ADM Final Project, League of Legends Game Result Prediction Based on First 10 Mins Data

Shan Chen, Ken Wang

5/18/2020

```
library(class)
library(MASS)
library(glmnet)
library(randomForest)
library(gbm)
library(factoextra)
library(ggrepel)
library(GGally)
library(e1071)
library(neuralnet)
library(tidyverse)
```

We will only use blue sides' data here to predict because the side does not affect win rate:

```
data.df <- read.csv("LOL.csv",header=T) # read in data
names(data.df)
```

```
##  [1] "gameId"                       "blueWins"
##  [3] "blueWardsPlaced"              "blueWardsDestroyed"
##  [5] "blueFirstBlood"               "blueKills"
##  [7] "blueDeaths"                   "blueAssists"
##  [9] "blueEliteMonsters"            "blueDragons"
## [11] "blueHeralds"                  "blueTowersDestroyed"
## [13] "blueTotalGold"                "blueAvgLevel"
## [15] "blueTotalExperience"          "blueTotalMinionsKilled"
## [17] "blueTotalJungleMinionsKilled" "blueGoldDiff"
## [19] "blueExperienceDiff"           "blueCSPerMin"
## [21] "blueGoldPerMin"               "redWardsPlaced"
## [23] "redWardsDestroyed"            "redFirstBlood"
## [25] "redKills"                     "redDeaths"
## [27] "redAssists"                   "redEliteMonsters"
## [29] "redDragons"                   "redHeralds"
## [31] "redTowersDestroyed"           "redTotalGold"
## [33] "redAvgLevel"                  "redTotalExperience"
## [35] "redTotalMinionsKilled"        "redTotalJungleMinionsKilled"
## [37] "redGoldDiff"                  "redExperienceDiff"
## [39] "redCSPerMin"                  "redGoldPerMin"
```

```
cor(data.df,data.df$blueWins)
```

```
##                                      [,1]
## gameId                       9.851279e-04
## blueWins                     1.000000e+00
## blueWardsPlaced              8.695109e-05
## blueWardsDestroyed           4.424680e-02
## blueFirstBlood               2.017693e-01
## blueKills                    3.373576e-01
```

```
## blueDeaths                    -3.392967e-01
## blueAssists                    2.766850e-01
## blueEliteMonsters              2.219442e-01
## blueDragons                    2.137677e-01
## blueHeralds                    9.238472e-02
## blueTowersDestroyed            1.155665e-01
## blueTotalGold                  4.172126e-01
## blueAvgLevel                   3.578198e-01
## blueTotalExperience            3.961407e-01
## blueTotalMinionsKilled         2.249095e-01
## blueTotalJungleMinionsKilled   1.314449e-01
## blueGoldDiff                   5.111191e-01
## blueExperienceDiff             4.895579e-01
## blueCSPerMin                   2.249095e-01
## blueGoldPerMin                 4.172126e-01
## redWardsPlaced                -2.367124e-02
## redWardsDestroyed             -5.540031e-02
## redFirstBlood                 -2.017693e-01
## redKills                      -3.392967e-01
## redDeaths                      3.373576e-01
## redAssists                    -2.710469e-01
## redEliteMonsters              -2.215511e-01
## redDragons                    -2.095159e-01
## redHeralds                    -9.717188e-02
## redTowersDestroyed            -1.036956e-01
## redTotalGold                  -4.113962e-01
## redAvgLevel                   -3.521268e-01
## redTotalExperience            -3.875876e-01
## redTotalMinionsKilled         -2.121715e-01
## redTotalJungleMinionsKilled   -1.109935e-01
## redGoldDiff                   -5.111191e-01
## redExperienceDiff             -4.895579e-01
## redCSPerMin                   -2.121715e-01
## redGoldPerMin                 -4.113962e-01
```

Seems like there are repetitions in the variables. Since the overall resources are rather constant, we only need to know the stats for the blue team to know the genearl game situation. Thus, we will just use the first half of the data that contains the blue team stats to save time and storage.

```
blue.df <- data.df[,-c(1,22:40)] # keep only blue
names(blue.df)
```

```
##  [1] "blueWins"                    "blueWardsPlaced"
##  [3] "blueWardsDestroyed"          "blueFirstBlood"
##  [5] "blueKills"                   "blueDeaths"
##  [7] "blueAssists"                 "blueEliteMonsters"
##  [9] "blueDragons"                 "blueHeralds"
## [11] "blueTowersDestroyed"         "blueTotalGold"
## [13] "blueAvgLevel"                "blueTotalExperience"
## [15] "blueTotalMinionsKilled"      "blueTotalJungleMinionsKilled"
## [17] "blueGoldDiff"                "blueExperienceDiff"
## [19] "blueCSPerMin"                "blueGoldPerMin"
```

Indicator Analysis with PCA

We want to start with PCA to look at how significant each indicators are and how their general relationships affect the game outcome.

Build the data and check the data, a pretty good break even on win rate.

```
names(blue.df) # column names
```

```
##  [1] "blueWins"                      "blueWardsPlaced"
##  [3] "blueWardsDestroyed"            "blueFirstBlood"
##  [5] "blueKills"                     "blueDeaths"
##  [7] "blueAssists"                   "blueEliteMonsters"
##  [9] "blueDragons"                   "blueHeralds"
## [11] "blueTowersDestroyed"           "blueTotalGold"
## [13] "blueAvgLevel"                  "blueTotalExperience"
## [15] "blueTotalMinionsKilled"        "blueTotalJungleMinionsKilled"
## [17] "blueGoldDiff"                  "blueExperienceDiff"
## [19] "blueCSPerMin"                  "blueGoldPerMin"
```

```
with(blue.df,table(blueWins)) # table
```

```
## blueWins
##    0    1
## 4949 4930
```

```
numPreds <- ncol(blue.df)-1 # humber of predictors
blue.x <- data.matrix(blue.df[,2:20]) # x
blue.y <- data.matrix(blue.df[,1]) # y
```

Check to see if the data are scaled

```
summary(apply(blue.x,2,mean))
```

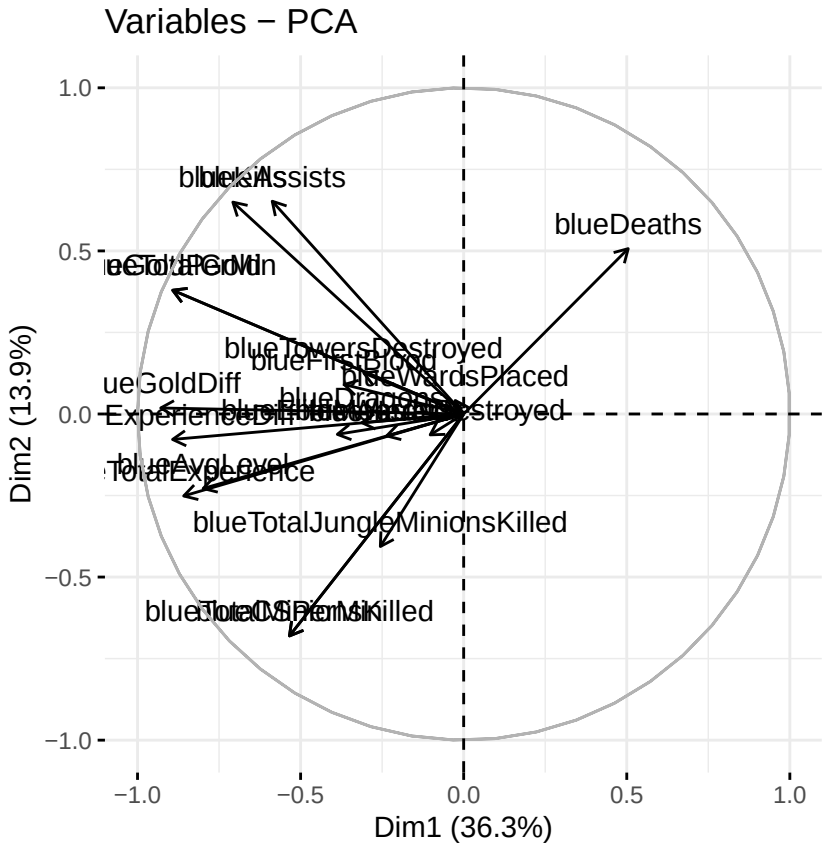
```
##      Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
##   -33.620     0.527     6.645  1916.012    36.399 17928.110
```

Nope

```
blue.x <- scale(blue.x,center=T) # scale and center
summary(apply(blue.x,2,mean)) # check if scaled again
```

```
##       Min.    1st Qu.     Median       Mean    3rd Qu.       Max.
## -5.024e-16 -1.189e-16 -4.529e-18  6.451e-17  2.337e-17  1.357e-15
```

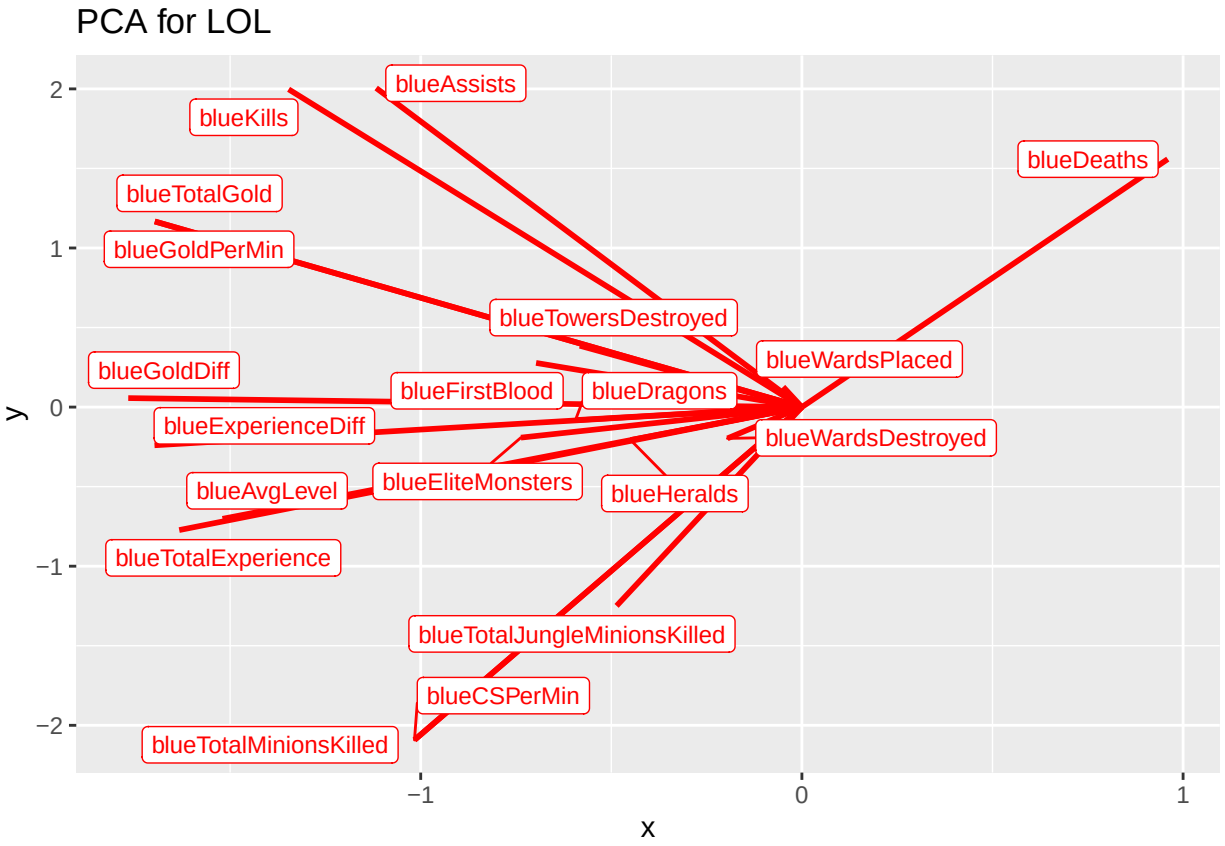
```
blue.mat = data.matrix(blue.x) # data matrix
indicatornames <- colnames(blue.mat) # get indicators
mod.pca <- prcomp(blue.mat) # pca
fviz_pca_var(mod.pca) # graph
```



The original graph is kinda hard to see. Let's build our own graph.

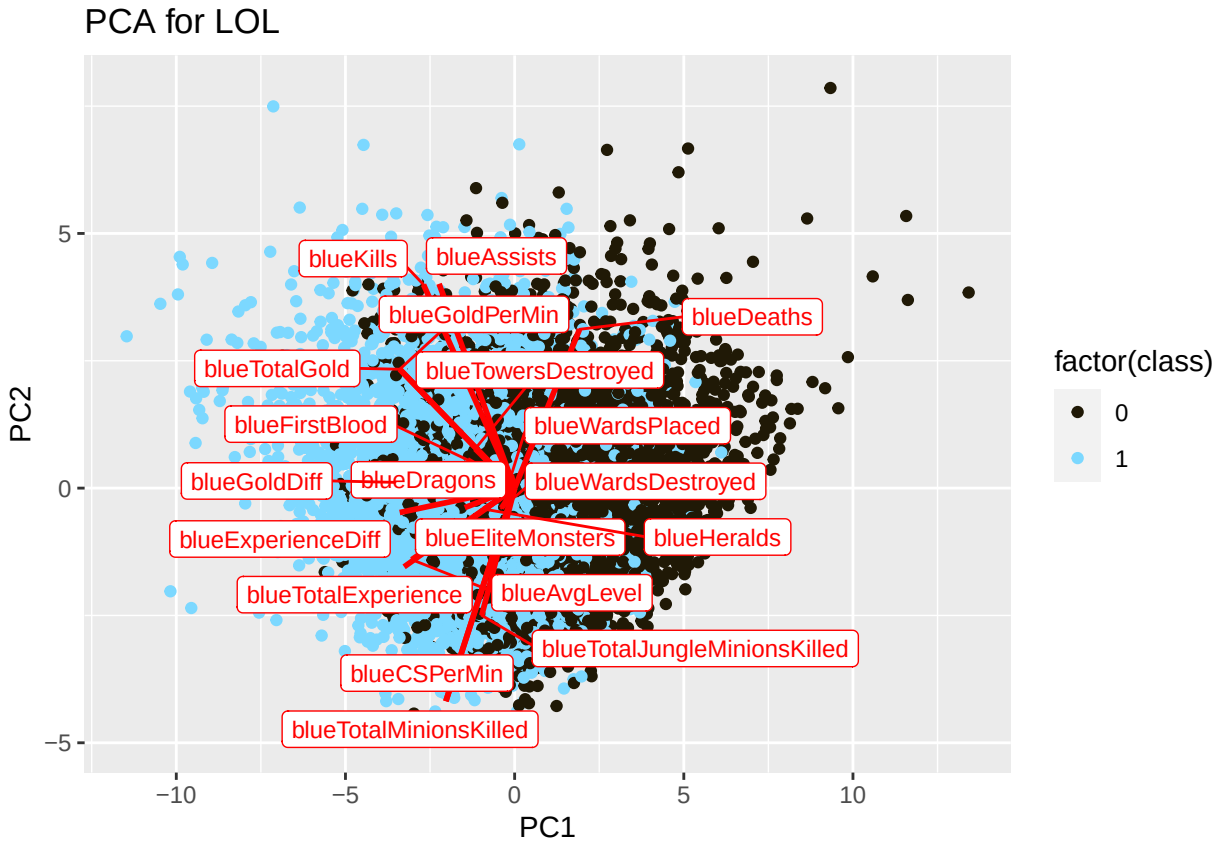
```
Rotated.df <- data.frame(mod.pca$x) # rotated vectors
rotation.df <- data.frame(mod.pca$rotation) # rotation data frame
rotation.df$indicators <- indicatornames # add indicators

sc <- 5 ## get everything on the same scale
Rotated.df %>%
  ggplot()+
  geom_segment(data=rotation.df,
               aes(x=0,y=0,xend=sc*PC1,yend=sc*PC2),size=1,color="red")+ # add loading
  geom_label_repel(data=rotation.df,
                   aes(sc*PC1,sc*PC2,label=indicators),size=3,color="red")+
  labs(title="PCA for LOL") # too much stuff on there
```



We can see that blue gold difference and blue experience diff are pretty influential factors for pc1. There are generally four categories of indicators: blueDeaths is a negative indicator on its own, which is not beneficial for winning the game; blueTotalJungleMinionsKilled, blueCSPerMin, blueTotalMinionsKilled are groups of indicators for basic income, which contributes to total number of gold; blueTotalExperience, blueAvgLevel, blueExperienceDiff, blueGoldDiff are groups of indicators for experience and levels. blueKills, blueAssists, blueTotalGold, blueGoldPerMin are groups of indicators that contributes to main income lead. This makes sense that kills and assists are main resource for gold; blueFirstBlood, blueEliteMonsters, blueDragons, blueHeralds, blueTowersDestroyed, blueWardsPlaced, blueWardsDestroyed are minor indicators. Among those, getting objects like dragons and towers seem to contribute more to gold lead than other indicators.

```
# Another graph with all the data points
sc=10
Rotated.df$class = blue.df[,1] # add classification
Rotated.df %>%
  ggplot()+
  geom_point(aes(PC1,PC2,color=factor(class)))+ # rotated values
  scale_colour_manual(values = c("#201906","#7CD8FF"))+
  geom_segment(data=rotation.df,
               aes(x=0,y=0,xend=sc*PC1,yend=sc*PC2),size=1,color="red")+ # add loading
  geom_label_repel(data=rotation.df,
                aes(sc*PC1,sc*PC2,label=indicators),size=3,color="red")+
  labs(title="PCA for LOL") # too much stuff on there
```



Method 1 Logistic Regression

Let's start with logistic regression

```
# A function that returns MSE with cross validation
errCV_log <- function(data.df,val){
  numFolds = 10 # number of folds
  dataSize <- nrow(data.df) # data size
  folds <- sample(1:numFolds,dataSize,rep=T) # folds
  err <- numeric(numFolds) # empty array
  for(fold in 1:numFolds){
    train.df <- data.df[folds != fold,] # train
    test.df <- data.df[folds == fold,] # test
    mod <- glm(blueWins ~ .,data=train.df,family="binomial")
    test.df$prob <- predict(mod,newdata=test.df,type="response") # predictions
    test.df$pred <- test.df$prob > val # classify into binary
    err[fold] <- with(test.df,mean(blueWins != pred)) # error rate
  }
  mean(err) # return mean error rate
}
```

```
vals = seq(0.4,0.6,by=0.01) # tried out and this seemed to be the best interval
errs_log = map_dbl(vals, function(x) errCV_log(blue.df,x))
vals[which.min(errs_log)] # best cutoff value
```

```
## [1] 0.48
```

```
(err_log = min(errs_log))
```

```
## [1] 0.2691779
```

```
final_results = data.frame(model=NA,error_rate=NA)
final_results[1,1]="Logistic"
final_results[1,2]=err_log
```

Not bad.

Method 2 KNN

Let's try KNN next.

```
# building function for knn
N <- nrow(blue.x)
knnERR <- function(kVal){
  numFolds = 10 # number of folds
  folds <- sample(1:numFolds,N,rep=T) # fold id
  errs <- numeric(numFolds) # array to store err rate
  for(fold in 1:numFolds){
    train.x <- blue.x[folds != fold,]
    train.y <- blue.y[folds != fold]
    test.x <- blue.x[folds == fold,]
    test.y <-  blue.y[folds == fold]
    mod.knn <- knn(train.x,test.x,train.y,k=kVal) # knn
    errs[fold] <- mean(mod.knn != test.y) # erra rate
  }
  mean(errs)
}

# Test out some values for k
knnERR(1)
```

```
## [1] 0.3724218
```

```
knnERR(10)
```

```
## [1] 0.303888
```

```
knnERR(30)
```

```
## [1] 0.2855677
```

```
knnERR(50)
```

```
## [1] 0.2805113
```

```
knnERR(70)
```

```
## [1] 0.2786099
```

```
knnERR(90)
```

```
## [1] 0.2777886
```

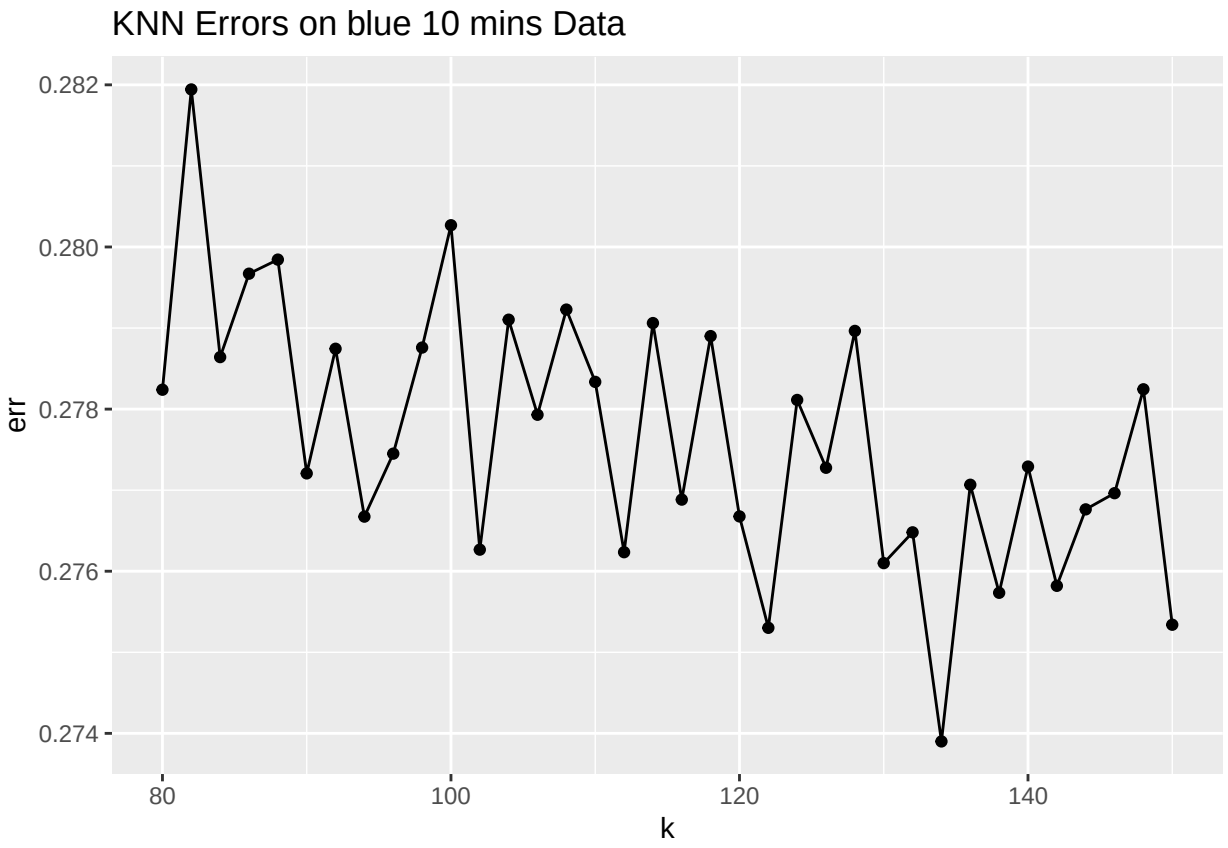
```
knnERR(110)
```

```
## [1] 0.2784909
```

Explore the k values

```
kVals <- seq(80,150,by=2) # tested out some best intervals.
# This seems to be a good interval
allErrs <- map_dbl(kVals,knnERR) # try out for all kvals

data.frame(k=kVals,err=allErrs) %>%
  ggplot()+
  geom_point(aes(k,err))+
  geom_line(aes(k,err))+
  labs(title="KNN Errors on blue 10 mins Data") # plot the data
```



They are quite the same, still monotonically decaying about 35 is pretty good

```
id = which.min(allErrs) # minimum error id
(min_kval = kVals[id]) # best k val
```

```
## [1] 134
```

```
(err.knn <-allErrs[id]) # minimum error
```

```
## [1] 0.2739004
```

```
final_results[2,1] = "KNN"
final_results[2,2] = err.knn
final_results
```

```
##      model error_rate
## 1 Logistic  0.2691779
## 2      KNN  0.2739004
```

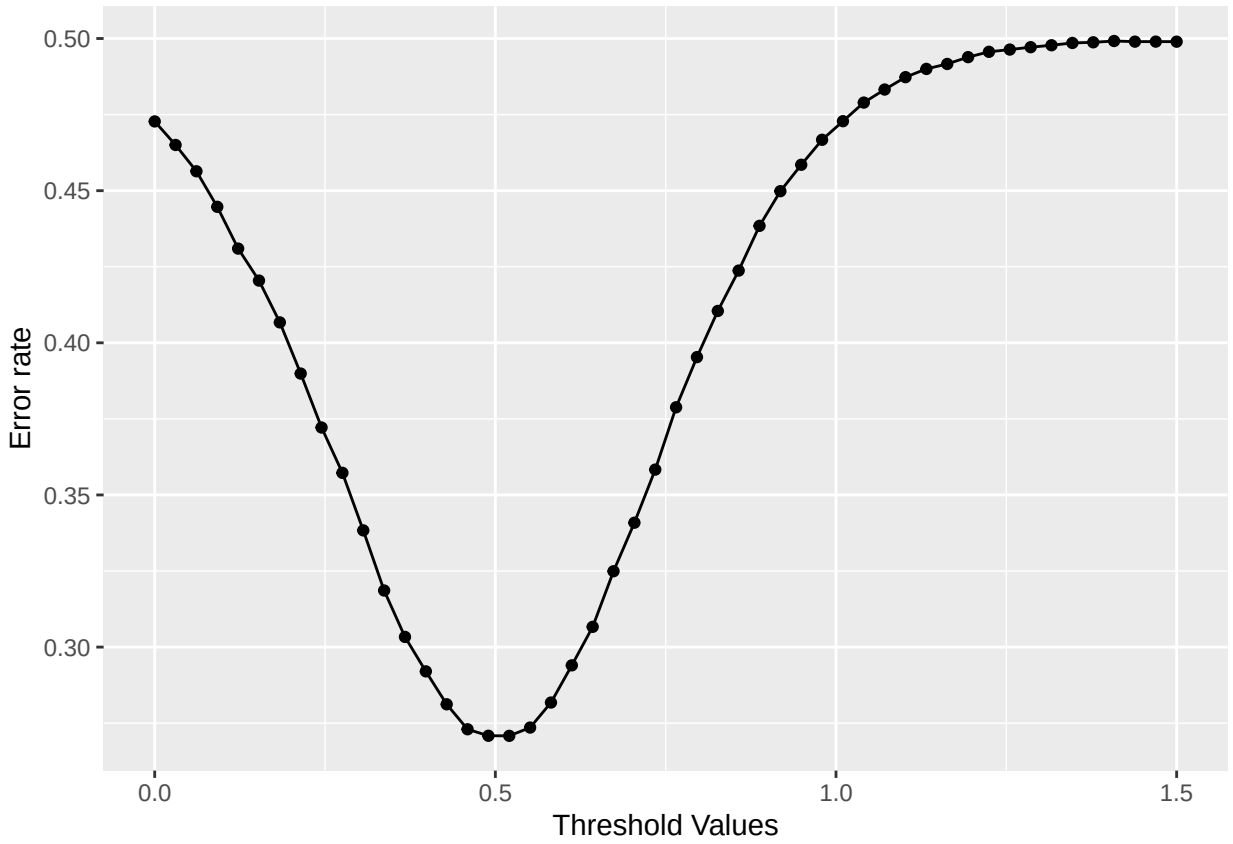
Ok, the error rate for KNN seems a bit higher than logistic, but still quite similar.

Method 3 Regular Linear Classifier

Let's try the really simple linear classifier and see how it does

```
# A function that returns error rate with cross validation
errCV_lc <- function(data.df,thresVal){
  numFolds=10 # number of folds
  dataSize <- nrow(data.df) # data size
  folds <- sample(1:numFolds,dataSize,rep=T) # folds
  err <- numeric(numFolds) # empty array
  for(fold in 1:numFolds){
    train.df <- data.df[folds != fold,] # train
    test.df <- data.df[folds == fold,] # test
    mod <- lm(blueWins ~ .,data=train.df) # model
    test.df$prob <- predict(mod,newdata=test.df) # predictions
    test.df$pred = ifelse(test.df$prob > thresVal,1,0) # classify
    err[fold] <- with(test.df,mean(blueWins != pred)) # error rate
  }
  mean(err) # return mean
}
```

```
threshVals <- seq(0,1.5,length=50) # threshold values
errs.lc <- map_dbl(threshVals, function(x) errCV_lc(blue.df,x)) # predict for all threshold
data.frame(threshVals,errs.lc) %>%
  ggplot(aes(x=threshVals,y=errs.lc))+
  geom_line()+
  geom_point()+
  labs(x="Threshold Values",y="Error rate") # plot
```




```
thresh_id = which.min(errs.lc) # best threshold id
(thresh_val = threshVals[thresh_id]) # best threshold value
```

```
## [1] 0.5204082
```

```
(err_lc = errs.lc[thresh_id]) # min err for linear classifier
```

```
## [1] 0.2708367
```

```
final_results[3,1]="Regular Linear Classifier"
final_results[3,2]=err_lc
final_results
```

```
##                         model error_rate
## 1                    Logistic  0.2691779
## 2                         KNN  0.2739004
## 3 Regular Linear Classifier  0.2708367
```

Actually, not bad! The error rate for linear classifier and logistic are similar. KNN does a bit worse but is still pretty close.

Method 4 One Hot Encoding

What if we try one hot encoding? How would that be different?

```
data.hot <- blue.df %>%
  mutate(blue = if_else(blueWins==1,1,0),
         red = if_else(blueWins==0,1,0)) # encode
```

```r
# A function that returns error rate with cross validation
col_hot = ncol(data.hot) # number of columns for the data
errCV_hot <- function(data.hot){
  numFolds=10 # number of folds
  dataSize <- nrow(data.hot) # data size
  folds <- sample(1:numFolds,dataSize,rep=T) # folds
  err <- numeric(numFolds) # empty array
  for(fold in 1:numFolds){
    train.blue <- data.hot[folds != fold,-c(1,col_hot)] # train for blue
    train.red <- data.hot[folds != fold,-c(1,col_hot-1)] # train for red
    test.df <- data.hot[folds == fold,] # test
    mod.blue <- lm(blue ~ .,data=train.blue) # model for Y0
    mod.red <- lm(red ~ .,data=train.red) # model for Y1
    test.df$blue_pred <- predict(mod.blue,newdata=test.df) # predictions for Y0
    test.df$red_pred <- predict(mod.red,newdata=test.df) # predictions for Y1
    test.df$pred = ifelse(test.df$blue_pred > test.df$red_pred,1,0) # classify
    err[fold] <- with(test.df,mean(blueWins != pred)) # error rate
  }
  mean(err) # return mean
}
```

```r
(err_hot = errCV_hot(data.hot)) # run one hot encoding
```

```
## [1] 0.2704098
```

```r
final_results[4,1]="One Hot Encoding"
final_results[4,2]=err_hot
final_results
```

```
##                         model error_rate
## 1                    Logistic  0.2691779
## 2                         KNN  0.2739004
## 3 Regular Linear Classifier  0.2708367
## 4          One Hot Encoding  0.2704098
```

One hot encoding does pretty good as well, similar to all the other ones. KNN does the worst.

Method 5 Linear Discriminant Analysis

Next, let's try some LDA.

```r
# A function that returns error rate for LDA with cross validation
errCV_lda <- function(data.df){
  numFolds=10 # number of folds
  dataSize <- nrow(data.df) # data size
  folds <- sample(1:numFolds,dataSize,rep=T) # folds
  err <- numeric(numFolds) # empty array
  for(fold in 1:numFolds){
    train.df <- data.df[folds != fold,] # train
    test.df <- data.df[folds == fold,] # test
    mod.lda <- lda(blueWins ~ .,data=train.df) # model
    pred.lda <- predict(mod.lda,newdata=test.df) # predictions
    test.df$pred = pred.lda$class # add classification to test
    err[fold] <- with(test.df,mean(blueWins != pred)) # error rate
  }
```

```
  mean(err) # return mean
}
```

```
(err_lda = errCV_lda(blue.df)) # run LDA
```

```
## [1] 0.2692143
```

```
final_results[5,1]="Linear Discriminant Analysis"
final_results[5,2]=err_lda
final_results
```

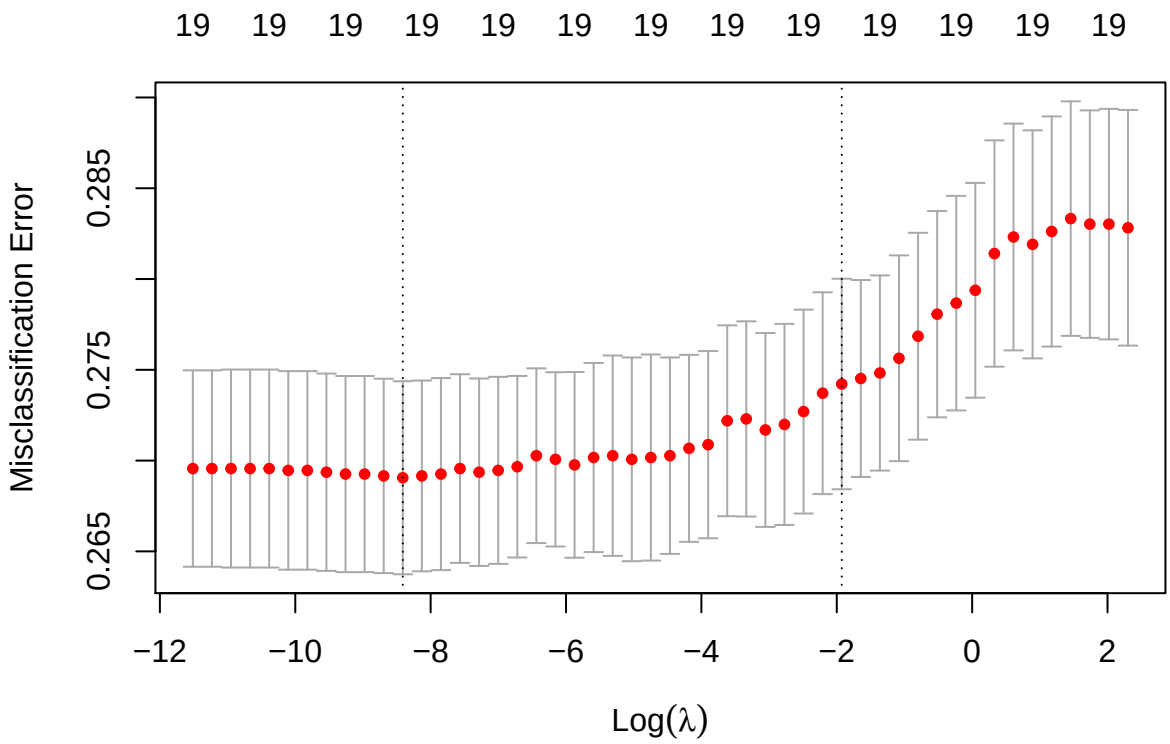
```
##                          model error_rate
## 1                     Logistic  0.2691779
## 2                          KNN  0.2739004
## 3    Regular Linear Classifier  0.2708367
## 4             One Hot Encoding  0.2704098
## 5 Linear Discriminant Analysis  0.2692143
```

Ok, the error rate for LDA is similar to all the others. Seems like the error rate is stuck around 0.27.

Method 6 Penalized Regression

Now let's try ridge and lasso and see how they do

```
# We tested out some lambda grid and this one seems pretty good
# This lambda grid places lambda 1se and min in the center of the plot
lambda.grid <- 10^seq(-5,1,length=50) # set up lambda grid
ridge.cv <- cv.glmnet(blue.x,blue.y, # we already have x and y setup
                      lambda=lambda.grid,
                      family="binomial",
                      type.measure="class",
                      alpha=0) # ridge
plot(ridge.cv) # plot
```



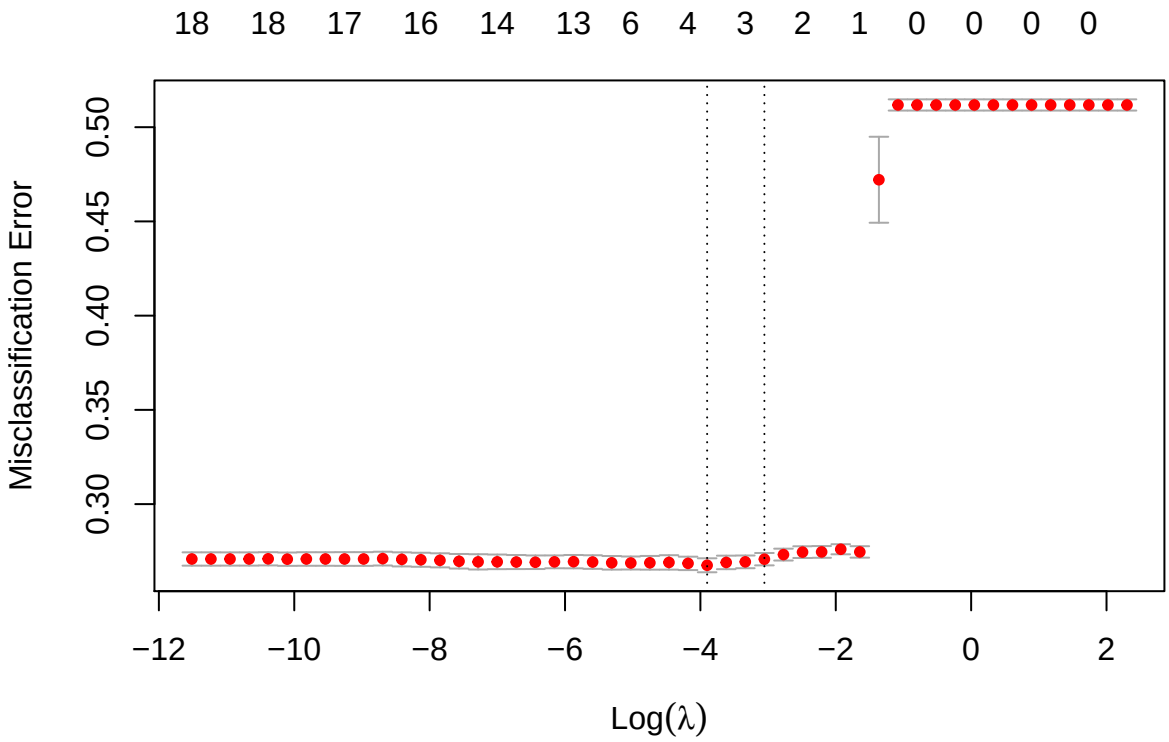

```
lambda.opt <- ridge.cv$lambda.1se # best lambda (1se)
id <- with(ridge.cv,which(lambda==lambda.opt)) # id at lambda 1se
(err_ridge <- with(ridge.cv,cvm[id])) # error rate
```

```
## [1] 0.274218
```

```
final_results[6,1]="Ridge"
final_results[6,2]=err_ridge
```

Now with Lasso

```
lasso.cv <- cv.glmnet(blue.x,blue.y,
                      lambda=lambda.grid,
                      family="binomial",
                      type.measure="class",
                      alpha=1) # lasso
plot(lasso.cv) # plot
```




```
lambda.opt <- lasso.cv$lambda.1se # best lambda (1se)
id <- with(lasso.cv,which(lambda==lambda.opt)) # id at lambda 1se
(err_lasso <- with(lasso.cv,cvm[id])) # error rate
```

```
## [1] 0.2707764
```

```
final_results[7,1]="Lasso"
final_results[7,2]=err_lasso
final_results
```

```
##                         model error_rate
## 1                    Logistic  0.2691779
## 2                         KNN  0.2739004
## 3    Regular Linear Classifier  0.2708367
## 4             One Hot Encoding  0.2704098
## 5 Linear Discriminant Analysis  0.2692143
## 6                       Ridge  0.2742180
## 7                       Lasso  0.2707764
```

The error rate is similar for all of them, lasso seems to be doing a bit better than ridge.

While we are here, see how many coefficients are nonzero in lasso 1se? There are 18 predictors in total

```
lasso.cv$nzero[id] # how many predictors are non-zero
```

```
## s19
##   3
```

```
coef(lasso.cv, s = "lambda.1se") # print coefficients
```

```
## 20 x 1 sparse Matrix of class "dgCMatrix"
##                                       1
## (Intercept)                -0.003680979
## blueWardsPlaced             .
```

```
## blueWardsDestroyed            .
## blueFirstBlood                .
## blueKills                     .
## blueDeaths                    .
## blueAssists                   .
## blueEliteMonsters             .
## blueDragons                   0.051760271
## blueHeralds                   .
## blueTowersDestroyed           .
## blueTotalGold                 .
## blueAvgLevel                  .
## blueTotalExperience           .
## blueTotalMinionsKilled        .
## blueTotalJungleMinionsKilled  .
## blueGoldDiff                  0.771653546
## blueExperienceDiff            0.279152228
## blueCSPerMin                  .
## blueGoldPerMin                .
```

Lasso zeroed out most predictors. Only 3 of them are nonzero: dragons, gold difference, and experience difference. These are probably significant predictors for whether a team wins the game. Indeed, killing minions, taking objectives and beating enemy champions contribute most to gold and experience lead which will offer advantages for winning the game.

```
N <- nrow(blue.df) # number of observations
train <- sample(1:N,N/2,rep=T) # train id
train.df <- blue.df[train,] # train
test.df <- blue.df[-train,] # test
```

Method 7 Bagging

```
# A function that returns error rate for bagging with cross validation
errCV_bag <- function(data.df){
  dataSize = nrow(data.df)
  numFolds=10 # number of folds
  numPreds = ncol(data.df) - 1 # number of predictors
  folds <- sample(1:numFolds,dataSize,rep=T) # folds
  err <- numeric(numFolds) # empty array
  for(fold in 1:numFolds){
    train.df <- data.df[folds != fold,] # train
    test.df <- data.df[folds == fold,] # test
    mod.bag <- randomForest(factor(blueWins) ~ .,
                            data=train.df,
                            mtry=numPreds,
                            ntree=1000,
                            importance=TRUE) # model
    pred.bag <- predict(mod.bag,newdata=test.df) # predictions
    err[fold] <- with(test.df,mean(blueWins != pred.bag)) # error rate
  }
  mean(err) # return mean
}
```

```
(err_bag <- errCV_bag(blue.df)) # bagging
```

```
## [1] 0.282143
```

```
final_results[8,1]="Bagging"
final_results[8,2]=err_bag
final_results
```

```
##                         model error_rate
## 1                    Logistic  0.2691779
## 2                         KNN  0.2739004
## 3    Regular Linear Classifier  0.2708367
## 4             One Hot Encoding  0.2704098
## 5 Linear Discriminant Analysis  0.2692143
## 6                       Ridge  0.2742180
## 7                       Lasso  0.2707764
## 8                     Bagging  0.2821430
```

Ok, seems like bagging actually does a bit worse than all of the previous ones. Interesting.

Method 8 Random Forest

```
# A function that returns error rate for random forest with cross validation
errCV_rf <- function(data.df){
  dataSize = nrow(data.df)
  numFolds=10 # number of folds
  numPreds = ncol(data.df) - 1 # number of predictors
  folds <- sample(1:numFolds,dataSize,rep=T) # folds
  err <- numeric(numFolds) # empty array
  for(fold in 1:numFolds){
    train.df <- data.df[folds != fold,] # train
    test.df <- data.df[folds == fold,] # test
    mod.rf <- randomForest(factor(blueWins) ~ .,
                           data=train.df,
                           mtry=numPreds/3,
                           ntree=500,
                           importance=TRUE) # model
    pred.rf <- predict(mod.rf,newdata=test.df) # predictions
    err[fold] <- with(test.df,mean(blueWins != pred.rf)) # error rate
  }
  mean(err) # return mean
}
```

```
(err_rf <- errCV_rf(blue.df)) # random forest
```

```
## [1] 0.2814832
```

```
final_results[9,1]="Random Forest"
final_results[9,2]=err_rf
final_results
```

```
##                         model error_rate
## 1                    Logistic  0.2691779
## 2                         KNN  0.2739004
## 3    Regular Linear Classifier  0.2708367
## 4             One Hot Encoding  0.2704098
## 5 Linear Discriminant Analysis  0.2692143
## 6                       Ridge  0.2742180
```

```
## 7                        Lasso  0.2707764
## 8                      Bagging  0.2821430
## 9                Random Forest  0.2814832
```

Random forest seems to do a bit better than bagging, but still not as good as some previous ones, like logistic and one hot encoding.

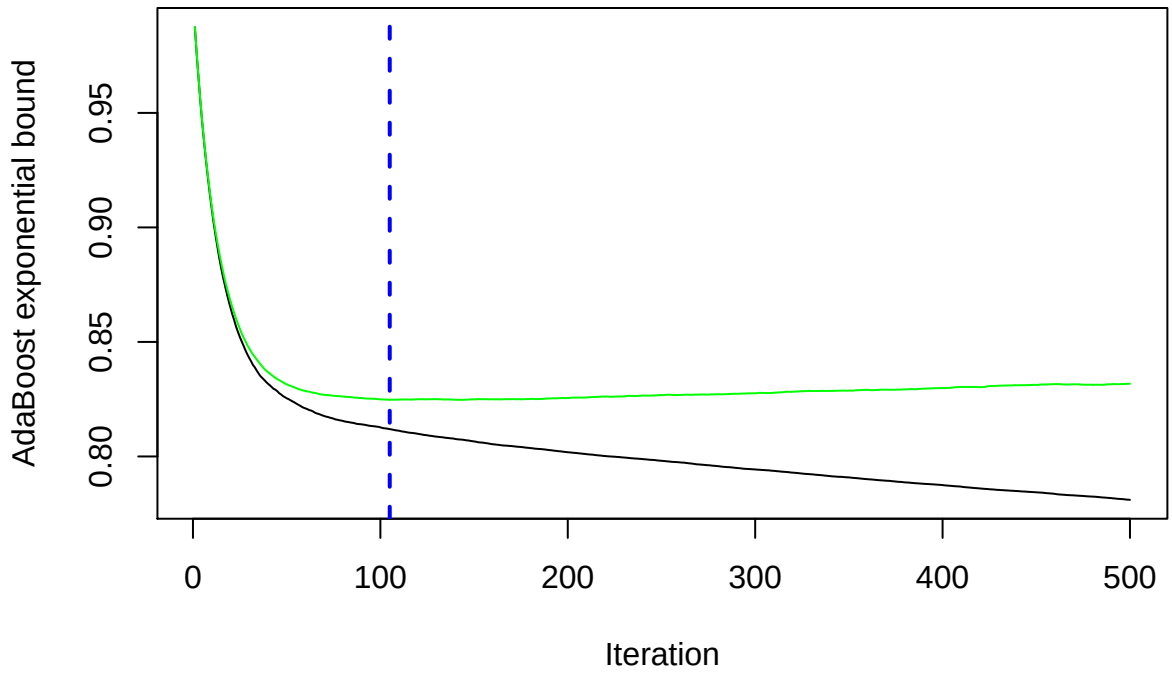
Method 9 Ada Boosting

Now we can try out some boosting. We had to explore the these parameters to find a good starting set. Start with depth = 3 and shrinkage = 0.05.

```
theDepth <- 3 # depth of tree
theShrinkage <- 0.05 # shrinkage
numTrees <- 500 # number of trees
```

Run cross-validated gbm

```
mod.gbm.cv <- gbm(blueWins ~ .,
                  data=blue.df,
                  distribution="adaboost",
                  interaction.depth = theDepth,
                  shrinkage=theShrinkage,
                  cv.folds = 10,
                  n.trees=numTrees) # ada boost
(numTreesOpt <- gbm.perf(mod.gbm.cv,method="cv")) # plot and best number of trees
```




```
## [1] 105
```

Having 105 trees seems pretty good

```
mod.gbm <- gbm(blueWins ~ .,
               data=blue.df,
               distribution="adaboost",
               shrinkage=theShrinkage,
```

```
               n.trees=numTreesOpt,
               interaction.depth = theDepth) # run with 108 trees
prob.gbm <- predict(mod.gbm,
                    newdata=test.df,
                    n.trees=numTreesOpt,
                    type="response") # predict
pred.gbm <- ifelse(prob.gbm > 0.5,1,0) # classify
with(test.df,table(blueWins, pred.gbm))
```

```
##         pred.gbm
## blueWins    0    1
##        0 2218  787
##        1  828 2143
```

```
(err.gbm <- with(test.df,mean(blueWins != pred.gbm))) # prelimenary error rate
```

```
## [1] 0.2702477
```

Oh error rate is still around 26 to 28%, seems to be similar to previous methods.

Now let's cross-validate on shrinkage and depth.

```
# Function to find best shrinkage, depth, and number of trees
N <- nrow(blue.df)
cvGBM <- function(theShrinkage,theDepth,numTrees,numFolds=10){
  # print the progress
  print(sprintf("Shrinkage: %s   Depth: %s: numTrees: %s",theShrinkage,theDepth,numTrees))
  folds <-  sample(1:numFolds,N,rep=T)
  errs <- numeric(numFolds)
  fold <- 1
  for(fold in 1:numFolds){
    train.df <- blue.df[folds != fold,]
    test.df <- blue.df[folds == fold,]
    mod.gbm <- gbm(blueWins ~ .,
                   data=train.df,
                   distribution="adaboost",
                   shrinkage=theShrinkage,
                   n.trees=numTreesOpt,
                   interaction.depth = theDepth) # adaboost
    prob.gbm <- predict(mod.gbm,
                        newdata=test.df,
                        n.trees=numTreesOpt,
                        type="response") # predict
    pred.gbm <- ifelse(prob.gbm > 0.5,1,0) # classify
    errs[fold] <- with(test.df,mean(blueWins != pred.gbm)) # get error
  }
  mean(errs) # return error
}
#testing..
cvGBM(theShrinkage,theDepth,numTreesOpt)
```

```
## [1] "Shrinkage: 0.05   Depth: 3: numTrees: 105"
```

```
## [1] 0.2705698
```

Create a grid of values of shrinkage and depths... don't be too precise here!

```
shrinks <- c(1,.1,.05,.01) # some values for shrinkage
depths <- c(1,2,3,4) # some values for depth
cv.vals <- expand.grid(shrinks,depths) # create a grid
cv.vals
```

```
##    Var1 Var2
## 1  1.00    1
## 2  0.10    1
## 3  0.05    1
## 4  0.01    1
## 5  1.00    2
## 6  0.10    2
## 7  0.05    2
## 8  0.01    2
## 9  1.00    3
## 10 0.10    3
## 11 0.05    3
## 12 0.01    3
## 13 1.00    4
## 14 0.10    4
## 15 0.05    4
## 16 0.01    4
```

Run the cvGBM against all of these. There are 16 total so it will take quite some time.

```
err.vals <- apply(cv.vals,1,function(row) cvGBM(row[1],row[2],numTreesOpt))
```

```
## [1] "Shrinkage: 1   Depth: 1: numTrees: 105"
## [1] "Shrinkage: 0.1   Depth: 1: numTrees: 105"
## [1] "Shrinkage: 0.05   Depth: 1: numTrees: 105"
## [1] "Shrinkage: 0.01   Depth: 1: numTrees: 105"
## [1] "Shrinkage: 1   Depth: 2: numTrees: 105"
## [1] "Shrinkage: 0.1   Depth: 2: numTrees: 105"
## [1] "Shrinkage: 0.05   Depth: 2: numTrees: 105"
## [1] "Shrinkage: 0.01   Depth: 2: numTrees: 105"
## [1] "Shrinkage: 1   Depth: 3: numTrees: 105"
## [1] "Shrinkage: 0.1   Depth: 3: numTrees: 105"
## [1] "Shrinkage: 0.05   Depth: 3: numTrees: 105"
## [1] "Shrinkage: 0.01   Depth: 3: numTrees: 105"
## [1] "Shrinkage: 1   Depth: 4: numTrees: 105"
## [1] "Shrinkage: 0.1   Depth: 4: numTrees: 105"
## [1] "Shrinkage: 0.05   Depth: 4: numTrees: 105"
## [1] "Shrinkage: 0.01   Depth: 4: numTrees: 105"
```

```
id <- which.min(err.vals)
(best.params <- cv.vals[id,]) # best parameter
```

```
##   Var1 Var2
## 3 0.05    1
```

```
(shrinkOpt <- best.params[1]) # best shrinkage value
```

```
##   Var1
## 3 0.05
```

```
(depthOpt <- best.params[2]) # best depth
```

```
##   Var2
## 3    1
```

```
(err_boost <- err.vals[id]) # err rate
```

```
## [1] 0.2691384
```

```
final_results[10,1]="Adaboost"
final_results[10,2]=err_boost
final_results
```

```
##                          model error_rate
## 1                     Logistic  0.2691779
## 2                          KNN  0.2739004
## 3     Regular Linear Classifier  0.2708367
## 4              One Hot Encoding  0.2704098
## 5  Linear Discriminant Analysis  0.2692143
## 6                        Ridge  0.2742180
## 7                        Lasso  0.2707764
## 8                      Bagging  0.2821430
## 9                Random Forest  0.2814832
## 10                    Adaboost  0.2691384
```

Adaboost does alright, but it seems like we stucked with around 27% error rate :(All of the error rates are quite similar. Let's rank the error rates.

```
final_results %>% arrange(error_rate)
```

```
##                          model error_rate
## 1                     Adaboost  0.2691384
## 2                     Logistic  0.2691779
## 3  Linear Discriminant Analysis  0.2692143
## 4              One Hot Encoding  0.2704098
## 5                        Lasso  0.2707764
## 6     Regular Linear Classifier  0.2708367
## 7                          KNN  0.2739004
## 8                        Ridge  0.2742180
## 9                Random Forest  0.2814832
## 10                     Bagging  0.2821430
```

Method 10 Support Vector Machine

We don't know much about SVM, but we tried to use it to classify our data. A linear kernel didn't work out so well, we tried polynomial. We ran the tuning part for 30 minutes, and it still didn't finish. So we just decided to pick the parameters manually

```
costVals <- 10^seq(2,6,by=1) # values for cost
degreeVals <- 1:4 # values for degree of polynomial kernal
numFolds = 5
svm.tune <- tune(svm,factor(blueWins)~ .,
     data=blue.df,
     kernel="poly",
     type = 'C-classification',
     ranges=list(cost=costVals,
                 degree=degreeVals),
     tunecontrol=tune.control(cross=numFolds) # tune the parameters
     )
```

```
summary(svm.tune) # summary
(cost.best <- svm.tune$best.parameters$cost) # best cost
(degree.best <- svm.tune$best.parameters$degree) # best degree
```

So we set degree = 2 and cost = 10

```
degree.best=2
cost.best=10

svm.cv = function(data.df){
  numFolds = 10 # number of folds
  dataSize = nrow(data.df) # rows
  folds <- sample(1:numFolds,dataSize,rep=T) # folds
  err <- numeric(numFolds) # empty array
  for(fold in 1:numFolds){
    train.df <- data.df[folds != fold,] # train
    test.df <- data.df[folds == fold,] # test
    mod.svm <- svm(factor(blueWins)~ .,
                   data=train.df,
                   kernel="poly",
                   degree=degree.best,
                   cost=cost.best,
                   scale=F)
    pred.svm <- predict(mod.svm,newdata=test.df)
    err[fold] = with(test.df,mean(blueWins != pred.svm))
  }
  mean(err)
}
```

```
(err_svm = svm.cv(blue.df))
```

```
## [1] 0.281404
```

```
final_results[11,1]="Support Vector Machine"
final_results[11,2]=err_svm
```

Ok. The error rate is a bit higher than what we have before. However, there might be better parameters for SVM that will produce a lower error rate.

Method 11 Neural Network

```
names(blue.df) # column names
```

```
##  [1] "blueWins"                    "blueWardsPlaced"
##  [3] "blueWardsDestroyed"          "blueFirstBlood"
##  [5] "blueKills"                   "blueDeaths"
##  [7] "blueAssists"                 "blueEliteMonsters"
##  [9] "blueDragons"                 "blueHeralds"
## [11] "blueTowersDestroyed"         "blueTotalGold"
## [13] "blueAvgLevel"                "blueTotalExperience"
## [15] "blueTotalMinionsKilled"      "blueTotalJungleMinionsKilled"
## [17] "blueGoldDiff"                "blueExperienceDiff"
## [19] "blueCSPerMin"                "blueGoldPerMin"
```

```
(fmla <- as.formula(paste("blueWins ~ ", paste(names(blue.df)[-1], collapse= "+")))) # formula
```

```
## blueWins ~ blueWardsPlaced + blueWardsDestroyed + blueFirstBlood +
##     blueKills + blueDeaths + blueAssists + blueEliteMonsters +
##     blueDragons + blueHeralds + blueTowersDestroyed + blueTotalGold +
##     blueAvgLevel + blueTotalExperience + blueTotalMinionsKilled +
##     blueTotalJungleMinionsKilled + blueGoldDiff + blueExperienceDiff +
##     blueCSPerMin + blueGoldPerMin
```

```
blue.scaled <- as.data.frame(scale(blue.df)) # scale
min.win <- min(blue.df$blueWins)
max.win <- max(blue.df$blueWins)
# response var must be scaled to [0 < resp < 1]
blue.scaled$blueWins <- scale(blue.df$blueWins,
                    center = min.win,
                    scale = max.win - min.win)
```

We looked at all the parameters for neural network, including stepMax, rep, learning rate, err.fct, and act.fct. It took a realy long time to run a neural net with just 1 layer of 10 perceptrons. So we will not worry about all the parameters. We use logisitc as activation function because it works well with classification. We will just run a cv on this simple neural net, calculate an error rate, and call it a day.

```
# A function that returns error rate for neural net with cross validation
errCV_nn <- function(data.df){
  numFolds = 5 # number of folds
  dataSize = nrow(data.df) # rows
  folds <- sample(1:numFolds,dataSize,rep=T) # folds
  err <- numeric(numFolds) # empty array
  for(fold in 1:numFolds){
    train.df <- as.matrix(data.df[folds != fold,]) # train
    test.df <- data.df[folds == fold,] # test
    blue.nn <- neuralnet(fmla,
                data=train.df,
                stepmax = 1e+05,
                rep = 1,
                hidden=10,
                act.fct="logistic",
                linear.output=FALSE) # model
    prob = predict(blue.nn,newdata = test.df) # predict
    pred = ifelse(prob > 0.5, 1, 0) # predict
    err[fold] <- with(test.df,mean(blueWins != pred)) # error rate
  }
  mean(err) # return mean
}
```

```
(err_nn = errCV_nn(blue.scaled))
```

```
## [1] 0.2883085
```

```
final_results[12,1]="Neural Net"
final_results[12,2]=err_nn
final_results
```

```
##                          model error_rate
## 1                     Logistic  0.2691779
## 2                          KNN  0.2739004
```

```
## 3     Regular Linear Classifier  0.2708367
## 4                 One Hot Encoding  0.2704098
## 5  Linear Discriminant Analysis  0.2692143
## 6                           Ridge  0.2742180
## 7                           Lasso  0.2707764
## 8                         Bagging  0.2821430
## 9                   Random Forest  0.2814832
## 10                      Adaboost  0.2691384
## 11       Support Vector Machine  0.2814040
## 12                    Neural Net  0.2883085
```

The CV for SVM and Neural Net took like forever to run. Both error rates came out to around 0.27 to 0.28. Again, seems like even SVM and NN could not lower our error rate significantly, although we did not actually test out all the parameters for SVM and NN. I'm pretty sure there would be way better neural network set ups than the ones we have here. Just from general knowledge, we know that SVM and NN are really complicated methods with fairly good error rates for some datasets. But they also come at a price of being really time consuming. If we have more time and computational power, we would test out some parameters grids for SVM and various neural network set ups for NN.

Conclusion

```
final_results %>% arrange(error_rate)
```

```
##                           model error_rate
## 1                       Adaboost  0.2691384
## 2                       Logistic  0.2691779
## 3  Linear Discriminant Analysis  0.2692143
## 4                 One Hot Encoding  0.2704098
## 5                           Lasso  0.2707764
## 6     Regular Linear Classifier  0.2708367
## 7                             KNN  0.2739004
## 8                           Ridge  0.2742180
## 9         Support Vector Machine  0.2814040
## 10                  Random Forest  0.2814832
## 11                        Bagging  0.2821430
## 12                    Neural Net  0.2883085
```

As we see from our results, all the method holds quite similar error rate (around 0.26 to 0.28) and thus accuracy rate, indicating that the first ten minutes of the game development are crucial but not definite. Some models do pretty good, like adaboost, logistic, one hot encoding, etc. Some complicated mpdels such as Bagging and Random Forest does a bit worse than we expected. SVM and NN does not perform well because we probably don't have the best parameters.

As League of Legends is a game with high variance, there are a lot more indicators that can impact the game outcome. For instance, the choice of champions impacts the game a lot. Different champions require different playing styles. Some late game champions need to get three items to make damages, which usually happens at about 25-30 minutes. There are also factors such as whether the team collaborates well in a fight. Some team fights late in game can turn the whole situation around. However, as our PCA and LASSO's result suggested, getting early gold advantages, kills lead and experience lead as well as securing objects like dragons and towers are essential factors for winning the game. This also coincides with the playing styles we see in professional games.