Binary Classification of the Wisconsin Breast Cancer Dataset

The objective of this project is to predict whether a cancer is malignant or benign from biopsy details.

This dataset is from the UCI Machine Learning Repository. For this binary classification project, I will be exploring, implementing, and comparing Random Forest, Gradient Boosting, and Logistic Regression.

```
library(dplyr)
library(mlbench)
library(randomForest)
library(gbm)
library(nnet)
data("BreastCancer")
BreastCancer_Raw<-BreastCancer #Copy of the raw data
set.seed(2928892)
```

We will explore the structure of the data, as well as remove the 16 instanes of N/A in the data.

```
BreastCancer<- na.omit(BreastCancer_Raw)
#str(BreastCancer)
```

The response variable is 'Class' which is a Factor with 2 levels - "benign" or "malignant"
The Id can column can be removed as it is redundant for the classification.
The response variable can also be move to the first column instead of the last. The variables that are in the data are:
1. Clump Thickness: 1 - 10 2. Uniformity of Cell Size: 1 - 10
3. Uniformity of Cell Shape: 1 - 10
4. Marginal Adhesion: 1 - 10
5. Single Epithelial Cell Size: 1 - 10
6. Bare Nuclei: 1 - 10
7. Bland Chromatin: 1 - 10
8. Normal Nucleoli: 1 - 10
9. Mitoses: 1 - 10

```
head(BreastCancer)
```

```
##       Class Cl.thickness Cell.size Cell.shape Marg.adhesion Epith.c.size
## 1    benign            5         1          1             1            2
## 2    benign            5         4          4             5            7
## 3    benign            3         1          1             1            2
## 4    benign            6         8          8             1            3
## 5    benign            4         1          1             3            2
## 6 malignant            8        10         10             8            7
##   Bare.nuclei Bl.cromatin Normal.nucleoli Mitoses
## 1           1           3               1       1
## 2          10           3               2       1
## 3           2           3               1       1
## 4           4           3               7       1
## 5           1           3               1       1
## 6          10           9               7       1
```

Random Forest

Random forests for classification are mostly the same as when we use them for regression. In addition to utilizing boostrap resampling, Random Forest also randomly selects a random subset of explanatory variables for each parent node. By default, it is sqaure root of the number of explanatory variables, and this is a parameter that could be tuned. Random Forest will make a forest of classification trees and uses a voting system. The objective of splits is to minimize the misclassification rate in terminal nodes. In classification, variable importance is calculated by the proportion in reduction of the Gini Index.

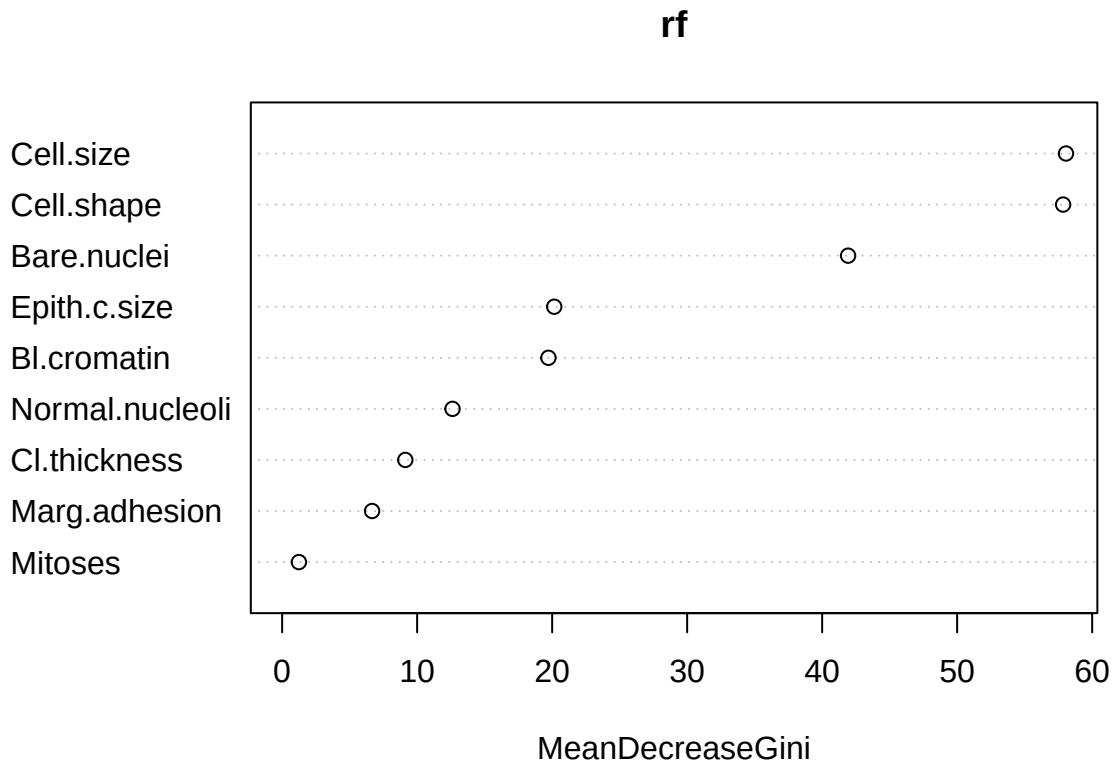
```
### Split the data into training and validation sets
p.train = 0.75
n = nrow(BreastCancer)
n.train = floor(p.train*n)
ind.random = sample(1:n)
data.train.rf = BreastCancer[ind.random <= n.train,]
data.valid.rf = BreastCancer[ind.random > n.train,]
Y.train.rf = data.train.rf$Class
Y.valid.rf = data.valid.rf$Class

rf<-randomForest(Class~.,data=data.train.rf)
rf$err.rate[500,1]*100
```

```
##      OOB
## 2.539062
```

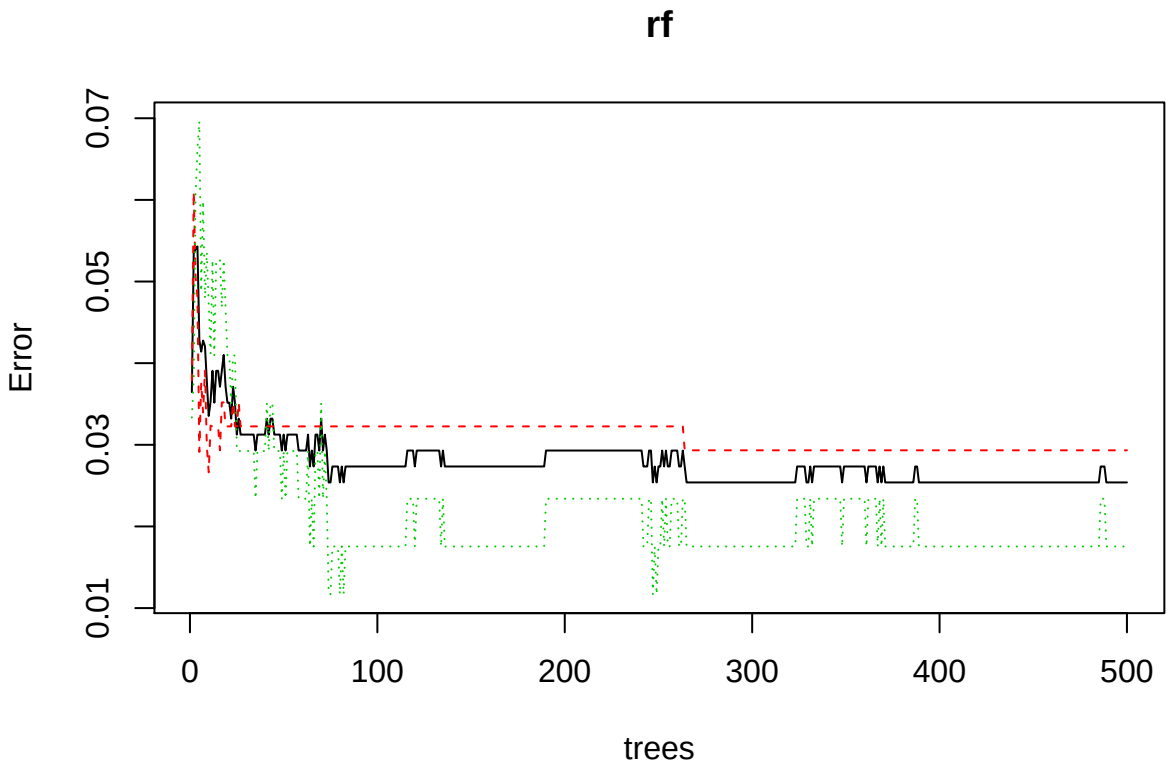
```
rf$confusion
```

```
##           benign malignant class.error
## benign       331        10  0.02932551
## malignant      3       168  0.01754386
```

```
varImpPlot(rf)
```


rf
Cell.size
Cell.shape
Bare.nuclei
Epith.c.size
Bl.cromatin
Normal.nucleoli
Cl.thickness
Marg.adhesion
Mitoses
0
10
20
30
40
50
60
MeanDecreaseGini

```
plot(rf)
```

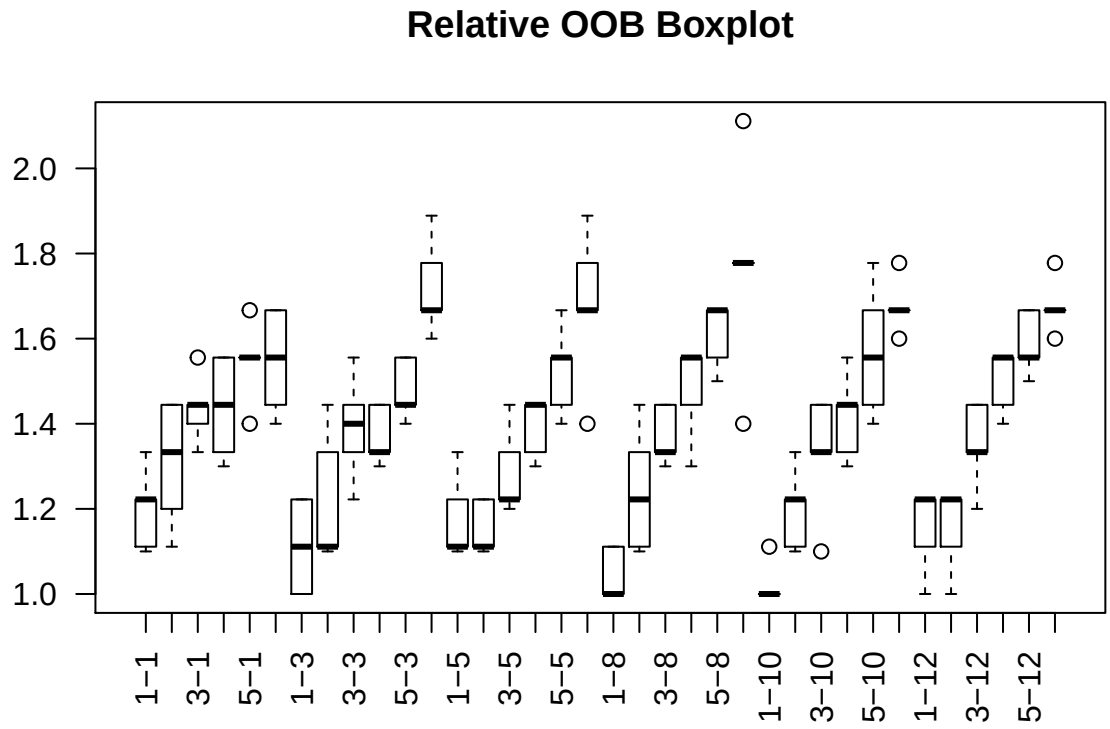


The default Random Forest model produces an Out-Of-Bag estimate of error rate of 2.54%, which is good. From the first plot, we can see that cell size, cell shape, and bare nuclei are the most implortant variables. From the second plot we can see that 300 trees should suffice (Default is 500). We will utilize tuning to try to improve our model. We will use out-of-bag error to tune. The tuning parameters are mtry(number of variables randomly selected at each split) and nodesize.

```
all.mtrys = 1:6
all.nodesizes = c(1, 3, 5, 8, 10,12)
all.pars.rf = expand.grid(mtry = all.mtrys, nodesize = all.nodesizes)
n.pars = nrow(all.pars.rf)

M = 5 # Number of times to repeat RF fitting. I.e. Number of OOB errors

### Container to store OOB errors. This will be easier to read if we name
### the columns.
all.OOB.rf = array(0, dim = c(M, n.pars))
names.pars = apply(all.pars.rf, 1, paste0, collapse = "-")
colnames(all.OOB.rf) = names.pars

for(i in 1:n.pars){
  ### Get tuning parameters for this iteration
  this.mtry = all.pars.rf[i, "mtry"]
  this.nodesize = all.pars.rf[i, "nodesize"]

  for(j in 1:M){
    ### Fit RF, then get and store OOB errors
    this.fit.rf = randomForest(Class ~ ., data = data.train.rf,
```

```
      mtry = this.mtry, nodesize = this.nodesize)

    pred.this.rf = predict(this.fit.rf)
    this.err.rf = mean(Y.train.rf != pred.this.rf)

    all.OOB.rf[j, i] = this.err.rf
  }
}
```

```
#boxplot(all.OOB.rf, las=2,main = "OOB Boxplot")
all.OOB.rf=all.OOB.rf
rel.OOB.rf = apply(all.OOB.rf, 1, function(W) W/min(W))
boxplot(t(rel.OOB.rf), las=2,  # las sets the axis label orientation
  main = "Relative OOB Boxplot")
```



A sensible choice seems to be mtry = 2 and nodesize = 5.

```
fit.rf = randomForest(Class ~ ., data = data.train.rf,
  mtry = 1, nodesize = 12,n.trees=300)

pred.rf = predict(fit.rf, data.valid.rf)

table(Y.valid.rf, pred.rf, dnn = c("Obs", "Pred"))
```

```
##            Pred
## Obs         benign malignant
```

```
##   benign        99         4
##   malignant      1        67
```

```
100-(mis.rf = mean(Y.valid.rf != pred.rf))*100 #Accuracy calculation
```

```
## [1] 97.07602
```

The final model with mtry=1, nodesize=3 and n.trees=300 was quite accurate on the test data with accuracy = 97.66

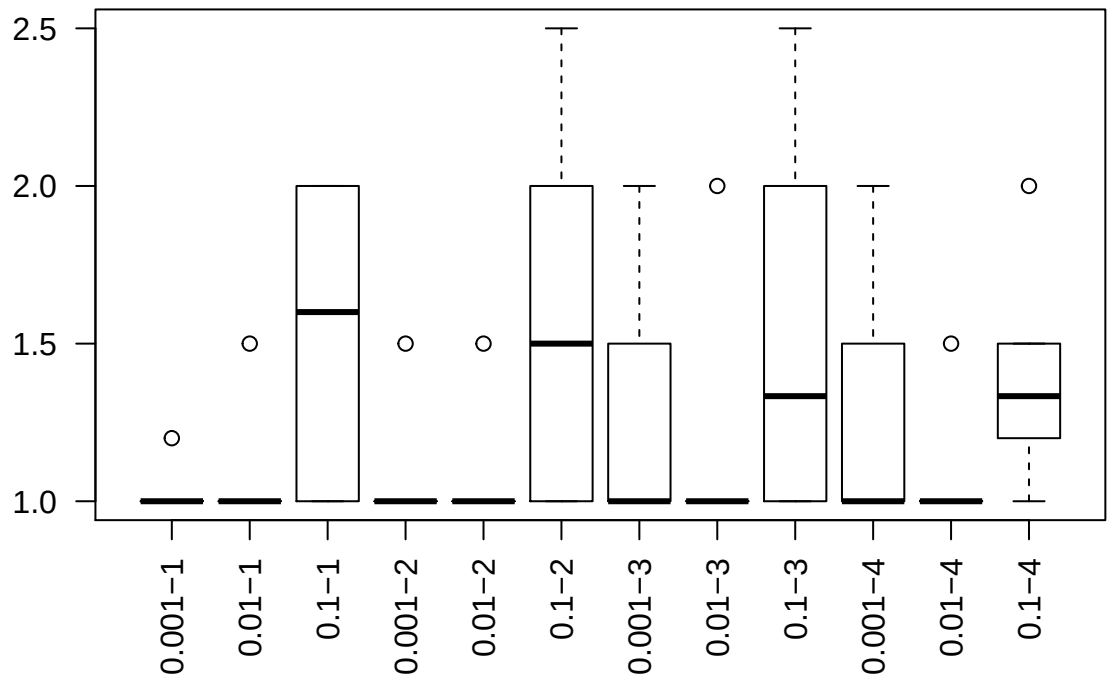
Gradient Boosting

Next we will utilize Gradient Boosting. There is a version of gradient boosting that does classification and regression. Gradient boosting works by finding local optimal weight coefficients for sequentially built decision trees by locally minimzing the sum of squares errors. Since the response variable is a factor in classification, it will build trees that explain the the optimization criterion which usually is a deviance criteria resembling a log-likelihood. Gradient boosting requires careful tuning of parameters. I will be utilizing 5-fold cross validation to tune combinations of the shrinkage and depth parameters.

```
max.trees = 10000
all.shrink = c(0.001, 0.01, 0.1)
all.depth = c(1, 2, 3)
all.pars.boost = expand.grid(shrink = all.shrink, depth = all.depth)
n.pars = nrow(all.pars.boost)

data.boost = BreastCancer
class.boost.fact = factor(data.boost$Class, levels = c("benign", "malignant"))
class.boost.num = as.numeric(class.boost.fact) - 1
data.boost$Class = class.boost.num

data.train.boost = data.boost[ind.random <= n.train,]
data.valid.boost = data.boost[ind.random > n.train,]
Y.train.boost = data.train.boost$Class
Y.valid.boost = data.valid.boost$Class

### We will stick to resampling rate of 0.8, maximum of 10000 trees, and ROT
### for choosing how many trees to keep.
max.trees = 10000
all.shrink = c(0.001, 0.01, 0.1)
all.depth = c(1, 2, 3, 4)
all.pars.boost = expand.grid(shrink = all.shrink, depth = all.depth)
n.pars = nrow(all.pars.boost)

### Number of folds
K = 5

### Get folds
n = nrow(data.train.boost)
folds = get.folds(n, K)

### Create container for CV MSPEs
CV.MSPEs = array(0, dim = c(K, n.pars))
names.pars = apply(all.pars.boost, 1, paste0, collapse = "-")
colnames(CV.MSPEs) = names.pars
```

```r
for(i in 1:K){
  ### Print progress update
  #print(paste0(i, " of ", K))

  ### Split data
  data.train.inner = data.train.boost[folds != i,]
  data.valid.inner = data.train.boost[folds == i,]
  Y.valid.inner = data.valid.inner$Class

  ### Fit boosting models for each parameter combination
  for(j in 1:n.pars){
    ### Get current parameter values
    this.shrink = all.pars.boost[j,"shrink"]
    this.depth = all.pars.boost[j,"depth"]

    ### Fit model using current parameter values.
    fit.gbm = gbm(Class ~ ., data = data.train.inner,
      distribution = "bernoulli", n.trees = max.trees,
       interaction.depth = this.depth, shrinkage = this.shrink,
      bag.fraction = 0.8)

    ### Choose how many trees to keep using ROT.
    n.trees = suppressMessages(gbm.perf(fit.gbm, plot.it = F) * 2)

    ### Check to make sure that ROT doesn't tell us to use more than 1000
    ### trees. If it does, add extra trees as necessary
    if(n.trees > max.trees){
      extra.trees = n.trees - max.trees
      fit.gbm = gbm.more(fit.gbm, extra.trees)
    }

    pred.gbm.prob = predict(fit.gbm, data.valid.inner, n.trees,
    type = "response")
    pred.gbm = round(pred.gbm.prob, 0)
    mis.gbm = mean(Y.valid.inner != pred.gbm)

    CV.MSPEs[i, j] = mis.gbm
  }
}

CV.RMSPEs = apply(CV.MSPEs, 1, function(W) W/min(W))
```

```r
boxplot(t(CV.RMSPEs), las = 2, main = "RMSPE Boxplot")
```




```
#Interaction.depth=1 and shrinkage = 0.1 is a sensible choice from the RMSPE Boxplot.

gbm = gbm(Class ~ ., data = data.train.boost,
      distribution = "bernoulli", n.trees = 10000,
       interaction.depth = 2, shrinkage = 0.01,
      bag.fraction = 0.8)
n.trees = suppressMessages(gbm.perf(gbm, plot.it = F) * 2) #Take number of treess as ROT

pred.gbm.prob = predict(gbm, data.valid.boost, n.trees,
  type = "response")
pred.gbm = round(pred.gbm.prob, 0)

table(Y.valid.boost, pred.gbm, dnn = c("Obs", "Pred"))
```

```
##    Pred
## Obs   0   1
##   0 101   2
##   1   3  65
```

```
100 - (mis.boost = mean(Y.valid.boost != pred.gbm))*100 #Accuracy Calculation
```

```
## [1] 97.07602
```

The tuned Gradient Boosting Model produced great results and outperformed Random Forest with the Accuracy = 98.25%.

Logistic Regression

Logistic Regression is a linear method for classification. It assumes a linear model for log-odds of success and uses maximum likelihood to estimate parameters and create hyperplanes boundaries between classes.

```
data.train.scale = data.train.rf
data.valid.scale = data.valid.rf
fit.log.nnet = multinom(Class ~ ., data = data.train.rf, maxit = 200)
```

```
## # weights:  82 (81 variable)
## initial  value 354.891356
## iter  10 value 11.190839
## iter  20 value 0.153044
## iter  30 value 0.000860
## final  value 0.000075
## converged
```

```
pred.log.nnet = predict(fit.log.nnet, data.valid.scale)

table(Y.valid.rf, pred.log.nnet,        ### Confusion matrix
  dnn = c("Observed", "Predicted"))
```

```
##            Predicted
## Observed    benign malignant
##   benign       100         3
##   malignant      8        60
```

```
100-(misclass.log.nnet = mean(pred.log.nnet != Y.valid.rf))*100
```

```
## [1] 93.56725
```

The Logistic model had Accuracy = 94.15%, so it still performed quite well, but did not perform as well as the other 2 models.

Conclusion

The Random Forest and Gradient Boosting models performed very, boosting provding better results having an accuracy of 98.25% when tuned. The Logistic model also had a good result, but not as good of a result as the first 2 models. Continuous optimization methods play a significant role behind the scenes in many predictive machines, and allows for powerful and accurate predictions to be made to improve the world around us.