Machine Learning for Finance Applications - Resampling and Ensemble Methods

Brian Clark

2020

Contents

1 Introduction

This vignette introduces resampling amd ensemble 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 Chapters 5 and 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]

The discussion around resampling will focus on cross validation and include a few examples. Bootstrapping will also be covered.

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 Resampling Methods

In this section, we discuss resampling methods. THe summary closely follows Chapter 5 of James et al. (2013). Resampling mehtods are ways of repeatedly drawing random samples from a dataset and successively

[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.

retraining a model on each sample. The goal is to gain additional information or insight that would not be available if we were to fit a single model to the full training dataset.

Of course, the downside to such methods is that they come at a relatively high computational cost because many models are fit.

2.1 Cross-Validation

Cross validation is a common (essentially standard) resampling method. The basic idea is to draw several samples from within the training set, fit a model, and test said model on a hold-out sample drawn from the training set. Note that the benefit is that we can see how the model performance changes across different random samples. This is related to the variance of the model. However, cross validation is generally limited in the sense that all testing is done within the training sample. While this is suitable for many applications, it can become problematic for time series problems that we often see in a financial context.

There are several common methods for implementing a cross validation approach which we discuss below.

2.1.1 The Validation Set Approach

The validation set approach serves as a simple introduction to cross validation. The basic idea is as follows:

1. Randomly split the training data in half
2. Trian the model on the first half
3. Test the model on the toher half, known as the "validation sample"

Let's re-create the example in Chapter 5 of James et al. (2013). The idea is to find the polynomial that best fits the variable horsepower to miles per gallon (MPG) in the Auto dataset.

```
rm(list=ls())
library("ISLR")
```

```
## Warning: package 'ISLR' was built under R version 3.6.2
```

```
set.seed(1)
train <- sample(392,196)

# The Auto data:
head(Auto)
```

```
##   mpg cylinders displacement horsepower weight acceleration year origin
## 1  18         8          307        130   3504         12.0   70      1
## 2  15         8          350        165   3693         11.5   70      1
## 3  18         8          318        150   3436         11.0   70      1
## 4  16         8          304        150   3433         12.0   70      1
## 5  17         8          302        140   3449         10.5   70      1
## 6  15         8          429        198   4341         10.0   70      1
##                        name
## 1 chevrolet chevelle malibu
## 2         buick skylark 320
## 3        plymouth satellite
## 4             amc rebel sst
## 5               ford torino
## 6          ford galaxie 500
```

```
lm.fit <- lm(mpg ~ horsepower, data=Auto, subset=train)
```

Next predict the response for all 392 observations and find the mean MSE of the validation set (i.e., -train).

```
attach(Auto)
mean((mpg-predict(lm.fit,Auto))[-train]^2)
```

```
## [1] 23.26601
```

Now let's estimate the MSe for quadratic and cubic regressions.

```
lm.fit2 <- lm(mpg~poly(horsepower,2),data=Auto,subset=train)
mean((mpg-predict(lm.fit2,Auto))[-train]^2)
```

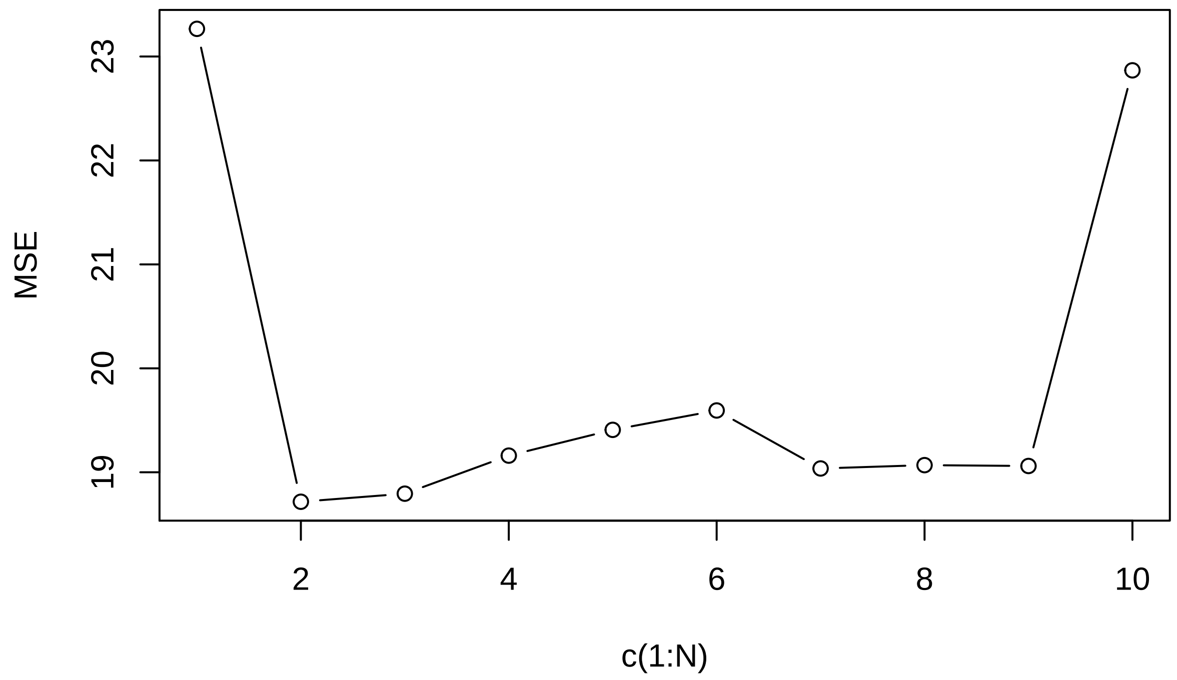
```
## [1] 18.71646
```

```
lm.fit3 <- lm(mpg~poly(horsepower,3),data=Auto,subset=train)
mean((mpg-predict(lm.fit3,Auto))[-train]^2)
```

```
## [1] 18.79401
```

Note that the MSE drops off from linear to quadratic but them seems to stabilize. Let's try to see if higher-order terms make a difference.

```
N <- 10
lm.fit <- list(NA)
MSE <- rep(NA,length=N)
for (i in 1:N){
  set.seed(1)
  lm.fit[[i]] <- lm(mpg~poly(horsepower,i),data=Auto,subset=train)
  MSE[i] <- mean((mpg-predict(lm.fit[[i]],Auto))[-train]^2)
}

plot(y=MSE,x=c(1:N),type="b")
```



Let's repeat for a bunch of different seeds.

```r
attach(Auto)
```

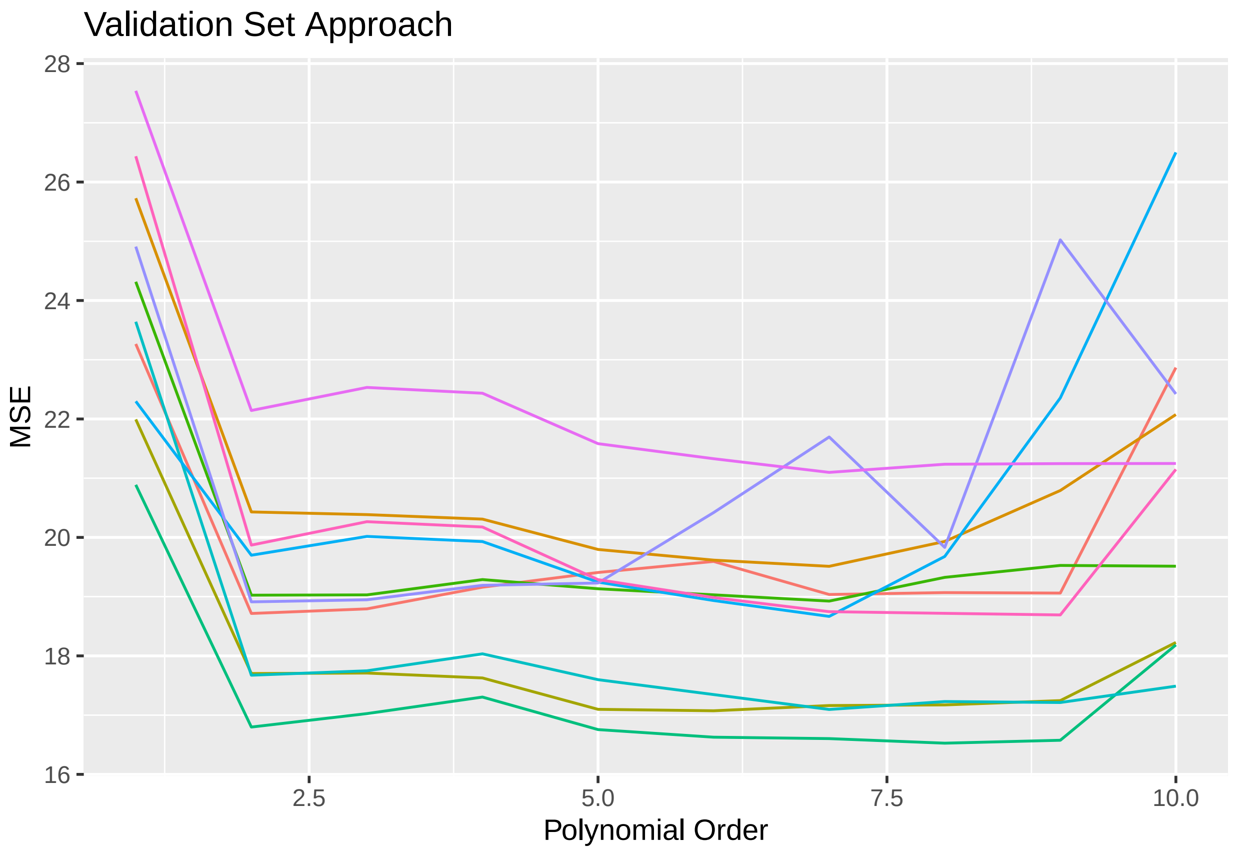
```
## The following objects are masked from Auto (pos = 3):
##
##     acceleration, cylinders, displacement, horsepower, mpg, name,
##     origin, weight, year
```

```r
S <- c(1:10)
N <- 10
MSE <- matrix(NA,nrow=N,ncol=10)
for(j in 1:10){
  set.seed(j)
  train <- sample(392,196)
  for (i in 1:N){
    lm.fit <- lm(mpg~poly(horsepower,i),data=Auto,subset=train)
    MSE[i,j] <- mean((mpg-predict(lm.fit,Auto))[-train]^2)
  }
}

library("reshape2")
library("ggplot2")
```

```
##
## Attaching package: 'ggplot2'

## The following object is masked from 'Auto':
```

```
##
##     mpg
```

```r
df <- as.data.frame(MSE)
df$k <- c(1:N)
df.melt <- melt(df,id="k")
p <- ggplot(df.melt,aes(x=k,y=value,col=variable)) +
  geom_line() + theme(legend.position = "none") +
  labs(x="Polynomial Order", y = "MSE",title= "Validation Set Approach")

print(p)
```



A few comments on the above plot. We can see the shapes of the MSE curves are all similar but there is some variaton, especially in the levels. As such, it highlights that there is some variation in the model. Another drawback of this approach is that only $\frac{N}{2}$ observations are used to fit the model - which we know to reduce model performance.

2.1.2 Leave-One-Out Cross-Validation (LOOCV)

$LOOCV$ is related to above approach but attempts to address the model's drawbacks. The idea is to fit the model on $N-1$ observations and test it on the single observation that is left out. The difference between the prediction and actual outcome for the simgle observation forms the basis for a single MSE_i. The individual $MSE_i's$ will have very little bias, but a lot of variance. As cuh, we can combine the individual $MSE_i's$ into a single metric as follows, which is the LOOCV estimtate of the test MSE:

$$CV_{(N)} = \frac{1}{N}\sum_{i=1}^{N} MSE_i. \tag{1}$$

This has a few advantages over the validation set approach:

1. Less bias (beacuase $N-1$ observations were used to fit the model)
2. Therefore, it does not tend to over-estimate the test error rate as much as teh validation set approach.
3. There is no randomness in the approach because all possible splits in teh data are considered.
4. The model is very general and can be applied in any setting.
5. There is a shortcut for reducing computational time when running a regression approach, which is shown below.

When using a least squares or polynomial regression, the following equation holds and is equivalent to the above LOOCV statistic:

$$CD_{(N)} = \frac{1}{N}\sum_{i=1}^{N}\left(\frac{y_i - \hat{y}_i}{1-h_i}\right)^2, \tag{2}$$

where h_i is the leverage of a single data point. The leverage refers the influence that a single data point has on the fit of the model and is defiend as

$$h_i = \frac{1}{N} + \frac{(x_i - \bar{x})^2}{\sum_{i'=1}^{N}(x_{i'} - \bar{x})^2}. \tag{3}$$

Note that the leverage increases with the distance of the datapoint x_i to the mean of the sample.

Unfortunately, this equation does not always hold. In cases such as a logistioc regression, we have to fit the model N times to compute the LOOCV, thus making it computationally very expensive.

To see how the method is implmented, we can use the `cv.glm` function. Note that the default setting when using a regression (i.e., Gaussian family) is to perform LOOCV. The result is stored in `glm.fit$delta`.

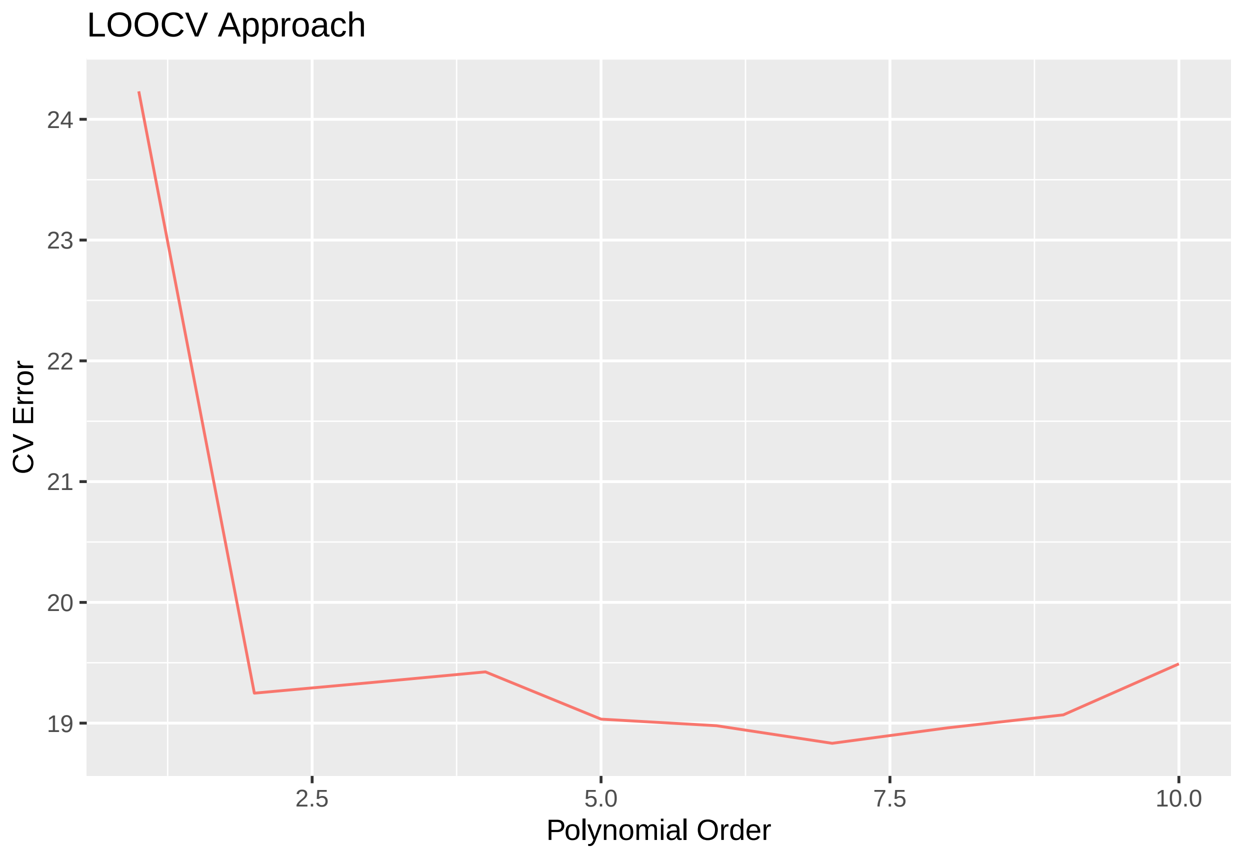
```
library("glmnet")
```

```
## Loading required package: Matrix

## Loading required package: foreach

## Loaded glmnet 2.0-18
```

```
library("boot")

attach(Auto)
```

```
## The following object is masked from package:ggplot2:
##
##     mpg

## The following objects are masked from Auto (pos = 9):
##
##     acceleration, cylinders, displacement, horsepower, mpg, name,
##     origin, weight, year

## The following objects are masked from Auto (pos = 10):
##
##     acceleration, cylinders, displacement, horsepower, mpg, name,
##     origin, weight, year
```

```r
S <- c(1:10)
N <- 10
cv.err <- matrix(NA,nrow=N,ncol=1)
for(j in 1:1){
  set.seed(j)
  train <- sample(392,196)
  for (i in 1:N){
    glm.fit <- glm(mpg~poly(horsepower,i),data=Auto)
    cv.err[i,j] <- cv.glm(Auto,glm.fit)$delta[1]
  }
}

library("reshape2")
library("ggplot2")
df <- as.data.frame(cv.err)
df$k <- c(1:N)
df.melt <- melt(df,id="k")
p <- ggplot(df.melt,aes(x=k,y=value,col=variable)) +
  geom_line() + theme(legend.position = "none") +
  labs(x="Polynomial Order", y = "CV Error",title= "LOOCV Approach")

print(p)
```



Note that the above approach takes longer. Also note that the results are similar in that there is a significant gain from the linear to quadratic model bu then things settle down.

2.1.3 k-Fold Cross-Validation

k-fold Cross-Validation (k-CV) is probbaly the most commonly used method. The idea lies somewhere between the validation test set approach and LOOCV. The method involves dividing your sample into k groups, or folds of approximately equal size. The first fold is treated as the validation set and the remaining $k-1$ folds are the training set. The MSE or other statistic is computed on the hold out (i.e., the k_{th}-fold). The procedure is then repeated k time are the results are averaged:

$$CV_{(k)} = \frac{1}{k}\sum_{i=1}^{k} MSE_i. \tag{4}$$

Based on the above equation, it is clear that LOOCV is a special case of k-CV with $k = N$. In practice, $k = 5, 10$ or some other relatively small number. The advatage is primarily computational.

There is one other important point to consider. The ultimate goal of a ML model is to minimize the test MSE. However, in practice the true test MSE is not known. It is also common that because all CV procedures discussed in this section are fit exclusinvely on the training set, they will tend to under-estimate the true test MSE. For example, one can imagine fitting a series of models to estimate the optimal λ parameter in a ridge regression using CV. Wile out CV MSE may be less than the true out of sample test MSE, the relationship between the hyperparamter (λ in this case) and the true test MSE many times exhibits a similar shape as the relationship between the hyperparameter and the CV MSE. As such, the value of the hyperparamter that minimizes the true test MSE curve is often similar to the value that minimizes the CV MSE. The key takeaway is that although we are likely to underestimate the test MSE, we are likely to choose a hyperparameter (or set of hyperparameters) that are close to "optimal."

Finally, k-fold CV has the advantage that it has less variance than the LOOCV. Even though k-fold CV has more bias than the LOOCV, it is often a better methods because it is likely to optimize the bias-variance tradoff. The basic reason is that the LOOCV is averaged over N models, but the models are highly positively correalted meaning that the are all essentially the same. The k-fold CV (which $k << N$) is more biased, but each training fold should be substantially different so when averaged, the bias is reduced.

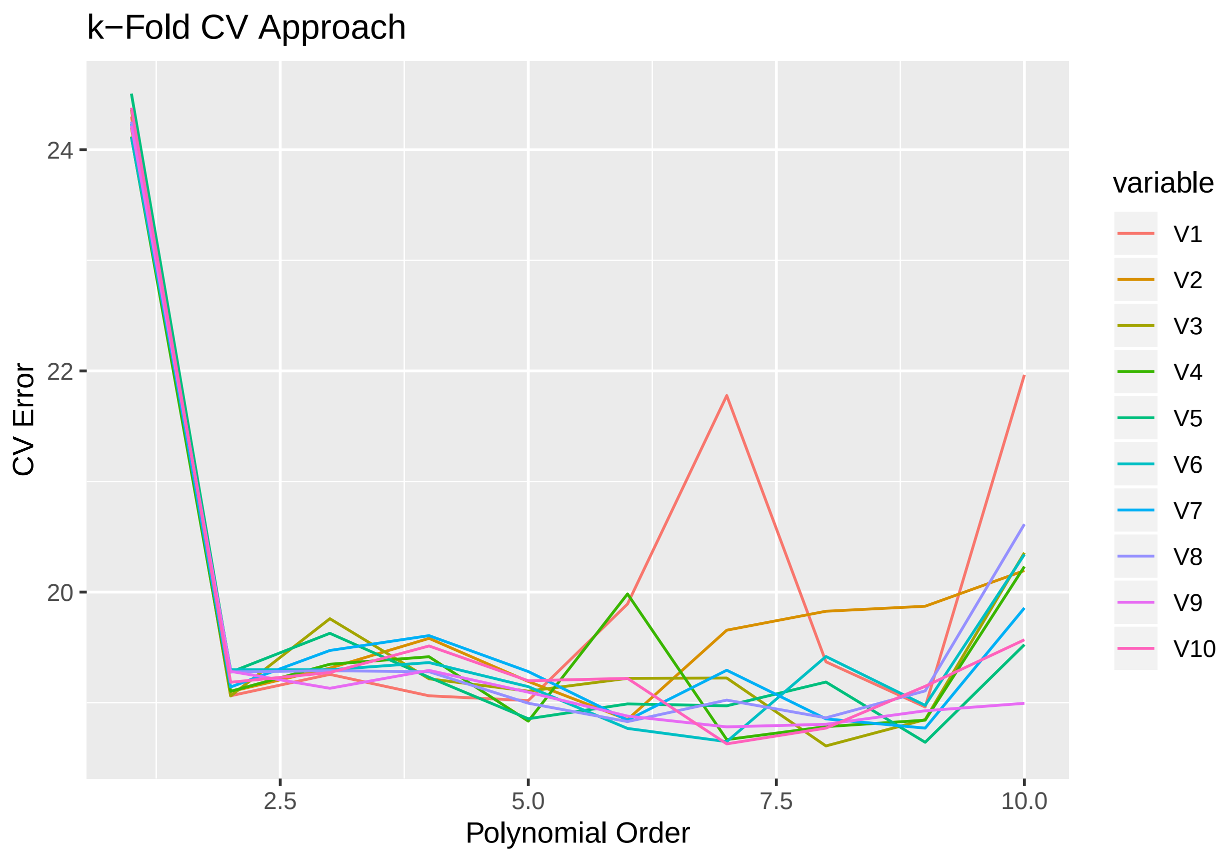
Let's now compare the k-fold CV method to the LOOCV method. Again, we'll use teh same data and repeat the lab in Chapter 5 of James et al. (2013).

```r
attach(Auto)
```

```
## The following objects are masked from Auto (pos = 3):
##
##     acceleration, cylinders, displacement, horsepower, mpg, name,
##     origin, weight, year

## The following object is masked from package:ggplot2:
##
##     mpg

## The following objects are masked from Auto (pos = 10):
##
##     acceleration, cylinders, displacement, horsepower, mpg, name,
##     origin, weight, year

## The following objects are masked from Auto (pos = 11):
##
##     acceleration, cylinders, displacement, horsepower, mpg, name,
##     origin, weight, year
```

```r
S <- c(1:10)
N <- 10
cv.err.k <- matrix(NA,nrow=N,ncol=10)
```

```r
for(j in 1:10){
  set.seed(1)
  for (i in 1:N){
    glm.fit <- glm(mpg~poly(horsepower,i),data=Auto)
    cv.err.k[i,j] <- cv.glm(Auto,glm.fit,K=j+1)$delta[1]
  }
}

df <- as.data.frame(cv.err.k)
df$k <- c(1:N)
df.melt <- melt(df,id="k")
p <- ggplot(df.melt,aes(x=k,y=value,col=variable)) +
  geom_line() +
  labs(x="Polynomial Order", y = "CV Error",title= "k-Fold CV Approach")

print(p)
```



In the above plot, each line represents a k-fold CV for a different value of k. The laebls on the lines correspond to $k-1$ (i.e., $V8$ means $k = 7$-fold CV). Note that this is much faster than the LOOCV approach.

2.1.4 Cross Validation for Classification

CV for classification problems is basically the same as for regression problems discussed above. However, the objective statistic is different. Typically we want a measure of the accuracy of the classifier. For example, the number (or %) of correctly classified observations.

2.1.5 Summary of Cross-Validation

There are a few practical points to be made here. There are a few common methods of cross validation that are used in practice and commonly implemented in statistical packages such as `caret`.

1. **Train/Test Split**: This refers to the minimum case where $k = 2$.
2. **LOOCV**: This refers to the case where $k = N$, the number of observations in the training sample.
3. **Stratified**: In this case, the model requires the split to be done in such a way that each fold has approximately the same number of observations with a certain categorical value. Usually, this refers to teh label or dependent variable in a classificaiton problem (e.g., you might want to have 100 defaulted customers in each fold).
4. **Repeated**: This refers to the case where a k-fold CV approach is repeated for several different random splits.

The other important point that we have yet to consider is how does CV fit into the overall model framework? Here is a link to a description of how CV fits into the overall modeling framework: https://scikit-learn.org/stable/modules/cross_validation.html. Generally, you don't want to use CV to directly make predictions. Rather, CV is commonly used to select hyperparamters. You then used these hyperparameters to fit your final model on the full training set of data and test on the test sample.

3 Part II: Ensemble Methods

Ensemble methods are related conceptually to CV methods in that they rely on resampling. Several potential model enhancements are dicussed below.

3.1 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

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

3.2 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,

$$\hat{f_{bag}}(x) = \frac{1}{B}\sum_{b=1}^{B}\hat{f^{*b}}(x). \tag{5}$$

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.

3.2.1 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.

3.3 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).

3.4 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:
   $$\hat{f}(x) \leftarrow \hat{f}(x) + \lambda \hat{f}^b(x). \tag{6}$$
   (c) Update the residuals,
   $$r_i \leftarrow r_i - \lambda \hat{f}^b(x_i). \tag{7}$$

3. Output the boosted model,
$$\hat{f}(x) = \sum_{b=1}^{B} \lambda \hat{f}^b(x). \tag{8}$$

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.

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.

```r
rm(list=ls()) # clear the memory
setwd("C:/Users/clarkb2/Documents/Classes/2020-Spring/AIML/ML Course")
library("ggplot2")
library("reshape")
library("plm")
library("rpart")
library("zoo")
library("plyr")
library("dplyr")
library("stringr")
library("reshape2")
library("ggplot2")
library("pander")
```

```
## Warning: package 'pander' was built under R version 3.6.2
```

```r
library("DataCombine")
```

```
## Warning: package 'DataCombine' was built under R version 3.6.2
```

```r
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`.

```r
# 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()

# 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"
```

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

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

```r
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.

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

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

```r
# 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.

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

```
## Warning: package 'caret' was built under R version 3.6.2

## Loading required package: lattice

##
## Attaching package: 'lattice'

## The following object is masked from 'package:boot':
##
##     melanoma
```

```r
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       92000      360    80    35 P               0
## 2 C           3.62   114000      360    95    37 P               0
## 3 R           4.5    389000      360    77    43 P               0
## 4 B           6.5    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>
```

```r
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)
```

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

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

5 Bagging using Decision Trees

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

```r
library("ipred")
```

```
## Warning: package 'ipred' was built under R version 3.6.2
```

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

```
## Warning: package 'ranger' was built under R version 3.6.2
```

```r
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
##   0.7342342  0.4684685
##
## 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:             29.30 %
```

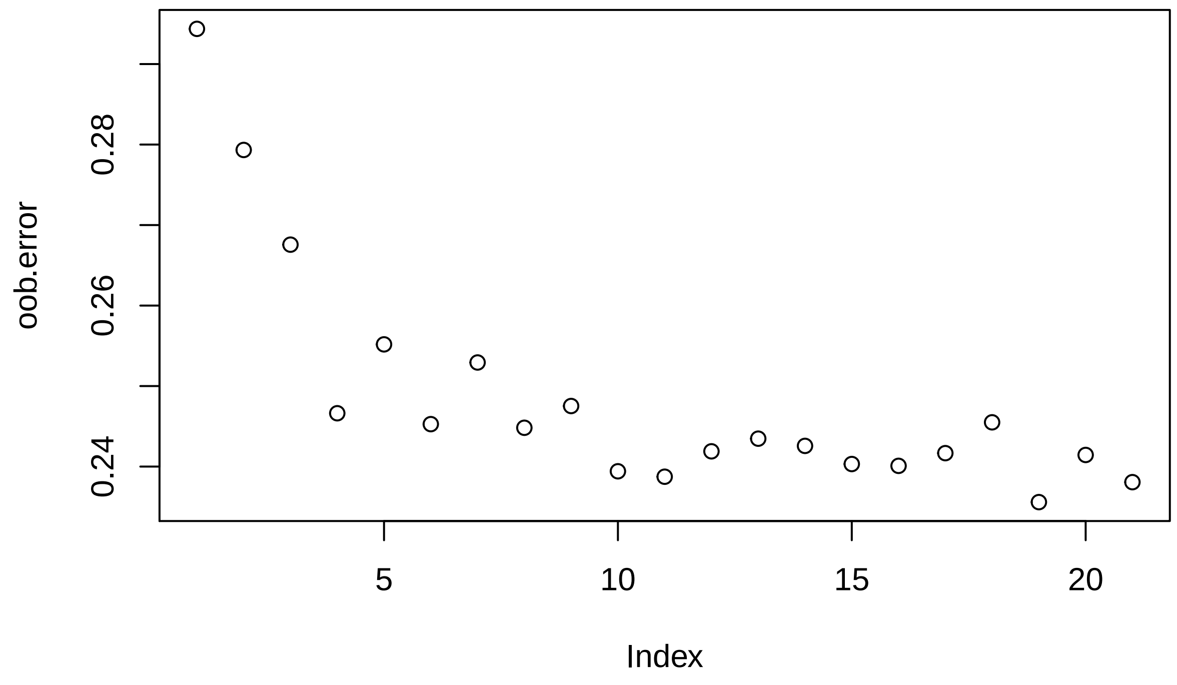
Now, let's run an experiment to see how the bagging works versus the number of bootstrap samples.

```
b <- c(1,seq(10,200,by=10))
oob.error <- c(NA)
bag.treebag <- list(NA)
```

```r
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
```

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



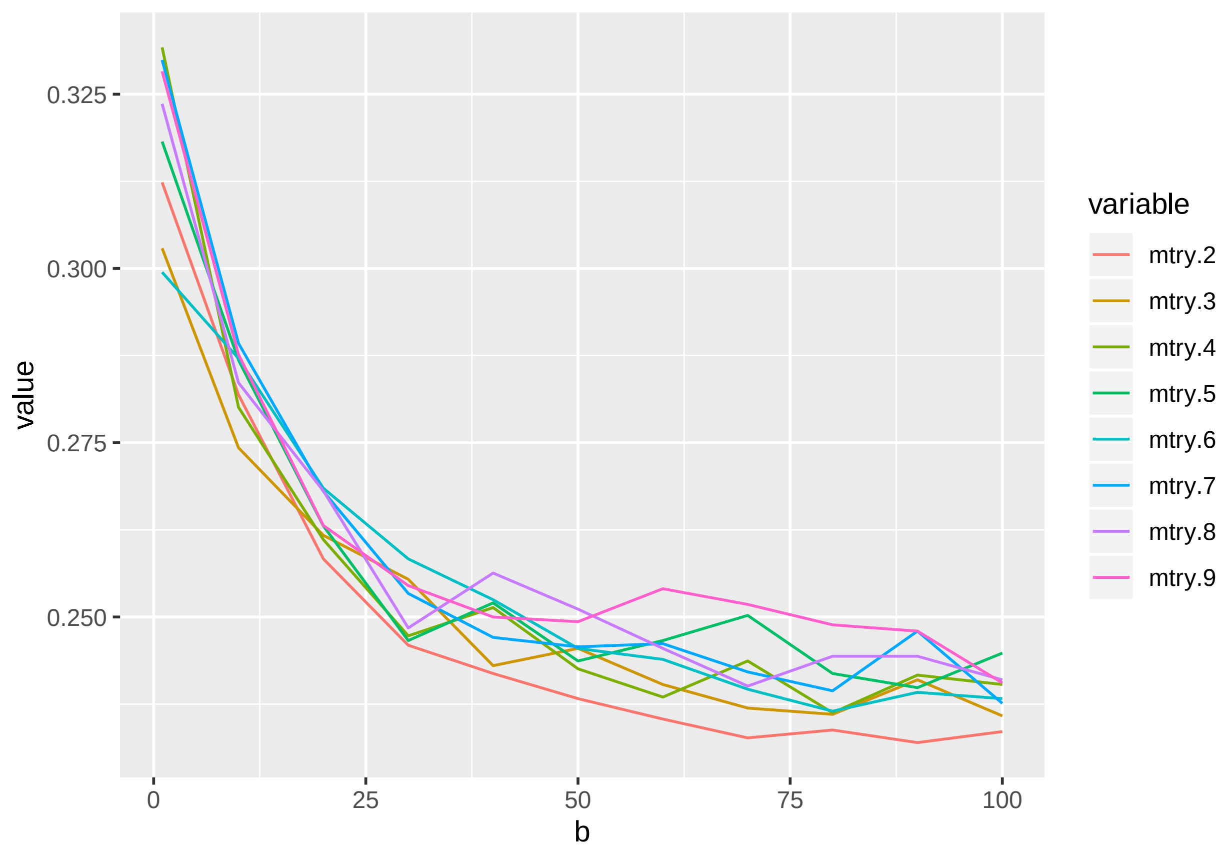
6 Random Forests

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
```

```r
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`).

7 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")
library("caret")
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),
```

```r
                    .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
# View(results.gbm)


library("knitr")
kable(as.data.frame(results.gbm),digits=c(4,0,0,0,4,4,4,4),format.args = list(big.mark=","))
```

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.