Feature Selection & Predictive Modelling For NBA Salary Prediction

Pranav Natarajan

06/05/2022

Loading Required Packages into workbook instance

```
require(doParallel) # for multicore processing
require(tidyr)# for easy data handling and pipeline creation
require(dplyr)# for easy data handling and pipeline creation
require(caret)# for machine learning models
require(Boruta)# for variable selection
require(xgboost)# for xgboost and SGD algorithms
require(glmnet) # for the elastic net
```

Loading comprehensive_stats from the rds file

```
player_stats<- readRDS(file="player_stats.rds") # 9628 rows of 76 variables! OK
```

getting summary of comprehensive stats

```
summary(player_stats)
```

Removing `SlugTeamsBREF` since `slugTeamBREF` already present.

```
player_stats<- player_stats %>% select(!c(slugTeamsBREF)) #76 variables
```

Removing further irrelevant columns

```
player_stats<- player_stats %>%
  select(!c(yearSeason:groupPosition, # do not need personal identifiers
  isHOFPlayer, # not required-- HOF attained only after retirement
  idPlayerNBA, # not required
  totSalary, salary_cap, # normalised salary already contains both their information
  )) # 67 columns left
```

getting revised summary of `player_stats`

```
summary(player_stats)
```

Removing `slugPlayerBREF` from feature set

```
player_stats<- player_stats %>%
  select(!c(slugPlayerBREF))
```

Creating the label set y = normalised_salary

```
y = player_stats$normalised_salary
```

Creating feature:- games not started as the difference between games and games started

```
player_stats<- player_stats %>%
  mutate(countGamesNotStarted = countGames - countGamesStarted,
         .before="pctFG")
```

Creating the dataframe of features, `X`

```
X = player_stats %>% select(!c(normalised_salary))
```

getting summary of `X`

```
summary(X)
```

Relocating the `slugTeamBREF` column to be after `slugPosition`

```
X<- X %>% relocate(slugTeamBREF, .after = slugPosition)
```

Variable Selection

Step 1:- removing redundancies

We will remove the per minute stats as advanced stats already has on floor percentages or specialised transformations of the same.

From the above and reading the spec text files, we note that we already have

1. eFG% <`pctEFG`> – Effective Field Goal Percentage, a percentage statistic that adjusts for the fact that a 3-point field goal is worth one more point than a 2-point field goal. This is, in essence, a transformation of the per game percentages of field goals, 2 pt field goals, 3 pt field goals present in the dataset. So, we will delete `pctFG:pctFG2`, (and `fgmPerGame:ftaPerGame`, on which the percentages depend upon).
2. Free throw percentage need not be computed since Free throw percentage throughout season is given by Advanced stat `pctFTRate`. So, we will delete `pctFT`
3. Note that the Advanced stats has percentages for steals, blocks, turnovers, assists, rebounds (including Offensive Rebounds and Defensive Rebounds encompassed by Total Rebounds). So, we can delete `orbPerGame:tovPerGame`. We will retain Personal Fouls and Points as per game stats
4. We will focus on Total rebounds, which factor in offensive and defensive rebounds, so we can delete `pctORB:pctDRB`

5. Similar logic as in 4. for Box Plus Minus, and Win shares. So, we will delete `ratioOWS:ratioDWS` and `ratioWSPer48:ratioDBPM`
6. Remove all per minute stats as mentioned above. So, delete `fgmPerMinute:ptsPerMinute`.

Doing so,

```
X<- X %>% select(!c(pctFG:pctFG2,
                    fgmPerGame:ftaPerGame,
                    pctFT,
                    orbPerGame:tovPerGame,
                    pctORB:pctDRB,
                    ratioOWS:ratioDWS,
                    ratioWSPer48:ratioDBPM,
                    fgmPerMinute:ptsPerMinute))
```

Revisiting the summary

```
summary(X)
```

Saving X and y for future use

```
# saving X
saveRDS(object = X,
        file = "X.rds")
# saving y
saveRDS(object = y,
        file = "y.rds")
```

Creating 80-20 training and testing set from X

```
set.seed(42) # for reproducability

# getting training tuple indices
# and splitting based on quantiles of y
# similar to stratified splittin in which the 'distribution'
# of the target is maintained in both training and test sets
train_indices<- createDataPartition(y = y,
                                    p = 0.8,
                                    list = F)

# creating training and testing sets
X_train<- X[train_indices,]
y_train<- y[train_indices]
X_test<- X[-(train_indices),]
y_test<- y[-(train_indices)]
```

```
# performing standard scaling of train features,
# and 'transforming' on the test features

# need a temporary variables to be able to extract center
# and scale attributes from the training continuous variables
X_train_cont_Scaled<- scale(x=X_train[, -(1:2)],
                            center=T, scale=T)
X_train[, -(1:2)]<- X_train_cont_Scaled
```

```
# "transforming" on the test set
X_test[, -(1:2)]<- scale(x=X_test[, -(1:2)],
                         center=attr(X_train_cont_Scaled,
                                     which = "scaled:center"),
                         scale=attr(X_train_cont_Scaled,
                                    which = "scaled:scale"))
```

Random Forest

The Boruta Algorithm for Variable Selection for Random Forest

The Boruta Algorithm uses ensemble methods with a `variable_importance` metric, and the principle of shadow features to select features.

```
Boruta_feat_selection<- Boruta(x = X_train,
                               y = y_train,
                               doTrace = 2,
                               maxRuns = 250)
```

Getting the feature importance plots

```
plot(Boruta_feat_selection,
     las=2,
     cex.axis=0.48,
     main = "Boruta Feature Importances for Random Forest")
```

```
plotImpHistory(Boruta_feat_selection)
```

we see that these plots both do not have any of our 23 variables as rejected, as shown by random forest feature importance.

fitting and tuning a Random Forest Regressor's number of random samples chosen as well as the number of trees using 10 fold CV, and the evaluation metric as `RMSE`

```
# instantiating the train control object
trControl<- trainControl(method = "cv",
                         number=5,
                         search = "grid",
                         verboseIter = T)
# instantiating the grid
tuneGrid<- expand.grid(mtry=seq(1:23),
                       splitrule=c("variance"),
                       min.node.size=c(5,6,7))
# setting the seed to 42 for reproducibility
set.seed(42)
```

```
# Running 5 fold CV grid Search of mtry values
# changed plans from repeated CV due to system inhibition
set.seed(42)
gridSearch_10FoldCV<- train(x = X_train,
                            y = y_train,
                            method = "ranger",
                            metric = "RMSE",
```

```
                                  trControl = trControl,
                                  tuneGrid = tuneGrid)
```

Getting the optimised hyperparameters

```
gridSearch_10FoldCV$bestTune
```

We have 14 (independent) variables to possibly split at in each node, with the node size at 5.

Getting the plot of the CV RMSEs vs the no. of predictors at each note by minimal Node size

```
plot(gridSearch_10FoldCV, main = "Random Forest Regressor")
```

Getting the best fit model, already fit on the training set.

```
nba_rfr<- gridSearch_10FoldCV$finalModel
```

getting the OOB scores (i.e., the overall MSE of out of bag samples) of the model

```
paste("sqrt(Random Forest OOB Score) = ",
      sqrt(nba_rfr$prediction.error))
#paste("Random Forest RMSE = ", sqrt(nba_rfr$prediction.error)) # note that both are the same thing
```

Note that the OOB Score is pretty good, while the training RMSE is around 5.5% of the yearly salary cap.

Getting the predictions on the test set:-

```
nba_rfr_preds<- predict(object = nba_rfr,
                        data = X_test)
```

Getting the RMSE on the test set as a metric of performance

```
rfr_RMSE<- RMSE(pred = nba_rfr_preds$predictions,
                obs = y_test)
paste("Random Forest RMSE = ",
      rfr_RMSE)
```

Note that the test set error quite agrees with the (square root of the) out of bag error.

Saving the model training object (with the best fit model) and RMSE Score

```
# saving model object
saveRDS(object=nba_rfr, "nba_rfr.rds")
```

```
# saving RMSE Score
saveRDS(object=rfr_RMSE, "nba_RMSE.rds")
```

Now, noting that in most cases, optimised tree based algorithms fit the training set better, possibly even overfit, we wish to compare this with a linear model.

First, we will compare an xgBoost Tree based regressor and an xgBoost Linear weighted function regressor. We will tune hyperparameters as well

E(X)treme (G)radient Boosting ensemble methods

Boruta Algorithm for XGBoost methods

Before we do CV, we will run the Boruta algorithm again, this time using xgBoost as the model object to derive variable importances from.

```
Boruta_feat_selection_xgboost<- Boruta(x = X_train,
                                       y = y_train,
                                       doTrace = 2,
                                       maxRuns = 250,
                                       getImp = getImpXgboost)
```

Viewing the variable stats upon running

```
#vars_for_selection<- attStats(Boruta_feat_selection_xgboost)
plot(Boruta_feat_selection_xgboost,
     las=2, cex.axis=0.5)
```

```
plotImpHistory(Boruta_feat_selection_xgboost)
```

```
vars_for_selection<- attStats(Boruta_feat_selection_xgboost)
vars_for_selection$decision
```

Note that there are confirmed, tentative and rejected. So, we can focus on the variables that are confirmed, and extract them as the training feature set for the xgBoost algorithms.

```
# getting subset of confirmed variables
vars_for_selection_conf<- vars_for_selection %>%
  filter(decision == "Confirmed") %>%
  select(decision)
```

```
# Getting the training Set for xgBoost
# note that the vector decision has rownames denoting each decision. So:
X_train_xgboost<- X_train %>%
  select(rownames(vars_for_selection_conf))
X_test_xgboost<- X_test %>%
  select(rownames(vars_for_selection_conf))
```

5 fold Cross Validation on the grid parameters to find the best hyperparameters for the xgBoost algorithms.

```
# instantiating the train control object
trControl<- trainControl(method = "cv", number=5,
                         search = "grid", verboseIter = T)
# instantiating the grid for xgbTree
tuneGridTree<- expand.grid(nrounds=100,
                           max_depth=c(5,6,7),
                           eta=c(0.1, 0.3, 0.5),
                           gamma=c(0, 0.05, 0.1, 0.5, 1),
                           colsample_bytree = 1, # default
                           min_child_weight =1, # default
                           subsample=1) # default
```

```
# instantiating the grid for xgbLinear
tuneGridLinear<- expand.grid(nrounds=100,
                             lambda=c(0, 0.05, 0.1, 0.5, 1),
                             alpha=c(0, 0.05, 0.1, 0.5, 1),
                             eta=c(0.1, 0.3, 0.5))
# setting the seed to 42 for reproducibility
set.seed(42)
```

XGBoost Decision Tree Regressor

```
# running the grid Search CV for the xgboost tree
set.seed(42) # for reproducibility

gridSearch_5FoldCV_Tree<- train(x=X_train_xgboost,
                                y=y_train,
                                method="xgbTree",
                                metric="RMSE",
                                trControl=trControl,
                                tuneGrid=tuneGridTree)
```

5 Fold CV Results of the XGBoost Decision Tree Regressor

```
# getting the optimised parameters for the xgbTree
gridSearch_5FoldCV_Tree$bestTune
```

```
plot(gridSearch_5FoldCV_Tree, main="XGBoost - Tree",
     xlab="Gamma (i.e., Minimum Loss Reduction)")
```

```
# saving the best model in a variable, and saving variable in an RDS file
nba_xgbTree<- gridSearch_5FoldCV_Tree$finalModel
saveRDS(object=nba_xgbTree, file="nba_xgbTree.rds")
```

XGBoost Linear Regularised Regressor

```
# running the grid Search CV for the xgboost linear

set.seed(42) # for reproducibility

gridSearch_5FoldCV_Linear<- train(x=X_train_xgboost,
                                  y=y_train,
                                  method="xgbLinear",
                                  metric="RMSE",
                                  trControl=trControl,
                                  tuneGrid=tuneGridLinear)
```

5 Fold CV Results of the XGBoost Linear Regularised Regressor

```
# getting the optimised parameters for the xgbTree
gridSearch_5FoldCV_Linear$bestTune
```

```
plot(gridSearch_5FoldCV_Linear, main="XGBoost - Linear")
```

```
# saving the best model in a variable, and saving variable in an RDS file
nba_xgbLinear<- gridSearch_5FoldCV_Linear$finalModel
saveRDS(object=nba_xgbLinear, file="nba_xgbLinear.rds")
```

Out of Sample Performances of XGBoost Regressors

```
# getting the predictions on the test set
nba_xgbTree_preds<- predict(object=nba_xgbTree,
                            newdata=as.matrix(X_test_xgboost))
nba_xgbLinear_preds<- predict(object=nba_xgbLinear,
                              newdata=as.matrix(X_test_xgboost))
```

```
# getting RMSEs
nba_RMSE_xgbTree<- RMSE(pred=nba_xgbTree_preds, obs=y_test)
nba_RMSE_xgbLinear<- RMSE(pred=nba_xgbLinear_preds, obs=y_test)
```

```
# saving RMSE values
saveRDS(object=nba_RMSE_xgbLinear,
        file="nba_RMSE_xgbLinear.rds")
saveRDS(object=nba_RMSE_xgbTree,
        file="nba_RMSE_xgbTree.rds")
```

```
paste("xgbTree RMSE = ", nba_RMSE_xgbTree)
paste("xgbLinear RMSE = ", nba_RMSE_xgbLinear)
```

We see that both xgBoost models (and the combination of hyperparameters chosen) perform poorer out of sample than the Random Forest Model.

Elastic Net

Note that the elastic Net is a linear combination of L1 and L2 norm regularisation, wherein we can handle overfitting (as well as underfitting by hyperparameter tuning of L2 Norm) and the selection of important features (by hyperparameter tuning of L1 Norm).

We will use the `glmnet` package to implement Elastic Net, and perform 5 fold CV on it as well to garner the best training performance. Read https://daviddalpiaz.github.io/r4sl/elastic-net.html for more reasoning, and noting that we only need optimise the mixing parameter, called `alpha`, and have the `glmnet` package compute the optimised lambda from its internal lambda sequence created.

```
# instantiating the train control object
trControl<- trainControl(method = "cv",
                         number=5,
                         search = "grid",
                         verboseIter = F)
# instantiating the grid for Elastic Net
# tuneGridElasticNet<- expand.grid(alpha= c(0, 0.001, 0.01, 0.1, 0.5, 0.9, 0.99, 0.999, 1),
#                                  lambda= c(seq(1,0, by=-0.01)))
# # Note that Alpha is the mixing Parameter,
# The penalty is defined as:-
# (1-alpha)/2||beta||_2^2+alpha||beta||_1.

# setting the seed to 42 for reproducibility
set.seed(42)
```

```r
# physically coercing to dummy matrix to ensure compilation with glmnet
x_train_mm<- data.frame(model.matrix(~.-1, data=X_train))
```

Running Grid Search CV for the Elastic Net

```r
# running the grid Search CV for the Elastic Net

set.seed(42) # for reproducibility

gridSearch_5FoldCV_ElasticNet<- train(x=x_train_mm,
                                      y=y_train,
                                      method="glmnet",
                                      family= "gaussian",
                                      metric="RMSE",
                                      trControl=trControl,
                                      tuneLength = 20,
                                      #tuneGrid=tuneGridElasticNet
                                      # 20 possible alpha values,
                                      # with 100 lambda values
                                      # determined by glmnet warm start
                                      standardize=F)
```

Results of Grid Search CV for the Elastic Net

```r
gridSearch_5FoldCV_ElasticNet$bestTune
```

```r
plot(gridSearch_5FoldCV_ElasticNet, cex=0.7)
```

```r
# saving the best model in a variable, and saving variable in an RDS file
nba_ElasticNet<- gridSearch_5FoldCV_ElasticNet$finalModel
saveRDS(object=nba_ElasticNet, file="nba_ElasticNet.rds")
```

Out of Sample Performance for the Elastic Net

```r
# creating the dumy variable matrix for X_test
x_test_mm<- data.frame(model.matrix(~.-1, data=X_test))
```

```r
# checking the number of variables present
# in the training set that are not in the test set
colnames(x_train_mm)[!(colnames(x_train_mm) %in%
                         colnames(x_test_mm))]
```

Now, we must add columns of zeroes to the test set to account for the nonexistence of these player (and their positions)

```r
# setting as a dataframe for easy manipulation of columns
x_test_mm<- data.frame(x_test_mm)
```

```r
# getting the number of
print(which(colnames(x_train_mm) == "slugPositionPG.SF")) #7
print(which(colnames(x_train_mm) == "slugPositionSF.C")) #10
print(which(colnames(x_train_mm) == "slugPositionSG.PF")) #14
```

```
# adding columns in respective locations corresponding to locations in x_train_mm
x_test_mm<- x_test_mm %>% mutate(slugPositionPG.SF = rep(0,nrow(x_test_mm)),
                                 .after=slugPositionPG)
x_test_mm<- x_test_mm %>% mutate(slugPositionSF.C = rep(0,nrow(x_test_mm)),
                                 .after=slugPositionSF)
x_test_mm<- x_test_mm %>% mutate(slugPositionSG.PF = rep(0,nrow(x_test_mm)),
                                 .after=slugPositionSG)
```

```
# getting the predictions on the test set and computing RMSE values
nba_ElasticNet_preds<- predict(object = nba_ElasticNet,
                               newx = as.matrix(x_test_mm),
                               type="response",
                               s = nba_ElasticNet$lambdaOpt)
```

```
# Getting the RMSE values on the test
nba_ElasticNet_RMSE<- RMSE(pred=nba_ElasticNet_preds, obs=y_test)
paste("Elastic Net RMSE = ", nba_ElasticNet_RMSE)
```

We further note, that the elastic net has performed the worst out of all the models so far, probably hinting at scope for further hyperparameter tuning using an iterative interval based alpha (and lambda) determination., or that the normality assumption of residuals does not hold, and a pure linear model is not suited to address this prediction. we would rather choose the xgBoosted Lienar model.

```
# saving the training and test model matrix dataframes
saveRDS(object=x_train_mm, file="x_train_mm.rds")
saveRDS(x=x_test_mm, file="x_test_mm.rds")

# saving the elasticNet RMSE predictions
saveRDS(object=nba_ElasticNet_RMSE,
        file="nba_ElasticNet_RMSE.rds")
```

Stochastic Gradient Boosted Decision Tree Regressor

Noting the realtively good performance of the xgBoost Tree model, we want to compare how it could perform with random subsampling – i.e., implememnting stochastic gradient descent. http://uc-r.github.io/gbm_regression is the source of understanding stochastic regression. Read CFRM 421 lecuter notes to understand SGD as an optimiser better if required.

Running 5 fold CV for the SGB Tree

```
# instantiating the train control object
trControl<- trainControl(method = "cv", number=5,
                         search = "grid", verboseIter = T)
# instantiating the grid for xbgTree
tuneGridSGDTree<- expand.grid(nrounds=200, # following advice in documentation to increase nrounds
                              max_depth=c(5,6,7),
                              eta=c(0.1, 0.3, 0.5),
                              gamma=c(0, 0.05, 0.1,
                                      0.5, 1),
                              colsample_bytree = 1,
                              min_child_weight =1,
                              subsample=c(0.5, 0.6, 0.75,
                                          0.9, 1))
```

```
# running the grid Search CV for the xgboost tree

set.seed(42) # for reproducibility

gridSearch_5FoldCV_SGDTree<- train(x=X_train_xgboost,
                                   y=y_train,
                                   method="xgbTree",
                                   metric="RMSE",
                                   trControl=trControl,
                                   tuneGrid=tuneGridSGDTree)
```

Results of 5 fold CV of SGB Tree

```
# printing CV train performance
plot(gridSearch_5FoldCV_SGDTree)
```

```
# getting the optimised parameters
gridSearch_5FoldCV_SGDTree$bestTune
```

```
# saving the best model in a variable, and saving variable in an RDS file
nba_SGDTree<- gridSearch_5FoldCV_SGDTree$finalModel
saveRDS(object=nba_SGDTree, file="nba_SGDTree.rds")
```

Out of Sample Performance of SGB Tree

```
# getting the predictions on the test set and computing RMSE values
nba_SGDTree_preds<- predict(object = nba_SGDTree,
                            newdata = as.matrix(X_test_xgboost))
```

```
# Getting the RMSE values on the test
nba_SGDTree_RMSE<- RMSE(pred=nba_SGDTree_preds,
                        obs=y_test)
paste("Stochastic GD RMSE = ", nba_SGDTree_RMSE)
```

We clearly see that the stochastic gradient descent algorithm with 90% resampling rate works better for the current parameter combination! The Computational Cost is also much lower, rationalising a case to prioritise it's further hyperparameter tuning.b

```
# saving the RMSE values of the SGD Linear Model
saveRDS(object=nba_SGDTree_RMSE,
        file="nba_RMSE_xgbTree.rds")
```

Visualising prediction 'deviance'

```
# Random Forest
scatter.smooth(x=nba_rfr_preds$predictions,
               y=y_test,
               lwd=2,
               xlab="Predicted salary as ratio of Yearly Salary Cap",
               ylab="True Salary as ratio of Yearly Salary Cap",
               main="Observed vs Predicted - Random Forest")
abline(0, 1, col="red")
legend("topleft", legend = c("Lowess Line", "Ideal Line"),
       lty = c(1,1), col=c("black", "red"))
```

```r
# xgbTree
scatter.smooth(x=nba_xgbTree_preds,
               y=y_test,
               lwd=2,
               xlab="Predicted salary as ratio of Yearly Salary Cap",
               ylab="True Salary as ratio of Yearly Salary Cap",
               main="Observed vs Predicted - XGBoost Tree")
abline(0, 1, col="red")
legend("topleft", legend = c("Lowess Line", "Ideal Line"),
       lty = c(1,1), col=c("black", "red"))
```

```r
# xgbLinear
scatter.smooth(x=nba_xgbLinear_preds,
               y=y_test,
               lwd=2,
               xlab="Predicted salary as ratio of Yearly Salary Cap",
               ylab="True Salary as ratio of Yearly Salary Cap",
               main="Observed vs Predicted - XGBoost Linear")
abline(0, 1, col="red")
legend("topleft", legend = c("Lowess Line", "Ideal Line"),
       lty = c(1,1), col=c("black", "red"))
```

```r
# Elastic Net
scatter.smooth(x=nba_ElasticNet_preds,
               y=y_test,
               lwd=2,
               xlab="Predicted salary as ratio of Yearly Salary Cap",
               ylab="True Salary as ratio of Yearly Salary Cap",
               main="Observed vs Predicted - Elastic Net")
abline(0, 1, col="red")
legend("topleft", legend = c("Lowess Line", "Ideal Line"),
       lty = c(1,1), col=c("black", "red"))
```

```r
# SGB Tree
scatter.smooth(x=nba_SGDTree_preds,
               y=y_test,
               lwd=2,
               xlab="Predicted salary as ratio of Yearly Salary Cap",
               ylab="True Salary as ratio of Yearly Salary Cap",
               main="Observed vs Predicted - SGD Tree algorithm x Gradient Boosting")
abline(0, 1, col="red")
legend("topleft", legend = c("Lowess Line", "Ideal Line"),
       lty = c(1,1), col=c("black", "red"))
```