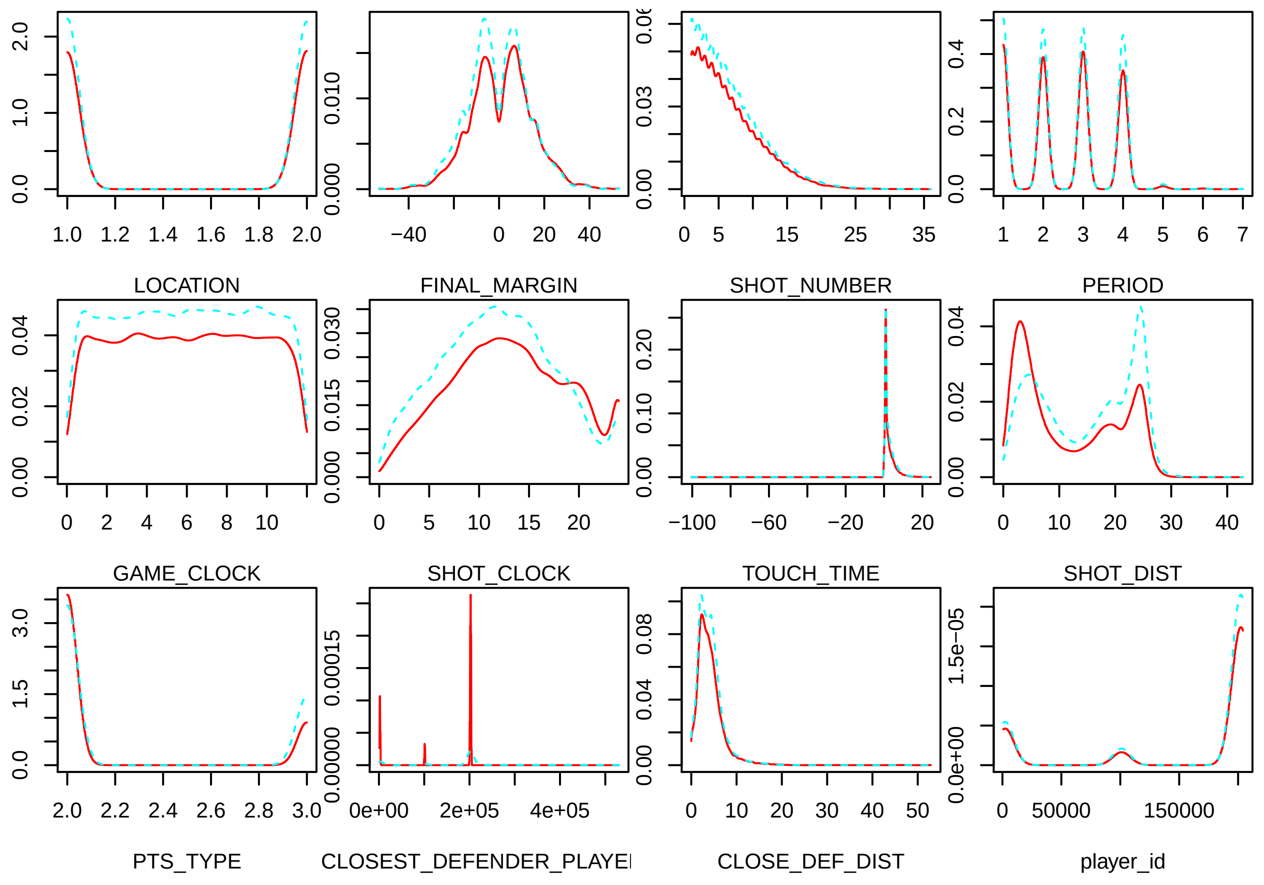
NBA Shot Logs: Makes and Misses

Vaakesan Sundrelingam

1. Introduction

The 'nba-shot-logs' dataset was scraped using the NBA's REST API by kaggle user 'dansbecker'. There are $n = 122,502$ observations of $p = 17$ variables taken over the 2014-2015 NBA season. This is a large dataset with $n >> p$. There wont be issues with high dimensionality, but there will be substantial issues related to computation.

The covariate 'SHOT_RESULT' indicates whether the shot was 'made' or 'missed'. In order to get a better idea of the distribution of the variables involved, we can look at their kernel density estimates conditional on the response 'SHOT_RESULT'. Some of the covariates are categorical (for which the density estimate is less useful) and many are continuous:



Note that as part of the data exploration, it was determined that there was a strong correlation (>0.80) between some of these covariates. In order to eliminate issues related to multicollinearity, these variables were not used in the analysis. These included the variables 'FINAL_MARGIN' and 'W', as well as 'TOUCH_TIME' and 'DRIBBLES'. Intuitively, using some domain knowledge in the context of the problem, it makes sense that these variables would be correlated. It was decided that 'FINAL_MARGIN' and 'TOUCH_TIME' would be kept, because they contain more information.

The objective of the analysis to follow is to accurately predict whether a shot will be ‘made’ or ‘missed’, and which variables are most important in predicting this result. The analysis will employ multiple classification techniques, including parametric, non-parametric, and ensemble approaches. The most appropriate model will be selected based on its performance relative to the other models tested. The pros and cons of each model will also be considered in the discussion.

2. Introduction to Model Selection

In order to perform model selection, we will have to estimate the prediction error on an independent validation set. This will be accomplished using the ‘bootstrap’ method. B samples of size $n(1-e^{-1}) \approx \frac{2}{3}n$ will be taken with replacement from the full training set. The model will be trained on this subset and the prediction error is estimated on the ‘out-of-bag’ subset of size $\approx \frac{1}{3}n$. The bootstrap error is given by:

$$\widehat{Err^{(1)}} = \frac{1}{n}\sum_{i=1}^{n}\frac{1}{|C^{-i}|}\sum_{b\in C^{-i}} L(y_i, \hat{f}^{*b}(x_i))$$

Where $\hat{f}^{*b}$ is the model trained on the bootstrap subset, (x_i, y_i) is the ith observation, C^{-i} is the set of bootstrap samples that don’t contain (x_i, y_i), and L is any loss function (generally $0-1$) in the majority of the models tested below. The performance between classifiers is to be compared using their bootstrap prediction error estimates. Once the best performing model is selected, its predicition error rate will be honestly assessed using an independent ‘test’ set.

3. Classifiers

The discussion to follow involves the underlying models, assumptions, and motivations of each classifier used in this analysis. The summary is a general overview, followed by an analysis specific to the context of the problem:

3.1. Discriminant Analysis

Linear discriminant analysis is a generative classifier that seeks to model $p_k(\mathbf{x}) = P(Y = k|\mathbf{X} = \mathbf{x})$ using bayes rule:

$$P(Y = k|\mathbf{X} = \mathbf{x}) = \frac{f_k(\mathbf{x})\pi_k}{\sum_{l=1}^{K} f_l(\mathbf{x})\pi_l}$$

The prior class probability π_k is either determined using knowledge about the population, or estimated using the proportion of class k in the training sample. There are two assumptions in Linear Discriminant Analysis.
1. $f_k(\mathbf{x})$ is given by a multivariate Gaussian distribution:

$$f_k(\mathbf{x}) = \frac{1}{(2\pi)^{\frac{p}{2}}|\Sigma_k|^{\frac{1}{2}}}e^{\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu_k})^T\Sigma_k^{-1}(\mathbf{x}-\boldsymbol{\mu_k})}$$

2. The covariance matrix is equal for all classes i.e. $\Sigma_k = \Sigma_l\ \forall k, l$, where the covariance matrix is given by:

$$\hat{\Sigma} = \sum_{k=1}^{K}\Sigma_{g_i=k}(\mathbf{x}_i - \hat{\boldsymbol{\mu_k}})(\mathbf{x}_i - \hat{\boldsymbol{\mu_k}})^T/(N-K)$$

With class mean $\hat{\boldsymbol{\mu_k}}$.

Quadratic Discriminant analysis follows the exact same procedure but relaxes the assumptions on the variance-covariance matrix to allow $\Sigma_1 \neq \Sigma_2 \neq \cdots \neq \Sigma_K$.

3.2. Logistic Regression

Logistic regression is a discriminative classifier that assumes Y follows a Multinomial distribution, and attempts to model the logit of any two class probabilities using a linear combination of the predictor variables:

$$log\left(\frac{p_k(\mathbf{x})}{p_K(\mathbf{x})}\right) = \beta_{k,0} + \boldsymbol{\beta}'_k\mathbf{x}$$

With class probabilities given by:

$$p(Y = k|\mathbf{X} = \mathbf{x}) = \frac{e^{\beta_{k,0}+\boldsymbol{\beta}'_k\mathbf{x}}}{1 + \sum_{l=1}^{K-1} e^{\beta_{l,0}+\boldsymbol{\beta}'_l\mathbf{x}}}$$

Generally a base class K is chosen to determine the logits, and the parameters β are estimated by maximum likelihood (which allows for inference due to asymptotic normality).

Regularized logistic regression (LASSO) was also tried, which penalizes the L_1 norm of the fitted coefficients. The objective function changes to the following:

$$\sum_{i=1}^{n}(y_i - \beta_0 - \sum_{j=1}^{p} x_{i,j}\beta_j)^2 + \lambda\sum_{j=1}^{p}|\beta_j|$$

These penalized parameters are not estimated by maximum likelihood, but instead through numerical techniques such as Newton-Raphson. For this reason, statistical inference cannot be used on the fitted coefficients.

3.3. K-Nearest Neighbours

K-Nearest Neighbours was the first non parametric technique used to predict the SHOT_RESULT. The classifier is fit by determining for any given point x_0, the k observations that are closest to x_0. The class l for which the proportion of these k observations that belong to class l is the largest among all classes is the predicted class label.

3.4. Generalized Additive Models

The generalized additive model is a generalization of the logistic regression model such that the logit of class probabilities is modelled by a linear combination of smooth functions of the predictors:

$$log\left(\frac{p_k(\mathbf{x})}{p_K(\mathbf{x})}\right) = \alpha_0 + f_1(X_1) + \cdots + f_p(X_p)$$

This model is a compromise between a linear combination of the predictors and a fully nonparametric model:

$$log\left(\frac{p_k(\mathbf{x})}{p_K(\mathbf{x})}\right) = f(\mathbf{x})$$

Note that the imposed additive structure does not allow for variable interactions. The smooth functions are fit using either:

1. By fitting a least squares regression model to an expanded basis (also known as piecewise polynomial regression), using cubic splines at K knots to guarantee continuity and differentiability:

$$E[Y|\mathbf{X} = \mathbf{x}] = \beta_0 + \beta_1X + \beta_2X^2 + \beta_3X^3 + \beta_4(X - \xi_1)^3_+ + \beta_5(X - \xi_2)^3_+ + \cdots + \beta_{K+3}(X - \xi_K)^3_+$$

2. Smoothing splines (using an unlimited basis of smooth functions) with a penalty on the complexity:

$$RSS(f, \lambda) = \sum_{i=1}^{n}(y_i - f(x_i))^2 + \lambda\int_{X_{min}}^{X_{max}}[f''(t)]^2dt$$

3. Kernel based local smoothers (e.g. weighted least squares), using a Gaussian kernel for example:

$$RSS(x_0; f, \lambda) = \sum_{i=1}^{n} K_\lambda(x_0; x_i)[y_i - f(x_i)]^2$$

3.5. Naive Bayes

Naive Bayes is similar to Discriminant analysis in that it seeks to model $p_k(\mathbf{x}) = P(Y = k|\mathbf{X} = \mathbf{x})$ using Bayes Rule. However, instead of assuming a multivariate Gaussian for $f_k(\mathbf{x})$, it is assumed that all p predictors are independent:

$$f_k(\mathbf{x}) = \prod_{j=1}^{p} f_{k,j}(x_j)$$

Where $f_{k,j}$ can take one of two forms:

1. A parametric form where it is assumed that each $f_{j,k}$ is given by a univariate Gaussian distribution. The difference then, from Discriminant Analysis is that in Discriminant Analysis it is not assumed that the x_j are independent from each other.
2. A nonparametric form where the density $f_{k,j}$ is estimated by Kernel Density Estimation as discussed above. The kernel function has the following form and is a function of the distance between all observations x_i and the point at which the density is being estimated:

$$\hat{f}(\mathbf{x}_0) = \frac{1}{n\lambda} \sum_{i=1}^{n} K\left(\frac{x_0 - x_i}{\lambda}\right)$$

Note that Principal Component Analysis could be used to decorrelate the predictors x_j and potentially improve the assumption of independence. However with the existence of categorical variables such as in our case, PCA cannot meaningfully decorrelate the predictors.

3.6. Tree Ensemble Methods

A single tree based classifier T is a partitioning of the predictor space $\mathbf{X}$ into M contiguous regions with class estimate c_j for each region R_m:

$$T(x; \{c_j, R_j\}_{j=1}^{J}) = \sum_{j=1}^{J} c_j I(x \in R_j)$$

The parameters $\Theta = \{c_j, R_j\}_{j=1}^{J}$ are chosen to minimize the loss L using a computationally feasible greedy strategy. However, a single tree based classifier is exceptionally variable, and so two ensemble approaches are used to improve the variance of the classifier:

1. Bagging grows B trees on B bootstrap samples from the original training data and uses the aggregate (majority vote) estimate for the predicted class. However, the B bootstrap samples have high overlap and so the B classifiers are highly correlated. A bagging variant known as Random Forests uses a subset of the predictors to grow each tree to further reduce correlation between trees and improve the aggregate estimate.
2. Boosting involves growing trees sequentially such that each successive tree is grown to best fit the residuals (or misclassified observations) of the previous tree. At each iteration m, the parameters are solved to minimize the following:

$$\hat{\Theta}_m = argmin_{\Theta_m} \sum_{i=1}^{n} L(y_i, f_{m-1}(\mathbf{x}) + T(\mathbf{x}_i; \Theta_m))$$

3.7. Support Vector Machines

Support Vectors Machines attempt to construct a linear boundary in a basis expanded feature space that achieves maximum separation between classes (equivalently "maximizing the margin"). This can be solved geometrically and it can be shown that the solution can be formulated as:

$$min_{\beta_0,\boldsymbol{\beta}} \frac{1}{2}||\boldsymbol{\beta}||^2 \; s.t. \; y_i(\beta_0 + \boldsymbol{\beta}'\mathbf{x}_i) \geq 1$$

And changes only slightly to accomodate for non separable classes. This formulation uses the original feature space, but holds equivalently for the expanded feature space given by $\mathbf{h}(\mathbf{x})$ by substituting the equation of the separating hyperplane $\beta_0 + \boldsymbol{\beta}'\mathbf{x}_i$ with $\beta_0 + \boldsymbol{\beta}'\mathbf{h}(\mathbf{x}_i)$, where $\mathbf{h}(\mathbf{x}_i)$ can be the Gaussian radial basis (the default basis used in our analysis): $e^{-\gamma||\mathbf{x}_0 - \mathbf{x}_i||^2}$

3.8. Neural Networks

A neural network is a multi layered model where each layer is a linear combination of the outputs from the previous layers (with some special considerations). There are inputs X, hidden nodes Z, an aggregation Y, and some final transformation T. The m hidden nodes are given by the following:

$$Z_m = \frac{1}{1 + e^{-(\alpha_{0,m} + \boldsymbol{\alpha}'_m \mathbf{X})}}$$

The hidden nodes are aggregated into k outputs $T_k = \beta_{k,0} + \boldsymbol{\beta}'\mathbf{Z}$ and then transformed into probabilities given by:

$$f_k(\mathbf{x}) = \frac{e^{T_k}}{\sum_{l=1}^{K} e^{T_l}}$$

And the predicted class is that for which $f_k(\mathbf{x})$ is maximum. The parameters $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ are fitted using "back propagation", which at every iteration, updates the parameters in the direction of the negative gradient of the loss function.

4. Hyperparameter Tuning

Many of the techniques that were used in the analysis of the 'nba-shot-logs' data allow for the tuning of various hyperparameters. Hyperparameters are paraemeters that are outside of the model, chosen by the user, and cannot be estimated from the data. Therefore, in order to determine the optimal hyperparameter, one could train the classifier on the data (having set a fixed value for the hyperparameter to be tuned), and evaluate the performance of the classifer on an independent test set or validation set. This can be done for many values of the hyperparameter and the optimal value would correspond to the hyperparameter which resulted in the smallest validation error.

There exist models for which there are many hyperparameters. This only marginally changes the procedure. Instead of training the model for many values of a single hyperparameter, the model is trained on every possible combination of any values of interest of the hyperparameters. This is called a 'grid-search'. Below are the results of the 'grid-search' used to tune the 'gradient boosted decision trees':

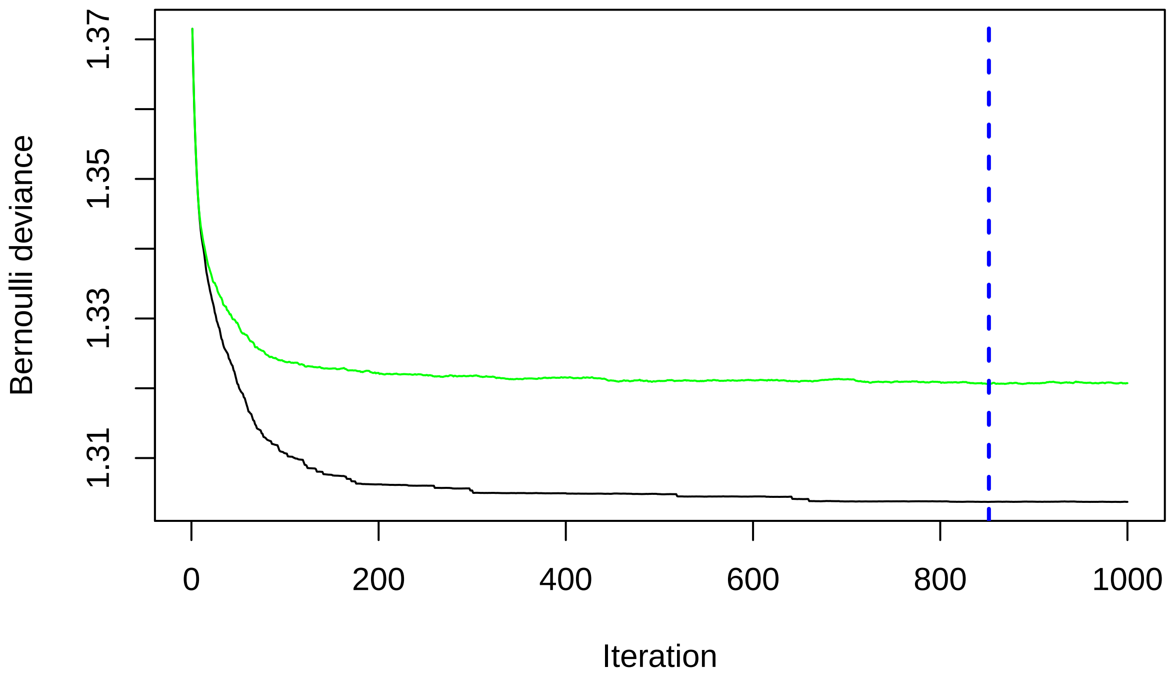
	shrinkage	interactionDepth	bag.fraction	n.trees	err
21	0.100	1	0.9	471	0.351
12	0.100	1	0.5	460	0.362
14	0.010	2	0.5	500	0.379
2	0.010	1	0.1	498	0.380
10	0.001	1	0.5	500	0.380

Although the performance of each classifier can be evaluated on an independent validation or test set in order to determine the optimal values for the hyperparameters, this requires an independent validation set (which can be difficult to obtain in small n datasets). This is not the case for the 'nba-shot-logs' data, and so generally an independent validation set was used. For example, this was the method used to tune the hyperparameters for 'k-nearest neighbours' and 'gradient boosted decision trees'.

However, some of the built in methods to functions in R are able to tune the hyperparameters via cross validation. Cross validation is used to estimate prediction error while using more of the training data to train the classifier. The training data is split into V "folds" (non overlapping partitions). The classifier is trained on $V-1$ folds and the prediction error is estimated on the remaining, hold-out fold. This is done for all V folds, and the prediction error is given by the following:

$$CV(\hat{f}) = \frac{1}{n}\sum_{v=1}^{V}\sum_{i=1}^{n_v} L(y_i, \hat{f}^{(-v)}(x_i))$$

Below are the results of using 5-fold cross validation to tune the number of trees in the 'gradient boosted decision trees' model:



```
## [1] 852
```

The results of the cross valiation show that the optimal number of trees is 852.

In the CV error formula, V is the number of folds, n is the number of observations, and n_v is the number of observations in fold V. (x_i, y_i) is observation i, L is any loss function (e.g. $0-1$ loss), and $\hat{f}^{(-v)}$ is the classifier trained on all $V-1$ folds (excluding all observations in the hold-out fold v). Note that the special case of CV is $LOOCV$ where $V = n$.

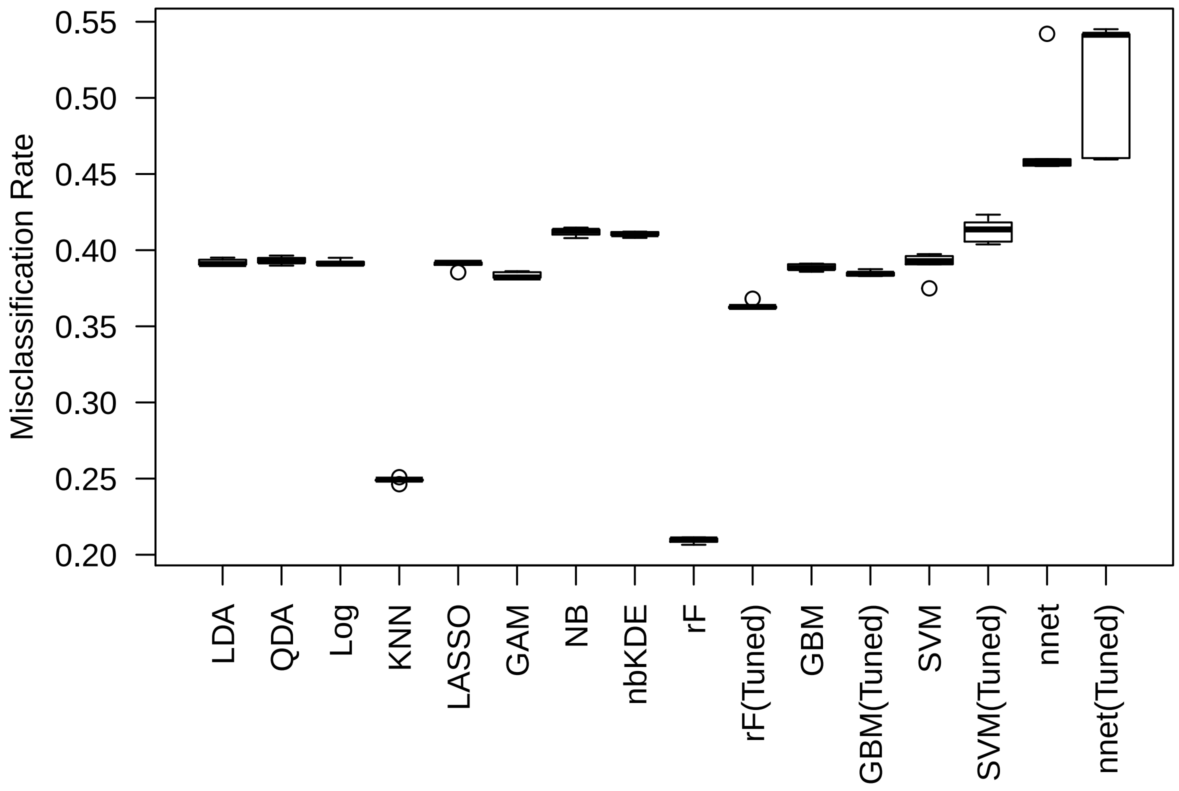
‘LASSO’, ‘gradient boosted decision trees’*, and ‘neural nets’ use cross validation to tune particular hyperparameters. ‘Gradient boosted decision trees’ use cross validation to determine the optimal number of trees. The cross validation uses the training set, but the grid search is performed on an independent test set. The number of folds is chosen by the user. It is generally considered that 5 and 10 are good choices for V. There are two considerations however, when choosing V.

The first is the bias-variance tradeoff. Larger V results in smaller bias, but higher variance. This is because the classifier is trained on more data. However, when training the classifier on each set of 9 folds, there is large overlap in the datasets. This creates correlation in the predictions, which increases the variance. A similar argument can be made for how the bias-variance tradeoff behaves for a smaller number of folds.

The second consideration is computational. The higher the number of folds, the more times the classifier will have to be trained on the data. For the ‘nba-shot-logs’ data, this was an important consideration, and was the main reason for choosing the number of folds for all hyperparameter tuning via cross validation to be 5.

5. Model Selection

Below is the out-of-bag error on 5 bootstrap samples for the 16 classifiers trained on the data:



We can see that the nonparametric classifiers such as ‘k-nearest neighbours’ and ‘random forest’ performed the best overall (by a substantial margin). Other nonparametric classifiers such as ‘naive bayes’ with kernel density estimation did not perform as well. We can assume that this is due to the assumption of independence of the covariates being violated. The ‘gradient boosted machines’ also performed poorly (performance in line with the parametric methods) even after tuning. Perhaps the use of a weighted, or penalized variant (such as ‘Adaboost’ or ‘XGBoost’) would have improved performance here.

The performance of the neural network was very poor, even though we would have expected it to perform well given the size of n in this dataset. This is likely because a seed was fixed before running the model, which fixed the initializations of the neural net. To improve the performance of the neural net, many different seeds/initializations should be tried, and the best initialization should be used to train and tune the net.

The performance of the linear and nonlinear parametric models were very similar. The models performed better than random guessing, but performed much worse than the nonparametric models. This is probably due to the violation of some of the assumptions in the parametric models.

Interestingly, the tuned methods did not work as well as the untuned methods. To save computational time, only a small subset of hyperparameters and hyperparameter combinations were compared. In order to improve the performance of the tuned models, more combinations should be tried in the future. Overall, the best performing classifier was the 'random forest'.

6. Model Assessment

As discussed in section 1.1, performing model selection in this way introduces selection bias, and so we cannot use the bootstrap prediction error to honestly assess the performance of the model. Instead we can assess the performance on an independent test set:

```r
mean(ifelse(predict(rForest, newdata = test, type = "response") ==
    test$SHOT_RESULT, yes = 0, no = 1))
```

```
## [1] 0.3929778
```

On the test set, the 'random forest' classifier performs about as well as any of the other classifiers (parametric or nonparametric), which suggests that the 'random forest' was tracing errors in the training set. However, the performance is better than random guessing, and given the difficulty of the problem, this is a good result.

A. Appendix

```r
# READ AND SEED
set.seed(20460842)
rawData <- read.csv("shot_logs.csv")
# head(nba); str(nba);

# removing NA cases
nba <- rawData[complete.cases(rawData), ]

# factor to variable (too many factor levels)
nba$GAME_CLOCK <- sapply(strsplit(as.character(nba$GAME_CLOCK),
    ":"), function(x) {
    x <- as.numeric(x)
    round(x[1] + x[2]/60, digits = 2)
})

# remove 1:=GAME_ID, 2:MATCHUP (meaningless variables) remove
# 18:= FGM, 19:=PTS (perfectly correlated w/response
# SHOT_RESULT)
nba <- nba[, -c(1, 2, 18, 19)]
```

```r
# NOTE: player information:= c(15,16,20,21) very few
# observations/factor level

# data split by bootstrap
bootstrap <- sample(x = nrow(nba), replace = TRUE)
train <- nba[bootstrap, ]
oob <- nba[-bootstrap, ]
bootstrap.v <- sample(x = nrow(oob), replace = TRUE)
validation <- oob[bootstrap.v, ]
test <- oob[-bootstrap.v, ]

# for LDA/QDA, require numeric variables
numericTrain <- train[, -c(13, 16)]
numericTrain$LOCATION <- as.numeric(numericTrain$LOCATION)
numericTrain$W <- as.numeric(numericTrain$W)

numericValidation <- validation[, -c(13, 16)]
numericValidation$LOCATION <- as.numeric(numericValidation$LOCATION)
numericValidation$W <- as.numeric(numericValidation$W)

numericTest <- test[, -c(13, 16)]
numericTest$LOCATION <- as.numeric(numericTest$LOCATION)
numericTest$W <- as.numeric(numericTest$W)

# study of correlations (requires numeric variables)
correlations <- cor(numericTrain[, -12])
for (i in 1:ncol(correlations)) {
    for (j in 1:i) {
        if (abs(correlations[j, i]) >= 0.8 & correlations[j,
            i] != 1) {
            print(c(colnames(correlations)[i], rownames(correlations)[j],
                correlations[j, i]))
        }
    }
}

# multicollinearity, choose FINAL_MARGIN over W & TOUCH_TIME
# over DRIBBLES
train <- train[, -c(2, 8)]
numericTrain <- numericTrain[, -c(2, 8)]
validation <- validation[, -c(2, 8)]
numericValidation <- numericValidation[, -c(2, 8)]
test <- test[, -c(2, 8)]
numericTest <- numericTest[, -c(2, 8)]

# kernel density estimates show the variable distributions
library(klaR)
densityEstimates <- NaiveBayes(x = numericTrain[, -10], grouping = numericTrain[,
    10], usekernel = TRUE)
par(mfrow = c(4, 4), mar = c(4, 2, 0.1, 0.1))
plot(densityEstimates, main = "", legend = FALSE)
```

```r
# LDA
library(MASS)

# fit the full model first to interpret parameters
ldaFit <- lda(x = numericTrain[, -10], grouping = numericTrain$SHOT_RESULT)
# plot(ldaFit) #1D plot doesn't work
ldaFit

# initialize vector of bootstrap validation errors
boot.LDA.err <- rep(0, times = 5)  #times=# bootstraps

# fit a model on 2/3 of the data n times
# (n=length(boot.LDA.err)) record error on 1/3
# holdout/validation set
for (i in 1:length(boot.LDA.err)) {
    boot <- sample(x = nrow(numericTrain), replace = TRUE)
    boot.Train <- numericTrain[boot, ]
    boot.Validation <- numericTrain[-boot, ]
    boot.LDA <- lda(x = boot.Train[, -10], grouping = boot.Train$SHOT_RESULT)
    boot.LDA.err[i] <- mean(ifelse(predict(boot.LDA, newdata = boot.Validation[,
        -10])$class == boot.Validation$SHOT_RESULT, yes = 0,
        no = 1))
}
```

```r
# QDA
qdaFit <- qda(x = numericTrain[, -10], grouping = numericTrain$SHOT_RESULT)
qdaFit

boot.QDA.err <- rep(0, times = 5)  #times=# bootstraps

for (i in 1:length(boot.QDA.err)) {
    boot <- sample(x = nrow(numericTrain), replace = TRUE)
    boot.Train <- numericTrain[boot, ]
    boot.Validation <- numericTrain[-boot, ]
    boot.QDA <- qda(x = boot.Train[, -10], grouping = boot.Train$SHOT_RESULT)
    boot.QDA.err[i] <- mean(ifelse(predict(boot.QDA, newdata = boot.Validation[,
        -10])$class == boot.Validation$SHOT_RESULT, yes = 0,
        no = 1))
}
```

```r
# Logistic regression
library(nnet)
# too slow w/non-numeric variables
logFit <- glm(data = numericTrain, formula = SHOT_RESULT ~ .,
    family = binomial(link = "logit"))

boot.Log.err <- rep(0, times = 5)  #times=# bootstraps

for (i in 1:length(boot.Log.err)) {
    boot <- sample(x = nrow(numericTrain), replace = TRUE)
    boot.Train <- numericTrain[boot, ]
    boot.Validation <- numericTrain[-boot, ]
    boot.Log <- glm(data = boot.Train, formula = SHOT_RESULT ~
        ., family = binomial(link = "logit"))
    boot.Log.err[i] <- mean(ifelse(ifelse(predict(boot.Log, newdata = boot.Validation[,
        -10], type = "response") >= 0.5, yes = "missed", no = "made") ==
        boot.Validation$SHOT_RESULT, yes = 0, no = 1))
}
```

```r
# KNN
library(FNN)
# using default k=1 (will tune later) must scale variables to
# make meaningful distance comparisons
KNNFit <- knn(train = scale(numericTrain[, -10]), test = scale(numericValidation[,
    -10]), cl = numericTrain[, 10])

boot.KNN.err <- rep(0, times = 5)  #times=# bootstraps

for (i in 1:length(boot.KNN.err)) {
    boot <- sample(x = nrow(numericTrain), replace = TRUE)
    boot.Train <- numericTrain[boot, ]
    boot.Validation <- numericTrain[-boot, ]
    boot.KNN <- knn(train = scale(boot.Train[, -10]), test = scale(boot.Validation[,
        -10]), cl = boot.Train[, 10])
    boot.KNN.err[i] <- mean(ifelse(boot.KNN == boot.Validation$SHOT_RESULT,
        yes = 0, no = 1))
}

# tune k on independent validation set only need to run
# once*** ks <- c(1,3,5,10,50,100) tuningKNN.err =
# rep(0,times=6) for (i in 1:length(ks)){ tuningKNN <-
# knn(train=scale(numericTrain[,-10]),
# test=scale(numericValidation[,-10]), cl=boot.Train[,10],
# k=ks[i]) tuningKNN.err[i] <-
# mean(ifelse(tuningKNN==numericValidation$SHOT_RESULT,
# yes=0, no=1)) } #100 is the optimal k (lowest validation
# error)

# assess prediction error on tuned k
boot.KNNtuned.err <- rep(0, times = 5)  #times=# bootstraps

for (i in 1:length(boot.KNNtuned.err)) {
    boot <- sample(x = nrow(numericTrain), replace = TRUE)
    boot.Train <- numericTrain[boot, ]
    boot.Validation <- numericTrain[-boot, ]
    boot.KNNtuned <- knn(train = scale(boot.Train[, -10]), test = scale(boot.Validation[,
        -10]), cl = boot.Train[, 10], k = 100)
    boot.KNNtuned.err[i] <- mean(ifelse(boot.KNNtuned == boot.Validation$SHOT_RESULT,
        yes = 0, no = 1))
}
```

```r
# LASSO rescale to speed up convergence time
library(glmnet)
# 5-fold cross validation to tune L2-norm penalty lambda
lassoFit <- cv.glmnet(x = scale(numericTrain[, -10]), y = numericTrain[,
    10], family = "binomial", nfolds = 5)
```

```r
lassoFit$lambda.1se

boot.LASSO.err <- rep(0, times = 5)  #times=# bootstraps

for (i in 1:length(boot.LASSO.err)) {
    boot <- sample(x = nrow(numericTrain), replace = TRUE)
    boot.Train <- numericTrain[boot, ]
    boot.Validation <- numericTrain[-boot, ]
    boot.LASSO <- cv.glmnet(x = scale(boot.Train[, -10]), y = boot.Train[,
        10], family = "binomial", nfolds = 5)
    boot.LASSO.err[i] <- mean(ifelse(predict(boot.LASSO, newx = scale(boot.Validation[,
        -10]), s = boot.LASSO$lambda.1se, type = "class") ==
        boot.Validation$SHOT_RESULT, yes = 0, no = 1))
}
```

```r
# GAM the additive version of sm.binomial sm.binomial can
# only do 1 covariate at a time so GAM is the only option for
# relevant comparison to other models

# requires 0-1 response variable
gamTrain <- train
library(dplyr)
gamTrain$SHOT_RESULT <- recode(gamTrain$SHOT_RESULT, made = 1,
    missed = 0)


library(mgcv)
gamFit <- gam(data = gamTrain, SHOT_RESULT ~ s(FINAL_MARGIN) +
    s(SHOT_NUMBER) + s(GAME_CLOCK) + s(SHOT_CLOCK) + s(TOUCH_TIME) +
    s(SHOT_DIST) + s(CLOSEST_DEFENDER_PLAYER_ID) + s(CLOSE_DEF_DIST) +
    s(player_id), family = binomial(link = "logit"))
# cannot use factor variables; could code LOCATION as numeric
# can't use period, pts_type (a term has fewer unique
# covariate combinations than specified max degrees of
# freedom)
summary(gamFit)

par(mfrow = c(3, 3), mar = c(4, 4, 0.1, 0.1))
plot(gamFit, ylim = c(-1, 1))

mean(ifelse(as.numeric(predict(gamFit, newdata = gamTrain, type = "response") >=
    0.5) == gamTrain$SHOT_RESULT, yes = 0, no = 1))

boot.GAM.err <- rep(0, times = 5)  #times=# bootstraps

for (i in 1:length(boot.GAM.err)) {
    boot <- sample(x = nrow(gamTrain), replace = TRUE)
    boot.Train <- gamTrain[boot, ]
    boot.Validation <- gamTrain[-boot, ]
    boot.GAM <- gamFit <- gam(data = boot.Train, SHOT_RESULT ~
        s(FINAL_MARGIN) + s(SHOT_NUMBER) + s(GAME_CLOCK) + s(SHOT_CLOCK) +
            s(TOUCH_TIME) + s(SHOT_DIST) + s(CLOSEST_DEFENDER_PLAYER_ID) +
            s(CLOSE_DEF_DIST) + s(player_id), family = binomial(link = "logit"))
    boot.GAM.err[i] <- mean(ifelse(as.numeric(predict(boot.GAM,
        newdata = boot.Validation, type = "response") >= 0.5) ==
        boot.Validation$SHOT_RESULT, yes = 0, no = 1))
}
```

```
# naive bayes
library(e1071)
nbFit <- naiveBayes(x = train[, -10], y = train[, 10])  #Gaussian parametric density
nbKdeFit <- NaiveBayes(x = train[, -10], grouping = train[, 10],
    useKernel = TRUE)  #Gaussian-kernel nonparametric density

boot.NB.err <- rep(0, times = 5)  #times=# bootstraps

for (i in 1:length(boot.NB.err)) {
    boot <- sample(x = nrow(train), replace = TRUE)
    boot.Train <- train[boot, ]
    boot.Validation <- train[-boot, ]
    boot.NB <- naiveBayes(x = boot.Train[, -10], y = boot.Train[,
        10])
    boot.NB.err[i] <- suppressWarnings(mean(ifelse(predict(boot.NB,
        newdata = boot.Validation[, -10], type = "class") ==
        boot.Validation$SHOT_RESULT, yes = 0, no = 1)))
}

boot.nbKDE.err <- rep(0, times = 5)  #times=# bootstraps

for (i in 1:length(boot.nbKDE.err)) {
    boot <- sample(x = nrow(train), replace = TRUE)
    boot.Train <- train[boot, ]
    boot.Validation <- train[-boot, ]
    boot.nbKDE <- naiveBayes(x = boot.Train[, -10], y = boot.Train[,
        10])
    boot.nbKDE.err[i] <- suppressWarnings(mean(ifelse(predict(boot.nbKDE,
        newdata = boot.Validation[, -10], type = "class") ==
        boot.Validation$SHOT_RESULT, yes = 0, no = 1)))
}

# could try PCA rotated Naive Bayes- but many categorical
# variables
```

```
# random forest
set.seed(20460842)
library(randomForest)
# cannot handle predictors with more than 53 categories
rForest <- randomForest(data = train[, -c(11, 14)], SHOT_RESULT ~
    ., keep.forest = TRUE)
plot(rForest)

boot.rForest.err <- rep(0, times = 5)  #times=# bootstraps

for (i in 1:length(boot.rForest.err)) {
    boot <- sample(x = nrow(train), replace = TRUE)
    boot.Train <- train[boot, ]
```

```r
    boot.Validation <- train[-boot, ]
    boot.rForest <- randomForest(data = boot.Train[, -c(11, 14)],
        SHOT_RESULT ~ ., keep.forest = TRUE)
    boot.rForest.err[i] <- mean(ifelse(predict(boot.rForest,
        newdata = boot.Validation, type = "response") == boot.Validation$SHOT_RESULT,
        yes = 0, no = 1))
}

# only tune mtry (built-in); could tune maxnodes via grid
# search in the future only needs to be run once***
# rForest.tuned <- tuneRF(x=train[,-c(10,11,14)],
# y=train[,10], mtryStart=1, stepFactor=1, ntreeTry=500)
# #ntreeTry:=randomForest default

boot.rForest.tuned.err <- rep(0, times = 5)  #times=# bootstraps

for (i in 1:length(boot.rForest.tuned.err)) {
    boot <- sample(x = nrow(train), replace = TRUE)
    boot.Train <- train[boot, ]
    boot.Validation <- train[-boot, ]
    boot.rForest.tuned <- randomForest(data = boot.Train[, -c(11,
        14)], SHOT_RESULT ~ ., mtry = 1, importance = TRUE, keep.forest = TRUE)
    boot.rForest.tuned.err[i] <- mean(ifelse(predict(boot.rForest.tuned,
        newdata = boot.Validation, type = "response") == boot.Validation$SHOT_RESULT,
        yes = 0, no = 1))
}
```

```r
# gradient boosted decision trees
set.seed(20460842)
# cv for n.trees currently does not work (matrix is too
# large)

# distribution='bernoulli' in gbm requries 0-1 response
gbmTrain <- train
gbmValidation <- validation
library(dplyr)
gbmTrain$SHOT_RESULT <- recode(gbmTrain$SHOT_RESULT, made = 1,
    missed = 0)
gbmValidation$SHOT_RESULT <- recode(gbmValidation$SHOT_RESULT,
    made = 1, missed = 0)

library(gbm)
gbFit <- gbm(data = gbmTrain, formula = SHOT_RESULT ~ ., distribution = "bernoulli",
    n.trees = 1000, cv.folds = 5)
gbm.perf(gbFit, method = "cv")  #need at least 500 trees

boot.GBM.err <- rep(0, times = 5)  #times=# bootstraps

for (i in 1:length(boot.GBM.err)) {
    boot <- sample(x = nrow(gbmTrain), replace = TRUE)
    boot.Train <- gbmTrain[boot, ]
    boot.Validation <- gbmTrain[-boot, ]
    boot.GBM <- gbm(data = boot.Train, formula = SHOT_RESULT ~
        ., distribution = "bernoulli")
    boot.GBM.err[i] <- mean(ifelse(as.numeric(predict(boot.GBM,
        newdata = boot.Validation, n.trees = 100, type = "response") >=
        0.5) == as.numeric(boot.Validation$SHOT_RESULT), yes = 0,
        no = 1))  #n.trees is default
}

# tune shrinkage, n.trees, interactionDepth w/CV using grid
# search parameters <-
# expand.grid(shrinkage=c(0.001,0.01,0.1),
# interactionDepth=c(1,2,5), bag.fraction=c(0.1,0.5,0.9),
# n.trees=0, err=0)

# independent holdout validation set will be used to tune
# parameters within training set, n.trees tuned by 5-fold
# cross validation only needs to be run once*** for (i in
# 1:nrow(parameters)){ tuning.GBM <- gbm(data=gbmTrain,
# formula=SHOT_RESULT~., distribution='bernoulli',
# interaction.depth=parameters$interactionDepth[i],
# shrinkage=parameters$shrinkage[i],
# bag.fraction=parameters$bag.fraction[i], n.trees=500,
# cv.folds=5) tuning.Trees <- gbm.perf(tuning.GBM,
# method='cv', plot.it=FALSE) parameters$n.trees[i] <-
# tuning.Trees tuning.Err <-
# mean(ifelse(as.numeric(predict(tuning.GBM,
# newdata=gbmValidation[1:1000,], n.trees=tuning.Trees,
# #type='response')>=0.5)==as.numeric(gbmValidation$SHOT_RESULT[1:1000]),
# yes=0, no=1)) parameters$err[i] <- tuning.Err }

# library(knitr) kable(parameters[order(parameters$err),])

# optimal parameters are: shrinkage=0.001,
# interactionDepth=1, bag.fraction=0.1, n.trees=471 estimate
# prediction error using the tuned parameters

boot.GBMtuned.err <- rep(0, times = 5)  #times=# bootstraps

for (i in 1:length(boot.GBMtuned.err)) {
    boot <- sample(x = nrow(gbmTrain), replace = TRUE)
    boot.Train <- gbmTrain[boot, ]
    boot.Validation <- gbmTrain[-boot, ]
    boot.GBMtuned <- gbm(data = boot.Train, formula = SHOT_RESULT ~
        ., distribution = "bernoulli", shrinkage = 0.1, interaction.depth = 1,
        bag.fraction = 0.9, n.trees = 471)
    boot.GBMtuned.err[i] <- mean(ifelse(as.numeric(predict(boot.GBMtuned,
        newdata = boot.Validation, n.trees = 471, type = "response") >=
        0.5) == as.numeric(boot.Validation$SHOT_RESULT), yes = 0,
        no = 1))
}
```

```
# SVM TAKES VERY LONG TO RUN ~3hrs (works fast on smaller
# subsets of data n~10k)
library(e1071)
```

```
# use default radial basis function, gamma=1/ncol(x), cost=1
svmFit <- svm(data = train[1:10000, ], SHOT_RESULT ~ .)
mean(ifelse(predict(svmFit, newdata = test, type = "vector") ==
    test$SHOT_RESULT, yes = 0, no = 1))  #0.39

boot.SVM.err <- rep(0, times = 5)  #times=# bootstraps

for (i in 1:length(boot.SVM.err)) {
    boot <- sample(x = nrow(train[1:10000, ]), replace = TRUE)
    boot.Train <- train[boot, ]
    boot.Validation <- train[1:10000, ][-boot, ]
    boot.SVM <- svm(data = boot.Train, SHOT_RESULT ~ .)
    boot.SVM.err[i] <- mean(ifelse(predict(boot.SVM, newdata = boot.Validation,
        type = "vector") == boot.Validation$SHOT_RESULT, yes = 0,
        no = 1))
}

# tune using tune.svm from e1071 on validation set only needs
# to be run once*** svmTuned <-
# tune.svm(data=train[1:10000,], SHOT_RESULT~.,
# tunecontrol=tune.control(sampling='fix'),
# validation.x=validation[,-10],
# validation.y=validation[,10], gamma = 10^(-3:-1), cost =
# 10^(2:7))

# optimal parameters: gamma=0.001; cost=100

# use bootstrap to estimate prediction error on tuned model

boot.SVMtuned.err <- rep(0, times = 5)  #times=# bootstraps

for (i in 1:length(boot.SVMtuned.err)) {
    boot <- sample(x = nrow(train[1:10000, ]), replace = TRUE)
    boot.Train <- train[boot, ]
    boot.Validation <- train[1:10000, ][-boot, ]
    boot.SVMtuned <- svm(data = boot.Train, SHOT_RESULT ~ .,
        gamma = 0.001, cost = 100)
    boot.SVMtuned.err[i] <- mean(ifelse(predict(boot.SVMtuned,
        newdata = boot.Validation, type = "vector") == boot.Validation$SHOT_RESULT,
        yes = 0, no = 1))
}
```

```
# neural network
set.seed(20460842)
# softmax requires an indicator variable response throws
# errors when categorical variables are used
nnetTrain <- numericTrain
nnetValidation <- numericValidation
library(dplyr)
nnetTrain$SHOT_RESULT <- recode(nnetTrain$SHOT_RESULT, made = 1,
    missed = 0)
nnetValidation$SHOT_RESULT <- recode(nnetValidation$SHOT_RESULT,
    made = 1, missed = 0)
```

```r
library(nnet)
nnetResponse <- class.ind(nnetTrain[, 10])

nnetFit <- nnet(x = nnetTrain[, -10], y = nnetResponse, size = 1,
    softmax = TRUE)

boot.nnet.err <- rep(0, times = 5)  #times=# bootstraps

for (i in 1:length(boot.nnet.err)) {
    boot <- sample(x = nrow(nnetTrain), replace = TRUE)
    boot.Train <- nnetTrain[boot, ]
    boot.nnetResponse <- class.ind(boot.Train$SHOT_RESULT)
    boot.Validation <- nnetTrain[-boot, ]
    boot.nnet <- nnet(x = boot.Train[, -10], y = boot.nnetResponse,
        size = 1, softmax = TRUE)
    boot.nnet.err[i] <- mean(ifelse(predict(boot.nnet, newdata = boot.Validation,
        type = "class") == as.factor(boot.Validation$SHOT_RESULT),
        yes = 0, no = 1))
}

# tune w/ 5 fold CV w/caret only needs to be done once***
# library(caret) tuning.nnet <- train(x=numericTrain[,-10],
# y=numericTrain[,10], method='nnet', preProcess='scale',
# maxit=1000,
# tuneGrid=expand.grid(.size=c(1,5,10),.decay=c(0,0.1,0.01)),
# trControl=trainControl(method='cv', number=5),
# verbose=FALSE) tuning.nnet$bestTune

# optimal parameters are: size=10; decay=0

boot.nnet.Tuned.err <- rep(0, times = 5)  #times=# bootstraps

for (i in 1:length(boot.nnet.Tuned.err)) {
    boot <- sample(x = nrow(nnetTrain), replace = TRUE)
    boot.Train <- nnetTrain[boot, ]
    boot.nnet.Tuned.Response <- class.ind(boot.Train$SHOT_RESULT)
    boot.Validation <- nnetTrain[-boot, ]
    boot.nnet.Tuned <- nnet(x = boot.Train[, -10], y = boot.nnet.Tuned.Response,
        size = 10, decay = 0, softmax = TRUE)
    boot.nnet.Tuned.err[i] <- mean(ifelse(predict(boot.nnet.Tuned,
        newdata = boot.Validation, type = "class") == as.factor(boot.Validation$SHOT_RESULT),
        yes = 0, no = 1))
}  #performs near to random guessing (very dependent on initial weights)
```

```r
# model selection save.image('GlobalEnvironment')
par(mar = c(7, 5, 1, 1))
boxplot(boot.LDA.err, boot.QDA.err, boot.Log.err, boot.KNN.err,
    boot.LASSO.err, boot.GAM.err, boot.NB.err, boot.nbKDE.err,
    boot.rForest.err, boot.rForest.tuned.err, boot.GBM.err, boot.GBMtuned.err,
    boot.SVM.err, boot.SVMtuned.err, boot.nnet.err, boot.nnet.Tuned.err,
    names = c("LDA", "QDA", "Log", "KNN", "LASSO", "GAM", "NB",
        "nbKDE", "rF", "rF(Tuned)", "GBM", "GBM(Tuned)", "SVM",
        "SVM(Tuned)", "nnet", "nnet(Tuned)"), las = 2)
```