Ensemble Methods

Taylor Shrode

November 15, 2020

Introduction

"Ensemble learning is a method to combine results produced by different learners into one format, with the aim of producing better classification results and regression results (Chiu, 2015)." The main goal of ensemble learning is to find the optimal classification or regression model by averaging, or voting, the results of the combined classifiers. Voting is a technique used in classification where multiple models are used to make predictions for each data point and those predictions from each model are considered as a 'vote' (Ensemble learning, 2018). The predictions we get from the majority of the models are used as the final prediction. On the other hand, averaging is used for regression. This method takes an average of predictions from all the models and uses it to make the final prediction (Ensemble learning, 2018). Averaging can be used for making predictions in regression problems or while calculating probabilities for classification problems (Ensemble learning, 2018).

The most common methods used in ensemble learning include bagging, boosting, and random forest. For this assignment, we will be exploring bagging and boosting using the *PimaIndiansDiabetes2* data set. This data set contains information about diabetes of Indian people (Diabetes of indian people, n.d.). The attributes in this dataset include (Diabetes of indian people, n.d.):

1. pregnant: Number of times pregnant (numeric)
2. glucose: Plasma glucose concentration a 2 hours in an oral glucose tolerance test (numeric)
3. pressure: Diastolic blood pressure (mm Hg) (numeric)
4. triceps: Triceps skin fold thickness (mm) (numeric)
5. insulin: 2-Hour serum insulin (mu U/ml)(numeric)
6. mass: Body mass index (weight in kg/(height in m)^2) (numeric)
7. pedigree: Diabetes pedigree function (numeric)
8. age: Age (years) (numeric)
9. diabetes class variable: diabetic or not (factor with 2 levels: neg and pos)

There are many methods and various packages that can be used for bagging and boosting. We will only explore a couple of these methods.

Libraries and Data Set

Before we begin, we need to load the dataset and the packages needed to throughout this assignment.

```
library(mlbench)
library(DataExplorer)
library(ggplot2)
#library(dplyr)
library("mice")
library("VIM")
library(ipred)
library(caret)
library(pROC)
```

The **mlbench** package is loaded so we can retrieve our dataset. The **DataExplorer** package is used for exploratory data analysis and the **ggplot2** package is loaded to create other various plots. The **dplyr** package is loaded for general data wrangling needs. The **mice** package is loaded so we can apply multiple imputation and the **VIM** package allows us to determine the pattern of missing values. To perform bagging, we need to use the **ipred** and **caret** packages. To perform boosting, we will be using the **caret** package. Finally, the **pROC** package is loaded to create ROC curves.

Now, we can load our dataset.

```
data("PimaIndiansDiabetes2")
diabetes <- PimaIndiansDiabetes2
```

Using our loaded dataset, we can perform exploratory data analysis.

Exploratory Data Analysis

To begin exploring our data, we will view the structure and a summary of our data.

```
str(diabetes)
```

```
## 'data.frame':    768 obs. of  9 variables:
##  $ pregnant: num  6 1 8 1 0 5 3 10 2 8 ...
##  $ glucose : num  148 85 183 89 137 116 78 115 197 125 ...
##  $ pressure: num  72 66 64 66 40 74 50 NA 70 96 ...
##  $ triceps : num  35 29 NA 23 35 NA 32 NA 45 NA ...
##  $ insulin : num  NA NA NA 94 168 NA 88 NA 543 NA ...
##  $ mass    : num  33.6 26.6 23.3 28.1 43.1 25.6 31 35.3 30.5 NA ...
##  $ pedigree: num  0.627 0.351 0.672 0.167 2.288 ...
##  $ age     : num  50 31 32 21 33 30 26 29 53 54 ...
##  $ diabetes: Factor w/ 2 levels "neg","pos": 2 1 2 1 2 1 2 1 2 2 ...
```

```
summary(diabetes)
```

```
##     pregnant         glucose         pressure         triceps
##  Min.   : 0.000   Min.   : 44.0   Min.   : 24.00   Min.   : 7.00
```

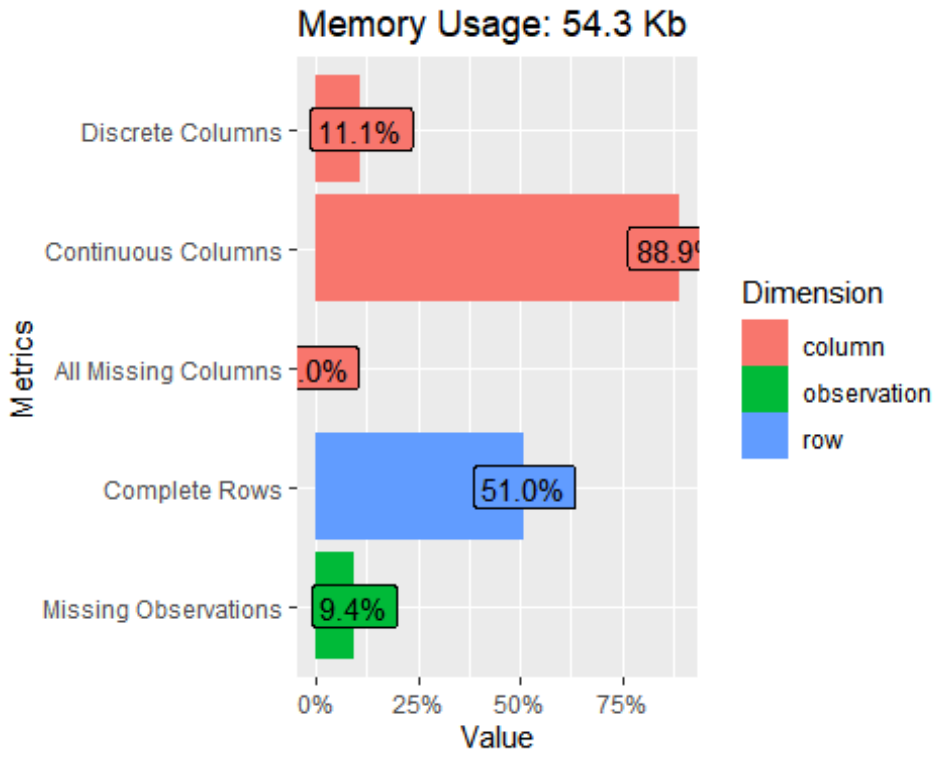
```
##  1st Qu.: 1.000   1st Qu.: 99.0   1st Qu.: 64.00   1st Qu.:22.00
##  Median : 3.000   Median :117.0   Median : 72.00   Median :29.00
##  Mean   : 3.845   Mean   :121.7   Mean   : 72.41   Mean   :29.15
##  3rd Qu.: 6.000   3rd Qu.:141.0   3rd Qu.: 80.00   3rd Qu.:36.00
##  Max.   :17.000   Max.   :199.0   Max.   :122.00   Max.   :99.00
##                   NA's   :5       NA's   :35       NA's   :227
##     insulin            mass          pedigree           age
diabetes
##  Min.   : 14.00   Min.   :18.20   Min.   :0.0780   Min.   :21.00   neg:500
##  1st Qu.: 76.25   1st Qu.:27.50   1st Qu.:0.2437   1st Qu.:24.00   pos:268
##  Median :125.00   Median :32.30   Median :0.3725   Median :29.00
##  Mean   :155.55   Mean   :32.46   Mean   :0.4719   Mean   :33.24
##  3rd Qu.:190.00   3rd Qu.:36.60   3rd Qu.:0.6262   3rd Qu.:41.00
##  Max.   :846.00   Max.   :67.10   Max.   :2.4200   Max.   :81.00
##  NA's   :374      NA's   :11
```

The **str()** output indicates that our dataset contains 768 observations of 9 attributes. We can also see that our data contains 8 numeric attributes and 1 factor attribute (response variable). Thus, we do not need to transform the data types of our attributes. Next, from the **summary()** output, we can see that several of our columns contain missing values.

```
introduce(diabetes)
```

```
##   rows columns discrete_columns continuous_columns all_missing_columns
## 1  768       9                1                  8                   0
##   total_missing_values complete_rows total_observations memory_usage
## 1                  652           392               6912        55592
```

```
plot_intro(diabetes)
```



From the plot above, only 51% of our rows are complete (do not contain any NA values). To handle these missing values, we could use one of the following methods:

1. Multiple Imputation
2. Quick Imputation
3. Omit NA Values

The first option requires more work, but is flexible with any kind of analysis, especially when we have limited knowledge of the missing values model. The second method is easy to perform, by may affect the variance of the data. By removing the incomplete rows, our dataset would only contain 392 rows.

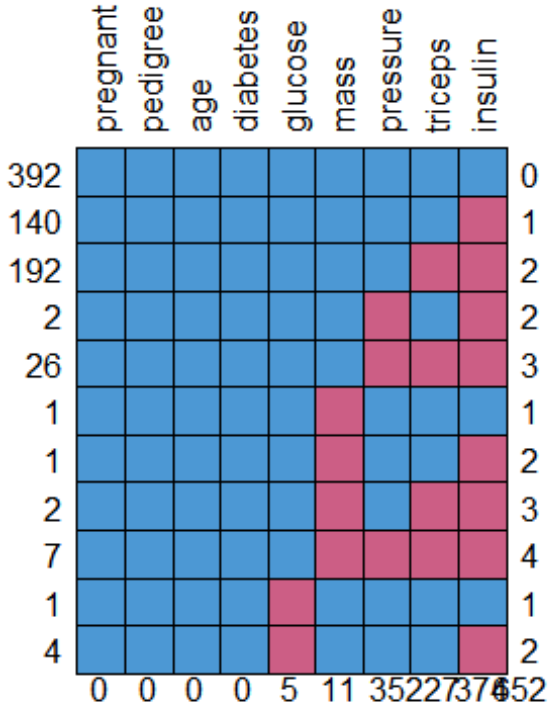
```
colSums(is.na(diabetes))
```

```
## pregnant  glucose pressure  triceps  insulin     mass pedigree      age
##        0        5       35      227      374       11        0        0
## diabetes
##        0
```

While we have a high percentage of missing values, and a high percentage of missing values in the columns **triceps** and **insulin**, we will not remove these attributes because this information may be useful for our predictive models. Thus, we will apply multiple imputation using the **mice** package. This package imputes missing values with plausible data values that are drawn from a distribution specifically designed for each missing data point (Imputing missing data with r, 2018).

To use this package, we assume that the missing data is classified as “missing completely at random” (MCAR) versus “missing not at random” (MNAR). To get a better understanding of the pattern of missing data, we can use the **md.pattern()** function (Imputing missing data with r, 2018).

```
md.pattern(diabetes, rotate.names = TRUE)
```



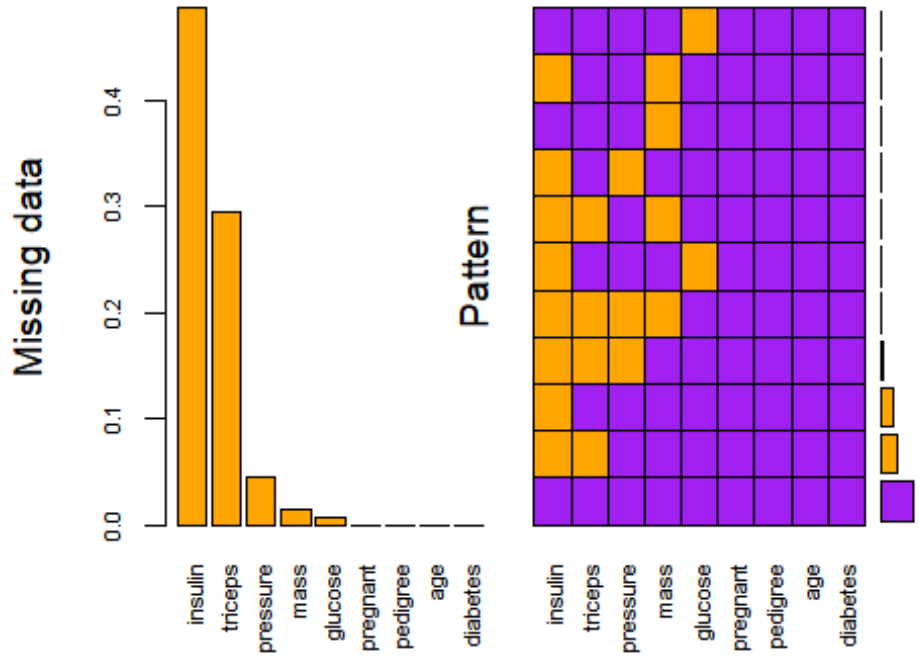

```
##     pregnant pedigree age diabetes glucose mass pressure triceps insulin
## 392        1        1   1        1       1    1        1       1       1
0
## 140        1        1   1        1       1    1        1       1       0
1
## 192        1        1   1        1       1    1        1       0       0
2
## 2          1        1   1        1       1    1        0       1       0
2
## 26         1        1   1        1       1    1        0       0       0
3
## 1          1        1   1        1       1    0        1       1       1
1
## 1          1        1   1        1       1    0        1       1       0
2
## 2          1        1   1        1       1    0        1       0       0
3
## 7          1        1   1        1       1    0        0       0       0
4
## 1          1        1   1        1       0    1        1       1       1
```

```
1
## 4           1         1   1         1         0    1         1         1         0
2
##            0         0   0         0         5   11        35       227       374
652
```

The output above tells us we have 392 samples that are complete, 140 samples that only miss the **insulin** measurement, 192 samples that miss the **triceps** and **insulin** measurement, and so on.

A more useful visual representation can be obtained using the **VIM** package (Imputing missing data with r, 2018).

```
mice_plot <- aggr(diabetes, col=c('purple','orange'),
                  numbers=TRUE, sortVars=TRUE,
                  labels=names(diabetes), cex.axis=.7,
                  gap=1, ylab=c("Missing data","Pattern"))
```




```
##
##  Variables sorted by number of missings:
##  Variable       Count
##   insulin 0.486979167
##   triceps 0.295572917
##  pressure 0.045572917
##      mass 0.014322917
##   glucose 0.006510417
##  pregnant 0.000000000
```

```
##  pedigree 0.000000000
##       age 0.000000000
##  diabetes 0.000000000
```

The plot above indicates that ~51% of the samples are not missing and information, ~25% are missing the **triceps** and **insulin** information, and so on.

To impute the missing data, we use the **mice()** function.

```
tempData <- mice(diabetes,m=5,maxit=50,meth='pmm',seed=500, printFlag =
FALSE)
summary(tempData)

## Class: mids
## Number of multiple imputations:  5
## Imputation methods:
## pregnant  glucose pressure  triceps  insulin     mass pedigree      age
##       ""    "pmm"    "pmm"    "pmm"    "pmm"    "pmm"       ""       ""
## diabetes
##       ""
## PredictorMatrix:
##          pregnant glucose pressure triceps insulin mass pedigree age
diabetes
## pregnant        0       1        1       1       1    1        1   1
1
## glucose         1       0        1       1       1    1        1   1
1
## pressure        1       1        0       1       1    1        1   1
1
## triceps         1       1        1       0       1    1        1   1
1
## insulin         1       1        1       1       0    1        1   1
1
## mass            1       1        1       1       1    0        1   1
1
```

The parameters above (Imputing missing data with r, 2018):

- $m = 5$ is the number of imputed datasets (5 is default value).
- $meth = 'pmm'$ refers to the imputation method. Here, we use the "predictive mean matching" imputation method.
- $maxit$ is the number of iteration (5 is default value).
- $seed$ is an integer that is used as argument by the **set.seed()** for offsetting the random number generator. Default is to leave the random number generator alone.
- $printFlag = False$ silences the computation history on the console.

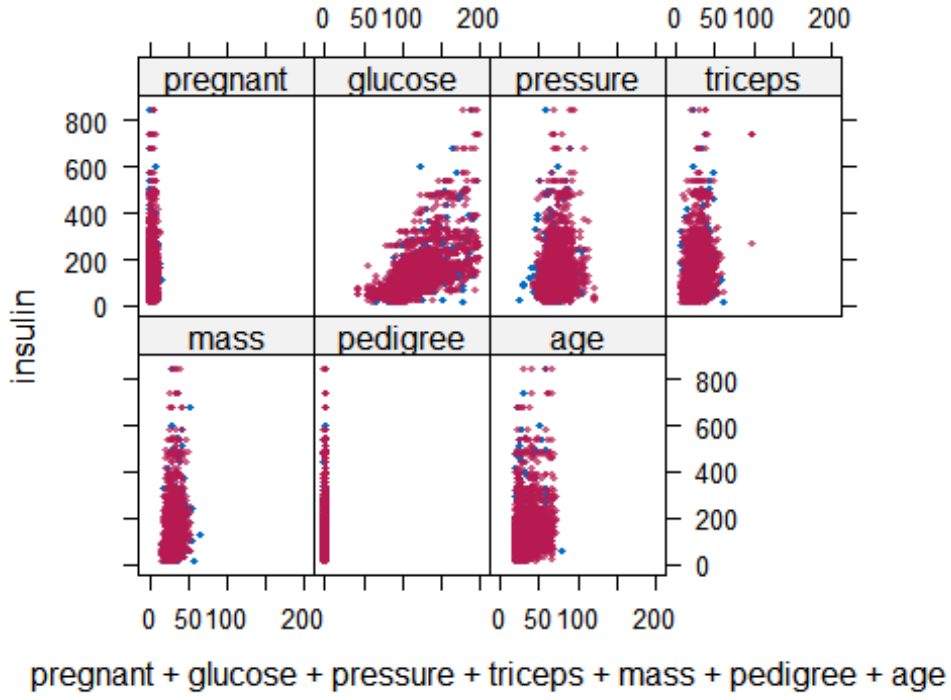
To check the imputed data for the variable **insulin**, we can use the following code (Imputing missing data with r, 2018).

```
head(tempData$imp$insulin)
```

```
##       1   2   3   4   5
## 1   105 237 342 545 144
## 2    32  56  82  16  36
## 3   495 225 245 300 328
## 6    78 105 265  87 116
## 8   191 110  85 100  91
## 10 130 200 122 167 171
```

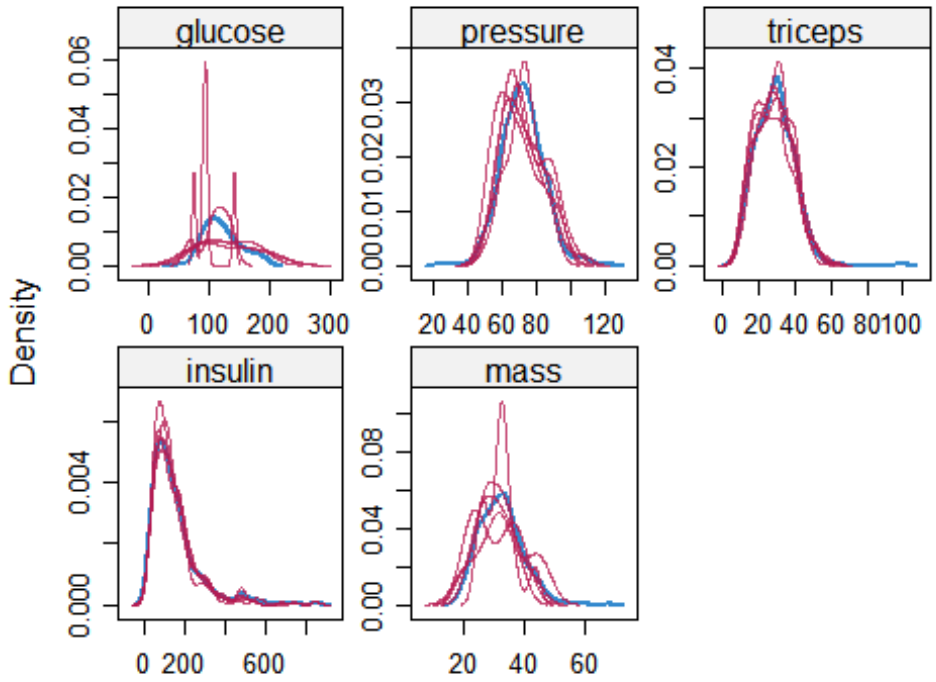
The output above shows the imputed data for each observation within each imputed dataset. We can inspect the distributions of the original and imputed data. First, we will use a scatterplot and plot **insulin** against all other variables (Imputing missing data with r, 2018).

```
xyplot(tempData,insulin ~ pregnant + glucose + pressure + triceps + mass +
pedigree + age ,pch=18,cex=0.5)
```



Ideally, we want to see the shape of the imputed points (magenta) to match the shape of the observed points (blue) (Imputing missing data with r, 2018). This indicates that the imputed values are indeed "plausible". We can also use a density plot.

```
densityplot(tempData)
```



Again, we want to see the distribution of the imputed points (magenta) to match the distribution of the observed points (blue) (Imputing missing data with r, 2018). This indicates that the imputed values are indeed "plausible". We can also use a density plot. We can see that the distributions for **glucose** and **mass** have datasets that do not indicate plausible values.

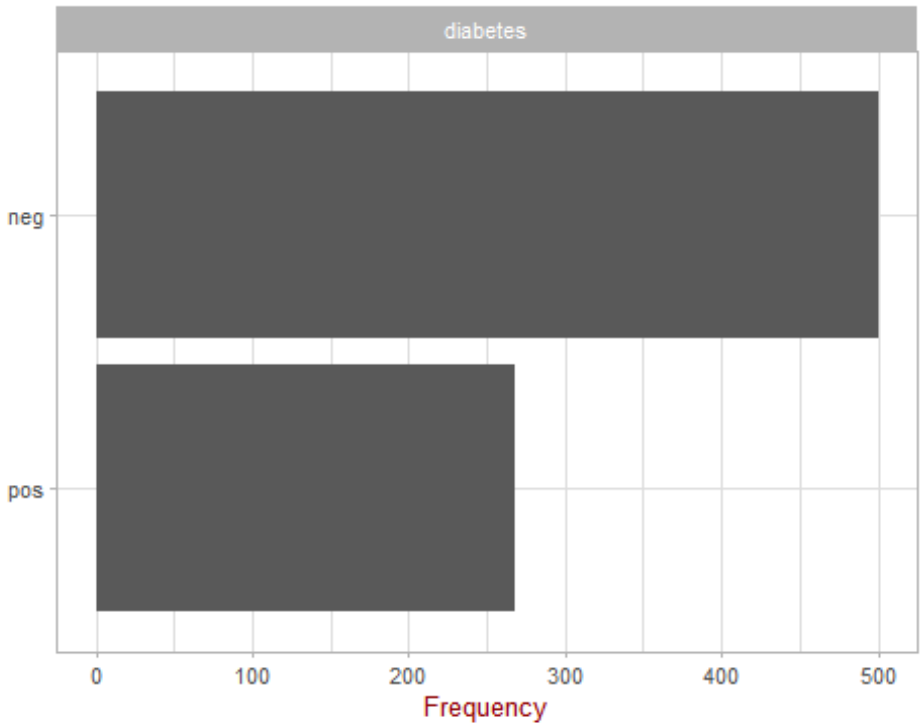
Now, to obtain the completed dataset, we can use the **complete()** function and choose which dataset we want to use. Below, we choose the first dataset.

```
completeData <- complete(tempData,1)
summary(completeData)
```

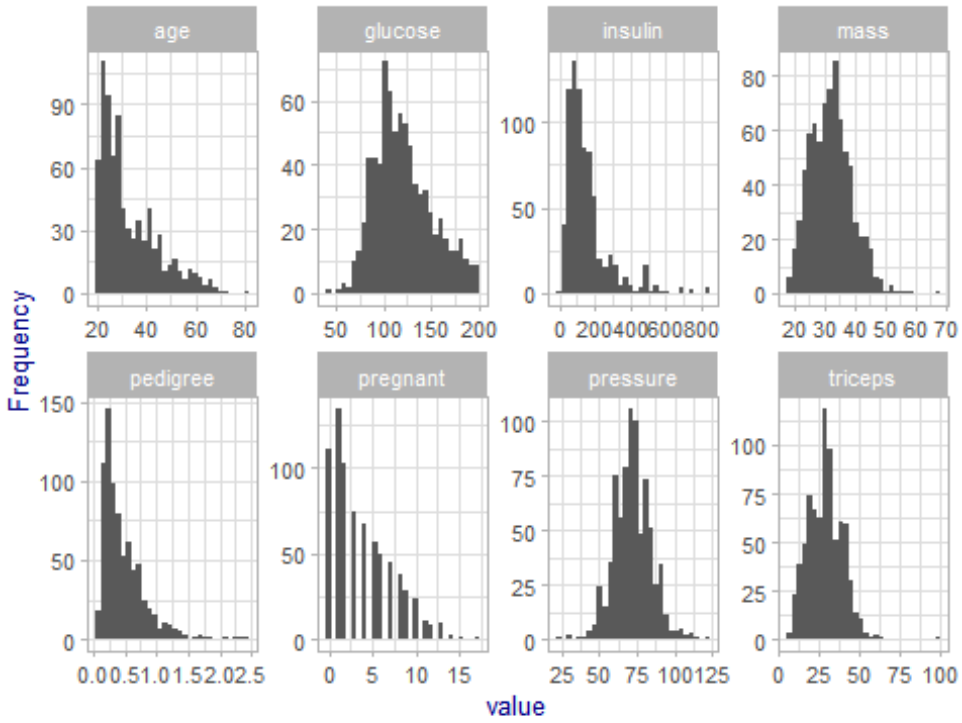
```
##     pregnant         glucose         pressure         triceps
##  Min.   : 0.000   Min.   : 44.0   Min.   : 24.00   Min.   : 7.00
##  1st Qu.: 1.000   1st Qu.: 99.0   1st Qu.: 64.00   1st Qu.:21.00
##  Median : 3.000   Median :117.0   Median : 72.00   Median :29.00
##  Mean   : 3.845   Mean   :121.6   Mean   : 72.37   Mean   :28.77
##  3rd Qu.: 6.000   3rd Qu.:140.2   3rd Qu.: 80.00   3rd Qu.:36.00
##  Max.   :17.000   Max.   :199.0   Max.   :122.00   Max.   :99.00
##     insulin           mass          pedigree           age         diabetes
##  Min.   : 14.0   Min.   :18.20   Min.   :0.0780   Min.   :21.00   neg:500
##  1st Qu.: 78.0   1st Qu.:27.48   1st Qu.:0.2437   1st Qu.:24.00   pos:268
##  Median :125.0   Median :32.25   Median :0.3725   Median :29.00
##  Mean   :154.7   Mean   :32.42   Mean   :0.4719   Mean   :33.24
##  3rd Qu.:183.2   3rd Qu.:36.60   3rd Qu.:0.6262   3rd Qu.:41.00
##  Max.   :846.0   Max.   :67.10   Max.   :2.4200   Max.   :81.00
```

Now, using our "completed" dataset, we can plot the distributions of our discrete and continuous variables.

```
plot_bar(completeData,ggtheme = theme_light(base_size = 10), theme_config =
list("text" = element_text(color = "darkred")))
```




```
plot_histogram(completeData,ggtheme = theme_light(base_size = 10),
theme_config = list("text" = element_text(color = "navyblue")))
```



From our barplot, we have more people without diabetes then people with diabetes. From our histogram, **age**, **insulin**, **pedigree**, and **pregnant** attributes are not normally distributed. These attributes each have right-skewed histograms, which indicate that the majority of the values in each column are lower values. For example, most of the people are younger in age.

Partition Data

Before we can use ensemble methods, we need to partition the data. We can do this by using the **createDataPartition()** function. First, we will partition the data set into a training and testing dataset with a 75:25 ratio.

```
set.seed(789) #make results reproducible
index <- createDataPartition(completeData$diabetes, p =0.75, list = FALSE)
```

Now, we will create our training dataset and labels.

```
diabetesTrain <- completeData[index,]
trainLabels <- completeData[9][index,]
```

Finally, we will create our testing dataset and labels.

```
diabetesTest <- completeData[-index,]
testLabels <- completeData[-index,9]
```

We can confirm our split using the **dim()** function.

```
## 75% of Data: 576

## Dimension of training dataset: 576 9

## 25% of Data: 192

## Dimension of testing dataset: 192 9
```

Now, we can proceed with bagging.

Bagging Method

The first type of ensemble methods we will use is *bagging*, also known as *bootstrap aggregating* (Lantz, 2015). Bagging generates a number of training datasets by bootstrap sampling (resampling method that repeatedly draws independent samples from our data set and provides a direct computational way of assessing uncertainty) the original training data (Lantz, 2015). In other words, bagging fits multiple versions of a prediction model and then combines/ensembles them into an aggregated prediction (Greenwell, n.d.-a). Using our dataset, we will be bagging classification trees (decision trees) using the package **ipred** and **caret**. Recall, decision trees can partition the feature space without data scaling, so there is no need to normalize the data. Bagging is usually applied to regression tree methods, but can be applied to any regression or classification model. This method provides the best improvement results when used on models with high variance (Greenwell, n.d.-a).

Regression and classification trees partition a dataset into smaller groups and then fit a simple model for each subgroup. These single tree models are often highly unstable, suffer from high variance, and are not good predictors (Greenwell, n.d.-a). Averaging across multiple trees reduces the variability of a single tree and reduces overfitting (Greenwell, n.d.-a). This results in an improvement of predictive performance. Another benefit of bagging is that a bootstrap sample will contain $\frac{2}{3}$ of the training data and leave the remaining $\frac{1}{3}$ of the data out of the sample, called the *out-of-bag (OOB)* sample (Greenwell, n.d.-a). The OOB data is used to estimate the model's accuracy.

We can perform bagging by utilizing the **bagging()** function in the **ipred** package and the following parameters (Greenwell, n.d.-a):

- **nbagg** controls the number of iterations to include in the model (25 is default value)
- **coob** indicates whether to use the OOB sample as the error rate

```
set.seed(789)
bagged_ipred <- bagging(
  formula = diabetes ~ .,
  data = diabetesTrain,
  coob = TRUE,
  nbagg = 25
  )
```

```
bagged_ipred
```

```
##
## Bagging classification trees with 25 bootstrap replications
##
## Call: bagging.data.frame(formula = diabetes ~ ., data = diabetesTrain,
##     coob = TRUE, nbagg = 25)
##
## Out-of-bag estimate of misclassification error:  0.2535
```

This model works as expected with the **predict()** function (Lantz, 2015). Thus, we can then create a confusion matrix of our predictions.

```
bagged_pred <- predict(bagged_ipred, diabetesTest)
confusionMatrix(bagged_pred, testLabels, mode = "everything")
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction neg pos
##        neg 102  19
##        pos  23  48
##
##                Accuracy : 0.7812
##                  95% CI : (0.716, 0.8376)
##     No Information Rate : 0.651
##     P-Value [Acc > NIR] : 6.137e-05
##
##                   Kappa : 0.5251
##
##  Mcnemar's Test P-Value : 0.6434
##
##             Sensitivity : 0.8160
##             Specificity : 0.7164
##          Pos Pred Value : 0.8430
##          Neg Pred Value : 0.6761
##               Precision : 0.8430
##                  Recall : 0.8160
##                      F1 : 0.8293
##              Prevalence : 0.6510
##          Detection Rate : 0.5312
##    Detection Prevalence : 0.6302
##       Balanced Accuracy : 0.7662
##
##        'Positive' Class : neg
##
```

Recall from Week 2, Kappa values range from 0 to a maximum of 1. A value of 1 indicates a perfect agreement between the model's predictions and the true values (Lantz, 2015). Further,

- Poor agreement = less than 0.20
- Fair agreement = 0.20 to 0.40
- Moderate agreement = 0.40 to 0.60
- Good agreement = 0.60 to 0.80
- Very good agreement = 0.80 to 1.00

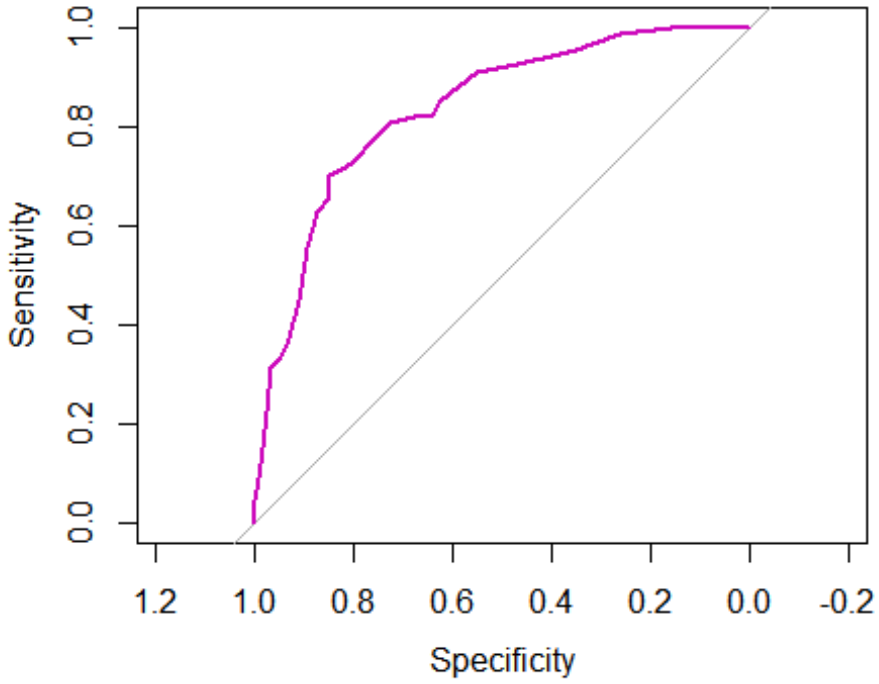
Thus, our Kappa value above indicates a moderate agreement between the model's predictions and the true values. Precision, also known as the positive predictive value, is defined as the proportion of positive examples that are truly positive. According to our value above, when our model predicts the positive class, it is correct ~84% of the time. Alternatively, recall is the measure of how complete the results are. In other words, when the observation is actually positive, how often does the model predict it's positive. Our model is approximately ~81% complete.

Finally, we can plot the ROC curve and calculate the area under the curve for our **ipred** model. To do this, we first need to calculate the probabilities that a person is classified as having diabetes or not having diabetes (Ensemble methods, n.d.). We do this by applying our model above to the testing data set by using the **predict()** function. Then, we can calculate and plot the ROC curve (Ensemble methods, n.d.).

```
bag_ipred <- predict(bagged_ipred, diabetesTest, type = "prob")
head(bag_ipred)
```

```
##       neg  pos
## [1,] 0.52 0.48
## [2,] 0.12 0.88
## [3,] 0.36 0.64
## [4,] 0.24 0.76
## [5,] 0.76 0.24
## [6,] 0.88 0.12
```

```
rocCurve_ipred <- roc(testLabels, bag_ipred[,"neg"])
plot(rocCurve_ipred, col= c(6))
```



To calculate the area under the curve, we can use the following code.

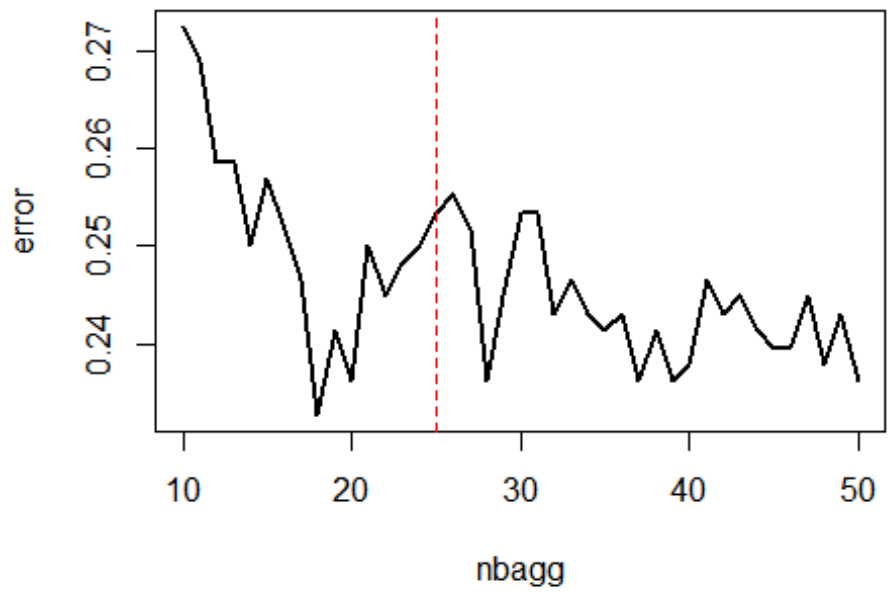
```
pROC::auc(rocCurve_ipred)

## Area under the curve: 0.8361
```

An *auc* value of 1 indicates that 100% of the predictions were correct (Ensemble methods, n.d.).

One thing to note is that (typically) more trees are better when performing bagging (Greenwell, n.d.-b). As more trees are added, we are averaging over more high variance, which results in a dramatic reduction in variance (hence the error) early on and eventually the reduction in error will flatline (Greenwell, n.d.-b). We can assess the error versus the number of trees by using the code below (Greenwell, n.d.-b).

```
nbagg <- 10:50

# create empty vector to store OOB RMSE values
error <- vector(mode = "numeric", length = length(nbagg))


for (i in seq_along(nbagg)) {
  # reproducibility
  set.seed(789)

  # perform bagged model
  tune_model <- bagging(
```

```
    formula = diabetes ~ .,
    data = diabetesTrain,
    coob = TRUE,
    nbagg = nbagg[i]
  )

  # get OOB error
  error[i] <- tune_model$err
}


plot(nbagg, error, type = 'l', lwd = 2)
abline(v = 25, col = "red", lty = "dashed") #indicated default 25 bootstrap
samples and trees
```



From the plot above, we can see that the error stabilizes around ~40 trees. Thus, we will use this in our final **ipred** model.

```
tuned_model <- bagging(
  formula = diabetes ~ .,
  data = diabetesTrain,
  coob = TRUE,
  nbagg = 40
)
```

```
confusionMatrix(predict(tuned_model, diabetesTest), testLabels)
```

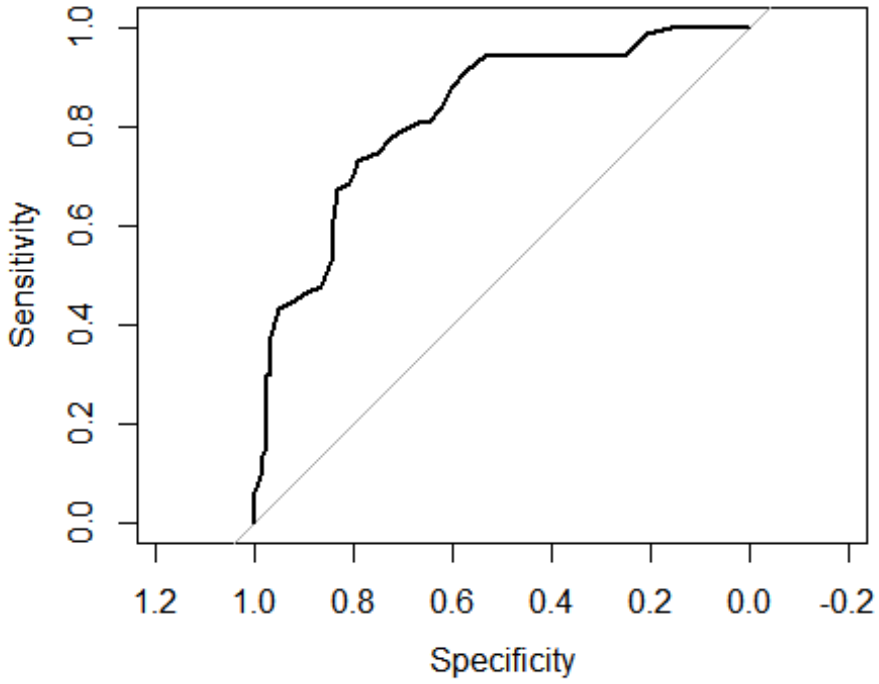
```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction neg pos
##        neg 103  21
##        pos  22  46
##
##                Accuracy : 0.776
##                  95% CI : (0.7104, 0.8329)
##     No Information Rate : 0.651
##     P-Value [Acc > NIR] : 0.0001183
##
##                   Kappa : 0.5088
##
##  Mcnemar's Test P-Value : 1.0000000
##
##             Sensitivity : 0.8240
##             Specificity : 0.6866
##          Pos Pred Value : 0.8306
##          Neg Pred Value : 0.6765
##              Prevalence : 0.6510
##          Detection Rate : 0.5365
##    Detection Prevalence : 0.6458
##       Balanced Accuracy : 0.7553
##
##        'Positive' Class : neg
##
```

Finally, we can plot the ROC curve and calculate the area under the curve for our **ipred** model.

```
tuned_bag_ipred <- predict(tuned_model, diabetesTest, type = "prob")
head(tuned_bag_ipred)
```

```
##        neg   pos
## [1,] 0.425 0.575
## [2,] 0.425 0.575
## [3,] 0.425 0.575
## [4,] 0.300 0.700
## [5,] 0.825 0.175
## [6,] 0.750 0.250
```

```
rocCurve_tuned_ipred <- roc(testLabels, tuned_bag_ipred[,"neg"])
plot(rocCurve_tuned_ipred, col= c(1))
```




```
## Area under the curve: 0.8237
```

Given the preceding results, our model seems to have fit the data moderately well. To see how this translates into future performance, we can use the bagged trees with 10-fold cross-validation using the **caret** package (Lantz, 2015). The method name for the **ipred** bagged trees function is *treebag* (Kuhn, n.d.-a). Below, we preform a 10-fold cross-validated model.

```
set.seed(789) #make results reproducible
ctrl <- trainControl(method = "cv",
                     number = 10,
                     classProbs = TRUE,
                     summaryFunction = twoClassSummary) #10-fold cross
validation

bagged_cv <- train(
  diabetes ~ .,
  data = diabetesTrain,
  method = "treebag",    #'treebag': bagging method for classification
problem in R
  trControl = ctrl,
  metric = "ROC",
  nbagg = 30
)
```

```
bagged_cv
```

```
## Bagged CART
##
## 576 samples
##   8 predictor
##   2 classes: 'neg', 'pos'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 519, 518, 519, 519, 518, 518, ...
## Resampling results:
##
##   ROC        Sens       Spec
##   0.8186848  0.8559033  0.6314286
```

Now, we create the confusion matrix for our cross-validated model.

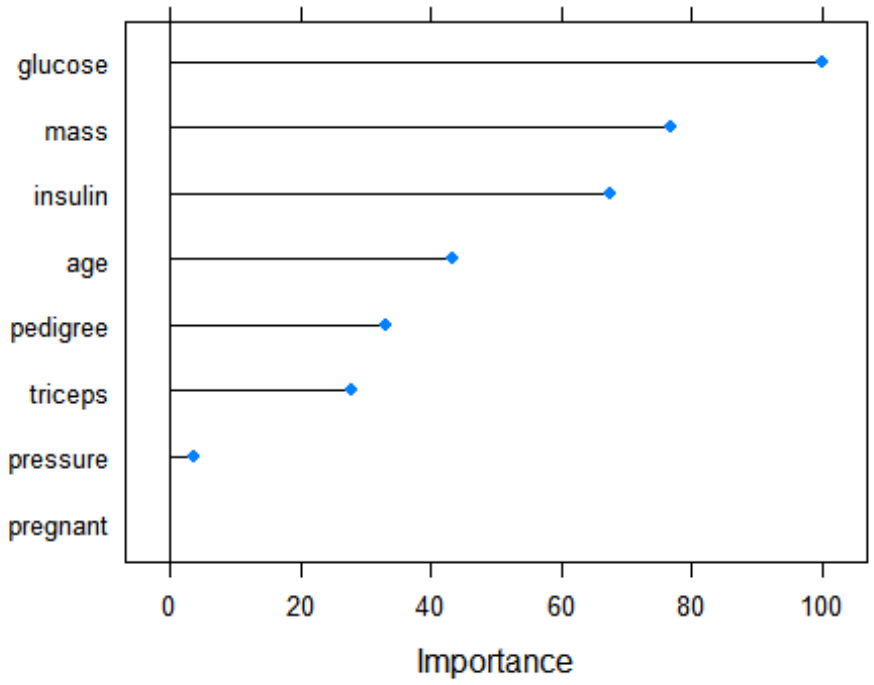
```
bagged_cv_pred <- predict(bagged_cv, diabetesTest)
confusionMatrix(bagged_cv_pred, testLabels)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction neg pos
##        neg 101  23
##        pos  24  44
##
##                Accuracy : 0.7552
##                  95% CI : (0.6881, 0.8143)
##     No Information Rate : 0.651
##     P-Value [Acc > NIR] : 0.001235
##
##                   Kappa : 0.4631
##
##  Mcnemar's Test P-Value : 1.000000
##
##             Sensitivity : 0.8080
##             Specificity : 0.6567
##          Pos Pred Value : 0.8145
##          Neg Pred Value : 0.6471
##              Prevalence : 0.6510
##          Detection Rate : 0.5260
##    Detection Prevalence : 0.6458
##       Balanced Accuracy : 0.7324
##
##        'Positive' Class : neg
##
```

Comparing the output above to our confusion matrix for the **ipred** model, we see that the accuracy and Kappa values are lower. This indicates less accuracy and less agreement between the cross-validated model's predictions and the true values.

We can also assess the top variables from our model.

```
plot(varImp(bagged_cv))
```

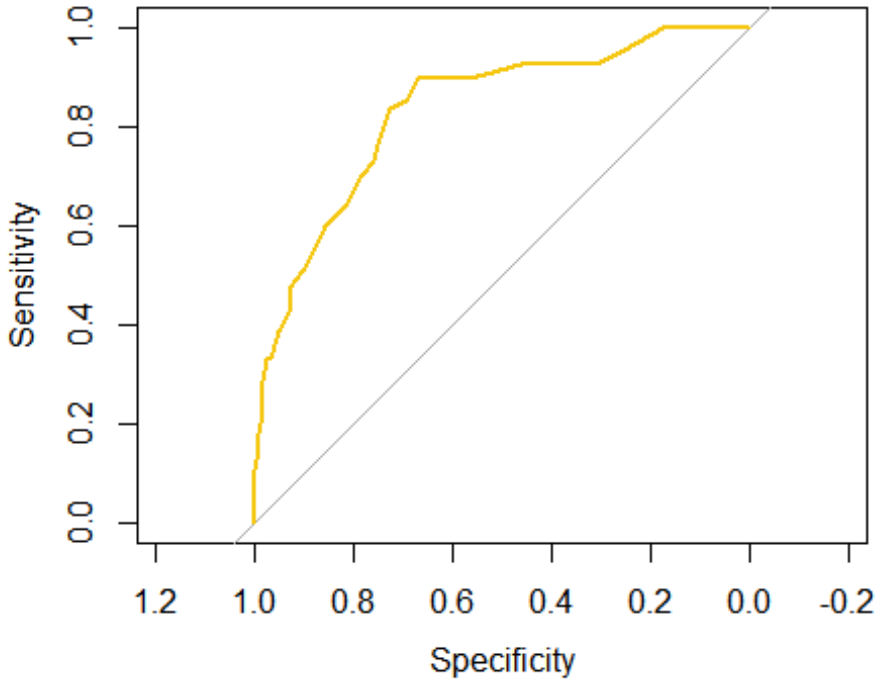


The plot above indicates that the top variable in determining the presence or absence of diabetes is **glucose**, **mass**, and **insulin**. Finally, we can also look at the ROC curve (performance of model) by extracting the probabilities of *neg* (Ensemble methods, n.d.).

```
bagged_probs <- predict(bagged_cv, diabetesTest, type = "prob")
head(bagged_probs)
```

```
##         neg       pos
## 1 0.4000000 0.6000000
## 2 0.4666667 0.5333333
## 3 0.4666667 0.5333333
## 4 0.4000000 0.6000000
## 5 0.6666667 0.3333333
## 6 0.8000000 0.2000000
```

```
rocCurve_bag <- roc(testLabels, bagged_probs[,"neg"])
plot(rocCurve_bag, col= c(7))
```



Finally, we can calculate the area under the ROC curve (Ensemble methods, n.d.).

```
pROC::auc(rocCurve_bag)

## Area under the curve: 0.8336
```

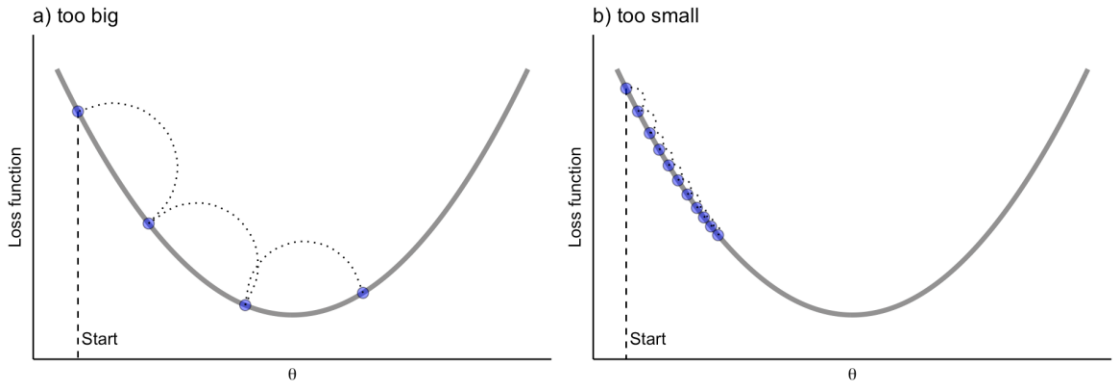
Recall, an *auc* value of 1 indicates that 100% of the predictions were correct.

Boosting Method

Another ensemble learning methods includes *boosting*. The idea behind boosting is to ensemble new models *sequentially*, which "boosts" the performance of the weak model and then continue with the next (Greenwell, n.d.-b). While boosting is a general algorithm for building an ensemble out of simpler modes, it is most effective on models with high bias and low variance. In addition, while boosting can be applied to any type of model, it is most effective when applied to regression trees (Greenwell, n.d.-b). By applying boosting to regression trees, we begin with a decision tree with only a few splits, and then sequentially boosts the performance by continuing to build new tree. Each new tree in the sequence looks to fix previous mistakes (Greenwell, n.d.-b).

One popular machine learning algorithm is *gradient boosting machines (GBMs)*. Gradient boosting machines build ensembles of shallow trees, whereas random forests build an ensemble of deep independent trees (Greenwell, n.d.-b). Shallow tress can be weak predictive models, but can be "boosted" to produce a powerful model. Gradient boosting is considered a *gradient decent* algorithm, which is a generic optimization algorithm capable

of finding optimal solutions to a wide range of problems (Greenwell, n.d.-b). The general idea is to tweak parameters iteratively to minimize a cost function. Gradient descent measures the local gradient of the loss (cost) function for a given set of parameters and takes steps in the direction of the descending gradient (Greenwell, n.d.-b). An important parameter in gradient descent is the size of the steps which is controlled by the *learning rate*. A learning rate that is too small means the algorithm will take many iterations (steps) to find the minimum (Greenwell, n.d.-b). Conversely, if the learning rate is too high, we risk missing the minimum. As seen in the image below (Greenwell, n.d.-b).



It should be noted that not all cost functions are "bowl shaped". There may be local minimas, plateaus, and other irregular terrain that makes finding the global minimum difficult (Greenwell, n.d.-b). To help address this problem, we can use *stochastic gradient descent*, which samples a fraction of the training observations and grows the next tree using that subsample (Greenwell, n.d.-b). While this random sampling adds some random nature in descending the loss function gradient and does not allow the algorithm to find the absolute global minimum, it helps the algorithm jump out of local minima and off plateaus and get near the global minimum (Greenwell, n.d.-b).

GBMs provide several tuning parameters, which allows them to be highly flexible. The main challenge is that they can be time consuming to tune and find the optimal combination of hyperparameters due to the high level of flexibility (Greenwell, n.d.-b). There are two categories of hyperparameters: *boosting* and *tree-specific* hyperparameters. The two main boosting hyperparameters include (Greenwell, n.d.-b):

- Number of trees: The total number of trees in the sequence or ensemble. GBMs often require many trees and can suffer from overfitting. Thus, we seek to find the optimal number of trees that minimizes the loss function.

- Learning rate (shrinkage): This controls how quickly the algorithm proceeds down the gradient descent. Smaller values reduce the chance of overfitting but also increases the time to find the optimal fit. Values range from 0–1 with typical values between 0.001–0.3.

The two main tree-specific hyperparameters include (Greenwell, n.d.-b):

- Tree depth: This controls the depth of the individual trees. Typical values range from a depth of 3–8 but it is not uncommon to see a tree depth of 1 (which is basically a stump).
- Minimum number of observations in terminal nodes: This controls the complexity of each tree. Typical values range from 5–15 where higher values help prevent a model from learning relationships, but smaller values help with imbalanced target classes.

To use the boosting algorithm, we will be using the **caret** package. We will be using the training and testing datasets and labels created above. By default, simple bootstrap resampling is used in the **createDataPartition()** function above (Kuhn, n.d.-b). Others are available such as repeated *K*-fold cross-validation, which we will use (Kuhn, n.d.-b). We can define the type of resampling in the function **trainControl()** below. We will specify the function to use 10-fold cross-validation .

```
fitControl = trainControl(method = "cv",
                          number = 10,
                          classProbs = TRUE,
                          summaryFunction = twoClassSummary)
```

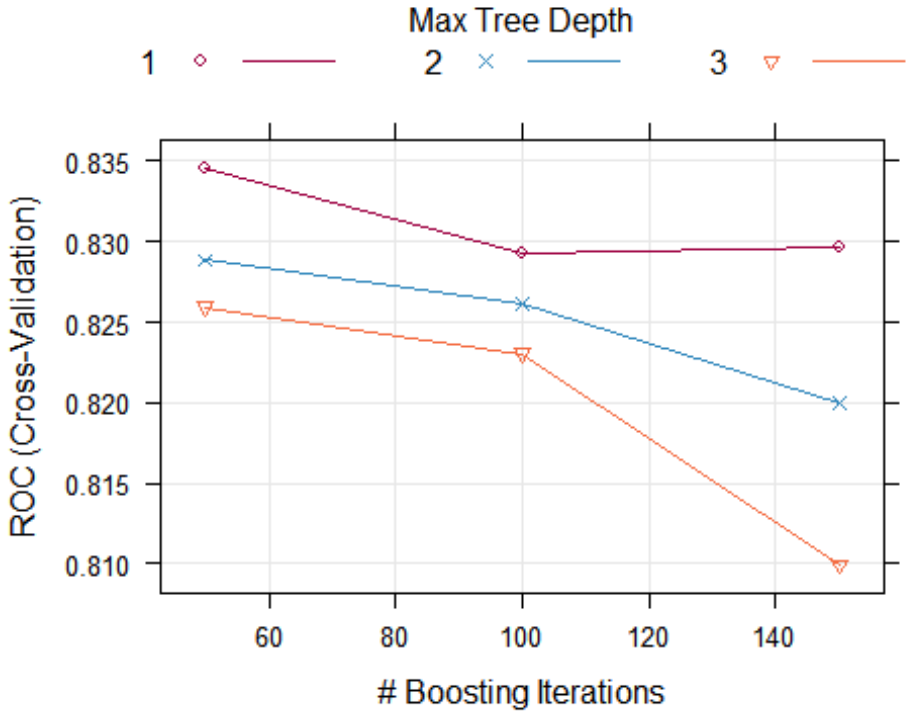
Now, we can fit our boosted model. We use the parameter **method = "gbm"** to use stochastic gradient boosting (Kuhn, n.d.-a).

```
set.seed(789) #make results reproducible
train_time <- system.time({
  gbm_fit1 <- train(diabetes ~ .,
                data = diabetesTrain,
                method = "gbm",
                trControl = fitControl,
                metric = "ROC",
                verbose = FALSE)
})

cat("Train Time (seconds):", train_time, "\n")

## Train Time (seconds): 1.56 0.03 1.59 NA NA

gbm_fit1

## Stochastic Gradient Boosting
##
## 576 samples
##   8 predictor
##   2 classes: 'neg', 'pos'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 519, 518, 519, 519, 518, 518, ...
## Resampling results across tuning parameters:
##
```

```
##   interaction.depth  n.trees  ROC        Sens       Spec
##   1                   50      0.8345772  0.8721906  0.5016667
##   1                  100      0.8292542  0.8667852  0.5466667
##   1                  150      0.8296525  0.8693457  0.5516667
##   2                   50      0.8288361  0.8614509  0.5414286
##   2                  100      0.8261424  0.8534139  0.5709524
##   2                  150      0.8199983  0.8453058  0.5809524
##   3                   50      0.8259336  0.8480797  0.5811905
##   3                  100      0.8230372  0.8346373  0.5661905
##   3                  150      0.8099196  0.8214083  0.5564286
##
## Tuning parameter 'shrinkage' was held constant at a value of 0.1
##
## Tuning parameter 'n.minobsinnode' was held constant at a value of 10
## ROC was used to select the optimal model using the largest value.
## The final values used for the model were n.trees = 50, interaction.depth =
##  1, shrinkage = 0.1 and n.minobsinnode = 10.
```

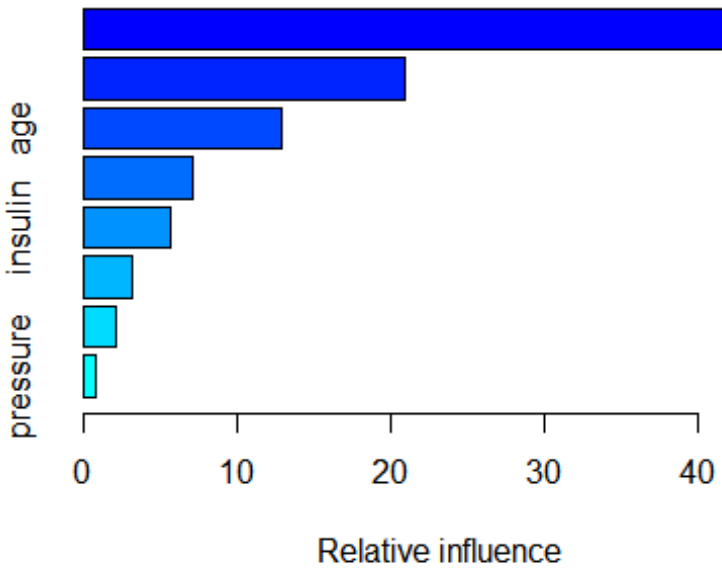
The default values tested for this model are shown in the first two columns: **n.trees** and **interaction.depth**. We can plot our model, which will plot the boosting iterations versus ROC and each color represents the tree depth used.

```
trellis.par.set(caretTheme())
plot(gbm_fit1)
```



From the plot above, the model with highest ROC occurs early on in the boosting iterations and when the max tree depth is 1. To determine the variable importance in this model, we can use the **summary()** function.

```
summary(gbm_fit1)
```




```
##                var    rel.inf
## glucose   glucose 47.0527445
## mass         mass 20.9590669
## age           age 12.8714101
## pedigree pedigree  7.1185600
## insulin   insulin  5.7224691
## pregnant pregnant  3.2360241
## triceps   triceps  2.1780042
## pressure pressure  0.8617213
```

From the output above, the variables with the most influence on predicting diabetes includes **glucose**, **mass**, and **age**. Now, we can create a confusion matrix of our model.

```
gbm1_pred <- predict(gbm_fit1, diabetesTest)
confusionMatrix(gbm1_pred, testLabels)

## Confusion Matrix and Statistics
##
##           Reference
## Prediction neg pos
##        neg 109  24
```

```
##        pos  16  43
##
##                Accuracy : 0.7917
##                  95% CI : (0.7273, 0.8468)
##     No Information Rate : 0.651
##     P-Value [Acc > NIR] : 1.513e-05
##
##                   Kappa : 0.5284
##
##  Mcnemar's Test P-Value : 0.2684
##
##             Sensitivity : 0.8720
##             Specificity : 0.6418
##          Pos Pred Value : 0.8195
##          Neg Pred Value : 0.7288
##              Prevalence : 0.6510
##          Detection Rate : 0.5677
##    Detection Prevalence : 0.6927
##       Balanced Accuracy : 0.7569
##
##        'Positive' Class : neg
##
```

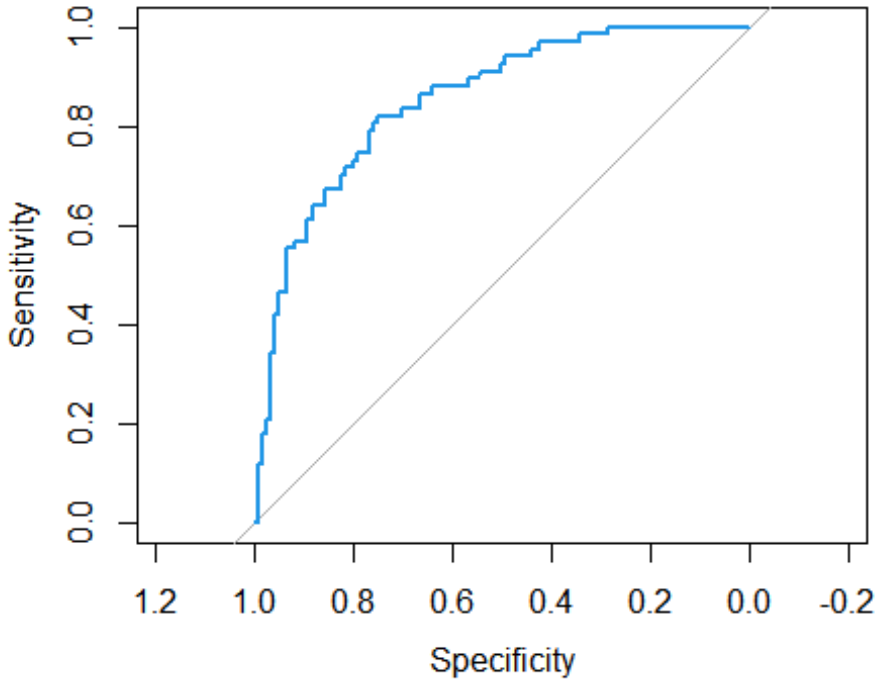
Notice, this model has the highest accuracy rate and the highest Kappa value of all the models thus far.

Now, we can create and ROC curve similar to the ones found above.

```
boosted_probs <- predict(gbm_fit1, diabetesTest, type = "prob")
head(boosted_probs)
```

```
##         neg       pos
## 1 0.2886449 0.7113551
## 2 0.6988645 0.3011355
## 3 0.3455781 0.6544219
## 4 0.2532326 0.7467674
## 5 0.7195886 0.2804114
## 6 0.6482272 0.3517728
```

```
rocCurve_boost1 <- roc(testLabels, boosted_probs[,"neg"])
plot(rocCurve_boost1, col= c(4))
```



Using the ROC information, we can calculate the area under the ROC curve (Ensemble methods, n.d.).

```
pROC::auc(rocCurve_boost1)

## Area under the curve: 0.8514
```

Now, there are a few ways to customize the process of tuning parameters and building the final model. The tuning parameters for this method include the number of trees (**n.trees**), tree depth (**interaction.depth**), learning rate (**shrinkage**), and the minimum number of observations in terminal nodes (**n.minobsinnode**) (Kuhn, n.d.-a). For our model, we will use a tuning parameter grid, where we can test several values for each parameter at once. This does take more time, but after the model runs, we can choose the optimal parameters for our model. Below, we specify our parameter grid.

```
gbm_grid <-  expand.grid(interaction.depth = c(1, 5, 9),
                         n.trees = (1:30)*50,
                         shrinkage = c(.01, .1, .3),
                         n.minobsinnode = c(5, 10, 15))
cat("Number of parameter combinations that will be tested:", nrow(gbm_grid))

## Number of parameter combinations that will be tested: 810
```

We can now use the grid above in our tuning model.

```
train_time <- system.time({
gbm_fit2 <- train(diabetes ~ .,
```

```
                  data = diabetesTrain,
                  method = "gbm",
                  trControl = fitControl,
                  metric = "ROC",
                  tuneGrid = gbm_grid,
                  verbose = FALSE)
})


cat("Train Time (seconds):", train_time, "\n")

## Train Time (seconds): 145.06 0.02 145.51 NA NA

#gbm_fit2
```

Instead of printing the results of the model, we can use either of the commands below to determine the final values for the model. The first command gives more specific information such as ROC, sensitivity, etc.

```
gbm_fit2$results[which.max(gbm_fit2$results$ROC),]

##     shrinkage interaction.depth n.minobsinnode n.trees       ROC      Sens
## 182      0.01                 9              5     100 0.8436532 0.9094595
##          Spec      ROCSD     SensSD     SpecSD
## 182 0.4585714 0.03696555 0.04546119 0.08416591

#gbm_fit2$bestTune
```

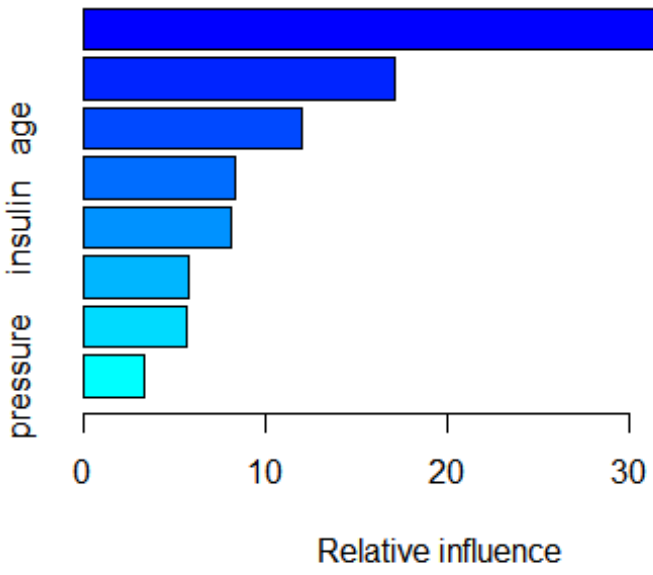
Using the parameters above, we can build our final model.

```
gbm_grid <-  expand.grid(interaction.depth = 9,
                         n.trees = 100,
                         shrinkage = .01,
                         n.minobsinnode = 5)

train_time <- system.time({
gbm_fit3 <- train(diabetes ~ .,
                  data = diabetesTrain,
                  method = "gbm",
                  trControl = fitControl,
                  metric = "ROC",
                  tuneGrid = gbm_grid,
                  verbose = FALSE)
})


cat("Train Time (seconds):", train_time, "\n")

## Train Time (seconds): 1.07 0.01 1.08 NA NA

gbm_fit3
```

```
## Stochastic Gradient Boosting
##
## 576 samples
##   8 predictor
##   2 classes: 'neg', 'pos'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 518, 519, 519, 519, 517, 519, ...
## Resampling results:
##
##   ROC        Sens       Spec
##   0.8360005  0.8938834  0.4780952
##
## Tuning parameter 'n.trees' was held constant at a value of 100
## Tuning
##
## Tuning parameter 'shrinkage' was held constant at a value of 0.01
##
## Tuning parameter 'n.minobsinnode' was held constant at a value of 5
```

To determine the variabls with the most influence on our predictions, we can use the **summary()** function.

```
summary(gbm_fit3)
```



```
##               var   rel.inf
## glucose   glucose 39.672821
## mass         mass 17.114185
## age           age 11.985345
## pedigree pedigree  8.337444
## insulin   insulin  8.111538
## pregnant pregnant  5.777136
## triceps   triceps  5.675555
## pressure pressure  3.325976
```

Again, **glucose**, **mass**, and **age** have the most influence on making predictions. Now, we create the confusion matrix for our model.

```
gbm3_pred <- predict(gbm_fit3, diabetesTest)
confusionMatrix(gbm3_pred, testLabels)
```

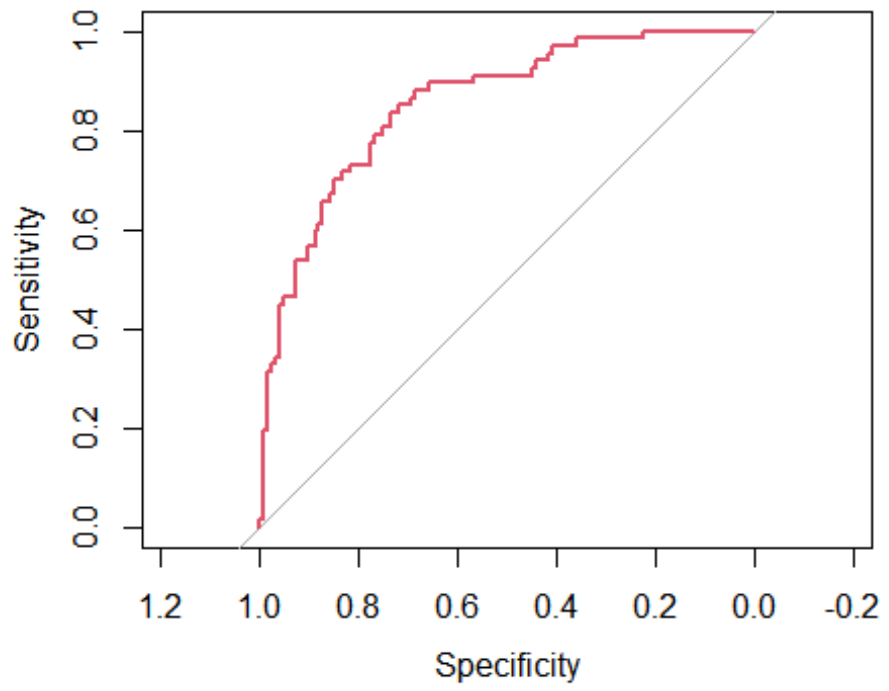
```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction neg pos
##        neg 113  29
##        pos  12  38
##
##                Accuracy : 0.7865
##                  95% CI : (0.7217, 0.8422)
##     No Information Rate : 0.651
##     P-Value [Acc > NIR] : 3.093e-05
##
##                   Kappa : 0.5006
##
##  Mcnemar's Test P-Value : 0.01246
##
##             Sensitivity : 0.9040
##             Specificity : 0.5672
##          Pos Pred Value : 0.7958
##          Neg Pred Value : 0.7600
##              Prevalence : 0.6510
##          Detection Rate : 0.5885
##    Detection Prevalence : 0.7396
##       Balanced Accuracy : 0.7356
##
##        'Positive' Class : neg
##
```

Now, we plot the ROC curve and calculate the area under the curve.

```
boosted_probs3 <- predict(gbm_fit3, diabetesTest, type = "prob")
head(boosted_probs3)
```

```
##         neg       pos
## 1 0.5607863 0.4392137
```

```
## 2 0.6043973 0.3956027
## 3 0.4640083 0.5359917
## 4 0.4044293 0.5955707
## 5 0.6938896 0.3061104
## 6 0.7126650 0.2873350

rocCurve_boost3 <- roc(testLabels, boosted_probs3[,"neg"])
plot(rocCurve_boost3, col= c(10))
```




```
## Area under the curve: 0.8546
```

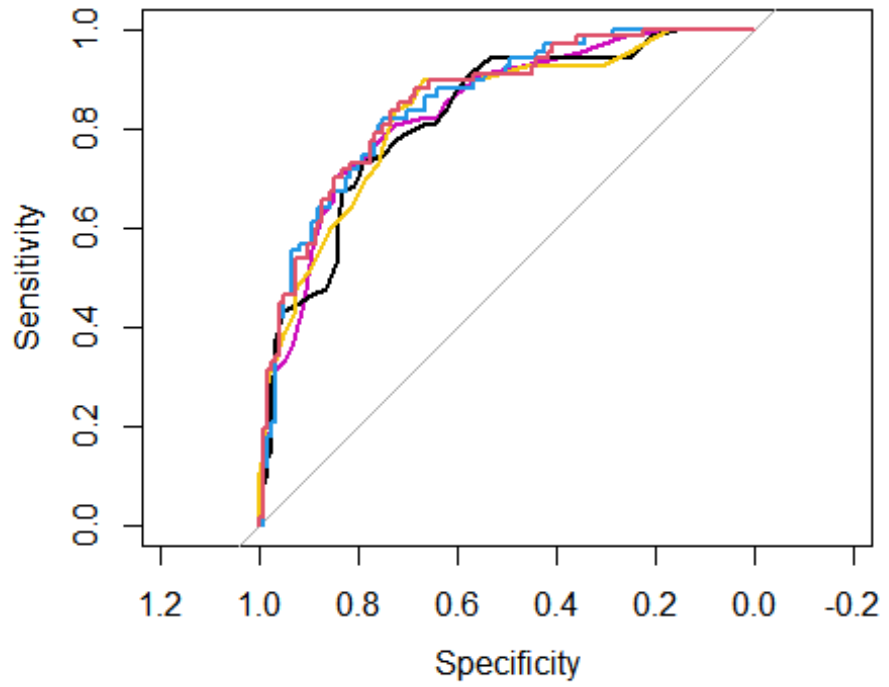
It appears that our tuned GBM model has the best performance of all the models tested throughout this assignment.

Conclusion

For this assignment we used a dataset called *PimaIndiansDiabetes2* and applied various ensemble learning methods to this data. After applying multiple imputation to our missing values and partitioning our data, we used a bagging method to predict the presence or absence of diabetes. Here, we found that our optimal model was the model built under the **caret** package. From here, we proceeded to use a boosting method. Here, we found our optimal model was the tuned gradient boosting machine model built under the **caret** package.

Perhaps the simplest method to compare how each model performed is by comparing the ROC curves and the area under each curve (Ensemble methods, n.d.).

```
plot(rocCurve_ipred, col= c(6)) #default ipred model, color = purple
plot(rocCurve_tuned_ipred, add=TRUE, col= c(1)) #tuned ipred model, color =
black
plot(rocCurve_bag, add=TRUE,col= c(7)) #bagged cv tree, color = yellow
plot(rocCurve_boost1 ,add=TRUE,col= c(4)) #default gbm, color = blue
plot(rocCurve_boost3,add=TRUE, col= c(10)) #tuned gbm, color = red
```



Now, we can calculate the area under the curve for each model.

```
## Bagging Model with ipred: 0.8360597

## Tuned Bagging Model with ipred: 0.8237015

## Bagged CV with Caret Model: 0.8336119

## Default GBM with Caret Model: 0.851403

## Tuned GBM with Caret Model 0.8545672
```

The model that appears to performed the "best" is our tuned gradient boosted machine model built using the **caret** package.

The goal of bagging is to improve the prediction accuracy for high variance, and low bias, models (Greenwell, n.d.-a). This is where decision trees suffer. However, when bagging trees, the trees are not completely independent of each other since all the original features

are considered at each split. Bagging provides the fundamental basis for more complex and powerful algorithms such as gradient boosting (Greenwell, n.d.-a). Gradient boosting machines are one of the powerful ensemble algorithms. While they are less intuitive and more computationally demanding than bagging and many other machine learning algorithms, they are essential (Greenwell, n.d.-b). From the results above, boosting appears to provide a better increase in accuracy than bagging.

Resources

Chiu, Y. (2015). Machine Learning with R cookbook. Packt Publishing (Chapter 8)

Diabetes of indian people. (n.d.). Retrieved November 16, 2020, from https://rdrr.io/github/quantide/qdata/man/pimaindiansdiabetes2.html

Ensemble learning. (2018, June 18). Analytics Vidhya. https://www.analyticsvidhya.com/blog/2018/06/comprehensive-guide-for-ensemble-models/

Ensemble methods. (n.d.). Retrieved November 16, 2020, from https://quantdev.ssri.psu.edu/sites/qdev/files/09_EnsembleMethods_2017_1127.html

Greenwell, B. B. & B. (n.d.-a). Chapter 10 bagging | hands-on machine learning with r. Retrieved November 16, 2020, from https://bradleyboehmke.github.io/HOML/bagging.html

Greenwell, B. B. & B. (n.d.-b). Chapter 12 gradient boosting | hands-on machine learning with r. Retrieved November 16, 2020, from https://bradleyboehmke.github.io/HOML/gbm.html

Imputing missing data with r. (2018, May 14). DataScience+. https://datascienceplus.com/imputing-missing-data-with-r-mice-package/

Kuhn, M. (n.d.-a). 5 model training and tuning | the caret package. Retrieved November 16, 2020, from https://topepo.github.io/caret/model-training-and-tuning.html

Kuhn, M. (n.d.-b). 7 train models by tag | the caret package. Retrieved November 16, 2020, from http://topepo.github.io/caret/train-models-by-tag.html

Lantz, B (2015). Machine Learning with R - second Edition, Packt Publishing. (Chapter 11)