Machine Learning for Finance Applications - Ensemble Methods

Brian Clark

2019

Contents

1 Introduction

This vignette introduces ensembe methods for finance applications. We will use decision trees as the underlying method but they could be applied to most any ML algorithm. The topics closely follow Chapter 8 of James et al. (2013).[1] Another resource is Chapters 9 and 10 of Friedman, Hastie, and Tibshirani (2001), which provides a more technical discussion.[2]

Because of the high variance nature of tree-based methods, they are often used as a base model with advanced methods overlaid. For example, the stability and performance of decision tree method can be enhanced by techniques such as bagging, random forests, and boosting. As with many ML techniques, these enhancements come at a cost in terms of computation time so the most appropriate model tends to be situation specific. These ensemble methods are the focus of this chapter.

2 Model Enhancements for Decision Trees

Like many other algorithms, decision trees can be improved using several techniques are common across ML applications. There are three main enhancements which are as follows (see Chapter 8 of James et al. (2013)):

1. Bagging
2. Random Forests
3. Boosting

[1] The full text is available at http://www-bcf.usc.edu/ gareth/ISL/.
[2] The full text is available at https://web.stanford.edu/ hastie/Papers/ESLII.pdf.

In this section, we will demonstrate each of these features, which are readily accesible using the `caret` package.

2.1 Bagging

Bagging is an application of bootstrapping. It involves selecting random samples and averaging the results. The basic idea stems from the Central Limit Theorem whereby variance of the mean of a set of observations decreases linearly with the number of observations n being averaged. In the context of decision tree models, we can train B trees $(\hat{f}^1(x), \hat{f}^2(x), ..., \hat{f}^B(x))$ using B separate training sets. In reality, we don't have B training sets so we repeatedly sample from a single training set and then average the predictions to get a single output,

$$\hat{f_{bag}}(x) = \frac{1}{B}\sum_{b=1}^{B}\hat{f^{*b}}(x). \tag{1}$$

For the case of decision trees, the aggregation process depends on the nature of the problem. For the case of regression trees, the predictions for each instance can be the average of the predicted values across the bagged trees. For classification problems, a different rule can be applied. The most simplistic rule would be to take the majority vote of across the models (e.g., classify based on the most frequently predicted class). More advanced rules - possibly probabilistic could also be used.

Of course a disadvantage of bagging is that there are now B trees so the nice interpretability feature of the goes away. However, we can still compute the variable importance scores for each feature. For regression trees, one could compute the change in the residual sum of squares. For classification trees, we could compute the drop in the Gini index (CART) or entropy information gain (C5.0) which would occur if we were to omit a given variable.

2.1.1 Bagging Example using CART Trees

The `caret` package has several bagging routines. A great resource for the `caret` package can be found **here**. For illustartive purposes, we will use the CART algorithm which has an associated bagging routine called that can be called using the `method = 'bag'`. The `bagControl` function contains options to control the method.

2.2 Random Forests

Random forests are similar to bagging with one key difference that has the intention of reducing the correlation between the trees. Random forests still work on randomly drawn samples of the training data. The difference from bagging is that for each node, only a subsample of the p features are considered as potential splitting variables. James et al. (2013) suggests that a typical value of parameters is $m \approx \sqrt{p}$. If we set $m = p$ then it is equivalent to bagging. The reason that random forests work is that the trees tend to look very different as compared to bagged trees. Random forests can be applied to regression or classification to different tree algorithms (e.g., CART of C5.0).

2.3 Boosting

Boosting is a third enhancement to decision tree models. The main difference between boosting and bagging is that boosted trees are build sequentially. The boosted trees are not build upon bootstrapped samples, rather they are build on a modified version of the original data. The method works by inferring information on the residuals at each step and using this information to improve the next iteration.

A basic algorithm is given on page 323 of James et al. (2013):

1. Set $\hat{f}(x) = 0$ and $r_i = y_i$ for all $i \in X$.

2. For $b = 1, 2, ..., B$, repeat:
   (a) Fit a tree $\hat{f}^b(x)$ with d splits to the training data (X, r).
   (b) Update $\hat{f}$ by adding in a shrunken version of the new tree:
   $$\hat{f}(x) \leftarrow \hat{f}(x) + \lambda \hat{f}^b(x). \tag{2}$$
   (c) Update the residuals,
   $$r_i \leftarrow r_i - \lambda \hat{f}^b(x_i). \tag{3}$$

3. Output the boosted model,
$$\hat{f}(x) = \sum_{b=1}^{B} \lambda \hat{f}^b(x). \tag{4}$$

There are three tuning parameters in terms of boosting decision trees:

1. The number of trees, B, which can be chosen using cross-validation. Boosting can overfit the data is B is too large.

2. The shrinkage parameter λ which controls the learning rate. Smaller values mean the algorithm learns slower because each sequential tree $\hat{f}^b(x)$ is less influential (see step 2 of the above algorithm).

3. The number of splits per tree, d. Typically a small value works best.

2.4 Mortgage Data

We will use the Fannie Mae mortgage data to test the CART method. The CART model can be implemented using the **rpart** package in R. Another resource to understand CART is **here**. The `rpart` documentation is **here**.

First, initialize the R session and load the data.

```
rm(list=ls()) # clear the memory
setwd("C:/Users/CLARKB2/Documents/Classes/ML Course")
library("ggplot2")
library("reshape")
library("plm")
library("rpart")
library("zoo")
library("plyr")
library("dplyr")
library("stringr")
library("reshape2")
library("ggplot2")
library("pander")
library("DataCombine")
library("plm")
```

```
library("quantmod")

# Import the mortgage data:
load("Mortgage_Annual.Rda")
```

The next step is to process the data. Within the .Rda file is a data frame called `p.mort.dat.annual`. As a matter of preference, rename `p.mort.dat.annual` as `df`. Set the data as a panel dataset (`pdata.frame()`) based on `LOAN_ID` and `year`.

```
# Rename the data (matter of preference):
df <- p.mort.dat.annual
rm(p.mort.dat.annual)

df <- pdata.frame(df, index=c("LOAN_ID","year"),
                  stringsAsFactors = F)

# Print the class of the variable:
class(df)
```

```
## [1] "pdata.frame" "data.frame"
```

Next, we can generate variables that we need. First, define default as the first instance of 180+ days delinquent, which is given in the data by `F180_DTE` (to refer to the variable in `df`, use `df$F180_DTE`). `F180_DTE` is the date in which a loan first becomes 180+ delinquent. If the loan never becomes delinquent, it is missing (i.e., `NA`). To make the default variable, first find the indices where `df$F180_DTE == df$date` and save it as a vector, `tmp`. The line `df$def[tmp] <- 1` sets the new variable `def = 1` for the year in which the loan defaults.

```
# Generate Variables we want:
# 1. Default 1/0 indicator (180+ DPD):
df$def <- 0
# Save the indices (rows) of
tmp <- which(df$F180_DTE == df$date)
df$def[tmp] <- 1
```

We may want to generate some other variables. For example, the variable `NUM_UNIT` gives the number of units is a house. Use `table(df$NUM_UNIT)` to print a frequency table of values of the number of units per house. Then, define a new variable to be a dummy if the number of units is greater than one (`MULTI_UN`).

```
# 2. Replace NUM_UNIT with MULTI_UNIT dummy:
table(df$NUM_UNIT)
```

```
##
##      1      2      3      4
## 305028   6452   1051   1085
```

```
df$MULTI_UN <- 0
tmp <- which(df$NUM_UNIT > 1)
df$MULTI_UN[tmp] <- 1
```

Finally, we can compress the data down to a single observation per loan. If you wanted to conduct a time-series or panal data analysis, you would skip this step. First, print the number of unique loans.

```
# 3. Count the number of loans:
print(length(unique(df$LOAN_ID)))
```

```
## [1] 66704
```

Next, compress the data down to a single observation per loan. First, we need to generate a variable equal to

one if the loan ever defaulted. We can do this using the `plm` package and grouping the data by `LOAN_ID`. Make a new dataset `df.annual` with a few additional variables: i) `def.max` and ii) `n` which is the row number to be used for keeping observaitons.

```
# Compress the data to single loans:
df.annual <-df %>%
  group_by(LOAN_ID) %>%
  mutate(def.max = max(def)) %>%
  mutate(n = row_number()) %>%
  ungroup()
```

```
## Warning: `as_dictionary()` is soft-deprecated as of rlang 0.3.0.
## Please use `as_data_pronoun()` instead
## This warning is displayed once per session.
```

```
## Warning: `new_overscope()` is soft-deprecated as of rlang 0.2.0.
## Please use `new_data_mask()` instead
## This warning is displayed once per session.
```

```
## Warning: The `parent` argument of `new_data_mask()` is deprecated.
## The parent of the data mask is determined from either:
##
##   * The `env` argument of `eval_tidy()`
##   * Quosure environments when applicable
## This warning is displayed once per session.
```

```
## Warning: `overscope_clean()` is soft-deprecated as of rlang 0.2.0.
## This warning is displayed once per session.
```

```
# Print the variable names in df.annual
names(df.annual)
```

```
##  [1] "year"               "ZIP_3"              "V1"
##  [4] "LOAN_ID"            "ORIG_CHN"           "Seller.Name"
##  [7] "ORIG_RT"            "ORIG_AMT"           "ORIG_TRM"
## [10] "ORIG_DTE"           "FRST_DTE"           "OLTV"
## [13] "OCLTV"              "NUM_BO"             "DTI"
## [16] "CSCORE_B"           "FTHB_FLG"           "PURPOSE"
## [19] "PROP_TYP"           "NUM_UNIT"           "OCC_STAT"
## [22] "STATE"              "MI_PCT"             "Product.Type"
## [25] "CSCORE_C"           "MI_TYPE"            "RELOCATION_FLG"
## [28] "Monthly.Rpt.Prd"    "Servicer.Name"      "LAST_RT"
## [31] "LAST_UPB"           "Loan.Age"           "Months.To.Legal.Mat"
## [34] "Adj.Month.To.Mat"   "Maturity.Date"      "MSA"
## [37] "Delq.Status"        "MOD_FLAG"           "Zero.Bal.Code"
## [40] "ZB_DTE"             "LPI_DTE"            "FCC_DTE"
## [43] "DISP_DT"            "FCC_COST"           "PP_COST"
## [46] "AR_COST"            "IE_COST"            "TAX_COST"
## [49] "NS_PROCS"           "CE_PROCS"           "RMW_PROCS"
## [52] "O_PROCS"            "NON_INT_UPB"        "REPCH_FLAG"
## [55] "TRANSFER_FLAG"      "CSCORE_MN"          "ORIG_VAL"
## [58] "PRIN_FORG_UPB"      "MODTRM_CHNG"        "MODUPB_CHNG"
## [61] "Fin_UPB"            "modfg_cost"         "C_modir_cost"
## [64] "C_modfb_cost"       "Count"              "LAST_STAT"
## [67] "lpi2disp"           "zb2disp"            "INT_COST"
## [70] "total_expense"      "total_proceeds"     "NET_LOSS"
## [73] "NET_SEV"            "Total_Cost"         "Tot_Procs"
```

```
## [76] "Tot_Liq_Ex"          "LAST_DTE"            "FMOD_DTE"
## [79] "FMOD_UPB"            "FCE_DTE"             "FCE_UPB"
## [82] "F180_DTE"            "F180_UPB"            "VinYr"
## [85] "ActYr"               "DispYr"              "MODIR_COST"
## [88] "MODFB_COST"          "MODTOT_COST"         "d.HPI"
## [91] "date"                "n"                   "n.obs"
## [94] "n.year"              "n.year.max"          "def"
## [97] "MULTI_UN"            "def.max"
```

Finally, we can save only one observation per loan.

```
# keep one obs per loan:
tmp <- which(df.annual$n == 1)
df.annual <- df.annual[tmp,]
dim(df.annual)
```

```
## [1] 66704    98
```

Notice that the number of rows is equal to the number of unique loans as shown above. Now, retain only the variables needed for the analysis.

```
# Keep only relevant variables for default analysis:
my.vars <- c("ORIG_CHN","ORIG_RT",
             "ORIG_AMT","ORIG_TRM","OLTV",
             "DTI","OCC_STAT",
             "MULTI_UN",
             "CSCORE_MN",
             "ORIG_VAL",
             "VinYr","def.max")
df.model <- subset(df.annual,select=my.vars)
names(df.model)
```

```
##  [1] "ORIG_CHN"  "ORIG_RT"   "ORIG_AMT"  "ORIG_TRM"  "OLTV"
##  [6] "DTI"       "OCC_STAT"  "MULTI_UN"  "CSCORE_MN" "ORIG_VAL"
## [11] "VinYr"     "def.max"
```

```
# Print the number of defaults/non-defaults
table(df.model$def.max)
```

```
##
##     0     1
## 64397  2307
```

```
tmp <- table(df.model$def.max)
df.rate <- tmp[2]/sum(tmp)*100
message(sprintf("The default rate is: %4.2f%%",df.rate))
```

```
## The default rate is: 3.46%
```

The last line prints the default rate. You can control the formatting just as you would in Matlab.

```
# Print the objects in memory:
ls()
```

```
## [1] "df"         "df.annual" "df.model"  "df.rate"   "my.vars"   "tmp"
```

```
# Remove all but df.model
rm(list=setdiff(ls(), "df.model"))
ls()
```

```
## [1] "df.model"
```

Now we have the data almost ready to use. We can downsample the data so the model results will look a bit nicer.

```
library("caret")
```

```
## Loading required package: lattice
```

```
head(df.model)
```

```
## # A tibble: 6 x 12
##   ORIG_CHN ORIG_RT ORIG_AMT ORIG_TRM  OLTV   DTI OCC_STAT MULTI_UN
##   <chr>      <dbl>    <dbl>    <int> <dbl> <dbl> <chr>       <dbl>
## 1 C           6.00   92000.      360   80.   35. P              0.
## 2 C           3.62  114000.      360   95.   37. P              0.
## 3 R           4.50  389000.      360   77.   43. P              0.
## 4 B           6.50  275000.      360   69.   40. P              0.
## 5 R           3.75   81000.      360   68.   38. P              0.
## 6 B           4.25  274000.      120   65.   23. P              0.
## # ... with 4 more variables: CSCORE_MN <dbl>, ORIG_VAL <dbl>, VinYr <chr>,
## #   def.max <dbl>
```

```
df.model.noNA <- df.model[complete.cases(df.model),]
# Select all except def.max
x <- subset(df.model.noNA, select=c(-def.max,-VinYr))
y <- as.factor(df.model.noNA$def.max)

# Up and down sampling examples:
down_train <- downSample(x = x[, -ncol(x)],
                         y = y)
table(down_train$Class)
```

```
##
##    0    1
## 2220 2220
```

3 Bagging using Decision Trees

Start with a bagging example. **Click here** for a list of bagging packages available via the `caret` wrapper.

```
library("ipred")
library("plyr")
library("dplyr")
library("e1071")
library("ranger")

x <- subset(down_train, select=c(-Class))
y <- as.factor(down_train$Class)
# 5 fold cross validation:
fitControl <- trainControl(method = "cv",number=5)

# Set the tuning parameters:
grid <- expand.grid(.vars=ncol(x))

# bag.treebag <- train(x=x,y=y,
#                      trControl = fitControl,
```

```
#                   method="bag",
#                   metric = "Accuracy",
#                   maximize=TRUE,
#                   tuneGrid=grid,
#                   bagControl = bagControl(fit = ctreeBag$fit,
#                               predict=ctreeBag$pred,
#                               aggregate=ctreeBag$aggregate))

# bag.treebag <- train(x=x,y=y,
#                      method="treebag",
#                      trControl=fitControl,
#                      metric="Accuracy")

# Set the tuning parameters:
grid <- expand.grid(.mtry=ncol(x),
                    .splitrule="gini",
                    .min.node.size=5)

bag.treebag <- train(x=x,y=y,
                     method="ranger",
                     trControl = fitControl,
                     metric="Accuracy",
                     tuneGrid=grid,
                     num.trees=10)

print(bag.treebag)
```

```
## Random Forest
##
## 4440 samples
##    9 predictor
##    2 classes: '0', '1'
##
## No pre-processing
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 3552, 3552, 3552, 3552, 3552
## Resampling results:
##
##   Accuracy   Kappa
##   0.7506757  0.5013514
##
## Tuning parameter 'mtry' was held constant at a value of 9
## Tuning
##  parameter 'splitrule' was held constant at a value of gini
##
## Tuning parameter 'min.node.size' was held constant at a value of 5
```

```
names(bag.treebag)
```

```
##  [1] "method"       "modelInfo"    "modelType"    "results"
##  [5] "pred"         "bestTune"     "call"         "dots"
##  [9] "metric"       "control"      "finalModel"   "preProcess"
## [13] "trainingData" "resample"     "resampledCM"  "perfNames"
## [17] "maximize"     "yLimits"      "times"        "levels"
```

```
print(bag.treebag$finalModel)
```

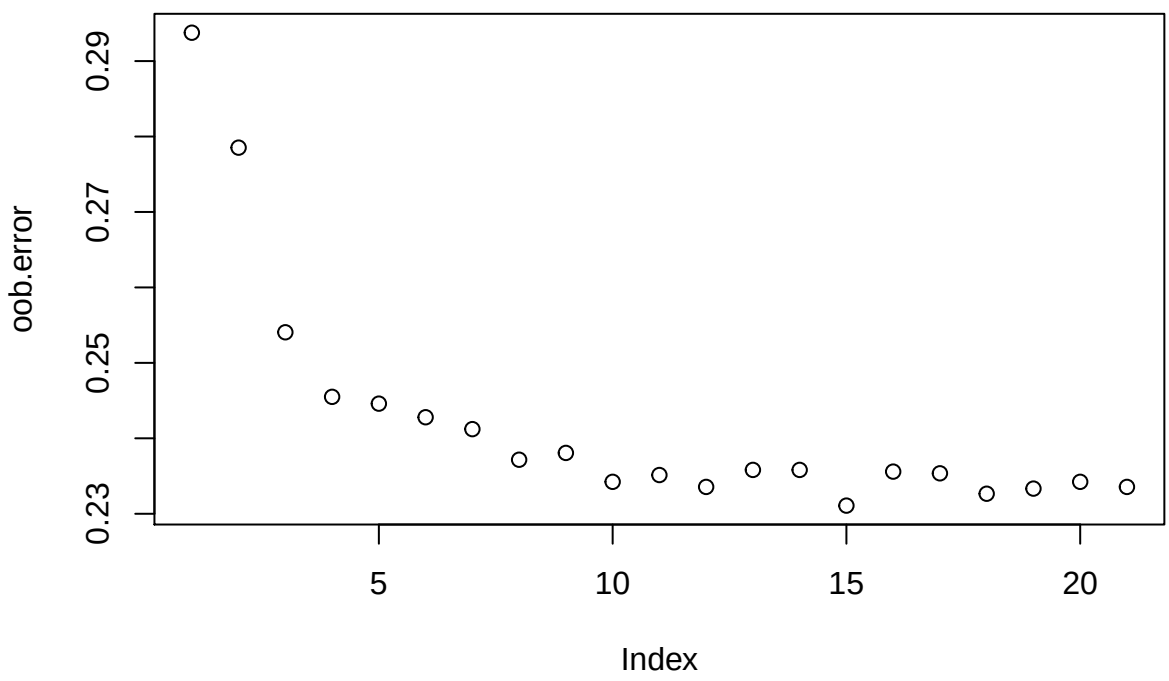
```
## Ranger result
##
## Call:
##  ranger::ranger(dependent.variable.name = ".outcome", data = x,      mtry = min(param$mtry, ncol(x))
##
## Type:                             Classification
## Number of trees:                  10
## Sample size:                      4440
## Number of independent variables:  9
## Mtry:                             9
## Target node size:                 5
## Variable importance mode:         none
## Splitrule:                        gini
## OOB prediction error:             27.56 %
```

Now, let's run an experiment to see how the bagging works versus the number of bootstrap samples.

```
b <- c(1,seq(10,200,by=10))
oob.error <- c(NA)
bag.treebag <- list(NA)
for (i in 1:length(b)){
  bag.treebag[[i]] <- train(x=x,y=y,
                      method="ranger",
                      trControl = fitControl,
                      metric="Accuracy",
                      tuneGrid=grid,
                      num.trees=b[i])
  oob.error[i] <- bag.treebag[[i]]$finalModel$prediction.error
  print(i)
}
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
## [1] 6
## [1] 7
## [1] 8
## [1] 9
## [1] 10
## [1] 11
## [1] 12
## [1] 13
## [1] 14
## [1] 15
## [1] 16
## [1] 17
## [1] 18
## [1] 19
## [1] 20
## [1] 21
```

```
plot(oob.error)
```



4 Random Forests

Recall that random forests are simply an application of bagging where the number of predictors is a subsample of the total number of predictors. Therefore, we can use the same basic code as above with different values of `mtry`.

```
b <- c(1,seq(10,100,by=10))
m <- c(2:ncol(x))
oob.error.rf <- matrix(NA,ncol=length(m),
                       nrow=length(b))
rf.treebag <- list(NA)
nn <- 1
for (j in 1:length(m)){
  grid <- expand.grid(.mtry=m[j],
                      .splitrule="gini",
                      .min.node.size=5)
  for (i in 1:length(b)){
    rf.treebag[[nn]] <- train(x=x,y=y,
                       method="ranger",
                       trControl = fitControl,
                       metric="Accuracy",
                       tuneGrid=grid,
```

```
                        num.trees=b[i])
    oob.error.rf[i,j] <- rf.treebag[[nn]]$finalModel$prediction.error
    print(i)
    nn <- nn + 1
    print(nn)
  }
}
```

```
## [1] 1
## [1] 2
## [1] 2
## [1] 3
## [1] 3
## [1] 4
## [1] 4
## [1] 5
## [1] 5
## [1] 6
## [1] 6
## [1] 7
## [1] 7
## [1] 8
## [1] 8
## [1] 9
## [1] 9
## [1] 10
## [1] 10
## [1] 11
## [1] 11
## [1] 12
## [1] 1
## [1] 13
## [1] 2
## [1] 14
## [1] 3
## [1] 15
## [1] 4
## [1] 16
## [1] 5
## [1] 17
## [1] 6
## [1] 18
## [1] 7
## [1] 19
## [1] 8
## [1] 20
## [1] 9
## [1] 21
## [1] 10
## [1] 22
## [1] 11
## [1] 23
## [1] 1
## [1] 24
```

```
## [1] 2
## [1] 25
## [1] 3
## [1] 26
## [1] 4
## [1] 27
## [1] 5
## [1] 28
## [1] 6
## [1] 29
## [1] 7
## [1] 30
## [1] 8
## [1] 31
## [1] 9
## [1] 32
## [1] 10
## [1] 33
## [1] 11
## [1] 34
## [1] 1
## [1] 35
## [1] 2
## [1] 36
## [1] 3
## [1] 37
## [1] 4
## [1] 38
## [1] 5
## [1] 39
## [1] 6
## [1] 40
## [1] 7
## [1] 41
## [1] 8
## [1] 42
## [1] 9
## [1] 43
## [1] 10
## [1] 44
## [1] 11
## [1] 45
## [1] 1
## [1] 46
## [1] 2
## [1] 47
## [1] 3
## [1] 48
## [1] 4
## [1] 49
## [1] 5
## [1] 50
## [1] 6
## [1] 51
```

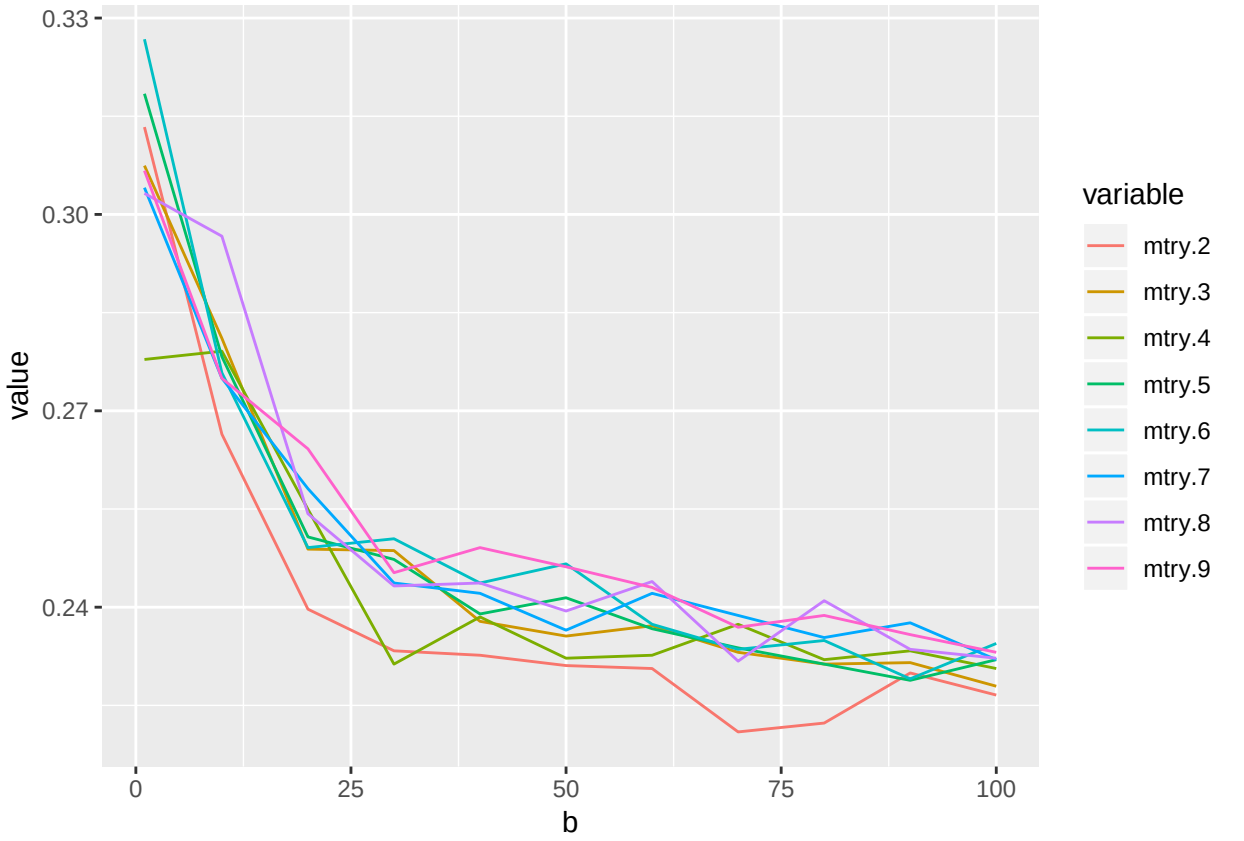
```
## [1] 7
## [1] 52
## [1] 8
## [1] 53
## [1] 9
## [1] 54
## [1] 10
## [1] 55
## [1] 11
## [1] 56
## [1] 1
## [1] 57
## [1] 2
## [1] 58
## [1] 3
## [1] 59
## [1] 4
## [1] 60
## [1] 5
## [1] 61
## [1] 6
## [1] 62
## [1] 7
## [1] 63
## [1] 8
## [1] 64
## [1] 9
## [1] 65
## [1] 10
## [1] 66
## [1] 11
## [1] 67
## [1] 1
## [1] 68
## [1] 2
## [1] 69
## [1] 3
## [1] 70
## [1] 4
## [1] 71
## [1] 5
## [1] 72
## [1] 6
## [1] 73
## [1] 7
## [1] 74
## [1] 8
## [1] 75
## [1] 9
## [1] 76
## [1] 10
## [1] 77
## [1] 11
## [1] 78
```

```
## [1] 1
## [1] 79
## [1] 2
## [1] 80
## [1] 3
## [1] 81
## [1] 4
## [1] 82
## [1] 5
## [1] 83
## [1] 6
## [1] 84
## [1] 7
## [1] 85
## [1] 8
## [1] 86
## [1] 9
## [1] 87
## [1] 10
## [1] 88
## [1] 11
## [1] 89
```

```
library("reshape")
library("ggplot2")
df.plot <- data.frame(b,oob.error.rf)
names.tmp <- c("b")
for (i in 1:length(m)){
  names.tmp[i+1] <- paste0("mtry.",m[i])
}
names(df.plot) <- names.tmp

melt.df.plot <- melt(df.plot,id="b")

p <- ggplot(melt.df.plot,
            aes(x=b,y=value,color=variable)) +
          geom_line()
print(p)
```



The out of bag error seems to drop substantially with the number of bootstrapped samples but levels off relatively quickly. However, there is not much difference between number of features tried (`mtry`).

5 Boosting

Boosting works very different from the bagging and random forest ensemble methods. For this example, we will use the `adabag` method via `caret`. First, install `adabag` and `plyr`. An intuitive explanation of AdaBoost is given **here**.

```
library("gbm")
fitControl <- trainControl(method = "cv",number=5)

# Convert character vars to factors
for (i in 1:ncol(x)){
  print(class(x[,i]))
}
x[,1] <- as.factor(x[,1])
x[,7] <- as.factor(x[,7])
for (i in 1:ncol(x)){
  print(class(x[,i]))
}

# Set the tuning parameters:
grid <- expand.grid(.n.trees=c(100,1000,5000),
                    .interaction.depth=c(1:4),
```

```
                    .shrinkage=c(0.001,0.1,0.5),
                    .n.minobsinnode=5)


boost.gbm <- train(x=x,y=y,
                     method="gbm",
                     trControl = fitControl,
                     metric="Accuracy",
                     tuneGrid=grid)

summary(boost.gbm)
print(boost.gbm$modelInfo)
results.gbm <- boost.gbm$results
```

References

Friedman, Jerome, Trevor Hastie, and Robert Tibshirani. 2001. *The Elements of Statistical Learning*. Vol. 1. 10. Springer series in statistics New York.

James, Gareth, Daniela Witten, Trevor Hastie, and Robert Tibshirani. 2013. *An Introduction to Statistical Learning*. Vol. 112. Springer.