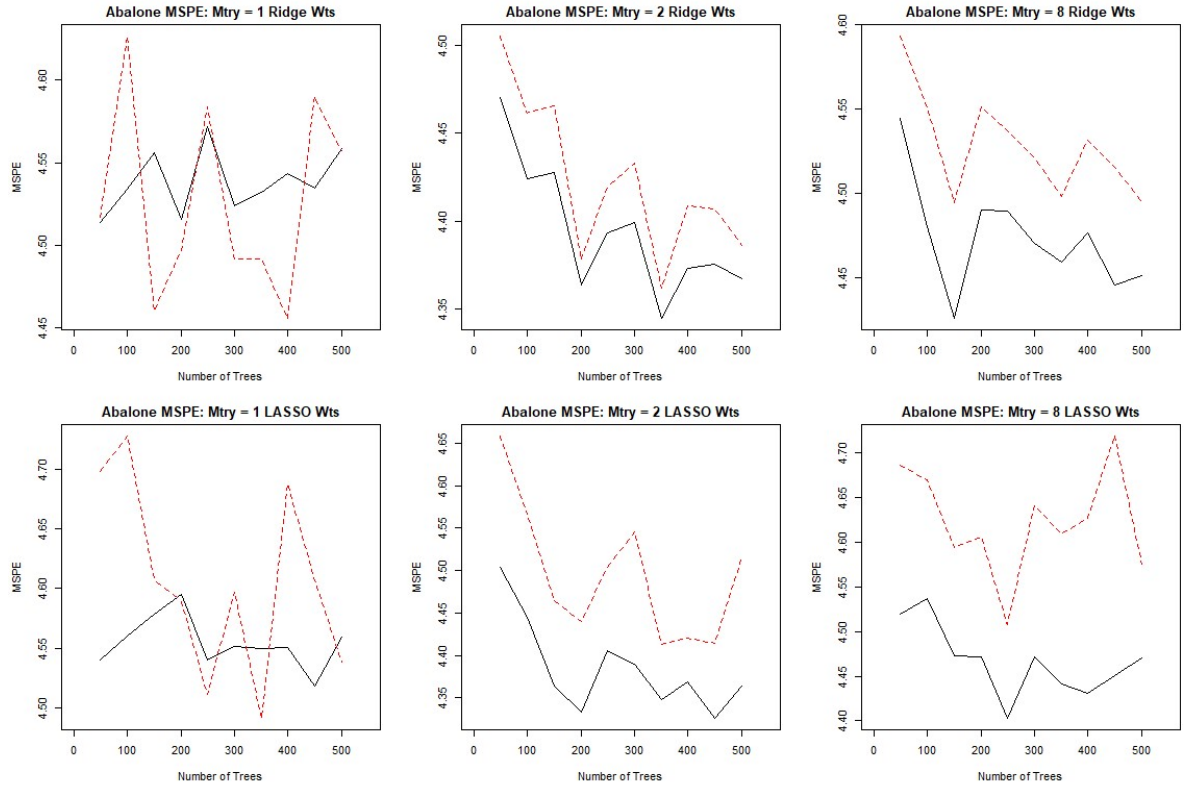
Dataset 3: Abalone

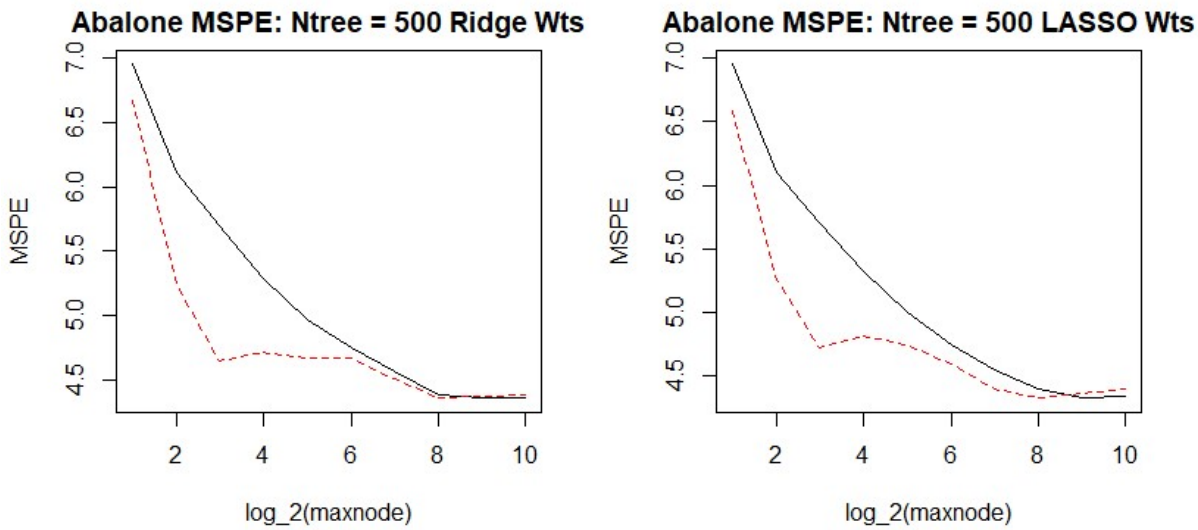
From UCI Machine Learning repository, the goal is to predict the age of various abalone, using only the physical attributes of the individual abalone temselves. This is a dataframe of 4177 rows and 8 columns.

Algorithm 1: OOB Weighted Random Forest



Algorithm 2: LASSO Weighted Random Forest





2.2.4 Interpretation of Results for Real-World Datasets

Similarly to the simulated data examples, it appears that Algorithm 1 and Algorithm 2 do an effective job in improving MSPE when `mtry` is low, but do poorly when `mtry` is high. As mentioned before, we believe this can be attributed to the bias-variance trade-off of our method. Our first algorithm aims to reduce bias by using optimal reweighting through by measuring proportion individual tree impact on OOB Error.

Our second algorithm aims to reduce bias by using LASSO or ridge regression to reduce the contribution of poorly-performing/biased trees on future predictions. However, the variability of the model may increase with the introduction of these non-deterministic weights. In the case when `mtry` $= p$, or equivalently, when bagging is used, each tree in the random forest is likely to be unbiased. This is because all variables available in the data set are candidate splitting variables, and as a result, splits chosen within each tree will be optimal.

Moreover, keeping the trend similar to the above, for Algorithm 1, in cases where `mtry` is lower than p, ie. either the default value or 1, the inverse scaling transform performs the best in most cases, althought there are instances when linear transformations also perform well. The reason for the improvement is proportional bias reduction for poorly performing and biased trees. The log scale transform tends to perform the worst, primarily due to the fact this is the penalty of the highest magnitude.

An interesting instance where our algorithms fall apart is in the Abalone dataset. Abalone is by far the largest dataset we tested our prediction on. Moreover, in preliminary investigations of the Abalone dataset, we noticed the presence of possibly multicollinear variables. According to the pairs scatterplot matrix, the variables length and diameter seem strongly positively corelated to each other. Height also seems to share a linear relationship with these variables (ie. higher height value correponds to a very high diameter and length). Moreover, the whole.weight, shucked.weight and viscera.weight variables are also strongly positively correlated. For Algorithm 1, it may be the case that our method fails to perform proportional bias reduction well for poorly performing and biased trees where multicollinearity is heavily present in the overall dataset. The same can be said for Algorithm 2, where multicollinearity is known to be a problem affecting multiple linear regression.

Unlike our simulated datasets, there are some differences in the performance in Algorithm 2 when LASSO or ridge regression is used to select the weights for each tree. This can be seen in the Prostate and Abalone datasets, where choosing weights with ridge regression performs better than choosing weights with LASSO. This may be because of the variable selection nature of the LASSO algorithm, which can introduce lots of variability into our models. We believe that this was not an issue with our simulated datasets because our simulated data sets contained several variables which were independent to the response, and hence, benefited from the variable selection. However, we cannot say the same about the Prostate and Abalone data sets. It thus appears that the sparsity of a dataset affects the performance of Algorithm 2 with LASSO vs. ridge regression.

Keeping the number of trees in the random forest fixed, we also see that changing the `maxnodes` setting does have an effect on the improvement of Algorithm 2. More specifically, when smaller pruned trees are built within the random forest, our LASSO weighted random forests tend to improve MSPE (relative to random forest) better when compared to building trees of full-size. This may be useful in the future when working with large datasets where small-trees are desired due to computational complexity. Algorithm 2 may improve predictive performance in such cases, combining many weak learners into a strong learner as a true ensemble method.

2.2.5 Comparison to Baseline Models

We build 5 baseline models using linear model, LASSO, MARS, Regression tree and XGBoost on the three real-life datasets. The Regression Tree model is pruned using both minimum cp and +1SE, and we tune the number of iterations in XGBoost. Here is a summary of MSPE using the different methods.

MSPE in Baseline Models vs. Proposed Algorithms

	Prostate	Concrete	Abalone
Linear Model	0.529	127.183	4.983
LASSO	0.717	134.023	5.081
MARS	1.430	42.405	4.618
Tree min-cp	0.630	86.506	5.716
Tree +1SE	0.633	90.117	5.939
XGBoost	0.810	16.417	3.834
OOB Weighted RF (Inverse)	0.595	37.663	4.385
glmnet Weighted RF (LASSO)	0.776	27.248	4.417

As observed, the OOB Weighted Random Forest and glmnet Weighted Random Forest algorithms fall in line with other baseline models, outperforming many of them. For the concrete dataset and the abalone dataset, OOB weighted and glmnet weighted algorithms are in the top 3 performing models alongside XGBoost. XGBoost generally performs better than the weighted random forest techniques that we presented.

3.0 Conclusions

We explored two weight assignment algorithms in this report - OOB weighted and glmnet weighted methods - in an effort to improve the predictive power of a random forest. In the OOB weighted algorithm, we transformed the OOB error using inverse, linear, and log transformations to assign weights to trees. In the glmnet algorithm, we used LASSO and ridge regression on tree weights to achieve an improved weight assignment.

We applied these algorithms to simulated datasets and real-world datasets. We examined the impact of "mry" and "ntree" on the results, as well as variable importance for the simulated data and "maxnodes" for the real-world datasets. Both algorithms demonstrated improvement in predictive power based on simulated data as well as real-world datasets, especially when "mtry" is low, due to bias-variance trade-off of our algorithms.

Both algorithms reduce bias by limiting the contribution of trees with poor predictive power. We also concluded through our investigation that "ntree" has little effect on the performance of the proposed algorithms. In the real-world datasets, we found that smaller pruned trees improve the MSPE compared to full-sized tree by tuning "maxnodes".

In the OOB algorithm, inverse scaling is the most effective while inverse of log had the worst performance, due to the extreme punishing nature of the log transformation. In the glmnet algorithm, when applied to simualted data, variables used in the algorithm did not match the ones used in the simulated model. The algorithm tends to select a smaller subset of trees to make the prediction, but it generally results in improvements. When we apply this algorithm on real-world datasets, ridge regression yields better results than LASSO, since LASSO introduces higher variability to the model due to its variable selection nature.

Overall, we believe we have introduced effective methods for improving the random forest algorithm. Further investigations may revolve around the relative variable implications of our algorithms, and sampling distributions within a random forest.

Appendix A

As suggested by Professor Peijun Sang, we also explored the possibility of shrinking coefficients at each terminal node for each tree within a random forest. Note that the predicted values generated by a tree can be written as a sum of indicator variables:

$$\hat{y}_i = \sum_{k=1}^{m} w_k I(X_i \in t_k) \qquad (A.1)$$

where w_k denotes the predicted value of the kth terminal node, and $I(X_i \in t_k) = 1$ if the ith obsevation lies within the kth terminal node of the tree and is 0 otherwise. It follows that a possible penalized objective function for fitting a tree with m nodes on n observations is:

$$\min \sum_{i=1}^{n} \left(y_i - \sum_{k=1}^{m} w_k I(X_i \in t_k) \right)^2 + \lambda \sum_{k=1}^{m} w_k^2 \qquad (A.2)$$

In general, this minimization problem is computationally infeasible due to the number of candidate trees being exponential with respect to the choices of splitting variables and variable split points. One approximation is then to minimize this objective function after choosing a specific tree structure. Since λ here is a tuning parameter, we must choose λ as well, and then minimize the function. Combining these steps together, we can build a random forest with trees minimized and tuned with respect to formula $(A.2)$ above. Below, we will refer to this as the shrunk tree method.

In `R`, we can implement this by building trees under the `randomForest` function, representing each tree as a linear combination of indicator functions, and then shrinking coefficients with the `lm.ridge` function. An implementation of our functions can be found below:

```
library(randomForest)
library(glmnet)

# input:
#   tree: matrix describing a tree from the getTree() function of random forest
#   x: dataframe of explanatory variables, where each column is a feature
#         and row is a different observation
#      assumed to be in the same order as
# output:
#   new matrix where each column is used to denote indicator variables for each
#   terminal node of the tree
indicatorMatrix <- function(tree, x){

  # determine the rows in tree which correspond to terminal nodes
  termRows <- as.vector(which(tree[, "status"] == -1))
  termRowNames <- paste0("row", termRows)
  ncol <- length(termRows)

  # create frame for output matrix
  output <- matrix(0, nrow = nrow(x), ncol = ncol,
                   dimnames = list(NULL, termRowNames))

  # populate each row of the output matrix
  for(i in 1:nrow(x)){

    # variable for row within tree we are currently checking
    treerow <- 1
```

```
    # determine terminal node the ith observation belongs in
    while(tree[treerow, "status"] != -1){

      splitvar <- tree[treerow, "split var"]
      splitpt <- tree[treerow, "split point"]

      if(x[i, splitvar] <= splitpt){
        treerow <- tree[treerow, "left daughter"]
      } else {
        treerow <- tree[treerow, "right daughter"]
      }

    }

    output[i, which(termRows == treerow)] <- 1
  }

  as.matrix(output)

}

# input:
#   x: data frame of explanatory variables
#   y: data frame of response variables
#   inputs for random forest:
#       maxnodes, mtry, ntree
# output:
#   list of: random forest object, list of shrunk trees
shrinkTrees <- function(x, y, maxnodes = NULL, mtry = max(floor(ncol(x)/3), 1), ntree = 500){

  # generate randomForest with specified parameters
  rf <- randomForest(x = x, y = y, ntree = ntree, maxnodes = maxnodes,
                     mtry = mtry, keep.forest = TRUE)

  # frame for output
  output <- list("rf" = rf, "st" = vector("list", ntree))

  # shrink each tree using ridge regression
  for(i in 1:ntree){

    tree_i <- getTree(rf, i)
    x_tree_i <- indicatorMatrix(tree_i, x)

    ridge_i <- cv.glmnet(x = x_tree_i, y = y, family = "gaussian",
                         standardize = FALSE, alpha = 0)

    output[["st"]][[i]] <- ridge_i

  }

  shrinkTrees <- output
}
```

```r
# input:
#   x: matrix of predictors, assumed to be in the same order as the matrix
#       used to fit models
#   st: shrinkTrees object, output from the shrinkTrees function
#   type: one of "st" or "rf" to indicate desired method of prediction
# output:
#   vector of predictions from the shrinkTrees object
predict_shrinkTrees <- function(x, st, type = c("st", "rf")){

  match.arg(type)

  # matrix of predictions, each column represents a different
  ntree <- length(st[["st"]])
  predict_all <- matrix(0, nrow = nrow(x), ncol = ntree)

  if(type == "st"){

    for(i in 1:ntree){

      # turn x into form of indicator functions
      tree_i <- getTree(st[["rf"]], i)
      x_tree_i <- indicatorMatrix(tree_i, x)

      # store predicted results in the predict_all matrix
      predict_all[, i] <- predict(st[["st"]][[i]], newx = x_tree_i,
                                  s = st[["st"]][[i]]$lambda.1se)

    }

    predict_shrinkTrees <- apply(predict_all, MARGIN = 1, FUN = mean)

  } else {

    predict_shrinkTrees <- predict(st[["rf"]], newdata = x)

  }

}
```

This heuristic was motivated by the implementation of XGBoost, where each boosted tree is fit with a penalty term of $\frac{1}{2}\lambda\|w\|^2$. In this case, λ is a tuning parameter for the penalty, and w denotes the vector of predicted values within each terminal node. While this penalty is able to significantly improve predictions and computational complexity in the XGBoost implementation, we did not find such shrinkage effective in random forest applications.

Firstly, shrinkage is applied in the XGBoost implementation to reduce convergence/overfitting in the boosting process. Shrinking the fit of each tree is more effective when the rate of each tree is a fixed learning rate ν. This is less problematic in a random forest, where the weight of each tree becomes smaller as more trees are added. In general, our discussions in lecture have noted that pruning trees in a random forest rarely yields significant benefits. Moreover, it is likely that the bias-variance trade-off introduced through our shrinkage is minimal. A large source of the variability in regression trees is introduced through the tree structure. Our method of shrinking terminal coefficients after fixing the tree structure does not address this source of variability.

Below is a numerical example on simulation 3 of the report, where $Y = \sin(X_1) + X_4^2 + \epsilon, \epsilon \sim N(0, 1)$.

```r
set.seed(1234)
x1 <- rnorm(1000,5,1)
x2 <- sample(c(0,1),replace = TRUE, size = 1000)
x3 <- runif(1000,0,10)
x4 <- rexp(1000,1)
trainRows <- runif(length(x1), 0, 1) < 0.75

x <- cbind(x1, x2, x3, x4)
x_train <- x[trainRows, ]
x_test <- x[!trainRows, ]

set.seed(123)
y3 <- sin(x1)+x4^2+rnorm(1000,0,1)
y3_train <- y3[trainRows]
y3_test <- y3[!trainRows]

set.seed(444)
# fit model
sim3_st <- shrinkTrees(x = x_train, y = y3_train)

# generate predictions
sim3_rf_pred <- predict_shrinkTrees(x_test, sim3_st, type = "rf")
sim3_st_pred <- predict_shrinkTrees(x_test, sim3_st, type = "st")

# calculate prediction error
c(mean((y3_test - sim3_rf_pred)^2), mean((y3_test - sim3_st_pred)^2))
```

```
## [1] 4.002227 9.181195
```

As seen above, the test set error for the shrunk tree is 9.181, more than double the error generated by the random forest. Such a discrepancy in the MSPE was not seen in our previous methods, even in the worst performing cases of our simulation. Hence, we decided to not proceed with this algorithm.

Appendix B

Appendix B.1

Below, we provide the R implementation for Algorithm 2.1, the OOB weighted random forest.

```
library(randomForest)
# input:
#   data: training data frame including explanatory variables and response
#   ntree: number of trees for the randomForest function
#   mtry: number of parameters to sample from for the randomForest function
#   type: one of "inverse", "log", "sqrt" to determine the transformation applied
#       on OOB errors
# output:
#   list of length 2, first is a random forest object, second is the OOB weights
rfReweighted <- function(x, y, ntree, mtry, type = c("inverse", "log", "linear")){

  OOB.Error <- c() # vector used to store OOB error
  rf <- randomForest(x = x, y = y, ntree=ntree, mtry=mtry,
                     keep.inbag=TRUE, keep.forest=TRUE)
  rf.pred <- predict(rf, newdata=x, predict.all=TRUE)

  # populate the OOB.Error vector
  for (i in 1:ntree){
    # determine which observations are OOB
    tree.usage <- rf$inbag[,i]
    indices <- which(tree.usage==0, arr.ind=T)
    data.response <- y[indices]
    # filter OOB observations
    data.pred <- rf.pred$individual[indices ,i]
    OOB.Error[i] <- mean((data.response-data.pred)^2)
  }

  # scale OOB error based on provided type
  rescale.OOB.Error <- OOB.Error
  if (type=="inverse"){

    rescale.OOB.Error <- 1/OOB.Error
    rescale.OOB.Error <- rescale.OOB.Error/sum(rescale.OOB.Error)

  }
  else if (type=="log"){

    rescale.OOB.Error <- OOB.Error/sum(OOB.Error)
    rescale.OOB.Error <- -log(rescale.OOB.Error)
    rescale.OOB.Error <- rescale.OOB.Error/sum(rescale.OOB.Error)

  }

  else if (type=="linear"){

    rescale.OOB.Error <- max(OOB.Error) - OOB.Error
    rescale.OOB.Error <- rescale.OOB.Error/sum(rescale.OOB.Error)

  }
```

```
  rfReweighted <- list("rf" = rf, "weights" = rescale.OOB.Error)
}

# input:
#   data: data frame of data (in same order used to fit model)
#   rfw: reweighted random forest object (output from the rfReweighted function)
#   weights: boolean; TRUE used to indicate that OOB weights should be used to
#     calculate predictions
# output:
#   predictions from the Reweighted Random Forest
predict_rfReweighted <- function(data, rfw, weights){

  rf.pred <- predict(rfw[["rf"]], newdata=data, predict.all = TRUE)

  if(weights == TRUE) {

    predict_rfReweighted <- rf.pred$individual %*% rfw[["weights"]]

  } else {

    predict_rfReweighted <- rf.pred$aggregate

  }
}
```

Appendix B.2

Below, we provide the R implementation for Algorithm 2.2, the LASSO weighted random forest.

```
library(randomForest)
library(glmnet)

# input:
#   x: matrix of predictors
#   y: response vector
#   inputs for random forest:
#       maxnodes, mtry, ntree
#   inputs for glmnet:
#       alpha (=1 for LASSO, =0 for ridge)
# output:
#   list of random forest object, lasso ensembled object, random forest predictions
#   random forest predictions are required for scaling explanatory variables
#     in future predictions
lassoEnsemble <- function (x, y, maxnodes = NULL, mtry = max(floor(ncol(x)/3), 1),
                           ntree = 500, alpha = 1){

  # generate randomForest with specified parameters
  rf <- randomForest(x = x, y = y, ntree = ntree, maxnodes = maxnodes,
                     mtry = mtry, keep.forest = TRUE)

  # matrices of random forest predictions by tree
  rf_pred <- predict(rf, newdata = x, predict.all = TRUE)$individual
  rf_pred_s <- scale(rf_pred)
```

```r
  # recombine parameters of random forest predictions with LASSO
  le <- cv.glmnet(x = rf_pred_s, y = y, family = "gaussian",
                  standardize = FALSE, alpha = alpha)

  list("rf" = rf, "le" = le, "rf_pred" = rf_pred)

}

# input:
#   x: data frame of predictors
#   le: lassoEnsemble object (created via the lassoEnsemble function)
#   type: one of "le" or "rf" to indicate desired method of prediction
# output: predictions from the lassoEnsemble object
predict_lassoEnsemble <- function(x, le, type = c("le", "rf")){

  match.arg(type)

  rf_newpred <- predict(le[["rf"]], newdata = x, predict.all = TRUE)

  if(type == "le"){

    rf_x <- rf_newpred$individual
    rf_x_s <- scale(rf_x,
                    center = apply(le[["rf_pred"]], MARGIN = 2, FUN = mean),
                    scale = apply(le[["rf_pred"]], MARGIN = 2, FUN = sd))

    predict_lassoEnsemble <- predict(le[["le"]], newx = rf_x_s, s = le[["le"]]$lambda.1se)


  } else {

    predict_lassoEnsemble <- rf_newpred$aggregate

  }
}
```

Appendix C

Appendix C.1

Below are various functions which were used to generate plots within our report. Note that these functions make use of the functions introduced within Appendix B.

```
# input:
#   x_train, y_train, x_test, y_test: matrices to represent training and test sets
#   alpha = 1 for LASSO, alpha = 0 for ridge (elastic net tuning parameter)
# output:
#   list containing errors for random forest and the lasso ensemble method, as well as
#   "all_mtry" which gives the list of mtry values used
MSPE.compare.mtry.le <- function(x_train, y_train, x_test, y_test, alpha = c(1, 0)){

  all_mtry <- c(1, floor(ncol(x_train)/3), ncol(x_train))

  for(mtry in 1:3){

    rf_error <- numeric(10)
    le_error <- numeric(10)
    i <- 0 # counter for loop

    for(ntree in seq(50, 500, 50)){

      i <- i + 1

      # fit lassoEnsemble with the specified mtry and ntree
      le_loop <- lassoEnsemble(x = x_train, y = y_train,
                               mtry = all_mtry[mtry], ntree = ntree, alpha = alpha)

      # random forest errors
      rf_pred <- predict_lassoEnsemble(x = x_test, le = le_loop, type = "rf")
      rf_error[i] <- mean((y_test - rf_pred)^2)

      # lassoEnsemble errors
      le_pred <- predict_lassoEnsemble(x = x_test, le = le_loop, type = "le")
      le_error[i] <- mean((y_test - le_pred)^2)

    }

    assign(paste0("rf_error_", mtry), rf_error)
    assign(paste0("le_error_", mtry), le_error)

  }

  list("rf_error_1" = rf_error_1, "rf_error_2" = rf_error_2, "rf_error_3" = rf_error_3,
       "le_error_1" = le_error_1, "le_error_2" = le_error_2, "le_error_3" = le_error_3,
       "all_mtry" = all_mtry, "alpha" = alpha)
}



# input:
#   compare_plot: output from the MSPE.compare.mtry.le function
```

```
#   mtry: vector of a subset of c(1,2,3) to determine which plots to generate
#   data_name: name of a dataset
# output:
#   plots detailing the differences between random forest and lasso ensemble MSPE
#     for different mtry and ntree combinations
make.MSPE.mtry.plot <- function(compare_plot, mtry=c(1,2,3), data_name){

  for (i in mtry){

    plot(x = seq(50, 500, 50), y = compare_plot[[paste0("rf_error_", i)]],
         xlim = c(0, 550),
         ylim = c(min(compare_plot[[paste0("rf_error_", i)]],
                      compare_plot[[paste0("le_error_", i)]]),
                  max(compare_plot[[paste0("rf_error_", i)]],
                      compare_plot[[paste0("le_error_", i)]])),
         main = paste0(data_name, " MSPE: Mtry = ",
                       compare_plot[["all_mtry"]][i],
                       ifelse(compare_plot["alpha"] == 1, " LASSO Wts", " Ridge Wts")),
         xlab = "Number of Trees", ylab = "MSPE", type = "l")

    lines(x = seq(50, 500, 50), y = compare_plot[[paste0("le_error_", i)]],
          lty = 2, col = "red")
  }
}



# input:
#   x_train, y_train, x_test, y_test: matrices to represent training and test sets
#   alpha = 1 for LASSO, alpha = 0 for ridge (elastic net tuning parameter)
#   maxlayer: maximum number of layers for each tree (maxnode = 2^maxlayer)
# output:
#   list comparing random forest and lasso ensemble errors for various values of maxnode
MSPE.compare.maxnode.le <- function(x_train, y_train, x_test, y_test,
                                    alpha = c(0,1), maxlayer){

  all_maxnode <- 2^(1:maxlayer)
  rf_error <- numeric(maxlayer)
  le_error <- numeric(maxlayer)

  for (i in 1:maxlayer){

    # fit lassoEnsemble with the specified mtry and ntree
    le_loop <- lassoEnsemble(x = x_train, y = y_train, maxnode = all_maxnode[i],
                             ntree = 500, alpha = alpha)

    # random forest errors
    rf_pred <- predict_lassoEnsemble(x = x_test, le = le_loop, type = "rf")
    rf_error[i] <- mean((y_test - rf_pred)^2)

    # lassoEnsemble errors
    le_pred <- predict_lassoEnsemble(x = x_test, le = le_loop, type = "le")
    le_error[i] <- mean((y_test - le_pred)^2)
```

```
  }

  list("rf_error" = rf_error, "le_error" = le_error,
       "alpha" = alpha, "maxlayer" = maxlayer)
}



# input:
#   compare_plot: output from the MSPE.compare.maxnode.le function
#   data_name: name of a dataset
# output:
#   plots detailing the differences between random forest and lasso ensemble MSPE
#     across different maxnode values
make.MSPE.maxnode.plot <- function(compare_plot, data_name){

  maxlayer <- compare_plot[["maxlayer"]]

  plot(x = 1:maxlayer, y = compare_plot[["rf_error"]],
       ylim = c(min(compare_plot[["rf_error"]], compare_plot[["le_error"]]),
                max(compare_plot[["rf_error"]], compare_plot[["le_error"]])),
       main = paste0(data_name, " MSPE: Ntree = 500 ",
                     ifelse(compare_plot[["alpha"]] == 1, "LASSO Wts", "Ridge Wts")),
       xlab = "log_2(maxnode)", ylab = "MSPE", type = "l")

  lines(x = 1:maxlayer, y = compare_plot[["le_error"]],
        lty = 2, col = "red")
}
```

Appendix C.2

This section provides code for our visualizations of results on the simulated datasets.

Appendix C.2.1

Below is our R implementation of our 7 simulated examples, where simulation 1 is introduced in section 1 of the report, and other simulations are introduced throughout section 2.

```
# explanatory variables
set.seed(1234)
x1 <- rnorm(1000,5,1)
x2 <- sample(c(0,1),replace = TRUE, size = 1000)
x3 <- runif(1000,0,10)
x4 <- rexp(1000,1)
x5 <- x1 + 5*x4
trainRows <- runif(length(x1), 0, 1) < 0.75

x.a <- cbind(x1, x2, x3, x4)
x.a.train <- x.a[trainRows, ]
x.a.test <- x.a[!trainRows, ]

x.b <- cbind(x1, x2, x3, x4, x5)
x.b.train <- x.b[trainRows, ]
x.b.test <- x.b[!trainRows, ]
```

```
# simulation 1
set.seed(123)
y1 <- x1+x4+rnorm(1000,0,1)
y1.train <- y1[trainRows]
y1.test <- y1[!trainRows]

# simulation 2
set.seed(123)
y2 <- x1+x4+x1*x2+rnorm(1000,0,1)
y2.train <- y2[trainRows]
y2.test <- y2[!trainRows]

# simulation 3
set.seed(123)
y3 <- sin(x1)+x4^2+rnorm(1000,0,1)
y3.train <- y3[trainRows]
y3.test <- y3[!trainRows]

# simulation 4
set.seed(123)
y4 <- sin(x1)+x4^2+x1*x2+rnorm(1000,0,1)
y4.train <- y4[trainRows]
y4.test <- y4[!trainRows]

# simulation 5
set.seed(123)
y5 <- sin(x1)*x4^2+x1*x2+rnorm(1000,0,1)
y5.train <- y5[trainRows]
y5.test <- y5[!trainRows]

# simulation 6
set.seed(123)
y6 <- x1+x4+x5+rnorm(1000,0,1)
y6.train <- y6[trainRows]
y6.test <- y6[!trainRows]

# simulation 7
set.seed(123)
y7 <- sin(x1)+x4^2+x5+rnorm(1000,0,1)
y7.train <- y7[trainRows]
y7.test <- y7[!trainRows]
```

Appendix C.2.2

Below, we provide code used to generate plots for the performance of the OOB Weighted Random Forest method on our first simulated dataset. Plots for the other simulations were generated similarly.

```
ntrees.vect <- as.vector(seq(50, 500, by=50))

# determine errors
sim1a.matrix <- matrix(data = NA, nrow=10, ncol=1+(3*2)*3)
sim1b.matrix <- matrix(data = NA, nrow=10, ncol=1+(3*2)*3)
```

```r
set.seed(444)
# mtry = 1
for (n in ntrees.vect){
  rf.objA <- rfReweighted(x.a.train, y1.train, n, 1, "inverse")
  rf.objB <- rfReweighted(x.a.train, y1.train, n, 1, "log")
  rf.objC <- rfReweighted(x.a.train, y1.train, n, 1, "linear")
  imp.predA <- predict_rfReweighted(x.a.test, rf.objA, TRUE)
  imp.predB <- predict_rfReweighted(x.a.test, rf.objB, TRUE)
  imp.predC <- predict_rfReweighted(x.a.test, rf.objC, TRUE)
  response <- y1.test
  predA <- predict(rf.objA[["rf"]], newdata=x.a.test)
  predB <- predict(rf.objB[["rf"]], newdata=x.a.test)
  predC <- predict(rf.objC[["rf"]], newdata=x.a.test)
  sim1a.matrix[n/50,1] <- n
  sim1a.matrix[n/50,2] <- mean((predA-response)^2)
  sim1a.matrix[n/50,3] <- mean((imp.predA-response)^2)
  sim1a.matrix[n/50,4] <- mean((predB-response)^2)
  sim1a.matrix[n/50,5] <- mean((imp.predB-response)^2)
  sim1a.matrix[n/50,6] <- mean((predC-response)^2)
  sim1a.matrix[n/50,7] <- mean((imp.predC-response)^2)
}

# mtry = p
for (n in ntrees.vect){
  rf.objA <- rfReweighted(x.a.train,y1.train, n, ncol(x.a.train), "inverse")
  rf.objB <- rfReweighted(x.a.train,y1.train, n, ncol(x.a.train), "log")
  rf.objC <- rfReweighted(x.a.train,y1.train, n, ncol(x.a.train), "linear")
  imp.predA <- predict_rfReweighted(x.a.test, rf.objA, TRUE)
  imp.predB <- predict_rfReweighted(x.a.test, rf.objB, TRUE)
  imp.predC <- predict_rfReweighted(x.a.test, rf.objC, TRUE)
  response <- y1.test
  predA <- predict(rf.objA[["rf"]], newdata=x.a.test)
  predB <- predict(rf.objB[["rf"]], newdata=x.a.test)
  predC <- predict(rf.objC[["rf"]], newdata=x.a.test)
  sim1a.matrix[n/50,14] <- mean((predA-response)^2)
  sim1a.matrix[n/50,15] <- mean((imp.predA-response)^2)
  sim1a.matrix[n/50,16] <- mean((predB-response)^2)
  sim1a.matrix[n/50,17] <- mean((imp.predB-response)^2)
  sim1a.matrix[n/50,18] <- mean((predC-response)^2)
  sim1a.matrix[n/50,19] <- mean((imp.predC-response)^2)
}
set.seed(444)
```

```r
# generate plots
par(mfrow=c(2,3))
# mtry = 1
plot(x = seq(50, 500, 50), y = sim1a.matrix[,2],  xlim = c(0, 550),
     ylim = c(min(sim1a.matrix[,2:3]), max(sim1a.matrix[,2:3])), type = "l",
     main = "sim1a MSPE: Mtry = 1, Scale = Inverse",
     xlab = "Number of Trees", ylab = "MSPE", cex.main=0.75)
lines(x = seq(50, 500, 50), y = sim1a.matrix[,3], col = "red", lty=2)

plot(x = seq(50, 500, 50), y = sim1a.matrix[,4],  xlim = c(0, 550),
     ylim = c(min(sim1a.matrix[,4:5]), max(sim1a.matrix[,4:5])), type = "l",
```

```
     main = "sim1a MSPE: Mtry = 1, Scale = Log",
     xlab = "Number of Trees", ylab = "MSPE", cex.main=0.75)
lines(x = seq(50, 500, 50), y = sim1a.matrix[,5], col = "red", lty=2)

plot(x = seq(50, 500, 50), y = sim1a.matrix[,6],  xlim = c(0, 550),
     ylim = c(min(sim1a.matrix[,6:7]), max(sim1a.matrix[,6:7])), type = "l",
     main = "sim1a MSPE: Mtry = 1, Scale = Linear",
     xlab = "Number of Trees", ylab = "MSPE", cex.main=0.75)
lines(x = seq(50, 500, 50), y = sim1a.matrix[,7], col = "red", lty=2)

# mtry = p
plot(x = seq(50, 500, 50), y = sim1a.matrix[,14],  xlim = c(0, 550),
     ylim = c(min(sim1a.matrix[,14:15]), max(sim1a.matrix[,14:15])), type = "l",
     main = "sim1a MSPE: Mtry = p, Scale = Inverse",
     xlab = "Number of Trees", ylab = "MSPE", cex.main=0.75)
lines(x = seq(50, 500, 50), y = sim1a.matrix[,15], col = "red", lty=2)

plot(x = seq(50, 500, 50), y = sim1a.matrix[,16],  xlim = c(0, 550),
     ylim = c(min(sim1a.matrix[,16:17]), max(sim1a.matrix[,16:17])), type = "l",
     main = "sim1a MSPE: Mtry = p, Scale = Log",
     xlab = "Number of Trees", ylab = "MSPE", cex.main=0.75)
lines(x = seq(50, 500, 50), y = sim1a.matrix[,17], col = "red", lty=2)

plot(x = seq(50, 500, 50), y = sim1a.matrix[,18],  xlim = c(0, 550),
     ylim = c(min(sim1a.matrix[,18:19]), max(sim1a.matrix[,18:19])), type = "l",
     main = "sim1a MSPE: Mtry = p, Scale = Linear",
     xlab = "Number of Trees", ylab = "MSPE", cex.main=0.75)
lines(x = seq(50, 500, 50), y = sim1a.matrix[,19], col = "red", lty=2)
```

Appendix C.2.3

Below, we provide code used to generate plots for the performance of the glmnet Weighted Random Forest method on our first simulated dataset. Plots for the other simulations were generated similarly. Also note that the functions below refer to functions introduced in Appendix C.1.

```
# plots with ridge
set.seed(444)
y1.a.plot0 <- MSPE.compare.mtry.le(x_train = x.a.train, y_train = y1.train,
                                   x_test = x.a.test, y_test = y1.test,
                                   alpha = 0)
y1.b.plot0 <- MSPE.compare.mtry.le(x_train = x.b.train, y_train = y1.train,
                                  x_test = x.b.test, y_test = y1.test,
                                  alpha = 0)

par(mfrow=c(2, 2))
make.MSPE.mtry.plot(y1.a.plot0, mtry = c(1, 3), data_name = "Sim 1a")
make.MSPE.mtry.plot(y1.b.plot0, mtry = c(1, 3), data_name = "Sim 1b")

# plots with LASSO
set.seed(444)
y1.a.plot1 <- MSPE.compare.mtry.le(x_train = x.a.train, y_train = y1.train,
                                   x_test = x.a.test, y_test = y1.test,
                                   alpha = 1)
y1.b.plot1 <- MSPE.compare.mtry.le(x_train = x.b.train, y_train = y1.train,
```

```
                                    x_test = x.b.test, y_test = y1.test,
                                    alpha = 1)

par(mfrow=c(2, 2))
make.MSPE.mtry.plot(y1.a.plot1, mtry = c(1, 3), data_name = "Sim 1a")
make.MSPE.mtry.plot(y1.b.plot1, mtry = c(1, 3), data_name = "Sim 1b")
```

Appendix C.2.4

Below, we provide the function used to generate variable frequency and importance tables for the glmnet Weighted Random Forests.

```
# input:
#   xnames: vector of names of explanatory variables, assumed to be in the same order
#       as the data used to fit model
#   le: lassoEnsemble object, output from the lassoEnsemble function
# output:
#   matrix indicating frequency of each variable
getLeFrequency <- function(xnames, le){

  # determine which trees in the random forest were actually selected
  # remove first coefficient since this represents intercept of LASSO (i.e. overall avg)
  le_wts <- coef(le[["le"]])[-1]
  trees_used <- which(le_wts > 0)

  output <- matrix(0, nrow = length(xnames), ncol = 7,
                   dimnames = list(xnames, c("LASSO Num Splits", "LASSO Splits Rel",
                                             "LASSO Trees Used", "RF Num Splits",
                                             "RF Splits Rel", "RF Trees Used",
                                             "% Chg in Rel Splits")))

  for(i in 1:length(le_wts)){

    tree_i <- getTree(le[["rf"]], i)
    lassoTree <- sum(trees_used == i) > 0

    # increment Num Splits for each variable
    for (j in 1:nrow(tree_i)){

      var_ij <- tree_i[j, "split var"]

      if (var_ij != 0){
        output[var_ij, "RF Num Splits"] <- output[var_ij, "RF Num Splits"] + 1

        if (lassoTree == TRUE){
          # case when tree is also selected by the LASSO weights
          output[var_ij, "LASSO Num Splits"] <- output[var_ij, "LASSO Num Splits"] + 1
        }
      }
    }

    # increment Trees Used for variables which are used by tree i
    varsused_i <- unique(tree_i[, "split var"])
```

```
    output[varsused_i, "RF Trees Used"] <- output[varsused_i, "RF Trees Used"] + 1
    if (lassoTree == TRUE){
      output[varsused_i, "LASSO Trees Used"] <- output[varsused_i, "LASSO Trees Used"] + 1
    }

  }


  # compute relative frequencies of the variables
  output[, "LASSO Splits Rel"] <- output[, "LASSO Num Splits"] /
    sum(output[, "LASSO Num Splits"])
  output[, "RF Splits Rel"] <- output[, "RF Num Splits"] /
    sum(output[, "RF Num Splits"])
  output[, "% Chg in Rel Splits"] <- (output[, "LASSO Splits Rel"] /
                                        output[, "RF Splits Rel"] - 1) * 100

  getLeFrequency <- output
}
```

The code used to generate the table for simulation 1 then follows below. Similar code was used for other simulations.

```
set.seed(444)
le_y1a <- lassoEnsemble(x = x.a.train, y = y1.train)
y1a_freq <- getLeFrequency(xnames = c("x1", "x2", "x3", "x4"), le_y1a)

le_y1b <- lassoEnsemble(x = x.b.train, y = y1.train)
y1b_freq <- getLeFrequency(xnames = c("x1", "x2", "x3", "x4", "x5"), le_y1b)
```

Appendix C.3

This section provides code for our visualizations of results on the non-simulated datasets.

Appendix C.3.1

Below, we provide code used to clean and generate training and test sets for the real, non-simulated datasets.

```
# prostate data set
prostate <-  read.table("Prostate.csv", header=TRUE, sep=",", na.strings=" ")
prostate <- prostate[-c(1, 11)] # remove ID and train/test set

set.seed(123)
smp_size <- floor(0.75 * nrow(prostate))
train_ind <- sample(seq_len(nrow(prostate)), size = smp_size)

prostate_train <- (prostate[train_ind, ])
prostate_train_x <- prostate_train[, -9]
prostate_train_y <- prostate_train[, 9]
prostate_test <- (prostate[-train_ind, ])
prostate_test_x <- prostate_test[, -9]
prostate_test_y <- prostate_test[, 9]
```

```
# concrete data set
concrete <- read.csv("concrete.csv")
```

```r
set.seed(123)
smp_size <- floor(0.75 * nrow(concrete))
train_ind <- sample(seq_len(nrow(concrete)), size = smp_size)

concrete_train <- (concrete[train_ind, ])
concrete_train_x <- concrete_train[, -9]
concrete_train_y <- concrete_train[, 9]
concrete_test <- (concrete[-train_ind, ])
concrete_test_x <- concrete_test[, -9]
concrete_test_y <- concrete_test[, 9]
```

```r
# abalone data set
Abalone <- read.csv("abalone.csv", header = TRUE)
# remove incorrect inputs (this was taken from Assignment 1 solutions)
Abalone <- Abalone[(0 < Abalone$Height)&(Abalone$Height < 0.5),]

set.seed(123)
trainRows = (runif(nrow(Abalone),0,1)<=0.75)

abalone_train_x <- Abalone[trainRows, -9]
abalone_train_y <- Abalone[trainRows, 9]
abalone_test_x <- Abalone[!trainRows, -9]
abalone_test_y <- Abalone[!trainRows, 9]
```

Appendix C.3.2

Below, we provide code used to generate plots for the performance of the OOB Weighted Random Forest method on the Prostate dataset. Similar code was used for the other non-simulated datasets.

```r
ntrees.vect <- as.vector(seq(50, 500, by=50))
prostatedata.matrix <- matrix(data = NA, nrow=10, ncol=1+(3*2)*3)

set.seed(444)
### mtry = 1
for (n in ntrees.vect){
  rf.objA <- rfReweighted(prostate_train_x,prostate_train_y, n, 1, "inverse")
  rf.objB <- rfReweighted(prostate_train_x,prostate_train_y, n, 1, "log")
  rf.objC <- rfReweighted(prostate_train_x,prostate_train_y, n, 1, "linear")
  imp.predA <- predict_rfReweighted(prostate_test_x, rf.objA, TRUE)
  imp.predB <- predict_rfReweighted(prostate_test_x, rf.objB, TRUE)
  imp.predC <- predict_rfReweighted(prostate_test_x, rf.objC, TRUE)
  response <- prostate_test_y
  predA <- predict(rf.objA[["rf"]], newdata=prostate_test_x)
  predB <- predict(rf.objB[["rf"]], newdata=prostate_test_x)
  predC <- predict(rf.objC[["rf"]], newdata=prostate_test_x)
  prostatedata.matrix[n/50,1] <- n
  prostatedata.matrix[n/50,2] <- mean((predA-response)^2)
  prostatedata.matrix[n/50,3] <- mean((imp.predA-response)^2)
  prostatedata.matrix[n/50,4] <- mean((predB-response)^2)
  prostatedata.matrix[n/50,5] <- mean((imp.predB-response)^2)
  prostatedata.matrix[n/50,6] <- mean((predC-response)^2)
  prostatedata.matrix[n/50,7] <- mean((imp.predC-response)^2)
}
```

```
## mtry = default
for (n in ntrees.vect){
  rf.objA <- rfReweighted(prostate_train_x,prostate_train_y, n,
                          floor(ncol(prostate_train_x)/3), "inverse")
  rf.objB <- rfReweighted(prostate_train_x,prostate_train_y, n,
                          floor(ncol(prostate_train_x)/3), "log")
  rf.objC <- rfReweighted(prostate_train_x,prostate_train_y, n,
                          floor(ncol(prostate_train_x)/3), "linear")
  imp.predA <- predict_rfReweighted(prostate_test_x, rf.objA, TRUE)
  imp.predB <- predict_rfReweighted(prostate_test_x, rf.objB, TRUE)
  imp.predC <- predict_rfReweighted(prostate_test_x, rf.objC, TRUE)
  response <- prostate_test_y
  predA <- predict(rf.objA[["rf"]], newdata=prostate_test)
  predB <- predict(rf.objB[["rf"]], newdata=prostate_test)
  predC <- predict(rf.objC[["rf"]], newdata=prostate_test)
  prostatedata.matrix[n/50,8] <- mean((predA-response)^2)
  prostatedata.matrix[n/50,9] <- mean((imp.predA-response)^2)
  prostatedata.matrix[n/50,10] <- mean((predB-response)^2)
  prostatedata.matrix[n/50,11] <- mean((imp.predB-response)^2)
  prostatedata.matrix[n/50,12] <- mean((predC-response)^2)
  prostatedata.matrix[n/50,13] <- mean((imp.predC-response)^2)
}

## mtry = p
for (n in ntrees.vect){
  rf.objA <- rfReweighted(prostate_train_x,prostate_train_y, n,
                          ncol(prostate_train_x), "inverse")
  rf.objB <- rfReweighted(prostate_train_x,prostate_train_y, n,
                          ncol(prostate_train_x), "log")
  rf.objC <- rfReweighted(prostate_train_x,prostate_train_y, n,
                          ncol(prostate_train_x), "linear")
  imp.predA <- predict_rfReweighted(prostate_test_x, rf.objA, TRUE)
  imp.predB <- predict_rfReweighted(prostate_test_x, rf.objB, TRUE)
  imp.predC <- predict_rfReweighted(prostate_test_x, rf.objC, TRUE)
  response <- prostate_test_y
  predA <- predict(rf.objA[["rf"]], newdata=prostate_test)
  predB <- predict(rf.objB[["rf"]], newdata=prostate_test)
  predC <- predict(rf.objC[["rf"]], newdata=prostate_test)
  prostatedata.matrix[n/50,14] <- mean((predA-response)^2)
  prostatedata.matrix[n/50,15] <- mean((imp.predA-response)^2)
  prostatedata.matrix[n/50,16] <- mean((predB-response)^2)
  prostatedata.matrix[n/50,17] <- mean((imp.predB-response)^2)
  prostatedata.matrix[n/50,18] <- mean((predC-response)^2)
  prostatedata.matrix[n/50,19] <- mean((imp.predC-response)^2)
}
```

```
par(mfrow=c(3,3))
# mtry = 1
plot(x = seq(50, 500, 50), y = prostatedata.matrix[,2],  xlim = c(0, 550),
     ylim = c(min(prostatedata.matrix[,2:3]), max(prostatedata.matrix[,2:3])),
     type = "l", main = "Prostate MSPE: Mtry = 1, Scale = Inverse",
     xlab = "Number of Trees", ylab = "MSPE", cex.main=0.75)
lines(x = seq(50, 500, 50), y = prostatedata.matrix[,3], col = "red", lty=2)
```

```r
plot(x = seq(50, 500, 50), y = prostatedata.matrix[,4],  xlim = c(0, 550),
     ylim = c(min(prostatedata.matrix[,4:5]), max(prostatedata.matrix[,4:5])),
     type = "l", main = "Prostate MSPE: Mtry = 1, Scale = Log",
     xlab = "Number of Trees", ylab = "MSPE", cex.main=0.75)
lines(x = seq(50, 500, 50), y = prostatedata.matrix[,5], col = "red", lty=2)

plot(x = seq(50, 500, 50), y = prostatedata.matrix[,6],  xlim = c(0, 550),
     ylim = c(min(prostatedata.matrix[,6:7]), max(prostatedata.matrix[,6:7])),
     type = "l", main = "Prostate MSPE: Mtry = 1, Scale = Linear",
     xlab = "Number of Trees", ylab = "MSPE", cex.main=0.75)
lines(x = seq(50, 500, 50), y = prostatedata.matrix[,7], col = "red", lty=2)


# default mtry
plot(x = seq(50, 500, 50), y = prostatedata.matrix[,8],  xlim = c(0, 550),
     ylim = c(min(prostatedata.matrix[,8:9]), max(prostatedata.matrix[,8:9])),
     type = "l", main = "Prostate MSPE: Mtry = Default, Scale = Inverse",
     xlab = "Number of Trees", ylab = "MSPE", cex.main=0.75)
lines(x = seq(50, 500, 50), y = prostatedata.matrix[,9], col = "red", lty=2)

plot(x = seq(50, 500, 50), y = prostatedata.matrix[,10],  xlim = c(0, 550),
     ylim = c(min(prostatedata.matrix[,10:11]), max(prostatedata.matrix[,10:11])),
     type = "l", main = "Prostate MSPE: Mtry = Default, Scale = Log",
     xlab = "Number of Trees", ylab = "MSPE", cex.main=0.75)
lines(x = seq(50, 500, 50), y = prostatedata.matrix[,11], col = "red", lty=2)

plot(x = seq(50, 500, 50), y = prostatedata.matrix[,12],  xlim = c(0, 550),
     ylim = c(min(prostatedata.matrix[,12:13]), max(prostatedata.matrix[,12:13])),
     type = "l", main = "Prostate MSPE: Mtry = Default, Scale = Linear",
     xlab = "Number of Trees", ylab = "MSPE",cex.main=0.75)
lines(x = seq(50, 500, 50), y = prostatedata.matrix[,13], col = "red", lty=2)


# mtry = p
plot(x = seq(50, 500, 50), y = prostatedata.matrix[,14],  xlim = c(0, 550),
     ylim = c(min(prostatedata.matrix[,14:15]), max(prostatedata.matrix[,14:15])),
     type = "l", main = "Prostate MSPE: Mtry = p, Scale = Inverse",
     xlab = "Number of Trees", ylab = "MSPE", cex.main=0.75)
lines(x = seq(50, 500, 50), y = prostatedata.matrix[,15], col = "red", lty=2)

plot(x = seq(50, 500, 50), y = prostatedata.matrix[,16],  xlim = c(0, 550),
     ylim = c(min(prostatedata.matrix[,16:17]), max(prostatedata.matrix[,16:17])),
     type = "l", main = "Prostate MSPE: Mtry = p, Scale = Log",
     xlab = "Number of Trees", ylab = "MSPE", cex.main=0.75)
lines(x = seq(50, 500, 50), y = prostatedata.matrix[,17], col = "red", lty=2)

plot(x = seq(50, 500, 50), y = prostatedata.matrix[,18],  xlim = c(0, 550),
     ylim = c(min(prostatedata.matrix[,18:19]), max(prostatedata.matrix[,18:19])),
     type = "l", main = "Prostate MSPE: Mtry = p, Scale = Linear",
     xlab = "Number of Trees", ylab = "MSPE", cex.main=0.75)
lines(x = seq(50, 500, 50), y = prostatedata.matrix[,19], col = "red", lty=2)
```

Appendix C.3.3

Below, we provide code used to generate plots for the performance of the glmnet Weighted Random Forest method on the Prostate dataset. Similar code was used for other non-simulated data set.

```
# mtry plots ------------------------------------------------------------------

set.seed(444)
pr_mtry_plot0 <- MSPE.compare.mtry.le(x_train = prostate_train_x,
                                      y_train = prostate_train_y,
                                      x_test = prostate_test_x,
                                      y_test = prostate_test_y,
                                      alpha = 0)
pr_mtry_plot1 <- MSPE.compare.mtry.le(x_train = prostate_train_x,
                                      y_train = prostate_train_y,
                                      x_test = prostate_test_x,
                                      y_test = prostate_test_y,
                                      alpha = 1)

par(mfrow=c(2, 3))
make.MSPE.mtry.plot(pr_mtry_plot0, mtry = c(1, 2, 3), data_name = "Prostate")
make.MSPE.mtry.plot(pr_mtry_plot1, mtry = c(1, 2, 3), data_name = "Prostate")

# maxnode plots ---------------------------------------------------------------

set.seed(444)
pr_maxnode_plot0 <- MSPE.compare.maxnode.le(x_train = prostate_train_x,
                                            y_train = prostate_train_y,
                                            x_test = prostate_test_x,
                                            y_test = prostate_test_y,
                                            alpha = 0, maxlayer = 6)
pr_maxnode_plot1 <- MSPE.compare.maxnode.le(x_train = prostate_train_x,
                                            y_train = prostate_train_y,
                                            x_test = prostate_test_x,
                                            y_test = prostate_test_y,
                                            alpha = 1, maxlayer = 6)

par(mfrow = c(1, 2))
make.MSPE.maxnode.plot(pr_maxnode_plot0, data_name = "Prostate")
make.MSPE.maxnode.plot(pr_maxnode_plot1, data_name = "Prostate")
```

Appendix C.4

This section provdes code for our baseline models and sample MSPE values on the non-simulated datasets. Immediately below is our code for tuning the baseline models we used as reference for our two algorithms.

```
#Abalone cleanse and divide
Abalone <- Abalone[(0 < Abalone$Height)&(Abalone$Height < 0.5),]
set.seed(123)
smp_size_aba <- floor(0.75 * nrow(Abalone))
train_ind_aba <- sample(seq_len(nrow(Abalone)), size = smp_size_aba)
#Concrete divide
set.seed(123)
smp_size_con <- floor(0.75 * nrow(Concrete))
train_ind_con <- sample(seq_len(nrow(Concrete)), size = smp_size_con)
```

```r
#Prostate divide
Prostate <- ProState[-c(1, 11)] # remove ID and train/test set
set.seed(123)
smp_size_pro <- floor(0.75 * nrow(Prostate))
train_ind_pro <- sample(seq_len(nrow(Prostate)), size = smp_size_pro)

#Define training and test sets
Train.aba <- Abalone[train_ind_aba, ][1:9]
Train.pro <- Prostate[train_ind_pro,][1:9]
Train.con <- (Concrete[train_ind_con, ])[1:9]
Test.aba <- Abalone[-train_ind,][1:9]
Test.pro <- Prostate[-train_ind_pro,][1:9]
Test.con <- (Concrete[-train_ind_con, ])[1:9]

#Test prediction:
real.aba <- Test.aba$Rings
real.pro <- Test.pro$lpsa
real.con <- Test.con$strength

####### 1. Linear Model #######

#Abalone
lm.abalone <- lm(data = Train.aba, Rings~.)
summary(lm.abalone)
pred.lm.aba <- predict(newdata=Test.aba,lm.abalone)
lm.aba.error <- mean((pred.lm.aba-real.aba)^2)

#Prostate
lm.prostate <- lm(data = Train.pro, lpsa~.)
summary(lm.prostate)
pred.lm.pro <- predict(newdata=Test.pro,lm.prostate)
lm.pro.error <- mean((pred.lm.pro-real.pro)^2)

#Concrete
lm.concrete <- lm(data = Train.con, strength~.)
summary(lm.concrete)
pred.lm.con <- predict(newdata=Test.con,lm.concrete)
lm.con.error <- mean((pred.lm.con-real.con)^2)


####### 2. LASSO #######

library(glmnet)

#Abalone
lasso.aba <- cv.glmnet(Train.aba$Rings, x= as.matrix(Train.aba[,1:8]), family="gaussian")
pred.lasso.aba <- predict(newx =as.matrix(Test.aba[,1:8]), lasso.aba)
lasso.aba.error <- mean((real.aba - pred.lasso.aba)^2)

#Prostate
lasso.pro <- cv.glmnet(Train.pro$lpsa, x= as.matrix(Train.pro[,1:8]), family="gaussian")
pred.lasso.pro <- predict(newx =as.matrix(Test.pro[,1:8]), lasso.pro)
lasso.pro.error <- mean((real.pro - pred.lasso.pro)^2)
```

```r
#Concrete
lasso.con <- cv.glmnet(Train.con$strength, x= as.matrix(Train.con[,1:8]), family="gaussian")
pred.lasso.con <- predict(newx =as.matrix(Test.con[,1:8]), lasso.con)
lasso.con.error <- mean((real.con - pred.lasso.con)^2)


####### 3. MARS #######

library(earth)

#Abalone
mars.aba <- earth(Rings~., data=Train.aba, trace=3)
summary(mars.aba)
pred.mars.aba <- predict(mars.aba,newdata=Test.aba)
mars.aba.error <- mean((real.aba - pred.mars.aba)^2)

#Prostate
mars.pro <- earth(lpsa~., data=Train.pro, trace=3)
summary(mars.pro)
pred.mars.pro <- predict(mars.pro,newdata=Test.pro)
mars.pro.error <- mean((real.pro - pred.mars.pro)^2)

#Concrete
mars.con <- earth(strength~., data=Train.con, trace=3)
summary(mars.con)
pred.mars.con <- predict(mars.con,newdata=Test.con)
mars.con.error <- mean((real.con - pred.mars.con)^2)

####### 4. Regression Tree #######

library(rpart)

#Abalone
tree.aba <- rpart(data=Train.aba, Rings~., method = "anova")
cpt.aba <- tree.aba$cptable
minrow.aba <- which.min(cpt.aba[,4])
serow.aba <- min(which(cpt.aba[,4] < cpt.aba[minrow.aba,4]+cpt.aba[minrow.aba,5]))

pruned.cp.aba <- prune.rpart(tree.aba,
    cp=sqrt(cpt.aba[minrow.aba,1]*ifelse(minrow.aba==1, yes=1, no=cpt.aba[minrow.aba-1,1])))
pruned.se.aba <- prune.rpart(tree.aba,
    cp=sqrt(cpt.aba[serow.aba,1]*ifelse(serow.aba==1, yes=1, no=cpt.aba[serow.aba-1,1])))

pred.tree.cp.aba <- predict(pruned.cp.aba, newdata = Test.aba)
pred.tree.se.aba <- predict(pruned.se.aba, newdata = Test.aba)

tree.cp.aba.error <- mean((real.aba-pred.tree.cp.aba)^2)
tree.se.aba.error <- mean((real.aba-pred.tree.se.aba)^2)

#Prostate
tree.pro <- rpart(data=Train.pro, lpsa~., method = "anova")
cpt.pro <- tree.pro$cptable
minrow.pro <- which.min(cpt.pro[,4])
```

```
serow.pro <- min(which(cpt.pro[,4] < cpt.pro[minrow.pro,4]+cpt.pro[minrow.pro,5]))

pruned.cp.pro <- prune.rpart(tree.pro,
    cp=sqrt(cpt.pro[minrow.pro,1]*ifelse(minrow.pro==1, yes=1, no=cpt.pro[minrow.pro-1,1])))
pruned.se.pro <- prune.rpart(tree.pro,
    cp=sqrt(cpt.pro[serow.pro,1]*ifelse(serow.pro==1, yes=1, no=cpt.pro[serow.pro-1,1])))

pred.tree.cp.pro <- predict(pruned.cp.pro, newdata = Test.pro)
pred.tree.se.pro <- predict(pruned.se.pro, newdata = Test.pro)

tree.cp.pro.error <- mean((real.pro-pred.tree.cp.pro)^2)
tree.se.pro.error <- mean((real.pro-pred.tree.se.pro)^2)

#Concrete
tree.con <- rpart(data=Train.con, strength~., method = "anova")
cpt.con <- tree.con$cptable
minrow.con <- which.min(cpt.con[,4])
serow.con <- min(which(cpt.con[,4] < cpt.con[minrow.con,4]+cpt.con[minrow.con,5]))

pruned.cp.con <- prune.rpart(tree.con,
    cp=sqrt(cpt.con[minrow.con,1]*ifelse(minrow.con==1, yes=1, no=cpt.con[minrow.con-1,1])))
pruned.se.con <- prune.rpart(tree.con,
    cp=sqrt(cpt.con[serow.con,1]*ifelse(serow.con==1, yes=1, no=cpt.con[serow.con-1,1])))

pred.tree.cp.con <- predict(pruned.cp.con, newdata = Test.con)
pred.tree.se.con <- predict(pruned.se.con, newdata = Test.con)

tree.cp.con.error <- mean((real.con-pred.tree.cp.con)^2)
tree.se.con.error <- mean((real.con-pred.tree.se.con)^2)


####### 5. XG Boost #######

library(xgboost)

#Abalone
iter.aba = 100
set.seed(1234)
xg.aba.cv <- xgb.cv(data=as.matrix(Train.aba[,1:8]), label=Train.aba$Rings,
                    max_depth=3, eta=.3, subsample=1,
                    nrounds=iter.aba, objective="reg:linear", nfold=5)
min.aba <- which.min(xg.aba.cv$evaluation_log$test_rmse_mean)
if(min.aba == iter.aba) {stop("test nrounds too small")}
set.seed(1234)
xg.aba<- xgboost(data=as.matrix(Train.aba[,1:8]), label=Train.aba$Rings,
                 max_depth=3, eta=.3, subsample=1,
                 nrounds=min.aba, objective="reg:linear")

pred.xg.aba.test <- predict(xg.aba, newdata=as.matrix(Test.aba[,1:8]))
xg.aba.error <- mean((pred.xg.aba.test-real.aba)^2)

#Prostate
iter.pro = 20
```

```
set.seed(1234)
xg.pro.cv <- xgb.cv(data=as.matrix(Train.pro[,1:8]), label=Train.pro$lpsa,
                    max_depth=3, eta=.3, subsample=1,
                    nrounds=iter.pro, objective="reg:linear", nfold=5)
min.pro <- which.min(xg.pro.cv$evaluation_log$test_rmse_mean)
if(min.pro == iter.pro) {stop("test nrounds too small")}
set.seed(1234)
xg.pro<- xgboost(data=as.matrix(Train.pro[,1:8]), label=Train.pro$lpsa,
                 max_depth=3, eta=.3, subsample=1,
                 nrounds=min.pro, objective="reg:linear")

pred.xg.pro.test <- predict(xg.pro, newdata=as.matrix(Test.pro[,1:8]))
xg.pro.error <- mean((pred.xg.pro.test-real.pro)^2)

#Concrete
iter.con = 600
set.seed(1234)
xg.con.cv <- xgb.cv(data=as.matrix(Train.con[,1:8]), label=Train.con$strength,
                    max_depth=3, eta=.3, subsample=1,
                    nrounds=iter.con, objective="reg:linear", nfold=5)
min.con <- which.min(xg.con.cv$evaluation_log$test_rmse_mean)
if(min.con == iter.con) {stop("test nrounds too small")}
set.seed(1234)
xg.con<- xgboost(data=as.matrix(Train.con[,1:8]), label=Train.con$strength,
                 max_depth=3, eta=.3, subsample=1,
                 nrounds=min.con, objective="reg:linear")

pred.xg.con.test <- predict(xg.con, newdata=as.matrix(Test.con[,1:8]))
xg.con.error <- mean((pred.xg.con.test-real.con)^2)


######## Summary ########
lm.err <- c(lm.pro.error,lm.con.error,lm.aba.error)
lasso.err <- c(lasso.pro.error, lasso.con.error,lasso.aba.error)
mars.err <- c(mars.pro.error, mars.con.error,mars.aba.error)
tree.cp.err <- c(tree.cp.pro.error, tree.cp.con.error,tree.cp.aba.error)
tree.se.err <- c(tree.se.pro.error, tree.se.con.error,tree.se.aba.error)
xg.err <- c(xg.pro.error, xg.con.error,xg.aba.error)

summary <- round(rbind(lm.err,lasso.err,mars.err,tree.cp.err,tree.se.err,xg.err),3)
colnames(summary) <- c("Prostate", "Concrete", "Abalone")
rownames(summary) <- c("Linear Model", "LASSO", "MARS",
                       "Tree min-cp", "Tree +1SE", "XGBoost")
```

Below is the code used to generate the MSPE for our OOB Weighted Random Forests. Note that these values were extracted from the loop we used to build our plots.

```
prostatedata.matrix[10, 9]
concretedata.matrix[10, 9]
abalonedata.matrix[10, 9]
```

Below is the code used to generate the MSPE for our glmnet Weighted Random Forests on the Prostate dataset. We refit a LASSO weighted random forest with 500 trees, and the default `mtry` value. The code is similar for the other datasets.

```r
set.seed(444)
le_pr <- lassoEnsemble(x = prostate_train_x, y = prostate_train_y)
pred_le_pr <- predict_lassoEnsemble(x = prostate_test_x, le = le_pr, type = "le")
MSPE_le_pr <- mean((prostate_test_y - pred_le_pr)^2)
```

Appendix D

Below is a documentation of the various real-world datasets that we applied the OOB Weighted Random Forest and LASSO Weighted Random Forest on.

Appendix D.1 Prostate Cancer Data

Pulled from `R`, these data come from a study that examined the correlation between the level of prostate specific antigen and a number of clinical measures in men who were about to receive a radical prostatectomy. It is a data frame with 97 rows and 9 columns.The goal is to predict the variable `lpsa`.

The data frame has the following components:

lcavol - log(cancer volume)

lweight -log(prostate weight)

age - age

lbph - log(benign prostatic hyperplasia amount)

svi -seminal vesicle invasion

lcp - log(capsular penetration)

gleason - Gleason score

pgg45 - percentage Gleason scores 4 or 5

lpsa- log(prostate specific antigen)

Stamey, T.A., Kabalin, J.N., McNeal, J.E., Johnstone, I.M., Freiha, F., Redwine, E.A. and Yang, N. (1989) Prostate specific antigen in the diagnosis and treatment of adenocarcinoma of the prostate: II. radical prostatectomy treated patients, Journal of Urology 141(5), 1076-1083.

Appendix D.2 Concrete Strength Data

Pulled from UCI Machine Learning Repository, concrete is the most important material in civil engineering. The concrete compressive strength is a highly nonlinear function of age and ingredients.The goal is to predict concrete compressive strength using available features. There are 9 columns and 1030 rows.

The data frame has the following components:

Cement (component 1) – quantitative – kg in a m3 mixture – Input Variable

Blast Furnace Slag (component 2) – quantitative – kg in a m3 mixture – Input Variable

Fly Ash (component 3) – quantitative – kg in a m3 mixture – Input Variable

Water (component 4) – quantitative – kg in a m3 mixture – Input Variable

Superplasticizer (component 5) – quantitative – kg in a m3 mixture – Input Variable

Coarse Aggregate (component 6) – quantitative – kg in a m3 mixture – Input Variable

Fine Aggregate (component 7) – quantitative – kg in a m3 mixture – Input Variable

Age – quantitative – Day (1~365) – Input Variable

Concrete compressive strength – quantitative – MPa – Output Variable

I-Cheng Yeh, "Modeling of strength of high performance concrete using artificial neural networks," Cement and Concrete Research, Vol. 28, No. 12, pp. 1797-1808 (1998)

Appendix D.3 Abalone Data

Pulled from UCI Machine Learning Repository, the goal is to predict the age of individual abalone on the basis of physical features alone. There are 9 columns and 4177 rows.

The data frame has the following components:

Sex / nominal / – / M, F, and I (infant)

Length / continuous / mm / Longest shell measurement

Diameter / continuous / mm / perpendicular to length

Height / continuous / mm / with meat in shell

Whole weight / continuous / grams / whole abalone

Shucked weight / continuous / grams / weight of meat

Viscera weight / continuous / grams / gut weight (after bleeding)

Shell weight / continuous / grams / after being dried

Rings / integer / – / +1.5 gives the age in years

Dua, D. and Graff, C. (2019). UCI Machine Learning Repository [http://archive.ics.uci.edu/ml]. Irvine, CA: University of California, School of Information and Computer Science

Appendix D.4 Final List of References

Dua, D. and Graff, C. (2019). UCI Machine Learning Repository [http://archive.ics.uci.edu/ml]. Irvine, CA: University of California, School of Information and Computer Science

I-Cheng Yeh, "Modeling of strength of high performance concrete using artificial neural networks," Cement and Concrete Research, Vol. 28, No. 12, pp. 1797-1808 (1998)

Stamey, T.A., Kabalin, J.N., McNeal, J.E., Johnstone, I.M., Freiha, F., Redwine, E.A. and Yang, N. (1989) Prostate specific antigen in the diagnosis and treatment of adenocarcinoma of the prostate: II. radical prostatectomy treated patients, Journal of Urology 141(5), 1076-1083.