Predicting the Outcome of MLB Games through Statistical Modeling

Douglas Byers, Matt Muller, TJ Rogers

11/2/2020

1) Setup and Load Packages, Libraries, and Data

2) Data scraped from Baseball Reference

BaseballReference is the leading online database for baseball statistics, ranging from overall numbers and situational splits to individual game by game statistics. Since the goal of this project is to predict the outcome of an individual game, we chose to scrape our data from the individual game logs. Our dataset includes offensive and defensive/pitching statistics for each game every year. Since 2019 was the last season in which a full 162 game schedule was played, we decided to use the 2019 season as our test dataset, with the 2018 season serving as our training dataset.

Since there were a few predictive statistics not included in the BaseballReference database, we were able to create our scraping function in such a way to build in the creation of those variables. We had two separate, but very similar, scraping functions. The function "scraping" was used to obtain the offensive statistics, with "pitching" serving as our way of getting the defensive/pitching statistics:

```
scraping <- function (url) {
  url1 <- read_html(url)
  nodes <- html_nodes(url1, css = "table")
  table <- html_table(nodes, header = TRUE, fill = TRUE)
  tibble <- as.data.frame(table)
  tibble <- rename(tibble, "2B" = "X2B", "3B" = "X3B")%>%
    filter(Opp != "Opp")
  tibble <- tibble %>%
    mutate(SOPercH = cumsum(SO)/cumsum(PA))
  tibble
}

pitching <- function (url) {
  url1 <- read_html(url)
  nodes <- html_nodes(url1, css = "table")
  table <- html_table(nodes, header = TRUE, fill = TRUE)
  tibble <- as.data.frame(table[[2]])
  mutated <- rename(tibble, c("HA" = "H", "RA" = "R", "BBA" = "BB", "OSO" = "SO",
                              "HRA" = "HR", "SBA" = "SB", "CSA" = "CS", "ABA" = "AB",
                              "2BA" = "2B", "3BA" = "3B", "SHA" = "SH", "SFA" = "SF",
                              "." = "","ROEA" = "ROE", "HBPA" = "HBP", "DPT" = "GDP",
                              "IBBA" = "IBB", "NumPitchers" = "#"))%>%
    filter(Opp != "Opp")
  mutated <- mutated%>%
    select(7:34)%>%
```

```
    mutate(BAA = cumsum(as.numeric(HA))/cumsum(as.numeric(ABA)),
           SLGA = (4 * cumsum(as.numeric(HRA)) + 3 * cumsum(as.numeric(`3BA`)) + 2 * cumsum(as.numeric(
                     (cumsum(as.numeric(HA)) - (cumsum(as.numeric(`2BA`)) + cumsum(as.numeric(`3BA`))
                                                + cumsum(as.numeric(HRA)))))/cumsum(as.numeric(ABA)),
           OBPA = (cumsum(as.numeric(HA)) + cumsum(as.numeric(BBA)) + cumsum(as.numeric(HBPA)))/(cumsum
                    cumsum(as.numeric(BBA)) + cumsum(as.numeric(HBPA)) + cumsum(as.numeric(SFA)) + cu
           OPSA = as.numeric(SLGA) + as.numeric(OBPA),
           WHIP = (cumsum(HA)+ cumsum(BBA))/cumsum(IP),
           FIP = ((13 * cumsum(HRA) + 3 * cumsum(BBA) - 2 * cumsum(OSO))/cumsum(IP)) + 3.214,
           StrPerc = cumsum(Str)/cumsum(Pit),
           SOPerc = cumsum(OSO)/cumsum(BF),
           K_9 = (cumsum(OSO) * 9)/cumsum(IP),
           K_BB = cumsum(OSO)/(cumsum(BBA) + 0.01),
           HR_9 = (cumsum(HRA) * 9)/cumsum(IP))

  mutated
}
```

3) Data manipulation, refining the datasets

Since our regressions and machine learning algorithms will be iterating over date to be able to predict the outcome of a particular game, we need to change our dataset to ignore games that are the second game being played between the same two teams on the same day (Game 2 of a double header). Also, since we want to predict the outcome of a particular game, a Win or not, as a binary response variable, we will add a variable for Win, with values 1 and 0, 1 representing a Win for that particular team.

```
games_2018 <- games_2018%>%
  separate(Date, into = c("Mth", "Day"), sep = " ")%>%
  mutate(Month = if_else(Mth == "Mar", 3,
                   if_else(Mth == "Apr", 4,
                     if_else(Mth == "May", 5,
                        if_else(Mth == "Jun", 6,
                           if_else(Mth == "Jul", 7,
                              if_else(Mth == "Aug", 8,
                                if_else(Mth == "Sep", 9, 10)))))))%>%
  unite(Date, Month, Day, Year, sep = "/")

games.train <- games_2018 %>%
  mutate(Home = if_else(is.na(Var.4), 1, 0))%>%
  separate(Rslt, into = c("Result", "Score"), sep = ",")%>%
  separate(Score, into = c("Runs Scored", "Runs Allowed"), sep = "-") %>%
  mutate(Home_Team = if_else(Home == 1, team, Opp))%>%
  mutate(Away_Team = if_else(Home == 0, team, Opp))%>%
  mutate(Matchup_1 = Home_Team)%>%
  mutate(Matchup_2 = Away_Team)%>%
  unite(Matchup, Matchup_1, Matchup_2, sep = ",")%>%
  mutate(Run_Diff = as.numeric(`Runs Scored`) - as.numeric(`Runs Allowed`))%>%
  mutate(Win = ifelse(Result == "W", 1, 0))%>%
  arrange(mdy(Date), Matchup, Home)

g1 <- games.train%>%
  filter(team == lag(team) & Opp == lag(Opp))
```

```
games.train <- setdiff(games.train, g1)

games_2019 <- games_2019%>%
  separate(Date, into = c("Mth", "Day"), sep = " ")%>%
  mutate(Month = if_else(Mth == "Mar", 3,
                   if_else(Mth == "Apr", 4,
                     if_else(Mth == "May", 5,
                        if_else(Mth == "Jun", 6,
                           if_else(Mth == "Jul", 7,
                              if_else(Mth == "Aug", 8,
                                if_else(Mth == "Sep", 9, 10)))))))%>%
  unite(Date, Month, Day, Year, sep = "/")

games.test <- games_2019 %>%
  mutate(Home = if_else(is.na(Var.4), 1, 0))%>%
  separate(Rslt, into = c("Result", "Score"), sep = ",")%>%
  separate(Score, into = c("Runs Scored", "Runs Allowed"), sep = "-") %>%
  mutate(Home_Team = if_else(Home == 1, team, Opp))%>%
  mutate(Away_Team = if_else(Home == 0, team, Opp))%>%
  mutate(Matchup_1 = Home_Team)%>%
  mutate(Matchup_2 = Away_Team)%>%
  unite(Matchup, Matchup_1, Matchup_2, sep = ",")%>%
  mutate(Run_Diff = as.numeric(`Runs Scored`) - as.numeric(`Runs Allowed`))%>%
  mutate(Win = ifelse(Result == "W", 1, 0))%>%
  arrange(mdy(Date), Matchup, Home)


g2 <- games.test %>%
  filter(team == lag(team) & Opp == lag(Opp))

games.test <- setdiff(games.test, g2)


cols.num <- c(8:32, 36:62, 65:75, 77, 81)
games.train[cols.num] <- sapply(games.train[cols.num],as.numeric)
games.test[cols.num] <- sapply(games.test[cols.num],as.numeric)
```

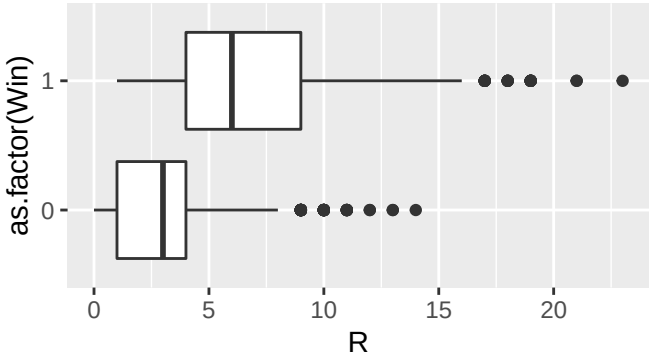
4) In Game Statistical Analysis

- First we want to run exploratory analysis and models for games with how teams perform in a given game. This should lead to models with a very low error rate since teams with better offensive and defensive statistics in a particular game are clearly more likely to win. In this way, we can see benchmarks for each statistic that teams can strive for when going into a game.

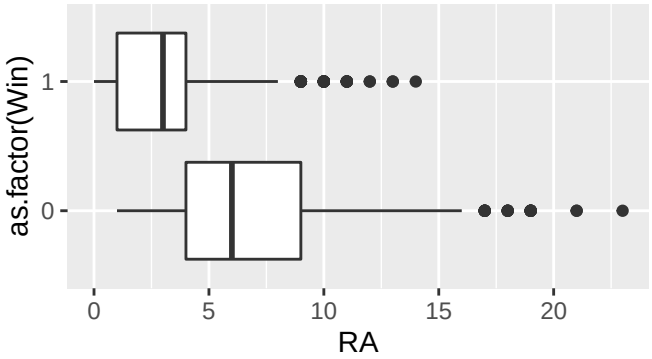
4.a) Variable Plots for EDA

- We run box and scatter plots to compare statistics we think may have an effect on a game with the outcome (win or loss).

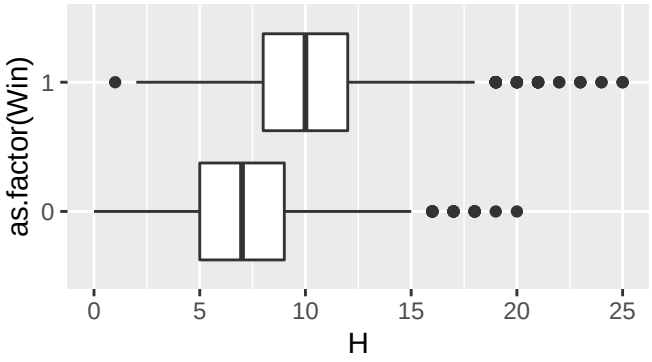
```
(RBox = ggplot (games.test, aes(y=as.factor(Win), x=R))+
  geom_boxplot())
```



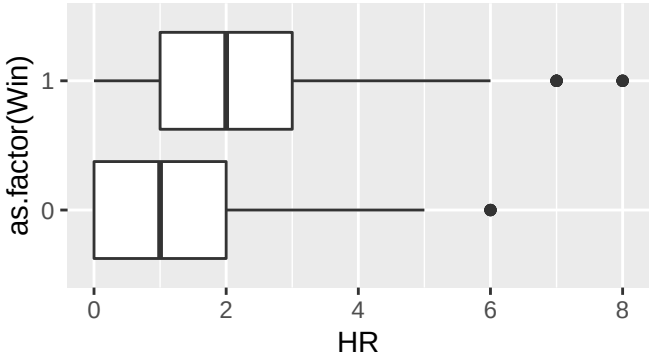

```
(RABox = ggplot (games.test, aes(y=as.factor(Win), x=RA))+
  geom_boxplot())
```



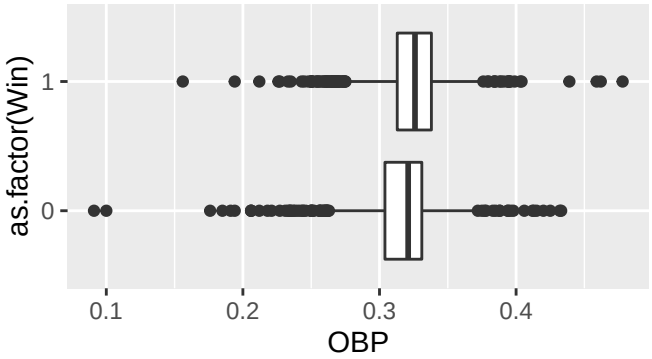

```
(HBox = ggplot (games.test, aes(y=as.factor(Win), x=H))+
  geom_boxplot())
```



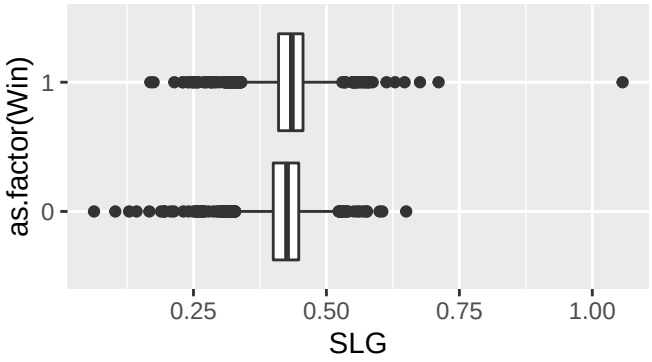

```
(HRBox = ggplot (games.test, aes(y=as.factor(Win), x=HR))+
  geom_boxplot())
```



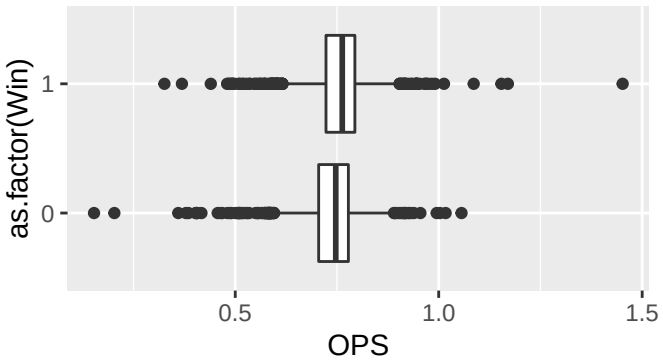

```
(OBPBox = ggplot (games.test, aes(y=as.factor(Win), x=OBP))+
  geom_boxplot())
```



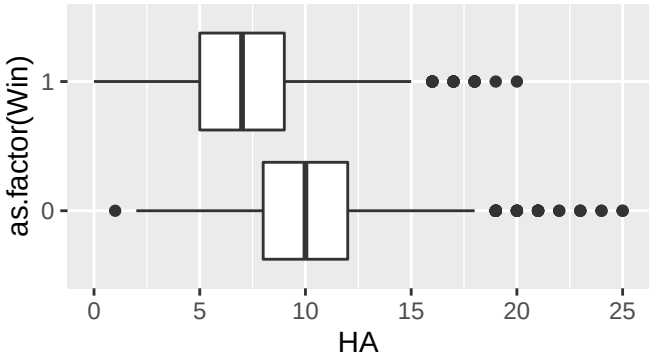

```
(SLGBox = ggplot (games.test, aes(y=as.factor(Win), x=SLG))+
  geom_boxplot())
```



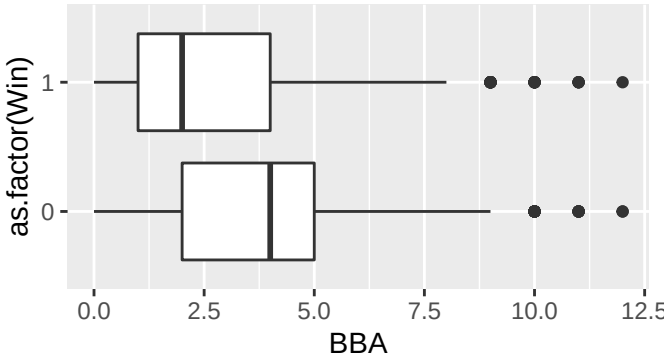

```
(OPSBox = ggplot (games.test, aes(y=as.factor(Win), x=OPS))+
  geom_boxplot())
```



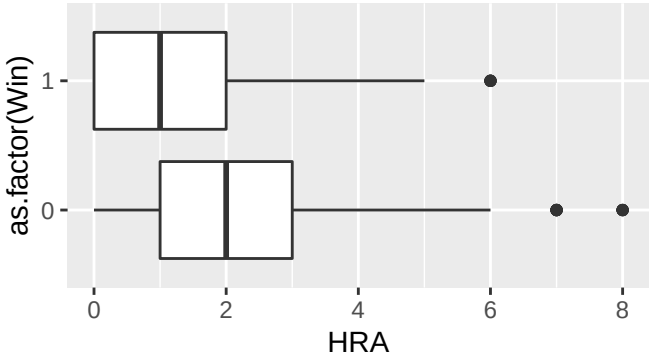

```
(HABox = ggplot (games.test, aes(y=as.factor(Win), x=HA))+
  geom_boxplot())
```



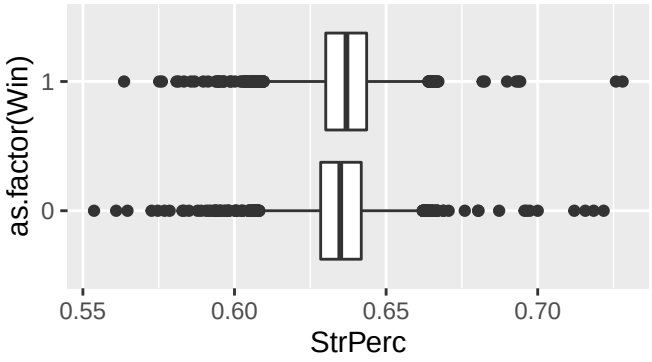

```
(BBABox = ggplot (games.test, aes(y=as.factor(Win), x=BBA))+
  geom_boxplot())
```



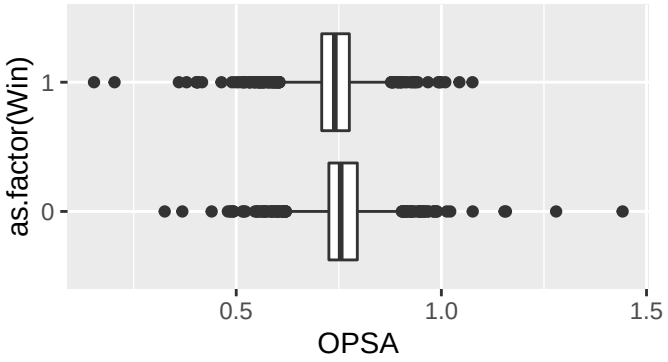

```
(HRABox = ggplot (games.test, aes(y=as.factor(Win), x=HRA))+
  geom_boxplot())
```



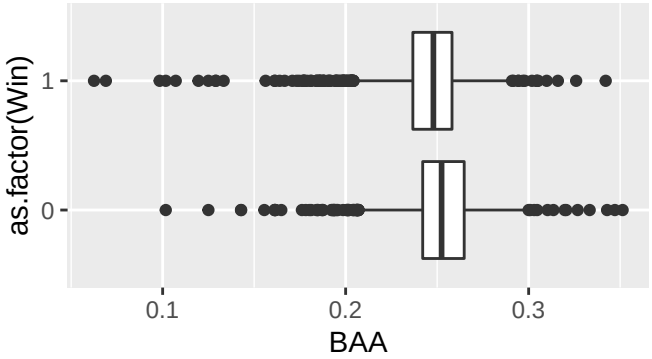

```
(StrPBox = ggplot (games.test, aes(y=as.factor(Win), x=StrPerc))+
  geom_boxplot())
```



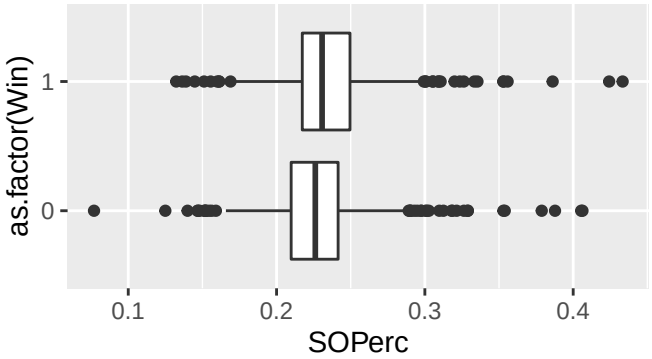

```
(OPSABox = ggplot (games.test, aes(y=as.factor(Win), x=OPSA))+
  geom_boxplot())
```



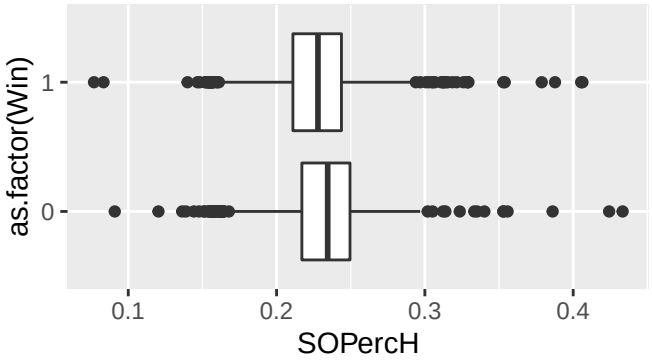

```
(BAABox = ggplot (games.test, aes(y=as.factor(Win), x=BAA))+
  geom_boxplot())
```




```
(SOPercBox = ggplot (games.test, aes(y=as.factor(Win), x=SOPerc))+
  geom_boxplot())
```

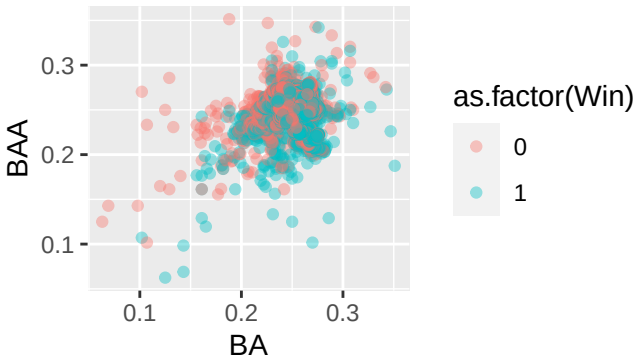



```
(SOPercHBox = ggplot (games.test, aes(y=as.factor(Win), x=SOPercH))+
  geom_boxplot())
```

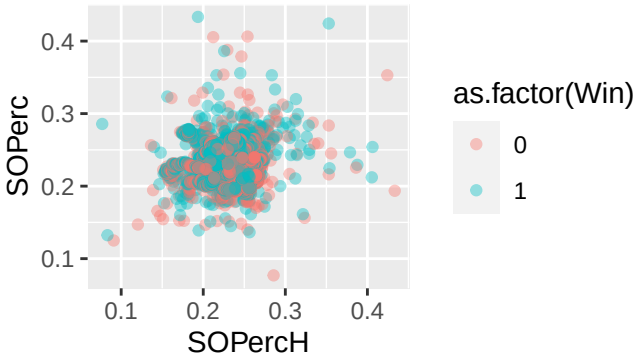


- We notice clear separation in Hits, Homeruns, Walks, and Runs, which is likely due to the fact that these are continuous and not percentage variables. It appears that On Base Percentage has the best separation for percentage variables that we displayed.

```
(BA_BAAplot = ggplot(games.test)+
  geom_point(aes(x= BA, y = BAA, color = as.factor(Win)), alpha = 0.4))
```




```
(SOPplot = ggplot(games.test)+
  geom_point(aes(x= SOPercH, y = SOPerc, color = as.factor(Win)), alpha = 0.4))
```



- There is not great separation between either of these offensive/defensive combinations, but slightly better significance shown through Batting Average and Batting Average Against.

4.b) Example Models

- As a mini test run, we look to run some models with a restricted amount of variables that we think may be key factors in predicting whether a team wins or not. We'll do this with two simple models, linear and logistic regression. The categorical variable Win (whether the team wins or not) is the response.

4.b.i) Linear Model

```
mod.lm.samp <- lm(Win ~ OBP + SOPerc + H + HR + WHIP + HRA,
             data = games.train)

games.test.samp <- games.test %>%
  add_predictions(mod.lm.samp,
                  type = "response") %>%
  mutate(class = if_else(pred > 0.5, 1,0))

table(games.test.samp$Win, games.test.samp$class)
```

```
##
##        0    1
##   0 1898  498
##   1  581 1815
```

```
(err.lm.samp <- with(games.test.samp, mean(Win != class)))
```

```
## [1] 0.2251669
```

```
summary(mod.lm.samp)
```

```
##
## Call:
## lm(formula = Win ~ OBP + SOPerc + H + HR + WHIP + HRA, data = games.train)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -1.13553 -0.33471 -0.01182  0.35453  1.11673
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept)  0.130323   0.179734   0.725    0.468
## OBP          1.705364   0.375202   4.545 5.62e-06 ***
## SOPerc       0.144070   0.290982   0.495    0.621
## H            0.047123   0.001895  24.868  < 2e-16 ***
## HR           0.090950   0.005759  15.792  < 2e-16 ***
## WHIP        -0.405312   0.068124  -5.950 2.88e-09 ***
## HRA         -0.154446   0.005216 -29.610  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.4047 on 4787 degrees of freedom
## Multiple R-squared:  0.3457, Adjusted R-squared:  0.3449
## F-statistic: 421.6 on 6 and 4787 DF,  p-value: < 2.2e-16
```

- Error rate of 22.5% is quite solid when choosing random explanatory variables, this should be reduced through unrestricted models later. The R-Squared value of 34.57% can definitely be improved.

4.b.ii) Log Model

```
mod.log.samp <- glm(Win ~ OBP + SOPerc + H + HR + WHIP + HRA,
             data = games.train,
             family = "binomial")

games.test.samp.log <- games.test %>%
  add_predictions(mod.log.samp,
                  type = "response") %>%
  mutate(class = if_else(pred > 0.5, 1,0)) #class instead of response

table(games.test.samp.log$Win, games.test.samp.log$class)
```

```
##
##        0    1
##   0 1888  508
##   1  575 1821
```

```
(err.log.samp <- with(games.test.samp.log, mean(Win != class)))
```

```
## [1] 0.2260017
```

```
summary(mod.log.samp)
```

```
##
## Call:
## glm(formula = Win ~ OBP + SOPerc + H + HR + WHIP + HRA, family = "binomial",
```

```
##     data = games.train)
##
## Deviance Residuals:
##      Min        1Q    Median        3Q       Max
## -2.93243  -0.74620   0.00587   0.78607   2.90873
##
## Coefficients:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept) -2.04838    1.14243  -1.793    0.073 .
## OBP         11.33747    2.46741   4.595 4.33e-06 ***
## SOPerc       0.27454    1.86827   0.147    0.883
## H            0.31077    0.01398  22.237  < 2e-16 ***
## HR           0.64822    0.04101  15.808  < 2e-16 ***
## WHIP        -2.85817    0.44333  -6.447 1.14e-10 ***
## HRA         -1.07706    0.04321 -24.929  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 6645.9  on 4793  degrees of freedom
## Residual deviance: 4519.5  on 4787  degrees of freedom
## AIC: 4533.5
##
## Number of Fisher Scoring iterations: 5
```

- We notice a similar error rate at 22.6% with the log model.

4.c) Unrestricted models

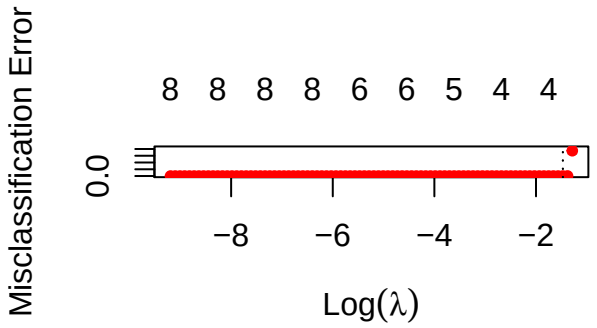
- We now look to examine models including all variables as the explanatory variables (other than the categorical variable Win as the response).

4.c.i) Lasso Model

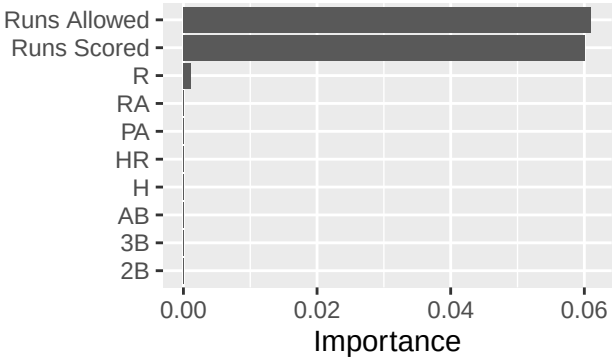
- Now we'll work on creating a lasso model. First we want to make sure all variables included are numeric and that the data is represented as a matrix.

```
games.y <- data.matrix(games.train$Win)


games.x.lasso <- games.train %>%
  select(8:32, 36:62, 65:75, 77)
games.x.lasso[is.na(games.x.lasso)] <- 0
games.x <- data.matrix(games.x.lasso)
```

```
lasso.cv <- cv.glmnet(games.x,
                      games.y,
                      family="binomial",
                      type.measure="class",
                      alpha=1)

plot(lasso.cv)
```




```
vip(lasso.cv)
```




```
lambda.opt <- lasso.cv$lambda.1se
id <- with(lasso.cv,which(lasso.cv$lambda==lambda.opt))
(err.lasso <- lasso.cv$cvm[id])
```

```
## [1] 0
```

- We notice an optimal lambda around .25 with 4 variables included. The error is 0 because clearly the team with more runs will win, and the model is reduced to only variables where "runs" is present.
- Obviously, RBI's and runs/runs against will have an impact on the outcome. These statistics refer to the amount of runs scored and given up by a team, which clearly directly correlate to wins. Let's look at more vague statistics to see how well they predict the model without these. Thus, we'll narrow down our explanatory variables even more by taking out run related variables.

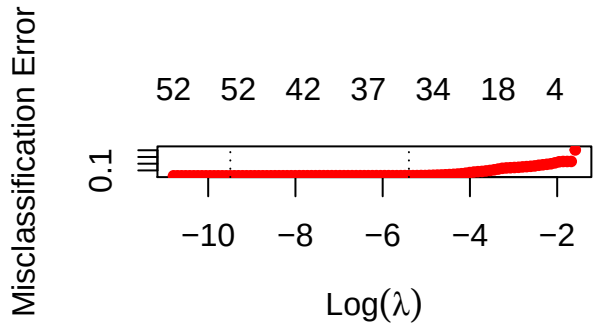
```
games.test.y <- data.matrix(games.test$Win)
games.test.x1 <- games.test %>%
  select(10:11, 13:16, 18:32, 36:38, 42:62, 65:75, 77)
games.test.x1[is.na(games.test.x1)] <- 0
games.test.x <- data.matrix(games.test.x1)
```

```
games.x.noR1 <- games.train %>%
  select(10:11, 13:16, 18:32, 36:38, 42:62, 65:75, 77)
games.x.noR1[is.na(games.x.noR1)] <- 0
games.x1 <- data.matrix(games.x.noR1)
```

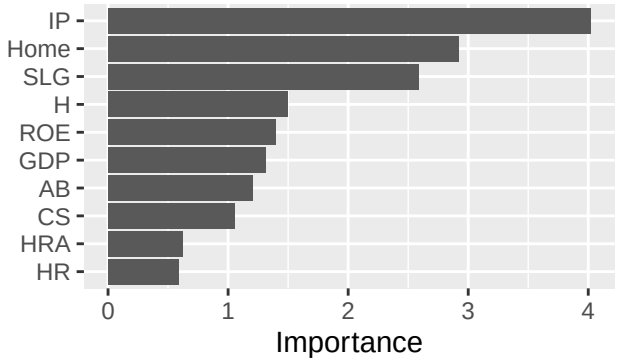
- We'll run another lasso model with a slightly reduced data matrix.

```
lasso.cv1 <- cv.glmnet(games.x1,
                       games.y,
                       family="binomial",
                       type.measure="class",
                       alpha=1)
```

```
plot(lasso.cv1)
```




```
vip(lasso.cv1)
```




```
res.vip1 <- vip(lasso.cv1,num_features=55)
vip.dat1 <- res.vip1[["data"]]
vip.dat1[52:55,]
```

```
## # A tibble: 4 x 3
##   Variable Importance Sign
##   <chr>         <dbl> <chr>
## 1 FIP               0 NEG
## 2 StrPerc           0 NEG
## 3 SOPerc            0 NEG
## 4 K_9               0 NEG
```

```
lambda.opt <- lasso.cv1$lambda.1se
id1 <- with(lasso.cv1,which(lasso.cv1$lambda==lambda.opt))
(err.lasso1 <- lasso.cv1$cvm[id1])
```

```
## [1] 0.02211097
```

```
preds.l <- predict(lasso.cv1, newx=games.x1,
                   type="class")
##The error rate on the testing dataset
table(games.y, preds.l)
```

```
##         preds.l
## games.y    0    1
##       0 2345   52
##       1   45 2352
```

- Here the optimal lambda is around 0 and there's an error rate of about 2%. This is a great error rate

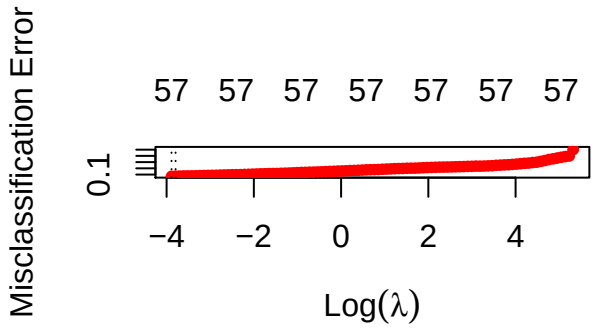
likely due to the reduction of multicollinearity when limiting the amount of predictors. Note that the error rate may differ from trial to trial due to cross validation

- Innings Pitched, whether a team is Home, and Slugging Percentage have a lot of relative importance, with Hits, Reach on Error, and Grounding into Double Plays (offensively) follow quite close behind.

4.c.ii) Ridge Model

- We'll use the same smaller dataset that we used for lasso for ridge as well (numeric explanatory variables) to predict a ridge model, along with an optimal lambda.

```
ridge.cv1 <- cv.glmnet(games.x1, games.y,
                       family="binomial",
                       type.measure="class",
                       alpha=0)
plot(ridge.cv1)
```




```
log(ridge.cv1$lambda.1se)
```

```
## [1] -3.794182
```

```
log(ridge.cv1$lambda.1se)
```

```
## [1] -3.794182
```

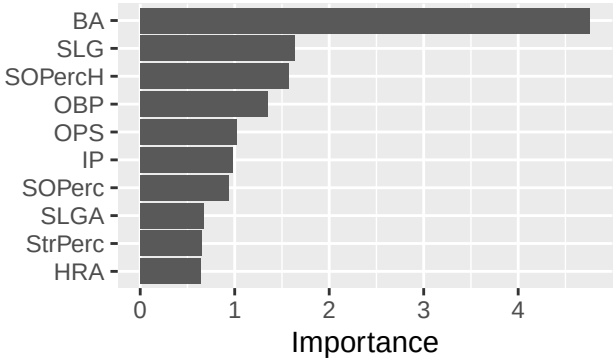
```
lambda.opt1 <- ridge.cv1$lambda.1se
id2 <- with(ridge.cv1,which(ridge.cv1$lambda==lambda.opt1))
(err.ridge1 <- ridge.cv1$cvm[id2])
```

```
## [1] 0.07363371
```

```
preds.r <- predict(ridge.cv1, newx=games.test.x,
                   type="class")
##The error rate on the testing dataset
table(games.test.y ,preds.r)
```

```
##             preds.r
## games.test.y    0    1
##            0 2189  207
##            1  190 2206
```

```
vip(ridge.cv1)
```




```
res.vip <- vip(ridge.cv1,num_features=55)
vip.dat <- res.vip[["data"]]
vip.dat[52:55,]
```

```
## # A tibble: 4 x 3
##   Variable Importance Sign
##   <chr>         <dbl> <chr>
## 1 PA          0.0319  NEG
## 2 HR_9        0.0305  POS
## 3 FIP         0.0212  POS
## 4 ABA         0.00967 NEG
```

- With an optimal lambda of about -3.8, we notice an MSE of 7.3%, which is much worse than the lasso model likely due to multicollinearity from the similarity of some of these variables in predicting Wins.
- Interesting, we actually see no statistic representing Runs through the ridge model which is pretty cool. The greatest importance seems to be represented by specific pitching and hitting stats that wouldn't directly correlate with wins at first sight (which runs and runs against clearly would). In the ridge model, Batting Average, Slugging Percentage, Strikeout Percentage, and On Base Percentage show the most significance to our model.
- Note once again that results may slightly vary from trial to trial due to the process of cross validation.

4.c.iii) Linear and Log models unrestricted

Now that we've seen these, let's try linear and log models with the reduced variables (numeric ones) that we've used.

```
##New data set, gotta adjust the numbers selected
games.test1 <- games.test %>%
  select(Win, 10:11, 13:16, 18:32, 36:38, 42:62, 65:75, 77)

games.train1 <- games.train %>%
  select(Win, 10:11, 13:16, 18:32, 36:38, 42:62, 65:75, 77)

games.train1[is.na(games.train1)] <- 0
games.test1[is.na(games.test1)] <- 0
```

```
mod.lm <- lm(Win ~ .,
             data = games.train1)

games.test.lm <- games.test1 %>%
  add_predictions(mod.lm,
```

```
                  type = "response") %>%
  mutate(class = if_else(pred > 0.5, 1,0))
```

```
## Warning in predict.lm(model, data, type = type): prediction from a rank-
## deficient fit may be misleading
```

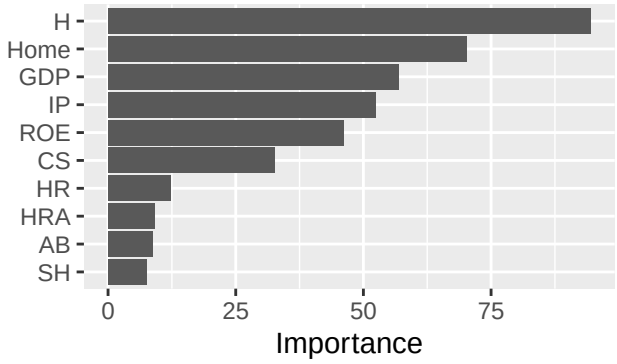
```
table(games.test.lm$Win, games.test.lm$class)
```

```
##
##        0    1
##   0 2354   42
##   1   76 2320
```

```
(err.lm <- with(games.test.lm, mean(Win != class)))
```

```
## [1] 0.02462437
```

```
vip(mod.lm)
```




```
summary(mod.lm)
```

```
##
## Call:
## lm(formula = Win ~ ., data = games.train1)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -0.87320 -0.10117  0.00942  0.11280  0.87885
##
## Coefficients: (1 not defined because of singularities)
##               Estimate Std. Error t value Pr(>|t|)
## (Intercept)  0.2285699  0.6526461   0.350 0.726189
## PA           0.0157424  0.0297584   0.529 0.596826
## AB          -0.2571634  0.0298275  -8.622  < 2e-16 ***
## H            0.2409676  0.0025502  94.490  < 2e-16 ***
## `2B`         0.0121561  0.0024982   4.866 1.18e-06 ***
## `3B`         0.0263373  0.0068026   3.872 0.000110 ***
## HR           0.0414541  0.0033702  12.300  < 2e-16 ***
## BB          -0.0119594  0.0297194  -0.402 0.687399
## IBB          0.0195944  0.0065878   2.974 0.002951 **
## SO           0.0002813  0.0010895   0.258 0.796299
## HBP         -0.0156488  0.0301244  -0.519 0.603455
## SH          -0.2281008  0.0305365  -7.470 9.51e-14 ***
## SF          -0.2234724  0.0305518  -7.315 3.02e-13 ***
```

```
## ROE          0.2507544  0.0054305  46.175  < 2e-16 ***
## GDP         -0.2350774  0.0041261 -56.973  < 2e-16 ***
## SB           0.0070855  0.0034664   2.044 0.041003 *
## CS          -0.2224702  0.0068294 -32.575  < 2e-16 ***
## BA           0.8694653  0.4197914   2.071 0.038396 *
## OBP          3.0205618  5.6605528   0.534 0.593632
## SLG          3.4949728  5.6591621   0.618 0.536883
## OPS         -3.4140911  5.6594792  -0.603 0.546370
## LOB          0.0019892  0.0020887   0.952 0.340960
## SOPercH      0.1997528  0.1692716   1.180 0.238030
## IP           0.8145433  0.0155486  52.387  < 2e-16 ***
## HA           0.0137060  0.0045817   2.991 0.002791 **
## BBA          0.0125742  0.0296697   0.424 0.671725
## OSO          0.0022795  0.0011903   1.915 0.055545 .
## HRA         -0.0277262  0.0030403  -9.120  < 2e-16 ***
## HBPA         0.0146513  0.0299976   0.488 0.625278
## ERA         -0.0157350  0.0111355  -1.413 0.157706
## BF          -0.0189736  0.0296494  -0.640 0.522249
## Pit         -0.0007153  0.0004734  -1.511 0.130838
## Str          0.0009649  0.0007194   1.341 0.179896
## IR           0.0039170  0.0023806   1.645 0.099959 .
## IS          -0.0102193  0.0040498  -2.523 0.011655 *
## SBA         -0.0080610  0.0034717  -2.322 0.020278 *
## CSA         -0.0410920  0.0076942  -5.341 9.70e-08 ***
## ABA         -0.0042866  0.0300087  -0.143 0.886420
## `2BA`       -0.0034806  0.0024932  -1.396 0.162769
## `3BA`       -0.0161430  0.0067905  -2.377 0.017479 *
## IBBA        -0.0229350  0.0067799  -3.383 0.000723 ***
## SHA         -0.0228536  0.0306854  -0.745 0.456447
## SFA         -0.0159542  0.0306290  -0.521 0.602471
## ROEA         0.0038780  0.0067094   0.578 0.563301
## DPT         -0.0181710  0.0055497  -3.274 0.001067 **
## NumPitchers  0.0005319  0.0028652   0.186 0.852734
## BAA          0.1784310  2.2117016   0.081 0.935703
## SLGA        -0.8181711  0.4143373  -1.975 0.048366 *
## OBPA         2.1009566  1.3415021   1.566 0.117387
## OPSA               NA         NA      NA       NA
## WHIP        -0.1173235  0.4022823  -0.292 0.770571
## FIP         -0.0365717  0.1602600  -0.228 0.819499
## StrPerc      0.3575343  0.4307890   0.830 0.406608
## SOPerc       1.0319239  1.8349187   0.562 0.573883
## K_9         -0.0440611  0.0700630  -0.629 0.529459
## K_BB        -0.0035065  0.0112812  -0.311 0.755947
## HR_9         0.1358604  0.2328045   0.584 0.559530
## Home        -0.6838682  0.0097399 -70.213  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.192 on 4737 degrees of freedom
## Multiple R-squared:  0.8542, Adjusted R-squared:  0.8525
## F-statistic: 495.7 on 56 and 4737 DF,  p-value: < 2.2e-16
```

```
res.vip2 <- vip(mod.lm,num_features=55)
vip.dat2 <- res.vip2[["data"]]
```

```
vip.dat2[52:55,]
```

```
## # A tibble: 4 x 3
##   Variable    Importance Sign
##   <chr>            <dbl> <chr>
## 1 SO               0.258 POS
## 2 FIP              0.228 NEG
## 3 NumPitchers      0.186 POS
## 4 ABA              0.143 NEG
```

- Our linear model shows a comparatively competitive MSE with the lasso model at 2.46%. Hits, Home, and Grounding into Double Plays interestingly enough prove to be the most significant variables in this model. Further, we see an extremely improved R-Squared from our example linear model to 85.42%

```
mod.log <- glm(Win ~ .,
               data = games.train1,
               family = "binomial")

games.test.log <- games.test1 %>%
  add_predictions(mod.log,
                  type = "response") %>%
  mutate(class = if_else(pred > 0.5, 1,0))
```

```
## Warning in predict.lm(object, newdata, se.fit, scale = 1, type = if (type == :
## prediction from a rank-deficient fit may be misleading
```

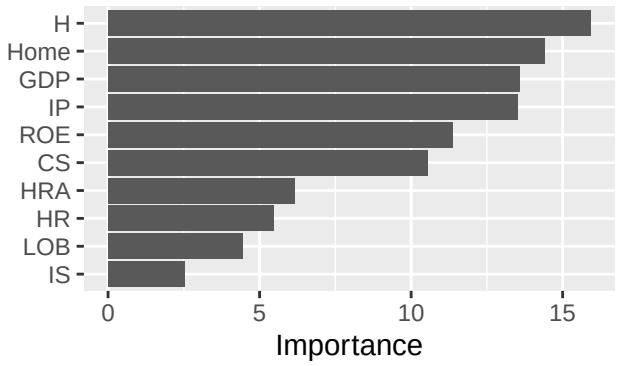
```
table(games.test.log$Win, games.test.log$class)
```

```
##
##        0    1
##   0 2349   47
##   1   66 2330
```

```
(err.log <- with(games.test.log, mean(Win != class)))
```

```
## [1] 0.02358097
```

```
vip(mod.log)
```




```
summary(mod.log)
```

```
##
## Call:
## glm(formula = Win ~ ., family = "binomial", data = games.train1)
##
```

```
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -3.9247  -0.0159   0.0000   0.0168   2.9850
##
## Coefficients: (1 not defined because of singularities)
##               Estimate Std. Error z value Pr(>|z|)
## (Intercept) -8.447e+00  3.081e+01  -0.274   0.7839
## PA           3.746e-01  4.746e+00   0.079   0.9371
## AB          -3.977e+00  4.751e+00  -0.837   0.4025
## H            4.310e+00  2.707e-01  15.922  < 2e-16 ***
## `2B`         2.122e-01  1.318e-01   1.610   0.1075
## `3B`         3.639e-01  3.829e-01   0.950   0.3420
## HR           1.085e+00  1.991e-01   5.450 5.05e-08 ***
## BB           3.065e-01  4.744e+00   0.065   0.9485
## IBB          1.361e-01  2.869e-01   0.474   0.6353
## SO          -6.993e-02  5.292e-02  -1.321   0.1864
## HBP          3.848e-01  4.749e+00   0.081   0.9354
## SH          -3.161e+00  4.768e+00  -0.663   0.5075
## SF          -3.042e+00  4.763e+00  -0.639   0.5230
## ROE          4.404e+00  3.869e-01  11.382  < 2e-16 ***
## GDP         -3.860e+00  2.841e-01 -13.584  < 2e-16 ***
## SB           1.718e-02  1.777e-01   0.097   0.9230
## CS          -3.912e+00  3.714e-01 -10.534  < 2e-16 ***
## BA           1.157e+01  1.962e+01   0.590   0.5553
## OBP          4.047e+02  2.811e+02   1.440   0.1500
## SLG          4.278e+02  2.808e+02   1.523   0.1276
## OPS         -4.155e+02  2.811e+02  -1.478   0.1394
## LOB         -5.936e-01  1.335e-01  -4.448 8.68e-06 ***
## SOPercH     -9.681e-01  8.332e+00  -0.116   0.9075
## IP           1.277e+01  9.439e-01  13.525  < 2e-16 ***
## HA          -8.674e-03  2.102e-01  -0.041   0.9671
## BBA          1.230e+00  1.794e+00   0.686   0.4929
## OSO          9.216e-02  5.923e-02   1.556   0.1197
## HRA         -1.144e+00  1.859e-01  -6.153 7.61e-10 ***
## HBPA         1.098e+00  1.807e+00   0.608   0.5435
## ERA         -1.254e+00  5.428e-01  -2.311   0.0208 *
## BF          -1.389e+00  1.793e+00  -0.775   0.4385
## Pit         -5.060e-02  2.471e-02  -2.048   0.0405 *
## Str          4.754e-02  3.708e-02   1.282   0.1998
## IR           1.389e-01  1.210e-01   1.148   0.2508
## IS          -5.425e-01  2.144e-01  -2.531   0.0114 *
## SBA          2.237e-03  1.671e-01   0.013   0.9893
## CSA         -7.580e-01  3.720e-01  -2.038   0.0416 *
## ABA          9.559e-01  1.806e+00   0.529   0.5966
## `2BA`       -2.750e-01  1.323e-01  -2.079   0.0376 *
## `3BA`       -8.033e-01  3.911e-01  -2.054   0.0400 *
## IBBA        -5.956e-01  3.010e-01  -1.979   0.0478 *
## SHA          4.418e-01  1.834e+00   0.241   0.8096
## SFA          7.006e-01  1.864e+00   0.376   0.7070
## ROEA         1.065e-01  3.186e-01   0.334   0.7381
## DPT         -3.417e-01  2.611e-01  -1.308   0.1908
## NumPitchers  6.063e-02  1.565e-01   0.387   0.6985
## BAA         -4.075e+01  1.046e+02  -0.390   0.6968
## SLGA         1.743e+01  2.246e+01   0.776   0.4377
```

```
## OBPA         3.529e+01  5.996e+01   0.589   0.5561
## OPSA               NA         NA      NA       NA
## WHIP        -4.645e-01  2.052e+01  -0.023   0.9819
## FIP         -9.106e-01  7.342e+00  -0.124   0.9013
## StrPerc      1.728e+01  2.243e+01   0.771   0.4410
## SOPerc      -5.696e+01  9.393e+01  -0.606   0.5442
## K_9          8.873e-01  3.499e+00   0.254   0.7998
## K_BB        -2.164e-01  3.561e-01  -0.608   0.5434
## HR_9         7.670e-01  1.065e+01   0.072   0.9426
## Home        -9.019e+00  6.259e-01 -14.408  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 6645.90  on 4793  degrees of freedom
## Residual deviance:  399.49  on 4737  degrees of freedom
## AIC: 513.49
##
## Number of Fisher Scoring iterations: 10
```

```
res.vip3 <- vip(mod.log,num_features=55)
vip.dat3 <- res.vip3[["data"]]
vip.dat3[52:55,]
```

```
## # A tibble: 4 x 3
##   Variable Importance Sign
##   <chr>         <dbl> <chr>
## 1 HR_9         0.0720 POS
## 2 BB           0.0646 POS
## 3 HA           0.0413 NEG
## 4 WHIP         0.0226 NEG
```

- The logistic model shows a slightly better MSE than the linear model at 2.36%. The important variables in this model are quite similar to the linear significant variables.

5) Cumulative Team Statistic Analysis

- Now we reach a point where we're ready to test models by using cumulative team statistics leading up to the predicted game. We use percentage statistics for these models to compare averages of the two different teams playing and separate variables into Home and Away Statistics by creating a new dataset from our previous one.

5.a) Data Scraping and Manipulation Part II

In our previous dataset, each game was represented by two separate rows, one with the predictors for each team. We want to have our dataset such that each game is fully encompassed by one row. To do that, we change our response variable to be whether or not the Home Team wins. Not only is this an easy way of organizing the data, as now we can have each predictor classified by whether or not it belongs to the Home Team or the Away Team, this new model also conveniently builds home field advantage into the model.

5.b) Modeling and Analysis

- We're looking for error rates between 38% and 42% for these models as some of the best professional predictive and betting models show success rates between 58% - 62%. With these models, we can use a

team's current statistics at a given point in the season to predict which team will win a game. This is a more realistic prediction method in general or when considering sports betting, since it's impossible to know the in game statistics before the game is played.

5.b.i) Logistic Model

- As before, we will begin with fitting simple linear and logistic models.

```
mod.log.cumu <- glm(Home_Win ~.,
                data = games.train3,
                family = "binomial")

games.test.log <- games.test3 %>%
  add_predictions(mod.log.cumu,
                  type = "response") %>%
  mutate(class = if_else(pred > 0.5, 1,0)) #class instead of response
```

```
## Warning in predict.lm(object, newdata, se.fit, scale = 1, type = if (type == :
## prediction from a rank-deficient fit may be misleading
```

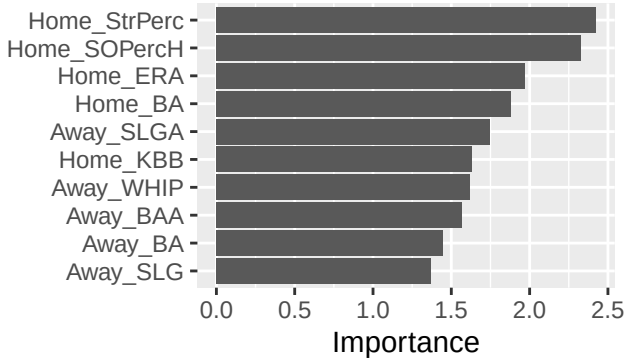
```
table(games.test.log$Home_Win, games.test.log$class)
```

```
##
##       0   1
##   0 628 504
##   1 444 820
```

```
(err.log.cumu <- with(games.test.log, mean(Home_Win != class)))
```

```
## [1] 0.3956594
```

```
vip(mod.log.cumu)
```




```
res.vip4 <- vip(mod.log.cumu,num_features=55)
vip.dat4 <- res.vip4[["data"]]
vip.dat4[29:32,]
```

```
## # A tibble: 4 x 3
##   Variable    Importance Sign
##   <chr>            <dbl> <chr>
## 1 Away_OBPA       0.222  NEG
## 2 Home_K9         0.191  POS
## 3 Away_SOPerc     0.134  NEG
## 4 Home_SOPerc     0.0384 NEG
```

```
summary(mod.log.cumu)
```

```
##
## Call:
## glm(formula = Home_Win ~ ., family = "binomial", data = games.train3)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -2.0941  -1.1175   0.6728   1.0516   1.9160
##
## Coefficients: (4 not defined because of singularities)
##                Estimate Std. Error z value Pr(>|z|)
## (Intercept)    -0.83296   14.91332  -0.056   0.9555
## Home_BA        14.36408    7.63842   1.881   0.0600 .
## Home_OBP       32.71263   89.51965   0.365   0.7148
## Home_SLG       35.62523   89.44698   0.398   0.6904
## Home_OPS      -28.63041   89.44729  -0.320   0.7489
## Home_ISO             NA         NA      NA       NA
## Home_SOPercH    6.71379    2.88487   2.327   0.0200 *
## Home_BAA       10.64739   41.63835   0.256   0.7982
## Home_OBPA       5.92285   22.77301   0.260   0.7948
## Home_SLGA       4.08886    7.14206   0.573   0.5670
## Home_OPSA            NA         NA      NA       NA
## Home_StrPerc   17.65606    7.28914   2.422   0.0154 *
## Home_ERA       -0.39769    0.20160  -1.973   0.0485 *
## Home_WHIP      -5.75031    8.03282  -0.716   0.4741
## Home_FIP        1.56545    3.01328   0.520   0.6034
## Home_SOPerc    -1.37420   35.77038  -0.038   0.9694
## Home_KBB       -0.33374    0.20490  -1.629   0.1034
## Home_K9         0.26515    1.38955   0.191   0.8487
## Home_HR9       -2.06176    4.39109  -0.470   0.6387
## Away_BA       -11.13144    7.69329  -1.447   0.1479
## Away_OBP     -111.58846   86.57912  -1.289   0.1974
## Away_SLG     -118.84800   86.58904  -1.373   0.1699
## Away_OPS      111.31342   86.62422   1.285   0.1988
## Away_ISO             NA         NA      NA       NA
## Away_SOPercH   -0.98019    2.87504  -0.341   0.7332
## Away_BAA      -61.53022   39.29561  -1.566   0.1174
## Away_OBPA      -5.29513   23.89003  -0.222   0.8246
## Away_SLGA      13.36227    7.64414   1.748   0.0805 .
## Away_OPSA            NA         NA      NA       NA
## Away_StrPerc   -2.06824    7.31709  -0.283   0.7774
## Away_ERA        0.04674    0.19326   0.242   0.8089
## Away_WHIP      11.49803    7.09987   1.619   0.1053
## Away_FIP       -3.81304    2.85443  -1.336   0.1816
## Away_SOPerc    -4.74403   35.31917  -0.134   0.8932
## Away_KBB       -0.35216    0.32266  -1.091   0.2751
## Away_K9        -0.68057    1.31334  -0.518   0.6043
## Away_HR9        4.62842    4.15001   1.115   0.2647
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
```

```
##     Null deviance: 3315.3  on 2396  degrees of freedom
## Residual deviance: 3048.6  on 2364  degrees of freedom
## AIC: 3114.6
##
## Number of Fisher Scoring iterations: 4
```

- The logistic regression model displays an error rate of 39.6%, which is right within our desired range. The most significant variables include Home Strike Percentage, Offensive Strikeout Percentage, and Earned Run Average.

5.b.ii) Linear Model

```
mod.lm.cumu <- lm(Home_Win ~.,
               data = games.train3)
```

```
games.test.lm <- games.test3 %>%
  add_predictions(mod.lm.cumu,
                  type = "response") %>%
  mutate(class = if_else(pred > 0.5, 1,0))
```

```
## Warning in predict.lm(model, data, type = type): prediction from a rank-
## deficient fit may be misleading
```

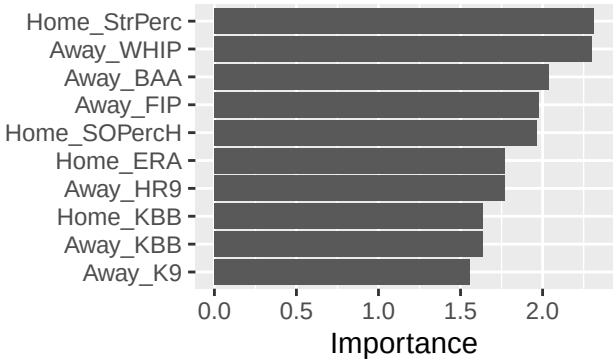
```
table(games.test.lm$Home_Win, games.test.lm$class)
```

```
##
##       0   1
##   0 647 485
##   1 459 805
```

```
(err.lm.cumu <- with(games.test.lm, mean(Home_Win != class)))
```

```
## [1] 0.39399
```

```
vip(mod.lm.cumu)
```




```
res.vip5 <- vip(mod.lm.cumu,num_features=55)
vip.dat5 <- res.vip5[["data"]]
vip.dat5[29:32,]
```

```
## # A tibble: 4 x 3
##   Variable     Importance Sign
##   <chr>             <dbl> <chr>
## 1 Away_SOPercH     0.142  NEG
## 2 Away_OBPA        0.127  POS
```

```
## 3 Home_SOPerc      0.0731 POS
## 4 Home_OBPA        0.0245 NEG
```

```
summary(mod.lm.cumu)
```

```
##
## Call:
## lm(formula = Home_Win ~ ., data = games.train3)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -0.8775 -0.4750  0.2125  0.4366  0.8337
##
## Coefficients: (4 not defined because of singularities)
##               Estimate Std. Error t value Pr(>|t|)
## (Intercept)    1.34196    3.14169   0.427   0.6693
## Home_BA        2.37241    1.66245   1.427   0.1537
## Home_OBP       7.56514   20.17669   0.375   0.7077
## Home_SLG       8.59496   20.17369   0.426   0.6701
## Home_OPS      -6.92661   20.17244  -0.343   0.7314
## Home_ISO            NA         NA      NA       NA
## Home_SOPercH   1.21135    0.61597   1.967   0.0494 *
## Home_BAA       2.38375    8.63428   0.276   0.7825
## Home_OBPA     -0.11802    4.80823  -0.025   0.9804
## Home_SLGA      0.63650    1.42947   0.445   0.6562
## Home_OPSA           NA         NA      NA       NA
## Home_StrPerc   3.56972    1.54413   2.312   0.0209 *
## Home_ERA      -0.07225    0.04076  -1.773   0.0764 .
## Home_WHIP     -1.04370    1.63700  -0.638   0.5238
## Home_FIP       0.39506    0.61319   0.644   0.5195
## Home_SOPerc    0.52162    7.13811   0.073   0.9418
## Home_KBB      -0.05877    0.03593  -1.635   0.1021
## Home_K9        0.04418    0.28078   0.157   0.8750
## Home_HR9      -0.48871    0.88856  -0.550   0.5824
## Away_BA       -2.24968    1.62208  -1.387   0.1656
## Away_OBP     -26.59583   19.47852  -1.365   0.1723
## Away_SLG     -27.82454   19.48265  -1.428   0.1534
## Away_OPS      26.50459   19.48695   1.360   0.1739
## Away_ISO            NA         NA      NA       NA
## Away_SOPercH  -0.08764    0.61677  -0.142   0.8870
## Away_BAA     -16.22570    7.96271  -2.038   0.0417 *
## Away_OBPA      0.64209    5.04560   0.127   0.8987
## Away_SLGA      2.40530    1.58772   1.515   0.1299
## Away_OPSA           NA         NA      NA       NA
## Away_StrPerc  -0.54791    1.55787  -0.352   0.7251
## Away_ERA       0.02340    0.04054   0.577   0.5638
## Away_WHIP      3.14613    1.36882   2.298   0.0216 *
## Away_FIP      -1.13139    0.57254  -1.976   0.0483 *
## Away_SOPerc    5.58361    6.73888   0.829   0.4074
## Away_KBB      -0.11055    0.06765  -1.634   0.1024
## Away_K9       -0.38817    0.24978  -1.554   0.1203
## Away_HR9       1.46640    0.82951   1.768   0.0772 .
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
```

```
## Residual standard error: 0.4769 on 2364 degrees of freedom
## Multiple R-squared:  0.0999, Adjusted R-squared:  0.08772
## F-statistic: 8.199 on 32 and 2364 DF,  p-value: < 2.2e-16
```

- The linear model displays an error rate of 39.4%, slightly better than our log model MSE. Important variables in our linear model include Home Strike Percentage, Away Walks and Hits per Inning Pitched (WHIP), and Away Batting Average Against. The R-Squared of these models is quite low, around 10%, most likely due to the difficulty of predicting these game outcomes from cumulative statistics.

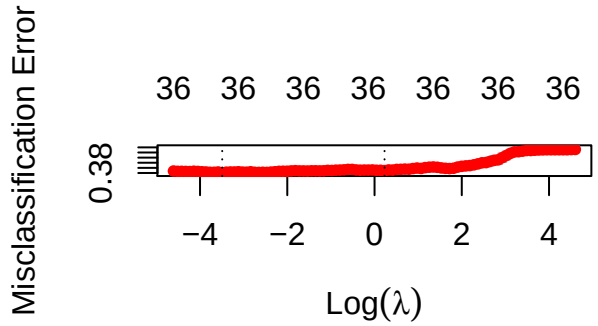
5.b.iii) Lasso and Ridge Models

- First we must create new data matrices to use in these models.

```
games.y3 <- data.matrix(games.train2$Home_Win)
games.test.y3 <- data.matrix(games.test2$Home_Win)

##New data set, gotta adjust the numbers selected
games.x3 <- games.train2 %>%
  select(4:21, 23:40)
games.x3[is.na(games.x3)] <- 0
games.x3 <- data.matrix(games.x3)

games.test.x3 <- games.test2 %>%
  select(4:21, 23:40)
games.test.x3[is.na(games.test.x3)] <- 0
games.test.x3 <- data.matrix(games.test.x3)
```

- From here, we can carry on as usual in building our ridge and lasso models.

```
ridge.cv.cumu <- cv.glmnet(games.x3, games.y3,
                      family="binomial",
                      type.measure="class",
                      alpha=0)
plot(ridge.cv.cumu)
```




```
log(ridge.cv.cumu$lambda.1se)
```

```
## [1] 0.2297342
```

```
log(ridge.cv.cumu$lambda.min)
```

```
## [1] -3.491615
```

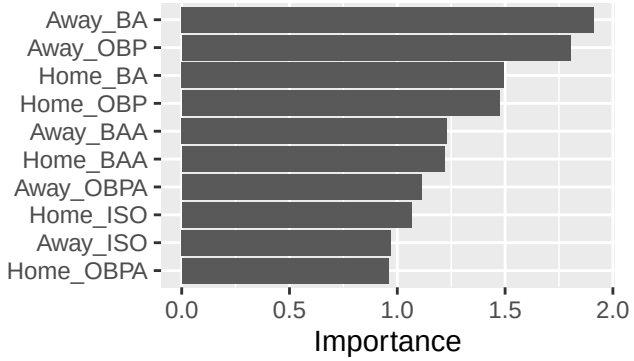
```
lambda.opt.cumu <- ridge.cv.cumu$lambda.1se
idrcumu <- with(ridge.cv.cumu,which(ridge.cv.cumu$lambda==lambda.opt.cumu))
(err.ridge.cumu <- ridge.cv.cumu$cvm[idrcumu])
```

```
## [1] 0.3892365
```

```
preds.ridge.cumu <- predict(ridge.cv.cumu, newx=games.test.x3,
                type="class")
##The error rate on the testing dataset
table(games.test.y3 ,preds.ridge.cumu)
```

```
##              preds.ridge.cumu
## games.test.y3   0   1
##             0 503 629
##             1 312 952
```

```
vip(ridge.cv.cumu)
```




```
res.vip6 <- vip(ridge.cv.cumu,num_features=55)
vip.dat6 <- res.vip6[["data"]]
vip.dat6[29:32,]
```

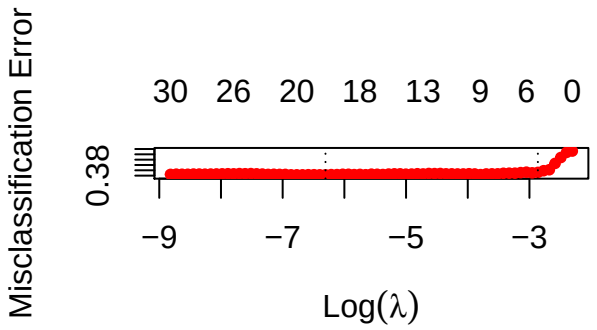
```
## # A tibble: 4 x 3
##   Variable Importance Sign
##   <chr>         <dbl> <chr>
## 1 Away_FIP     0.0466 POS
## 2 Away_ERA     0.0372 POS
## 3 Home_ERA     0.0332 NEG
## 4 Away_KBB     0.0308 NEG
```

- We see an optimal lambda of 1.25 for our ridge model and an MSE of 39.42%. This is in between the error rates of the log and linear models. As previously seen in our ridge models, Batting Average is one of the most important variables, with On Base Percentage also very important. Again, we must note that the lasso and ridge models are completed through cross validation, so we can assume that the error rate may vary around 39% to 40% from trial to trial. Thus, it's also safe to assume that it's safer to use the linear or log model instead of ridge.
- We'll look at the lasso model error to see if the MSE is significantly better there.

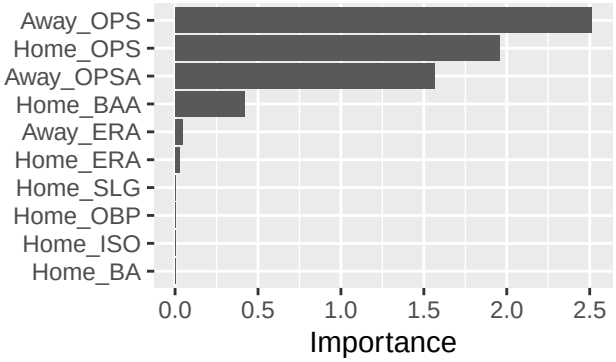
```
games.y3 <- data.matrix(games.train2$Home_Win)

##New data set, gotta adjust the numbers selected
games.x.lasso <- games.train2 %>%
  select(4:21, 23:40)
games.x.lasso[is.na(games.x.lasso)] <- 0
games.x3 <- data.matrix(games.x.lasso)
```

```
lasso.cv.cumu <- cv.glmnet(games.x3,
                    games.y3,
```

```
                    family="binomial",
                    type.measure="class",
                    alpha=1)

plot(lasso.cv.cumu)
```




```
vip(lasso.cv.cumu)
```




```
preds.lcumu <- predict(lasso.cv.cumu, newx=games.test.x3,
                  type="class")
##The error rate on the testing dataset
table(games.test.y3 ,preds.lcumu)
```

```
##              preds.lcumu
## games.test.y3    0    1
##             0  395  737
##             1  233 1031
```

```
log(lasso.cv.cumu$lambda.1se)
```

```
## [1] -2.863638
```

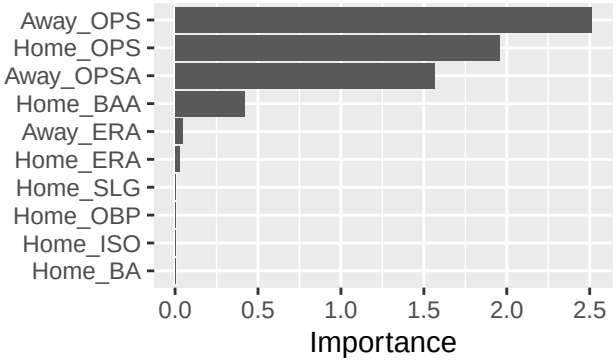
```
log(lasso.cv.cumu$lambda.1se)
```

```
## [1] -2.863638
```

```
lambda.opt.l <- lasso.cv.cumu$lambda.1se
idlcumu <- with(lasso.cv.cumu,which(lasso.cv.cumu$lambda==lambda.opt.l))
(err.lasso.cumu <- lasso.cv.cumu$cvm[idlcumu])
```

```
## [1] 0.3921569
```

```
vip(lasso.cv.cumu)
```



- For the lasso model, we see an optimal lambda of -2.96 with 6 explanatory variables maintained. This model leads to a 38.9% error rate, significantly better than our previous models.
- Note that optimal lasso and ridge models were considered but did not prove to provide better MSEs than the cross-validated versions for in-game or cumulative statistics. Thus we chose to remove these from our outline.

5.b.iv) Boosting

We also considered a number of more advanced machine learning models. First, we considered a boosting model. We left the distribution as "gaussian" as opposed to "bernoulli" so that our prediction not only gives us who we believe will win, we will get a probability of victory for the home team.

```
numTrees <- 2000
theShrinkage <- 0.01
theDepth <- 2
mod.gbm <- gbm(Home_Win ~ .,
               data=games.train3,
               distribution="gaussian",
               n.trees=numTrees,
               shrinkage=theShrinkage,
               interaction.depth = theDepth)


games.test3$predGBM <- predict(mod.gbm,newdata=games.test3)
```

```
## Using 2000 trees...
```

```
games.test3.boost <- games.test3%>%
  mutate(pred = if_else(predGBM > 0.5, 1, 0))
table(games.test3.boost$Home_Win, games.test3.boost$pred)
```

```
##
##       0   1
##   0 546 586
##   1 335 929
```

```
(mean((games.test3.boost$Home_Win != games.test3.boost$pred)))
```

```
## [1] 0.3843907
```

- Even with just an arbitrary choice for the number of trees, the depth, and the shrinkage parameter, we still see an error rate of around .383, which is right in our expected range, and is also arguably our best model thus far.

- However, we still want to see if we can improve our model. So, we can build a cross-validation function to optimize our parameters.

```
games.test3 <- games.test3%>%
  select(1:37)

cvGBM <- function(data.df, theShrinkage, theDepth, numTrees, numFolds=5){
  N <- nrow(data.df)
  folds <- sample(1:numFolds,N,rep=T)
  errs <- numeric(numFolds)
  for(fold in 1:numFolds){
    train.df.cv <- data.df %>%
      filter(folds != fold)
    test.df.cv <- data.df %>%
      filter(folds == fold)
    mod.gbm <- gbm(Home_Win ~ .,
                   data=train.df.cv,
                   distribution="gaussian",
                   n.minobsinnode=10,
                   interaction.depth = theDepth,
                   shrinkage=theShrinkage,
                   n.trees=numTrees)
    test.df.cv$pred.gbm <- predict(mod.gbm,
                         newdata=test.df.cv,
                         n.trees=numTrees)
    test.df.cv <- test.df.cv%>%
      mutate(pred = if_else(pred.gbm > 0.5, 1, 0))
    errs[fold] <- with(test.df.cv,mean((Home_Win != pred)))
    }
  mean(errs)
}
```

```
lambda <- 0.01
depth <- 1
numTrees <- 100
cvGBM(games.train3,lambda,depth,numTrees)
```

```
## [1] 0.4008022
```

```
cvGBM(games.train3,lambda,depth,100 * numTrees)
```

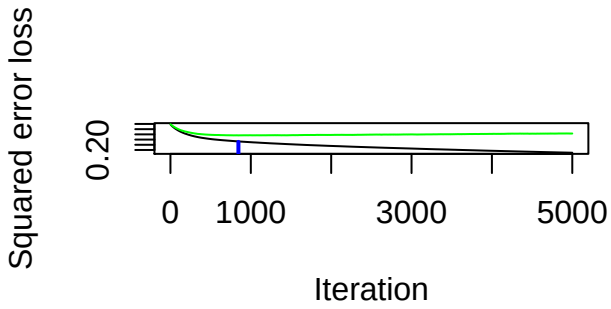
```
## [1] 0.3874434
```

- We again see decent and expected error rates in spite of our arbitrarily chosen parameters for our boosting. While this second error rate is within expectation, we must be wary of an overfit with that many trees used.
- We can also use the built in cross validation function to try to get our optimal number of trees.

```
numTrees <- 5000
mod.gbm.cv <- gbm(Home_Win ~ .,
                  data=games.train3,
                  distribution="gaussian",
                  n.trees=numTrees,
                  shrinkage=lambda,
                  interaction.depth = depth,
                  ##indicate folds here
```

```
                    cv.folds = 5,
                    ## restrict the minimum number of observations in a node
                    n.minobsinnode=10,
                    n.cores = 4) ## <-- use the cores on your computer
```

```
gbm.best <- gbm.perf(mod.gbm.cv,method="cv")
```




```
(numTreesOpt <- gbm.best)
```

```
## [1] 846
```

- With our optimal number of trees, we can fit an optimal model.

```
mod.gbm.opt <- gbm(Home_Win ~ .,
                   data=games.train3,
                   distribution="gaussian", ## for regression
                   n.trees=numTreesOpt,
                   shrinkage=lambda,
                   interaction.depth = depth)
games.test3a <- games.test3 %>%
  add_predictions(mod.gbm.opt)
```

```
## Using 846 trees...
```

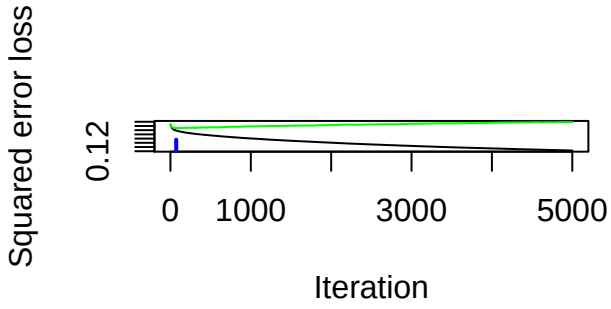
```
games.test3a <- games.test3a%>%
  mutate(class = if_else(pred > 0.5, 1, 0))
(mean(games.test3a$Home_Win != games.test3a$class))
```

```
## [1] 0.3843907
```

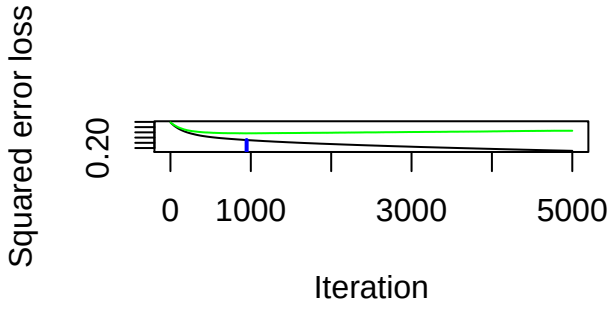
- Not bad. However, we still haven't attempted to optimize our shrinkage or depth. Our next function does just that.

```
shrink.vals <- c(0.1,0.01,0.001)
depth.vals <- c(1,2,3)
numTreesMax <- 5000
vals <- expand.grid(s=shrink.vals,d=depth.vals)
errs <- matrix(nrow=3,ncol=3)
i <- 1
errs <- numeric(9)
numTreesOpt <- numeric(9)
for(i in 1:9){
  lambda <- vals[i,1]
  depth <- vals[i,2]
  mod.gbm.cv <- gbm(Home_Win ~.,
                    data=games.train3,
                    distribution="gaussian",
```

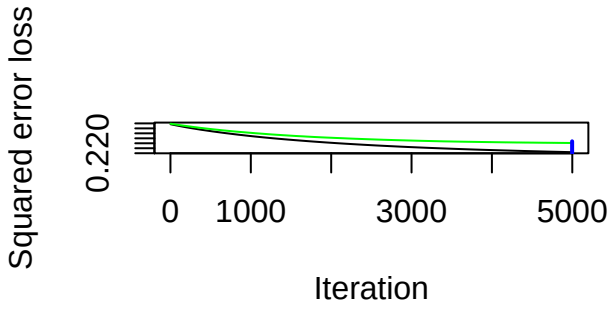
```
                    n.trees=numTreesMax,
                    shrinkage=lambda,
                    interaction.depth = depth,
                    ##indicate folds here
                    cv.folds = 5,
                    ## restrict the minimum number of observations in a node
                    n.minobsinnode=10,
                    n.cores = 4)
gbm.best <- gbm.perf(mod.gbm.cv,method="cv")
numTreesOpt[i] <- gbm.best
errs[i] <- cvGBM(games.train3,
                 lambda,
                 depth,
                 numTreesOpt[i])
print(sprintf("shrink=%s, depth=%s, numTrees=%s, MSE=%s",lambda,depth,numTreesOpt[i],errs[i]))
}
```



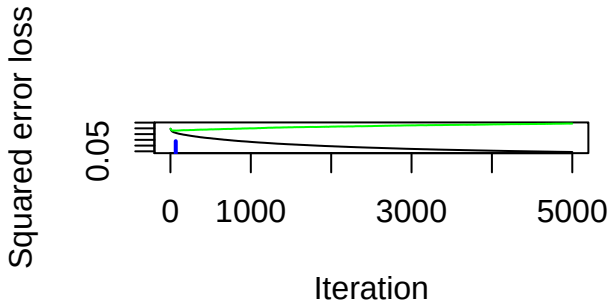

```
## [1] "shrink=0.1, depth=1, numTrees=72, MSE=0.388806960111814"
```



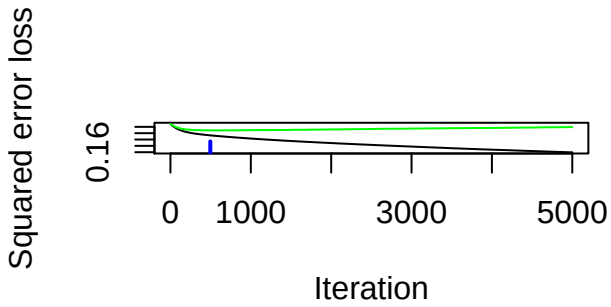

```
## [1] "shrink=0.01, depth=1, numTrees=948, MSE=0.377876576044784"
```



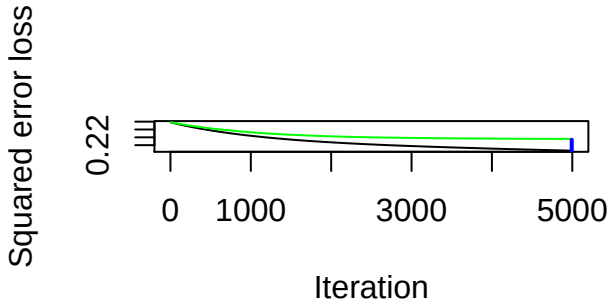

```
## [1] "shrink=0.001, depth=1, numTrees=4998, MSE=0.380744367648491"
```



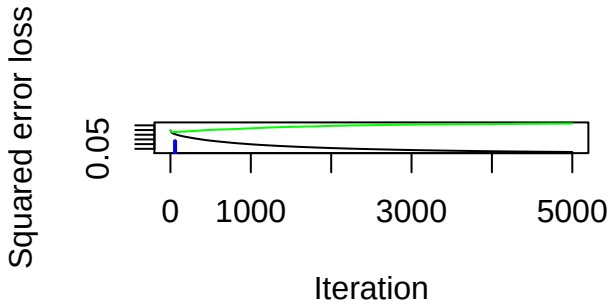

```
## [1] "shrink=0.1, depth=2, numTrees=66, MSE=0.388091131944797"
```



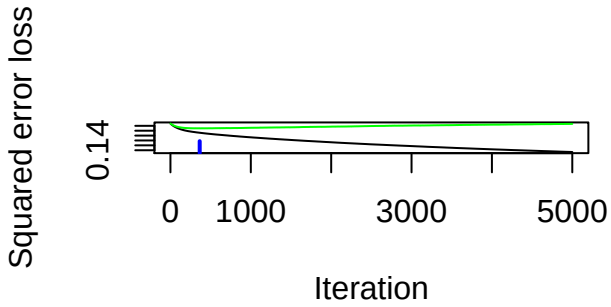

```
## [1] "shrink=0.01, depth=2, numTrees=496, MSE=0.381972889267075"
```



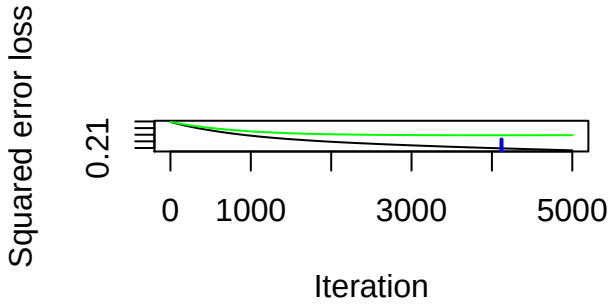

```
## [1] "shrink=0.001, depth=2, numTrees=4993, MSE=0.37524456806413"
```




```
## [1] "shrink=0.1, depth=3, numTrees=58, MSE=0.380498988639893"
```




```
## [1] "shrink=0.01, depth=3, numTrees=364, MSE=0.375454595789177"
```




```
## [1] "shrink=0.001, depth=3, numTrees=4119, MSE=0.378558179444252"
```

```
(err.opt = min(errs))
```

```
## [1] 0.3752446
```

```
id <- which.min(errs)
(lambdaOpt <- vals[id,1])
```

```
## [1] 0.001
```

```
(depthOpt <- vals[id,2])
```

```
## [1] 2
```

```
(treesOpt <- numTreesOpt[i])
```

```
## [1] 4119
```

- From this model we take the shrinkage/depth/trees combination that yields the best cross-validation error rate to compute our optimal train/test error rate.

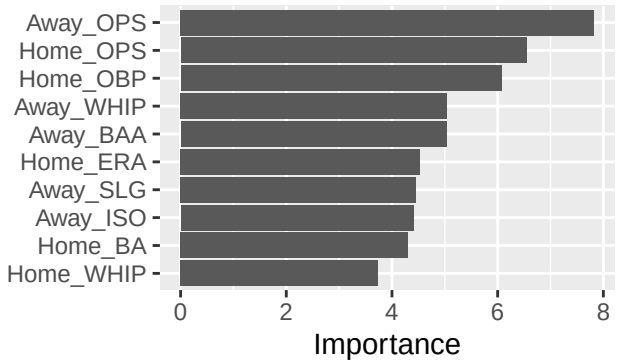
```
mod.gbm.opt <- gbm(Home_Win ~ .,
                   data=games.train3,
                   distribution="gaussian", ## for regression
                   n.trees=treesOpt,
                   shrinkage=lambdaOpt,
                   interaction.depth = depthOpt)
```

```
games.test3 %>%
  add_predictions(mod.gbm.opt) %>%
  mutate(class = if_else(pred > 0.5, 1, 0))%>%
  with(mean((Home_Win != class)))
```

```
## Using 4119 trees...
```

```
## [1] 0.3856427
```

```
vip(mod.gbm.opt)
```

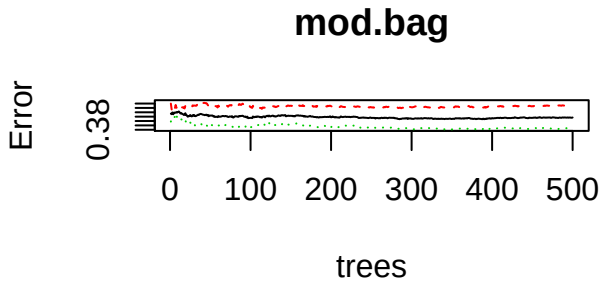


- This model yields an error rate of roughly .387 and has important variables that we've seen throughout the analysis, with On Base Plus Slugging, On Base Percentage, and WHIP showing high importance.

5.b.v) Bagging

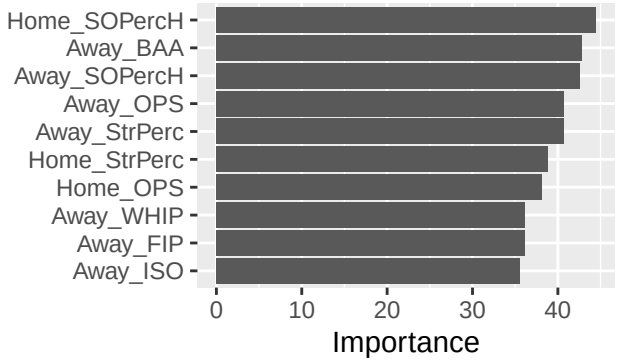
Our next machine learning model we created was a bagging model. Since, as is the case with any boostrapping model, the higher the number of boots/trees the better, we set our number of trees to be 500 to give us a large number of trees without increasing our computational time too much.

```
numTree <- 500
numPred <- ncol(games.train3)-1
set.seed(10)
mod.bag <- randomForest(as.factor(Home_Win) ~ .,
                        data=games.train3,
                        ntree=numTree,
                        mtry=numPred)

plot(mod.bag)
```



- Our plot shows the error rate oscillating between roughly .46 and .43, which is higher than we would expect. However, we will still fit our train/test model.

```
vip(mod.bag)
```




```
res.vipx <- vip(mod.bag,num_features=55)
vip.datx <- res.vip2[["data"]]
vip.datx[52:55,]
```

```
## # A tibble: 4 x 3
##   Variable    Importance Sign
##   <chr>            <dbl> <chr>
## 1 SO               0.258 POS
## 2 FIP              0.228 NEG
## 3 NumPitchers      0.186 POS
## 4 ABA              0.143 NEG
```

- We see strikeouts having a much higher importance in this model, with Offensive Strikeout Percentage for both the home and away teams represented in the top 3 most important variables.

```
bagtest.df <- games.test3 %>%
  add_predictions(mod.bag,
                  var="win.pred",
                  type = "class")

mse.bag <- bagtest.df %>%
  with(mean((Home_Win != win.pred)))

mse.bag
```

```
## [1] 0.3989983
```

```
table(bagtest.df$Home_Win, bagtest.df$win.pred)
```

```
##
##       0   1
##   0 621 511
##   1 445 819
```

Unsurprisingly, based on our original plot, our bagging error rate, while still within our range, doesn't compete with the error rates of our other models.

5.b.vi) KNN

We chose to fit two different KNN models, one with one train/test dataset and one with multiple boostrapped train/test datasets. Unfortunately, with the size of our data, the dimensionality of our data made the KNN model almost unusable. For this algorithm to be able to compete in accuracy with our other models, we needed to use values of k of 100+, which became computationally unrealistic.

```
calc_one_MSE_knn <- function(kNear){
    train.df <- games.train3
    test.df <- games.test3

    mod.knn <- knn3(as.factor(Home_Win) ~ .,
                    data=train.df,
                    k=kNear)

    test.df <- test.df %>%
      add_predictions(mod.knn,
                      type="class",
                      var="knnClass")

    with(test.df,mean(Home_Win != knnClass))
}


M <- 1
kvalue <- 1
calc_one_MSE_knn_aux <- function(iter) {
  calc_one_MSE_knn(kvalue)
}
mse.vec <- map_dbl(1:M, calc_one_MSE_knn_aux)


M <- 1
kvalue <- 1
mse.vec <- map_dbl(1:M, ~calc_one_MSE_knn(kvalue))


M <- 1
calc_MSE_knn<- function(kvalue) {
  mse.vec <- map_dbl(1:M, ~calc_one_MSE_knn(kvalue))
  mean <- mean(mse.vec)
  tibble(error = mean)
}


kMax <- 100
mse.tbl <-map_df(1:kMax, calc_MSE_knn)
mse.tbl <- mse.tbl %>%
  mutate(k=row_number())


ggplot(mse.tbl) +
  geom_line(aes(x=k, y=error), color="blue")+
  geom_smooth(aes(x=k, y=error), color="blue",se=F)

(mse.opt.k <- mse.tbl%>%
    filter(error == min(error))%>%
    filter(k == max(k)))
```

```
numTrain <- nrow(games.train3) #Number of observations in training
numTest <- nrow(games.train3)  #Number of observations in testing
```

```r
#Function to calculate the MSE for one simulation with a fixed kNear for KNN
calc_one_MSE_knn <- function(kNear){
    train.df <- sample_n(games.train3,numTrain, rep = T)
    test.df <- sample_n(games.test3,numTest, rep = T)

    mod.knn <- knn3(as.factor(Home_Win) ~ .,
                    data=train.df,
                    k=kNear)

    test.df <- test.df %>%
      add_predictions(mod.knn,
                      type="class",
                      var="knnClass")

    with(test.df,mean(Home_Win != knnClass))
}

M <- 100 #Number of simulation
#Function to calculate the mean and sd of MSE with a fixed kNear for KNN
calc_MSE_knn<- function(kvalue) {
  mse.vec <- map_dbl(1:M, ~calc_one_MSE_knn(kvalue))
  mse.mean <- mean(mse.vec)
  mse.sd <- sd(mse.vec)
  tibble(mse.mean=mse.mean, mse.sd=mse.sd)
}

kMax <- 100
mse.tbl <-map_df(1:kMax, calc_MSE_knn) #We calculate the mean,sd of MSE for all parameters

mse.tbl <- mse.tbl %>%
  mutate(k=row_number()) # We add the parameter (k)

# We plot the MSE bands as a function of k
ggplot(mse.tbl) +
  geom_line(aes(x=k, y=mse.mean), color="blue")+
  geom_smooth(aes(x=k, y=mse.mean), color="blue",se=F)+
  geom_line(aes(x=k, y=mse.mean-mse.sd), color="red")+
  geom_smooth(aes(x=k, y=mse.mean-mse.sd), color="red", se=F)+
  geom_line(aes(x=k, y=mse.mean+mse.sd), color="red")+
  geom_smooth(aes(x=k, y=mse.mean+mse.sd), color="red",se =F)

(mse.opt.boot <- mse.tbl%>%
    mutate(val = mse.mean - mse.sd)%>%
    filter(val <= mse.mean)%>%
    filter(k == max(k)))


opt.K <- mse.opt.boot$k

mod.knn.opt <- knn3(as.factor(Home_Win) ~.,
                              data = games.train3,
                              k = opt.K)
```

6) Alternate dataset

With 36 predictor variables, we wondered if there was a logical way to reduce that while still keeping all the information present. We decided to do so by examining the difference between the Home and Away teams for a given statistic. For example, instead of having the variable "Home_BA" for the home team's batting average and "Away_BA" for the away team's batting average, we have the variable "BA_Diff", which is the home team's batting average minus the away team's batting average. Along with these raw differences, we also considered finding the percentage difference between the two teams for each given statistic by taking the difference described previously and dividing by the home team's predictor.

6.a) Difference Variables

6.a.i) Log Model

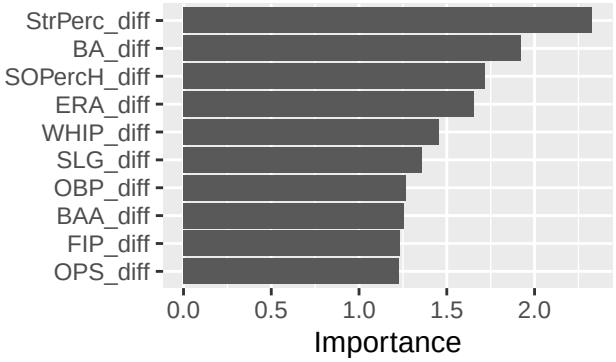
```
games.train7 <- games.train5 %>%
  select(5:22)

games.test7 <- games.test5 %>%
  select(5:22)

mod.log.diff <- glm(Home_Win ~.,
                data = games.train7,
                family = "binomial")

games.test7 <- games.test7 %>%
  add_predictions(mod.log.diff,
                  type = "response") %>%
  mutate(class = if_else(pred > 0.5, 1,0)) #class instead of response

table(games.test7$Home_Win, games.test7$class)
```

```
##
##       0   1
##   0 609 523
##   1 410 854
```

```
(err.log.diff <- with(games.test7, mean(Home_Win != class)))
```

```
## [1] 0.389399
```

```
vip(mod.log.diff)
```

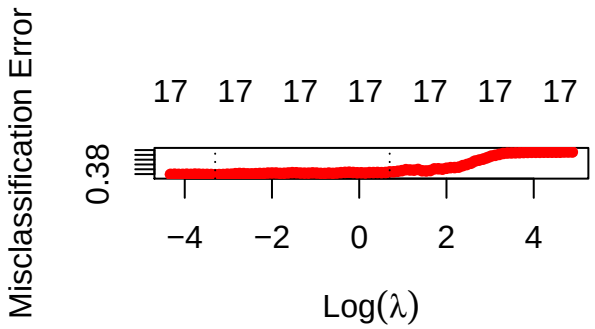


Our error rate and important variables are quite similar to those of our regular log model, which is not surprising. This gives us a model with similar error but fewer predictors, which is preffered.

6.a.ii) Lasso and Ridge

```
games.y.diff <- data.matrix(games.train7$Home_Win)


##New data set, gotta adjust the numbers selected
games.x.diff <- games.train7 %>%
  select(1:17)
games.x.diff[is.na(games.x.diff)] <- 0
games.x.diff <- data.matrix(games.x.diff)
```

```
ridge.cv.diff <- cv.glmnet(games.x.diff, games.y.diff,
                      family="binomial",
                      type.measure="class",
                      alpha=0)
plot(ridge.cv.diff)
```




```
log(ridge.cv.diff$lambda.1se)
```

```
## [1] 0.697915
```

```
log(ridge.cv.diff$lambda.min)
```

```
## [1] -3.302536
```

```
lambda.opt.diff <- ridge.cv.diff$lambda.1se
id <- with(ridge.cv.diff,which(ridge.cv.diff$lambda==lambda.opt.diff))
(err.ridge.diff <- ridge.cv.diff$cvm[id])
```

```
## [1] 0.3892365
```

```
mod.ridge.opt.diff <- glmnet(games.x.diff, games.y.diff,
                        family="binomial",
                        type.measure="class",
                        alpha=0,
                        lambda=lambda.opt.diff)

##New data set, gotta adjust the numbers selected
games.test.y.diff <- data.matrix(games.test7$Home_Win)
games.test.x.diff <- games.test7 %>%
  select(1:17)
games.test.x.diff[is.na(games.test.x.diff)] <- 0
games.test.x.diff <- data.matrix(games.test.x.diff)
```

```
preds.ridge.diff <- predict(mod.ridge.opt.diff, newx=games.test.x.diff,
                  type="class")
```

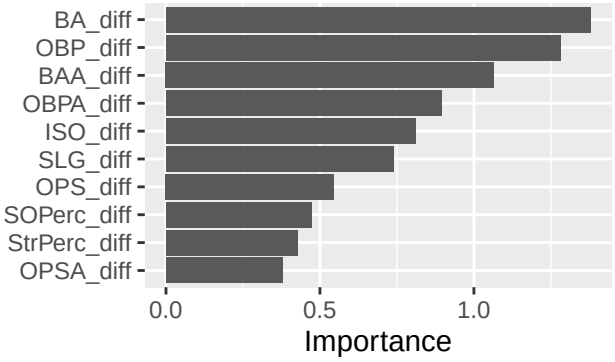
```
##The error rate on the testing dataset
table(games.test.y.diff, preds.ridge.diff)
```

```
##                   preds.ridge.diff
## games.test.y.diff   0   1
##                 0 495 637
##                 1 297 967
```

```
(err.ridge.opt.diff <- mean((games.test.y.diff != preds.ridge.diff)))
```

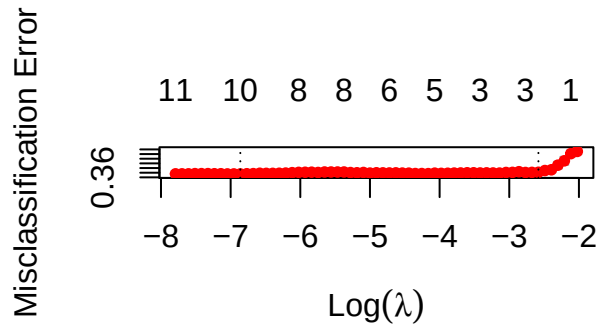
```
## [1] 0.3898164
```

```
vip(mod.ridge.opt.diff)
```



As with the log model, the ridge model has an error rate similar to the original ridge model of roughly .389. Also similar to our other ridge models, we see that predictors using Batting Average are the most important.

```
lasso.cv.diff <- cv.glmnet(games.x.diff,
                      games.y.diff,
                      family="binomial",
                      type.measure="class",
                      alpha=1)

plot(lasso.cv.diff)
```




```
log(lasso.cv.diff$lambda.1se)
```

```
## [1] -2.581524
```

```
log(lasso.cv.diff$lambda.min)
```

```
## [1] -6.861076
```

```
lambda.opt.diff <- lasso.cv.diff$lambda.1se
id <- with(lasso.cv.diff,which(lasso.cv.diff$lambda==lambda.opt.diff))
```

```
(err.lasso.diff <- lasso.cv.diff$cvm[id])
```

```
## [1] 0.3817272
```

- This cross-validation error rate for the lasso model is by far our best one yet at .381.

```
mod.lasso.opt.diff <- glmnet(games.x.diff, games.y.diff,
                         family="binomial",
                         type.measure="class",
                         alpha=1,
                         lambda=lambda.opt.diff)
```

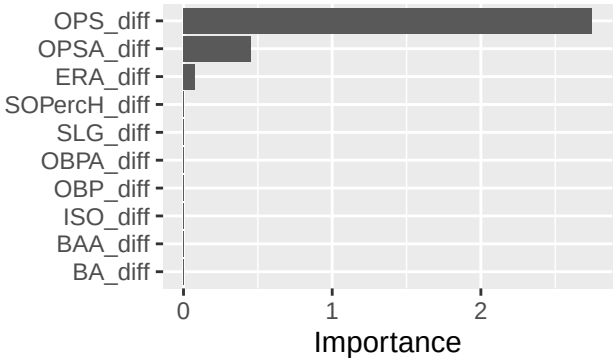
```
preds.lasso.diff <- predict(mod.lasso.opt.diff, newx=games.test.x.diff,
                  type="class")
##The error rate on the testing dataset
table(games.test.y.diff ,preds.lasso.diff)
```

```
##                   preds.lasso.diff
## games.test.y.diff   0   1
##                 0 488 644
##                 1 285 979
```

```
(err.lasso.opt.diff <- mean((games.test.y.diff != preds.lasso.diff)))
```

```
## [1] 0.3877295
```

```
vip(mod.lasso.opt.diff)
```



- Our train/test lasso model doesn't quite perform as well as the cross-validation model. However, we still see a reasonable error rate of .387, as well as seeing the same trend of repeated important variables, with On Base Plus Slugging still the most important.

6.a.iii) Boosting

We carry out the same process as above with the boosting model.

```
numTrees <- 2000
theShrinkage <- 0.01
theDepth <- 2
mod.gbm.diff <- gbm(Home_Win ~ .,
                data=games.train7,
                distribution="gaussian",
                n.trees=numTrees,
                shrinkage=theShrinkage,
                interaction.depth = theDepth)
```

```
games.test7$predGBM <- predict(mod.gbm.diff,newdata=games.test7)
```

```
## Using 2000 trees...
```

```
games.test7.boost <- games.test7%>%
  mutate(pred = if_else(predGBM > 0.5, 1, 0))
table(games.test7.boost$Home_Win, games.test7.boost$pred)
```

```
##
##       0   1
##   0 593 539
##   1 406 858
```

```
(mean((games.test7.boost$Home_Win != games.test7.boost$pred)))
```

```
## [1] 0.3944073
```

- The error rate for this arbitrary shrinkage, depth, and number of trees isn't as nice as our previous arbitrary values at 0.394. However, it still falls within our expected performance range.

```
games.test7 <- games.test7%>%
  select(1:18)

cvGBM <- function(data.df, theShrinkage, theDepth, numTrees, numFolds=5){
  N <- nrow(data.df)
  folds <- sample(1:numFolds,N,rep=T)
  errs <- numeric(numFolds)
  for(fold in 1:numFolds){
    train.df.cv <- data.df %>%
      filter(folds != fold)
    test.df.cv <- data.df %>%
      filter(folds == fold)
    mod.gbm <- gbm(Home_Win ~ .,
                   data=train.df.cv,
                   distribution="gaussian",
                   n.minobsinnode=10,
                   interaction.depth = theDepth,
                   shrinkage=theShrinkage,
                   n.trees=numTrees)
    test.df.cv$pred.gbm <- predict(mod.gbm,
                         newdata=test.df.cv,
                         n.trees=numTrees)
    test.df.cv <- test.df.cv%>%
      mutate(pred = if_else(pred.gbm > 0.5, 1, 0))
    errs[fold] <- with(test.df.cv,mean((Home_Win != pred)))
    }
  mean(errs)
}
```

```
lambda <- 0.01
depth <- 1
numTrees <- 100
cvGBM(games.train7,lambda,depth,numTrees)
```

```
## [1] 0.3920643
```

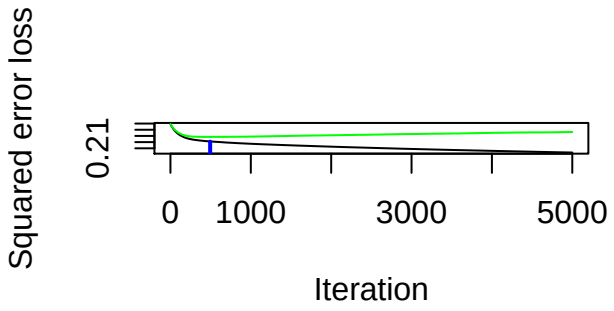
```
cvGBM(games.train7,lambda,depth,100 * numTrees)
```

```
## [1] 0.4018027
```

- Our cross-validation function performs slightly better than our raw model. Again, we see the issue with overfitting the boosted model, with the error rate jumpin over .4.

```
numTrees <- 5000
mod.gbm.cv.diff <- gbm(Home_Win ~ .,
                  data=games.train7,
                  distribution="gaussian",
                  n.trees=numTrees,
                  shrinkage=lambda,
                  interaction.depth = depth,
                  ##indicate folds here
                  cv.folds = 5,
                  ## restrict the minimum number of observations in a node
                  n.minobsinnode=10,
                  n.cores = 4) ## <-- use the cores on your computer
```

```
gbm.best.diff <- gbm.perf(mod.gbm.cv.diff,method="cv")
```




```
(numTreesOpt.diff <- gbm.best.diff)
```

```
## [1] 494
```

```
mod.gbm.opt.diff <- gbm(Home_Win ~ .,
                   data=games.train7,
                   distribution="gaussian", ## for regression
                   n.trees=numTreesOpt.diff,
                   shrinkage=lambda,
                   interaction.depth = depth)
games.test7a <- games.test7 %>%
  add_predictions(mod.gbm.opt.diff)
```

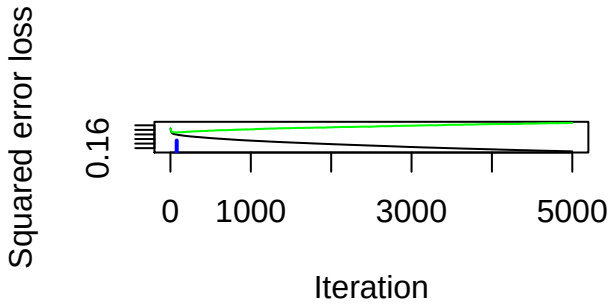
```
## Using 494 trees...
```

```
games.test7a <- games.test7a%>%
  mutate(class = if_else(pred > 0.5, 1, 0))
(mean(games.test7a$Home_Win != games.test7a$class))
```

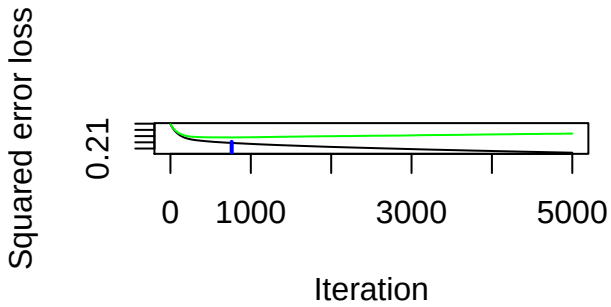
```
## [1] 0.3835559
```

- This optimal train/test model is our best train/test model thus far, tied with the previous boosting model, with an error of 0.383.
- We can carry out the same process as above to optimize our other parameters.

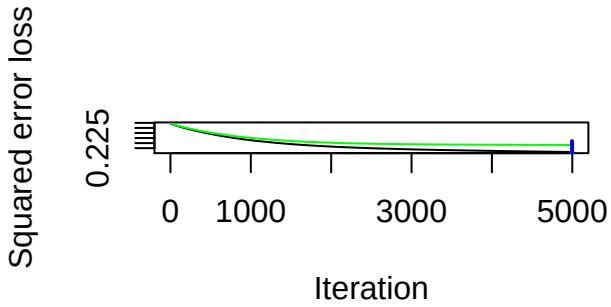
```
shrink.vals <- c(0.1,0.01,0.001)
depth.vals <- c(1,2,3)
numTreesMax <- 5000
vals <- expand.grid(s=shrink.vals,d=depth.vals)
errs <- matrix(nrow=3,ncol=3)
i <- 1
errs <- numeric(9)
numTreesOptDiff <- numeric(9)
for(i in 1:9){
  lambda <- vals[i,1]
  depth <- vals[i,2]
  mod.gbm.cv.diff2 <- gbm(Home_Win ~.,
                    data=games.train7,
                    distribution="gaussian",
                    n.trees=numTreesMax,
                    shrinkage=lambda,
                    interaction.depth = depth,
                    ##indicate folds here
                    cv.folds = 5,
                    ## restrict the minimum number of observations in a node
                    n.minobsinnode=10,
                    n.cores = 4)
gbm.best.diff2 <- gbm.perf(mod.gbm.cv.diff2,method="cv")
numTreesOptDiff[i] <- gbm.best.diff2
errs[i] <- cvGBM(games.train7,
                 lambda,
                 depth,
                 numTreesOptDiff[i])
print(sprintf("shrink=%s, depth=%s, numTrees=%s, MSE=%s",lambda,depth, numTreesOptDiff[i],errs[i]))
}
```



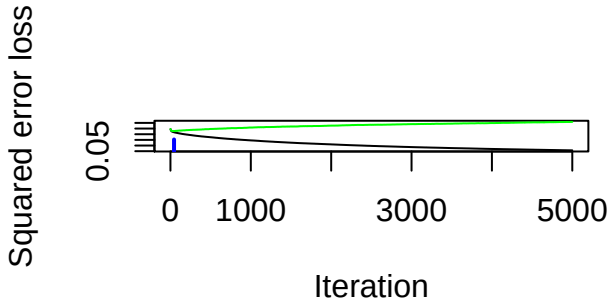

```
## [1] "shrink=0.1, depth=1, numTrees=78, MSE=0.397724406314277"
```



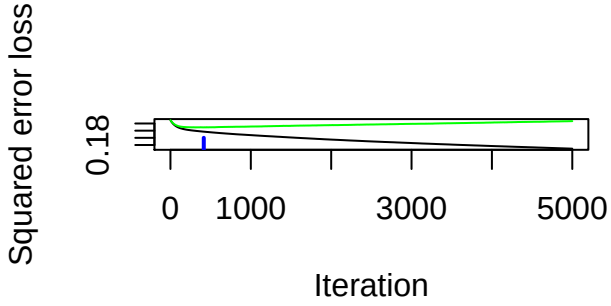

```
## [1] "shrink=0.01, depth=1, numTrees=762, MSE=0.39087802241386"
```



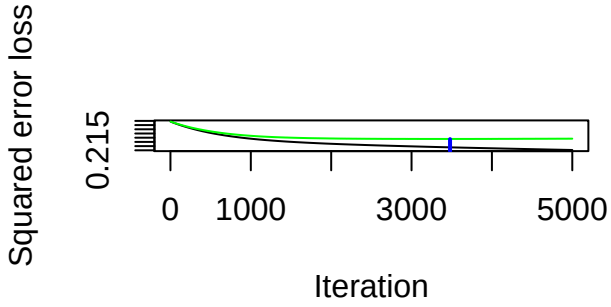

```
## [1] "shrink=0.001, depth=1, numTrees=4997, MSE=0.388437347779886"
```



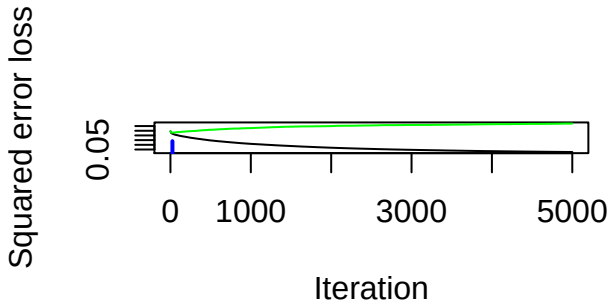

```
## [1] "shrink=0.1, depth=2, numTrees=45, MSE=0.386862203963062"
```



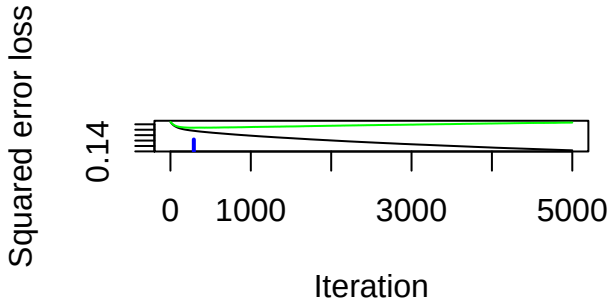

```
## [1] "shrink=0.01, depth=2, numTrees=415, MSE=0.392441794318044"
```



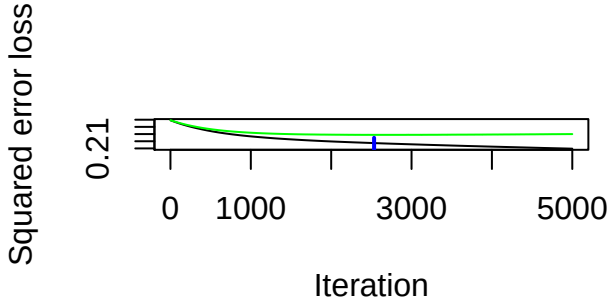

```
## [1] "shrink=0.001, depth=2, numTrees=3479, MSE=0.38825822261061"
```




```
## [1] "shrink=0.1, depth=3, numTrees=25, MSE=0.391494057526517"
```




```
## [1] "shrink=0.01, depth=3, numTrees=291, MSE=0.398209087485933"
```




```
## [1] "shrink=0.001, depth=3, numTrees=2534, MSE=0.389506434319924"
```

```
(err.opt = min(errs))
```

```
## [1] 0.3868622
```

```
id <- which.min(errs)
(lambdaOpt <- vals[id,1])
```

```
## [1] 0.1
```

```
(depthOpt <- vals[id,2])
```

```
## [1] 2
```

```
(treesOpt <- numTreesOptDiff[i])
```

```
## [1] 2534
```

```
mod.gbm.optDiff <- gbm(Home_Win ~ .,
                       data=games.train7,
                       distribution="gaussian", ## for regression
                       n.trees=treesOpt,
```

```
                    shrinkage=lambdaOpt,
                    interaction.depth = depthOpt)
```

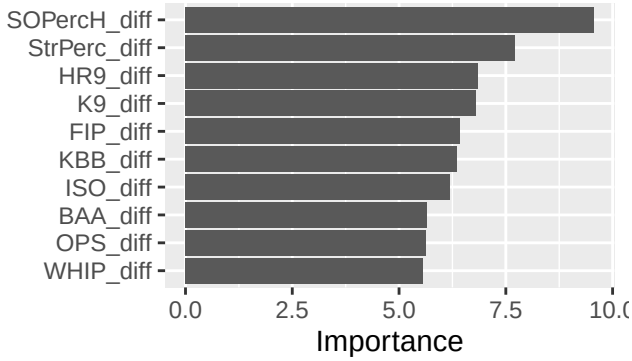
```
games.test7 %>%
  add_predictions(mod.gbm.optDiff) %>%
  mutate(class = if_else(pred > 0.5, 1, 0))%>%
  with(mean((Home_Win != class)))
```

```
## Using 2534 trees...
```

```
## [1] 0.4344741
```

- For some reason, this model doesn't perform nearly as well as our previous model, and is our only model that doesn't fit within our expected range with an error rate of 0.434.

```
vip(mod.gbm.optDiff)
```

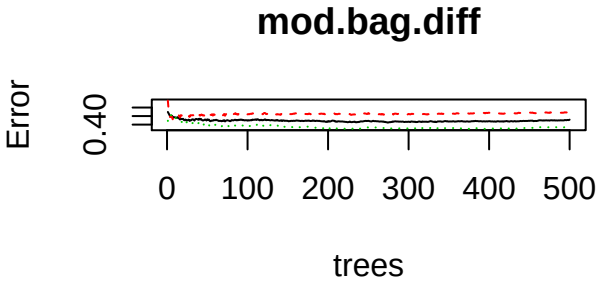


- Potentially one of the reasons this model doesn't perform as well as the others is the variable selection. Home runs allowed per 9 innings played is one of the most important variables, but isn't included anywhere in the other models.

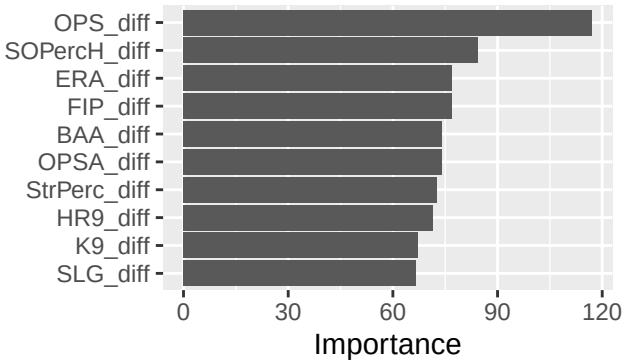
6.a.iv) Bagging

Although this was our worst performing model with the previous dataset, it is still worthwhile to see how this model will perform on the new dataset.

```
numTree <- 500
numPred <- ncol(games.train7)-1
set.seed(10)
mod.bag.diff <- randomForest(as.factor(Home_Win) ~ .,
                         data=games.train7,
                         ntree=numTree,
                         mtry=numPred)

plot(mod.bag.diff)
```



```
vip(mod.bag.diff)
```




```
res.vipx <- vip(mod.bag,num_features=55)
vip.datx <- res.vip2[["data"]]
vip.datx[52:55,]
```

```
## # A tibble: 4 x 3
##   Variable    Importance Sign
##   <chr>            <dbl> <chr>
## 1 SO               0.258 POS
## 2 FIP              0.228 NEG
## 3 NumPitchers      0.186 POS
## 4 ABA              0.143 NEG
```

```
bagtest.diff.df <- games.test7 %>%
  add_predictions(mod.bag.diff,
                  var="win.pred",
                  type = "class")

mse.bag.diff <- bagtest.diff.df %>%
  with(mean((Home_Win != win.pred)))

mse.bag.diff
```

```
## [1] 0.3973289
```

```
table(bagtest.diff.df$Home_Win, bagtest.diff.df$win.pred)
```

```
##
##       0   1
##   0 628 504
##   1 448 816
```

- We again see the trend of similar error rates and similar important variables, with an error rate of 0.39 and offensive strikeout rate being one of the most important variables. However, On Base Plus Slugging, the important variable from our arguably best model lasso, is now near the top.

6.a.iv)KNN

- As before, the KNN model needed too much computational power to be realistic in this instance.

6b) Percentage Difference Variables

6.b.i) Log Model

This dataset involves the percentage differences in home and away statistics. We will again begin by fitting the log model.

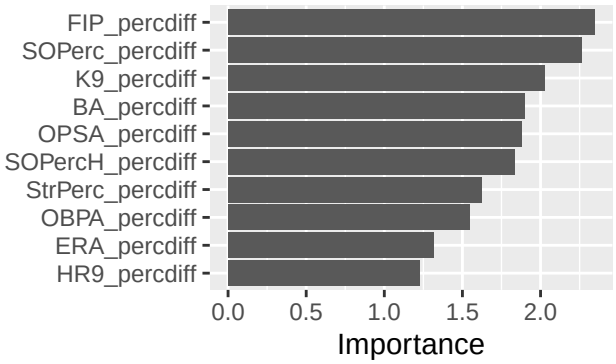
```
games.train8 <- games.train6 %>%
  select(5:22)

games.test8 <- games.test6 %>%
  select(5:22)

mod.log.perc <- glm(Home_Win ~.,
                data = games.train8,
                family = "binomial")

games.test8 <- games.test8 %>%
  add_predictions(mod.log.perc,
                  type = "response") %>%
  mutate(class = if_else(pred > 0.5, 1,0)) #class instead of response

table(games.test8$Home_Win, games.test8$class)
```

```
##
##       0   1
##   0 590 542
##   1 390 874
```

```
(err.log.perc <- with(games.test8, mean(Home_Win != class)))
```

```
## [1] 0.3889816
```

```
vip(mod.log.perc)
```



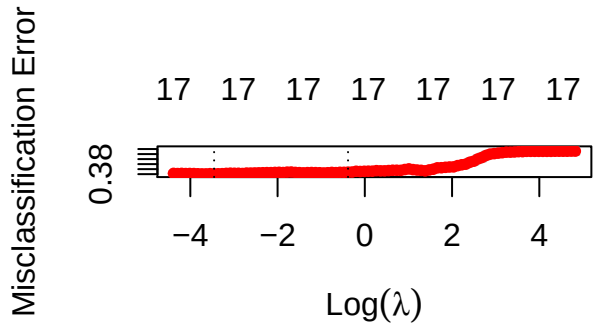
- Somewhat surprisingly, the error rate of this log model got slightly worse. We also see considerably different importatn variables, with 3 pitching statistics, Fielding Independant Pitching, Strikeout Percentage, and Strikeouts per 9 Innings being the top predictors.

6.b.ii) Ridge and Lasso

```
games.y.perc <- data.matrix(games.train8$Home_Win)


##New data set, gotta adjust the numbers selected
```

```
games.x.perc <- games.train8 %>%
  select(1:17)
games.x.perc[is.na(games.x.perc)] <- 0
games.x.perc <- data.matrix(games.x.perc)
```

```
ridge.cv.perc <- cv.glmnet(games.x.perc, games.y.perc,
                      family="binomial",
                      type.measure="class",
                      alpha=0)
plot(ridge.cv.perc)
```




```
log(ridge.cv.perc$lambda.1se)
```

```
## [1] -0.3847704
```

```
log(ridge.cv.perc$lambda.min)
```

```
## [1] -3.454884
```

```
lambda.opt <- ridge.cv.perc$lambda.1se
id <- with(ridge.cv.perc,which(ridge.cv.perc$lambda==lambda.opt))
(err.ridge.perc <- ridge.cv.perc$cvm[id])
```

```
## [1] 0.387985
```

```
mod.ridge.opt.perc <- glmnet(games.x.perc, games.y.perc,
                        family="binomial",
                        type.measure="class",
                        alpha=0,
                        lambda=lambda.opt)

##New data set, gotta adjust the numbers selected
games.test.y.perc <- data.matrix(games.test8$Home_Win)
games.test.x.perc <- games.test8 %>%
  select(1:17)
games.test.x.perc[is.na(games.test.x.perc)] <- 0
games.test.x.perc <- data.matrix(games.test.x.perc)
```

```
preds.perc <- predict(mod.ridge.opt.perc,
                      newx=games.test.x.perc,
                  type="class")
##The error rate on the testing dataset
table(games.test.y.perc ,preds.perc)
```

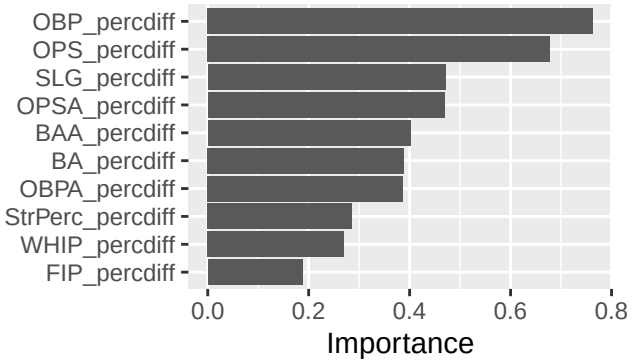
```
##                  preds.perc
## games.test.y.perc   0   1
```

```
##                  0 523 609
##                  1 337 927
```

```
(err.ridge.opt.perc <- mean((games.test.y.perc != preds.perc)))
```

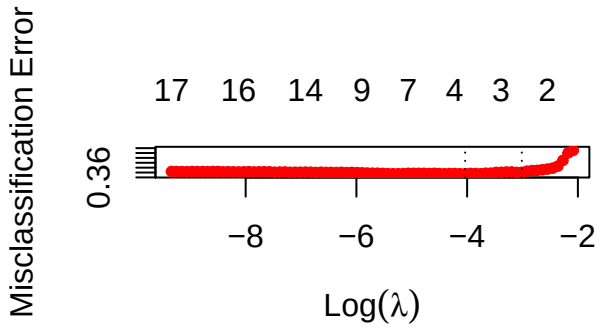
```
## [1] 0.3948247
```

```
vip(mod.ridge.opt.perc)
```

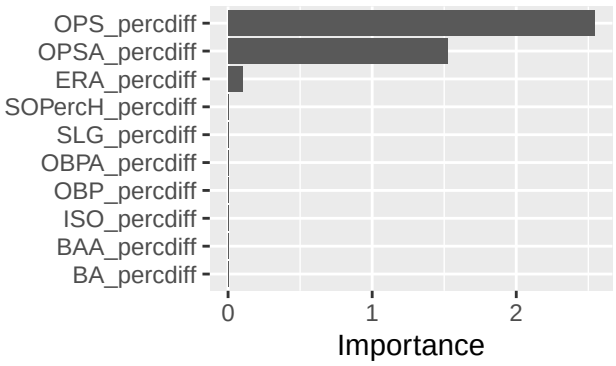


- Again, we see our models performing worse, with our ridge model having an error rate of .39. Also again, we see different important variables, with Batting Average moving down to the 6th most important variable, and On Base Percentage and On Base Plus Slugging becoming the most important.

```
lasso.cv.perc <- cv.glmnet(games.x.perc,
                           games.y.perc,
                           family="binomial",
                           type.measure="class",
                           alpha=1)

plot(lasso.cv.perc)
```




```
vip(lasso.cv.perc)
```



```
log(lasso.cv.perc$lambda.1se)
```

```
## [1] -3.012974
```

```
log(lasso.cv.perc$lambda.min)
```

```
## [1] -4.036345
```

```
lambda.opt <- lasso.cv.perc$lambda.1se
id <- with(lasso.cv.perc,which(lasso.cv.perc$lambda==lambda.opt))
(err.lasso.perc <- lasso.cv.perc$cvm[id])
```

```
## [1] 0.3808928
```

```
mod.lasso.opt.perc <- glmnet(games.x.perc, games.y.perc,
                        family="binomial",
                        type.measure="class",
                        alpha=1,
                        lambda=lambda.opt)
```

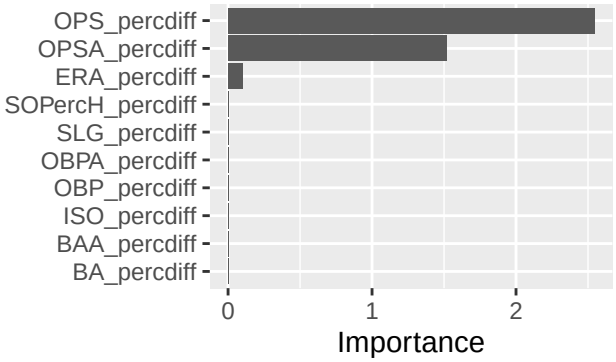
```
preds.perc <- predict(mod.lasso.opt.perc, newx=games.test.x.perc,
                  type="class")
##The error rate on the testing dataset
table(games.test.y.perc ,preds.perc)
```

```
##                  preds.perc
## games.test.y.perc   0   1
##                 0 532 600
##                 1 329 935
```

```
(err.lasso.opt.perc <- mean((games.test.y.perc != preds.perc)))
```

```
## [1] 0.3877295
```

```
vip(mod.lasso.opt.perc)
```



- This cross-validation error rate of 0.380 is definitely our best one yet. In our train/test model, we see similar performance, with an error rate of .387. We also see similar important variables here, with On Base Plus Slugging for and against being the most important variables in all our lasso models.

6.b.iii) Boosting

```
numTrees <- 2000
theShrinkage <- 0.01
theDepth <- 2
mod.gbm.diff <- gbm(Home_Win ~ .,
```

```
                data=games.train8,
                distribution="gaussian",
                n.trees=numTrees,
                shrinkage=theShrinkage,
                interaction.depth = theDepth)

games.test8$predGBM <- predict(mod.gbm.diff,newdata=games.test8)
```

```
## Using 2000 trees...
```

```
games.test8.boost <- games.test8%>%
  mutate(pred = if_else(predGBM > 0.5, 1, 0))
table(games.test8.boost$Home_Win, games.test8.boost$pred)
```

```
##
##       0   1
##   0 595 537
##   1 419 845
```

```
(mean((games.test8.boost$Home_Win != games.test8.boost$pred)))
```

```
## [1] 0.3989983
```

```
games.test8 <- games.test8%>%
  select(1:18)

cvGBM <- function(data.df, theShrinkage, theDepth, numTrees, numFolds=5){
  N <- nrow(data.df)
  folds <- sample(1:numFolds,N,rep=T)
  errs <- numeric(numFolds)
  for(fold in 1:numFolds){
    train.df.cv <- data.df %>%
      filter(folds != fold)
    test.df.cv <- data.df %>%
      filter(folds == fold)
    mod.gbm <- gbm(Home_Win ~ .,
                   data=train.df.cv,
                   distribution="gaussian",
                   n.minobsinnode=10,
                   interaction.depth = theDepth,
                   shrinkage=theShrinkage,
                   n.trees=numTrees)
    test.df.cv$pred.gbm <- predict(mod.gbm,
                         newdata=test.df.cv,
                         n.trees=numTrees)
    test.df.cv <- test.df.cv%>%
      mutate(pred = if_else(pred.gbm > 0.5, 1, 0))
    errs[fold] <- with(test.df.cv,mean((Home_Win != pred)))
    }
  mean(errs)
}
```

```
lambda <- 0.01
depth <- 1
numTrees <- 100
cvGBM(games.train8,lambda,depth,numTrees)
```

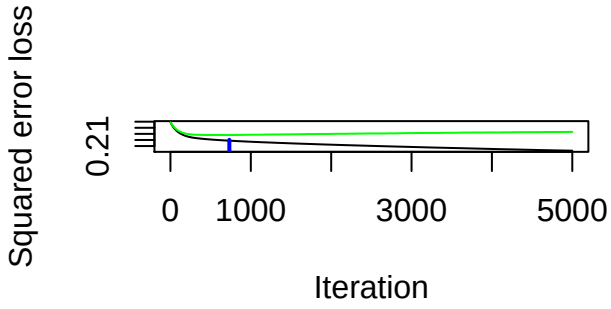
```
## [1] 0.3839062
```

```
cvGBM(games.train8,lambda,depth,100 * numTrees)
```

```
## [1] 0.3993899
```

```
numTrees <- 5000
mod.gbm.cv.perc <- gbm(Home_Win ~ .,
                  data=games.train8,
                  distribution="gaussian",
                  n.trees=numTrees,
                  shrinkage=lambda,
                  interaction.depth = depth,
                  ##indicate folds here
                  cv.folds = 5,
                  ## restrict the minimum number of observations in a node
                  n.minobsinnode=10,
                  n.cores = 4) ## <-- use the cores on your computer
```

```
gbm.best.perc <- gbm.perf(mod.gbm.cv.perc,method="cv")
```




```
(numTreesOpt.perc <- gbm.best.perc)
```

```
## [1] 733
```

```
mod.gbm.opt.perc <- gbm(Home_Win ~ .,
                   data=games.train8,
                   distribution="gaussian", ## for regression
                   n.trees=numTreesOpt.perc,
                   shrinkage=lambda,
                   interaction.depth = depth)
games.test7a <- games.test8 %>%
  add_predictions(mod.gbm.opt.perc)
```

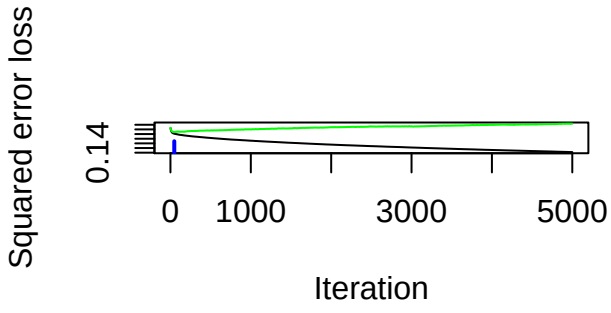
```
## Using 733 trees...
```

```
games.test7a <- games.test7a%>%
  mutate(class = if_else(pred > 0.5, 1, 0))
(mean(games.test7a$Home_Win != games.test7a$class))
```

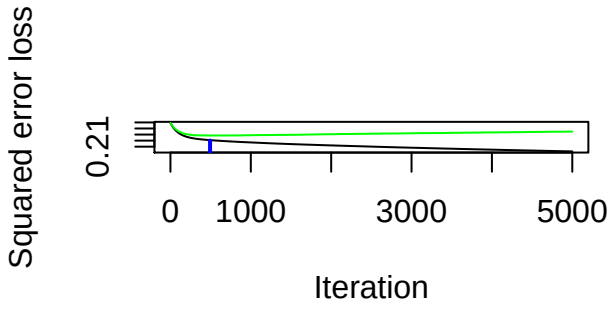
```
## [1] 0.3843907
```

```
shrink.vals <- c(0.1,0.01,0.001)
depth.vals <- c(1,2,3)
numTreesMax <- 5000
vals <- expand.grid(s=shrink.vals,d=depth.vals)
errs <- matrix(nrow=3,ncol=3)
i <- 1
```

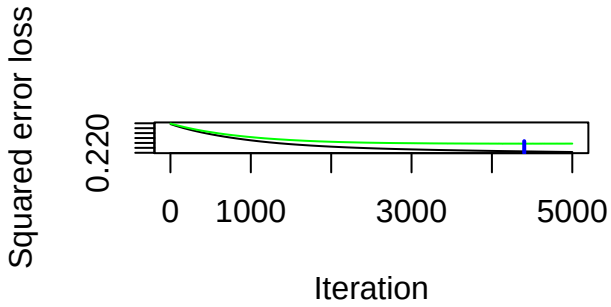
```
errs <- numeric(9)
numTreesOptPerc <- numeric(9)
for(i in 1:9){
  lambda <- vals[i,1]
  depth <- vals[i,2]
  mod.gbm.cv.perc2 <- gbm(Home_Win ~.,
                    data=games.train8,
                    distribution="gaussian",
                    n.trees=numTreesMax,
                    shrinkage=lambda,
                    interaction.depth = depth,
                    ##indicate folds here
                    cv.folds = 5,
                    ## restrict the minimum number of observations in a node
                    n.minobsinnode=10,
                    n.cores = 4)
gbm.best.perc2 <- gbm.perf(mod.gbm.cv.perc2,method="cv")
numTreesOptPerc[i] <- gbm.best.perc2
errs[i] <- cvGBM(games.train8,
                 lambda,
                 depth,
                 numTreesOptPerc[i])
print(sprintf("shrink=%s, depth=%s, numTrees=%s, MSE=%s",lambda,depth, numTreesOptPerc[i],errs[i]))
}
```



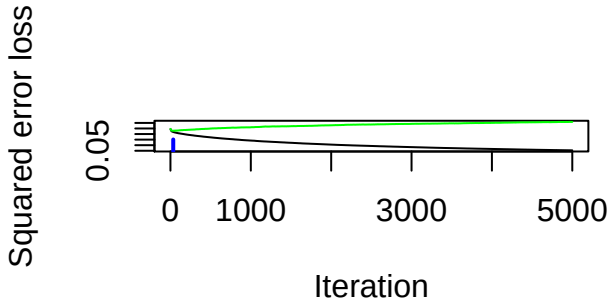

```
## [1] "shrink=0.1, depth=1, numTrees=48, MSE=0.390288545870398"
```



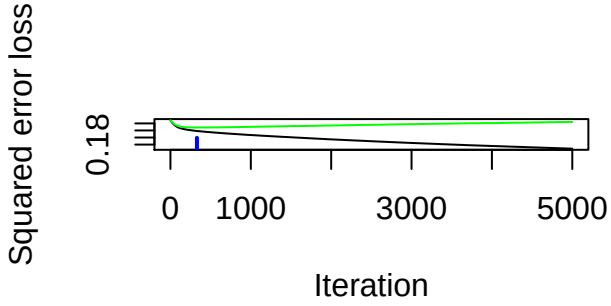

```
## [1] "shrink=0.01, depth=1, numTrees=493, MSE=0.385584508813369"
```



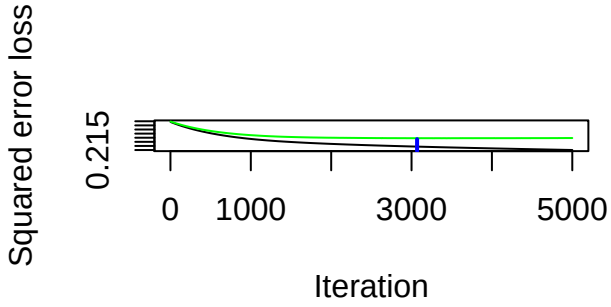

```
## [1] "shrink=0.001, depth=1, numTrees=4402, MSE=0.383341861120449"
```



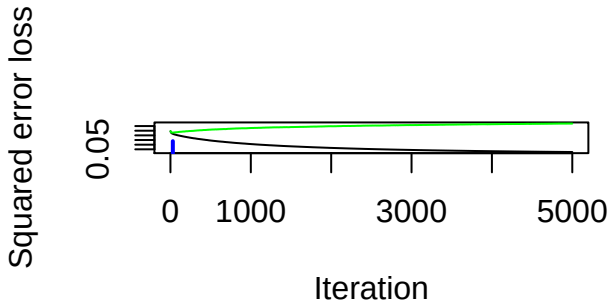

```
## [1] "shrink=0.1, depth=2, numTrees=35, MSE=0.383823407081747"
```



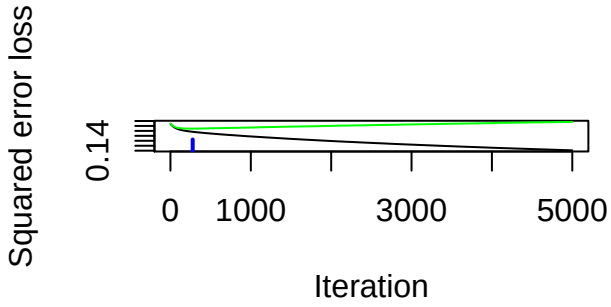

```
## [1] "shrink=0.01, depth=2, numTrees=330, MSE=0.391792952446878"
```



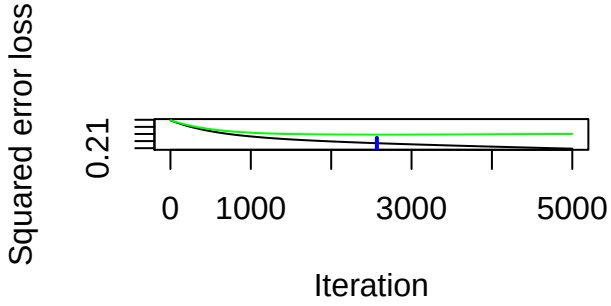

```
## [1] "shrink=0.001, depth=2, numTrees=3069, MSE=0.3814492456156"
```




```
## [1] "shrink=0.1, depth=3, numTrees=30, MSE=0.390636075116281"
```




```
## [1] "shrink=0.01, depth=3, numTrees=277, MSE=0.393722202759132"
```




```
## [1] "shrink=0.001, depth=3, numTrees=2569, MSE=0.397032404947173"
```

```
(err.opt = min(errs))
```

```
## [1] 0.3814492
```

```
id <- which.min(errs)
(lambdaOpt <- vals[id,1])
```

```
## [1] 0.001
```

```
(depthOpt <- vals[id,2])
```

```
## [1] 2
```

```
(treesOpt <- numTreesOptPerc[i])
```

```
## [1] 2569
```

```
mod.gbm.optPerc <- gbm(Home_Win ~ .,
                   data=games.train8,
                   distribution="gaussian", ## for regression
                   n.trees=treesOpt,
```

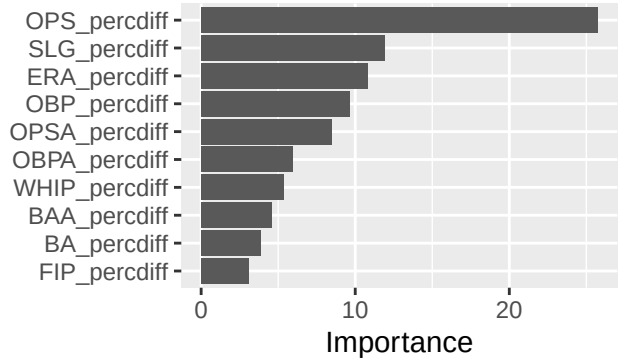
```
                    shrinkage=lambdaOpt,
                    interaction.depth = depthOpt)
```

```
games.test8 %>%
  add_predictions(mod.gbm.optPerc) %>%
  mutate(class = if_else(pred > 0.5, 1, 0))%>%
  with(mean((Home_Win != class)))
```

```
## Using 2569 trees...
```

```
## [1] 0.3860601
```

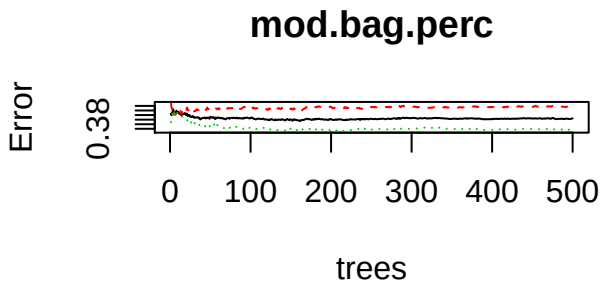
```
vip(mod.gbm.optPerc)
```



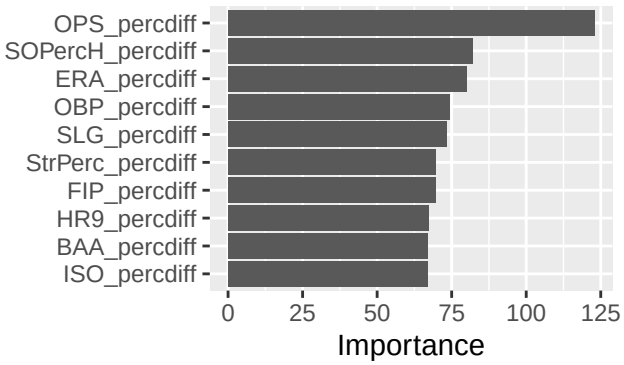
- After carrying out the boosting process one last time, we saw again that the boosting model performs very well in trying to predict these outcomes. Our error rate of .383 using our arbitrary number of trees and cross-validation compared nicely to our optimal error rate in a train/test model of 0.384. We again see On Base Plus Slugging as an important variable, with ERA also showing major importance.

6.b.iv) Bagging

```
numTree <- 500
numPred <- ncol(games.train3)-1
set.seed(10)
mod.bag.perc <- randomForest(as.factor(Home_Win) ~ .,
                        data=games.train8,
                        ntree=numTree,
                        mtry=numPred)

plot(mod.bag.perc)
```




```
vip(mod.bag.perc)
```




```
res.vipx <- vip(mod.bag.perc,num_features=55)
vip.datx <- res.vip2[["data"]]
vip.datx[52:55,]
```

```
## # A tibble: 4 x 3
##   Variable    Importance Sign
##   <chr>            <dbl> <chr>
## 1 SO               0.258 POS
## 2 FIP              0.228 NEG
## 3 NumPitchers      0.186 POS
## 4 ABA              0.143 NEG
```

```
bagtest.perc.df <- games.test8 %>%
  add_predictions(mod.bag.perc,
                  var="win.pred",
                  type = "class")

mse.bag.perc <- bagtest.perc.df %>%
  with(mean((Home_Win != win.pred)))

mse.bag.perc
```

```
## [1] 0.3944073
```

```
table(bagtest.perc.df$Home_Win, bagtest.perc.df$win.pred)
```

```
##
##       0   1
##   0 631 501
##   1 444 820
```

- After one more iteration of our bagging model, we still see that the bagging model performs worse than all our other models, with error rates above .39. We also see the continued trend of offensive strikeout percentage as one of the most important variables in the bagging model.

6.b.v) KNN

Even though we reduced the number of predictors, the dimensionality of the dataset was still too much for the KNN model to be computationally reasonable for its performance level.

```
calc_one_MSE_knn <- function(kNear){
    train.df <- games.train8
    test.df <- games.test8

    mod.knn <- knn3(as.factor(Home_Win) ~ .,
```

```
                    data=train.df,
                    k=kNear)

    test.df <- test.df %>%
      add_predictions(mod.knn,
                      type="class",
                      var="knnClass")

    with(test.df,mean(Home_Win != knnClass))
}


M <- 1
kvalue <- 1
calc_one_MSE_knn_aux <- function(iter) {
  calc_one_MSE_knn(kvalue)
}
mse.vec <- map_dbl(1:M, calc_one_MSE_knn_aux)


M <- 1
kvalue <- 1
mse.vec <- map_dbl(1:M, ~calc_one_MSE_knn(kvalue))


M <- 1
calc_MSE_knn<- function(kvalue) {
  mse.vec <- map_dbl(1:M, ~calc_one_MSE_knn(kvalue))
  mean <- mean(mse.vec)
  tibble(error = mean)
}



kMax <- 100
mse.tbl <-map_df(1:kMax, calc_MSE_knn)
mse.tbl <- mse.tbl %>%
  mutate(k=row_number())


ggplot(mse.tbl) +
  geom_line(aes(x=k, y=error), color="blue")+
  geom_smooth(aes(x=k, y=error), color="blue",se=F)

(mse.opt.k <- mse.tbl%>%
    filter(error == min(error))%>%
    filter(k == max(k)))
```

```
numTrain <- nrow(games.train8) #Number of observations in training
numTest <- nrow(games.train8)  #Number of observations in testing

#Function to calculate the MSE for one simulation with a fixed kNear for KNN
calc_one_MSE_knn <- function(kNear){
    train.df <- sample_n(games.train8,numTrain, rep = T)
```

```
    test.df <- sample_n(games.test8,numTest, rep = T)

    mod.knn <- knn3(as.factor(Home_Win) ~ .,
                    data=train.df,
                    k=kNear)

    test.df <- test.df %>%
      add_predictions(mod.knn,
                      type="class",
                      var="knnClass")

    with(test.df,mean(Home_Win != knnClass))
}

M <- 100 #Number of simulation
#Function to calculate the mean and sd of MSE with a fixed kNear for KNN
calc_MSE_knn<- function(kvalue) {
  mse.vec <- map_dbl(1:M, ~calc_one_MSE_knn(kvalue))
  mse.mean <- mean(mse.vec)
  mse.sd <- sd(mse.vec)
  tibble(mse.mean=mse.mean, mse.sd=mse.sd)
}

kMax <- 100
mse.tbl <-map_df(1:kMax, calc_MSE_knn) #We calculate the mean,sd of MSE for all parameters

mse.tbl <- mse.tbl %>%
  mutate(k=row_number()) # We add the parameter (k)

# We plot the MSE bands as a function of k
ggplot(mse.tbl) +
  geom_line(aes(x=k, y=mse.mean), color="blue")+
  geom_smooth(aes(x=k, y=mse.mean), color="blue",se=F)+
  geom_line(aes(x=k, y=mse.mean-mse.sd), color="red")+
  geom_smooth(aes(x=k, y=mse.mean-mse.sd), color="red", se=F)+
  geom_line(aes(x=k, y=mse.mean+mse.sd), color="red")+
  geom_smooth(aes(x=k, y=mse.mean+mse.sd), color="red",se =F)

(mse.opt.boot <- mse.tbl%>%
    mutate(val = mse.mean - mse.sd)%>%
    filter(val <= mse.mean)%>%
    filter(k == max(k)))


opt.K <- mse.opt.boot$k

mod.knn.opt <- knn3(as.factor(Home_Win) ~.,
                              data = games.train8,
                              k = opt.K)
```

7 Conclusions

Our boosting models continually performed as good or better than all our other models, with our lasso models coming in a close second. The bagging models routinely performed worse than all our other models.

Throughout the analysis, we see that our models performed better in accurately predicting wins than they did in accurately predicting losses. This could be do in large part to the random aspects of the game of baseball, where even if a team is completely overmatched in every major statistical category going into a game, an underdog team will win roughly 30% of the time.

While our models performed comparably to other models present in mainstream sports betting or sports mathematical literature, most of which use more statistical and theoretical methods as opposed to our machine learning methods, we can still improve. One way to improve would be to include starting pitcher. Our model does not account for starting pitcher, which can greatly influence the outcome of a particular game, moreso than they can impact a teams overall statistical output or overall team record.