STAT 441 Group Project

Group 1

Contents

Executive Summary

This report uses NFL Big Data Bowl 2021 data to classify passes in American Football as "Success"/"Fail". This classification is made based on whether or not the pass was successful. To train our models, we use the offensive team's players' movement data up to the point the pass was made, as well as the positions of some key defenders.

This study compares several models in classifying the pass result based on the tracking data. The top 2 models to highlight are XG Boost and Random Forest. Their test error rates are 28.6% and 28.7%. and the best tuning parameters for Random Forest are 100 trees and 150 maximum terminal nodes per tree. Other models considered include GLM Net Single Tree and Bagging, and their test error rate almost match the top 2 models. All models excels at classifying complete passes but not incomplete ones, and this could due to the fact that complete passes outnumber incomplete passes.

This report concludes that the best classification model either XG Boost, or Random Forest, or Bagging, for it minimizes test error and balances sensitivity and specificity. Lasso Logistic Regression is also a competitive candidate depending on modelling preference, Additional work needs to be done to incorporate the tracking data of more players as well as including individual player data into the models.

Introduction

Gridiron Football, also known as American Football (and hereafter simply referred to as football) is on of the most popular sports in North America. It is played as a contest between 2 teams of 11, with the aim being to score as many points as possible. Teams can score points by either advancing the ball into the opposing teams' end-zone, or by kicking the ball through the opponents' goal posts.

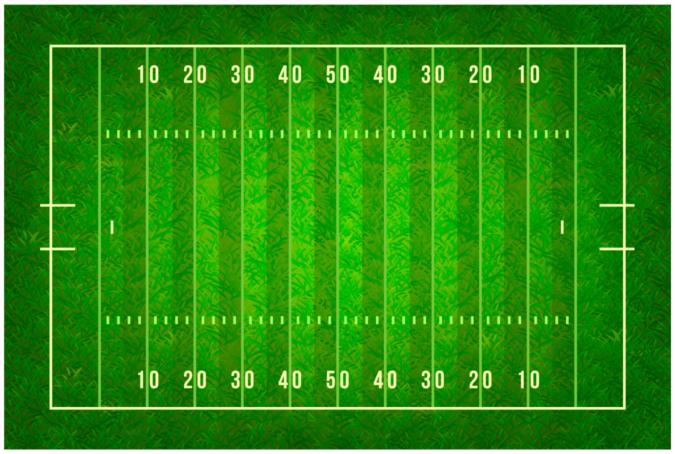


Figure 1: Football Field Illustration

As with many other contemporary sports, football has undergone a data revolution. With the development of new technology it is now possible to collect vast amounts of data in real time, and use that data to analyze the flow and progression of the game. Each players' exact location, direction of movement and field of vision can be precisely captured for the duration of the game. This data can be leveraged to identify strengths and weaknesses in the teams' plays, and choose optimal strategies. The dataset chosen by us for this project is from an annual competition knows as the NFL data bowl. While the specifics change each year, the organizers provide participants with real second-by-second data from the previous season of the NFL. The participants must come up with novel and useful ways to use this data to provide insight into the performance of the teams in the NFL.

Problem of Interest

In a game of football, the single most important role is the quarterback, this is a fact that can be proved by the average salary across all positions. Thus it is clear that in order to be successful, a team must have a high quality quarterback. In our study, we wish to construct a way to analyze quarterbacks. We believe that the most important aspect of ability for a quarterback is to pass the ball, more specifically, passing to a wide receiver. This is a task that can potentially convert to a instant touch down. We use various machine learning algorithm to predict the result of a QB-passing-to-WR play. When there exist a mismatch between the actual pass result and our prediction, it is a sign that something happened during the play that need further investigation. It might be that the QB simply threw a bad pass, or that there is a lack of synergy between the quarterback and the wide receiver. Our models can be further expand to perform live prediction during the games to help audience better understanding the game.

Data cleaning varibles manipulation

The initial idea of our project is to build and find proper models to classify whether given passing in one play could be a successful ball passing or not based on information from the moment the ball was thrown by the quarterback. This requires microdata to support the modeling process and be fed into models as input variables. We obtained our original dataset from Big Data Bowl which is an open-source data science competition on Kaggle. We focused on tracking data sets of each play which are Frame by frame data containing the 2-dimension Cartesian coordinates of players on the field, along with their speed, movement direction, acceleration, orientation, and other detailed features that can be used to profile what the situation on the field.

Instead of using some massive but ‘black box’ models, we were considering making some more compact and interpretable models. In that case, we can not just put every variable provided by the raw data sets we have. We have to manually reduce the number of variables with prior knowledge we have about football games and try to construct new descriptive features to summarize information we have. With these kinds of empirical approaches, ideally, we could capture more relevant information and also get rid of some redundant information, before we used other modeling-based methods to do further selections and explanations.

Therefore, the data cleaning was performed in two stages. First, we identified five key players in each frame the ball was about to be thrown. Two of them are from possession team, quarterback and wide receiver (also can be denoted as target). Rest three of them are from defense team, who are the top 3 closest players to the wide receiver based on Euclidean distance. We then extracted variables which are shown in Table.1.

Table 1: Selected Features

WR based info	QB based info	Defender(1∼3) based info	Other
(x, y)	(x, y)	(x, y)	GameId
Speed	Δx.toWR	Speed	PlayId
Acceleration	Δy.toWR	Acceleration	Yardline Number
Distance	Euclidean Distance.toWR	Distance	Pass Result
Orientation		Orientation	Passing Route
Dircetion		Direction	
		Position	

Pairs of (GameId, PlayId) serve as keys that are unique among each row of instances. 'Pass Result' is the binary response (Complete or Incomplete) variable of subsequent models we built. Beside that, the passing route and Positions of three defenders are also categorical variables with multi class within each. Rest of variables are all continuous scalars. The origin point of the Cartesian coordinate system is set in the left bottom corner of the top view of football pitch. 'Direction' and 'Orientation' of each recorded player are measured based on degree. Therefore, the zero degree has to be clearly defined, which according to the Big Data Bowl description documents is the positive direction of y-axis and increases monotonically clock-wise. 'Distance' measures the distance a certain player traveled from prior time point, in yards.

Then for the second stage, based on the variables extracted from raw data sets, we tried to seek some more conclusive features, so that we can further shrink the size of training data on the column side. All the variables we have for now regarding the player movements or locations are stored in scalar form. But in real life, instead of speed, vectorized velocity is actually what we mostly used for analysis purposes dealing with multi-dimensional movement. It can be properly decomposed and projected in designated directions. So the most intuitive idea is, instead of handing in scalar data to models and letting them figure out potential relationships, we should do it before by reintegrating variables and generating projections of velocities.

The basic idea of projection is to consider the line between players' positions as the target projection plane. Since we are focusing on the relative trend of movement between defenders and wide receiver, it is natural that we set the chaser to target as the positive direction. So the distance vector can be calculated by:

$$D = (X_{WR}, Y_{WR}) - (X_D, Y_D) \tag{1}$$

where (X_{WR}, Y_{WR}) is the wide receiver's location and (X_D, Y_D) is the location of the defender player we are interest in.

The polar coordinate system defined by the original data set is not the same as we commonly used. But luckily it can be easily converted into what we are familiar with by:

$$d^\star = 90 - d_{original} \tag{2}$$

After we converted 'Direction' variable for wide receiver and all defender players using Equ.2, the vectorized velocity can be calculated by:

$$v = s \times (\cos(d^\star), \sin(d^\star)) \tag{3}$$

Then with a inner product of velocity calculated by Eqn.3 and unitized distance vector from Eqn.1, the projection is:

$$v_p = v \cdot \frac{D}{||D||_2}$$

Based on previously mentioned equations, for each wide receiver - Defender pair in our data set we can then summarize 2 velocity projections in a total of 6 newly made variables as shown in tab.2

Table 2: New Features

Defender1→WR	Defender2→WR	Defender3→WR
v_1^{WR}	v_2^{WR}	v_3^{WR}
v_1^D	v_2^D	v_3^D

where v_i^{WR} is the WR's velocity projection on distance vector from i^{th} closest defender to himself, v_i^D is the i^{th} closest defender's velocity projection on distance vector from himself to WR.

After velocity projection variables were constructed, we went through the whole data set we have, found out the NA value, and simply dropped the rows which contain NA value since the number of rows with NA (About 50) is way too small compared to the total instance we have(More than 9 thousand).

Methods

GLM Net and Logistic Regression

We used logistic regression as a starting point of the modelling process since it is based on simple linear model and gives binary outcome: complete or incomplete pass. Since the original dataset contains around 40 predictors, model regularization is necessary to avoid overfitting and variance inflation caused by multi-collinearity. We used elastic net to select variables and maintain grouping effects for pairwise correlated variable groups (Hastie, 2005). Building a GLM Net model involves selecting 2 tuning parameters: alpha and lambda. Lambda scales the L_1 and L_2 penalty terms, while α distribute different weights to L_1 and L_2 penalty terms. We selected α and λ in the following way (Andreis, 2017): 1. Supply a grid of α ranging from 0 to 1, with each step be 0.1. 2. For each α in the grid, apply `cv.glmnet()` to find the suitable lambda and associated binomial deviance. 3. Store the α-λ pair and deviance in a data frame, choose the pair that yields as least deviance as the tuning parameter to train the final model.

At the second, step, 1-standard-error rule is used to choose the best lambda, this helps use simpler models to yield the similar test performance as more complex models (Hastie, 2021).

Random Forest and Bagging

Random forest bagging draw bootstrap samples from the training set and build tree predictors for each bootstrap samples. `randomForest()` from randomForest package was used to build both models. By default, 500 trees were grown for each predictor, and out-of-bag error (OOB) and test error were plotted against the number trees for each predictor. Since the plots are fuzzy curves, we picked the number of trees that roughly settled the OOB error to be build the final model. The random forest randomly drew 6 predictors to make splits in trees, while the bagging method used all predictors to make splits. Variable importance was plotted for both predictors. The maximum of nodes is inherited from the single tree predictor.

XGBoost

XGBoost is one of the most used Machine Learning algorithm in the data science industry as well as competitions. The Python package Optuna was used to perform the hyperparameter tuning for its power and efficiency.

The objective function used is the binary logistic objective function, which is also known as the log-loss function:

$$yln(p) + (1-y)ln(1-p) \quad where \quad p = \frac{1}{1+e^{-z}} \tag{4}$$

Hyperparameter tuning

GLMNET logistic regression

We applied cross-validation `cv.glmnet()` to a grid of α's to choose the optimal values of α and λ to use while fitting the GLMNET model. The optimal values of the parameters are $\alpha = 1, \lambda = 0.00465$, so the final GLM model is a Lasso regression with L_1 penalty term scaled by 0.00465.

Results

A bar chart and a scatter plot are useful to compare and highlight difference in model performance. While the test error rate shows proportion of misclassfied cases, area under ROC (AUC) summarizes type I and type II errors at all possible thresholds. On the bar chart, AUC show subtle difference between models, the 5 attempted models had 0.7 of AUC on average, with XG Boost being the top performer for this criterion, and the Single Tree be the bottom performer. All models had roughly the same level test error which is between 28% to 30%.

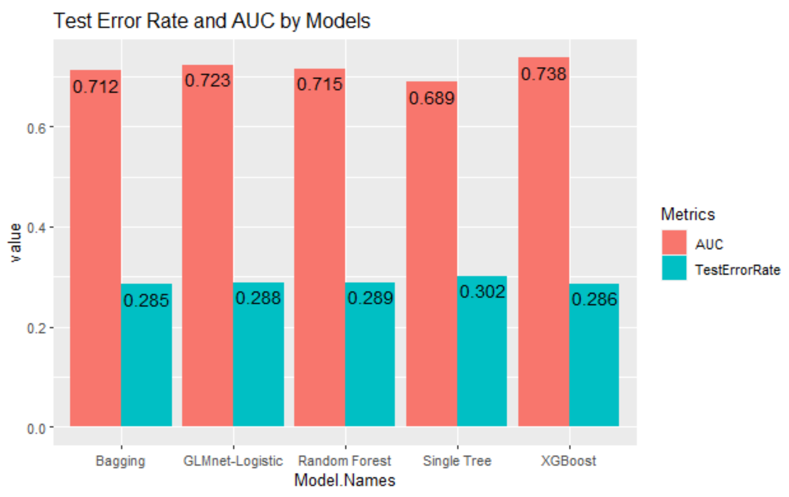


Figure 2: Random Forest Feature Importance

The scatter plot shows a trade-off between sensitivity and specificity. Setting "incomplete pass" as a control case, a single tree seems to be best at identifying complete passes but bad at identifying incomplete passes. XG Boost behaved oppositely. Random Forest and Bagging are in the middle of the scale, and little difference can be found between these two models.

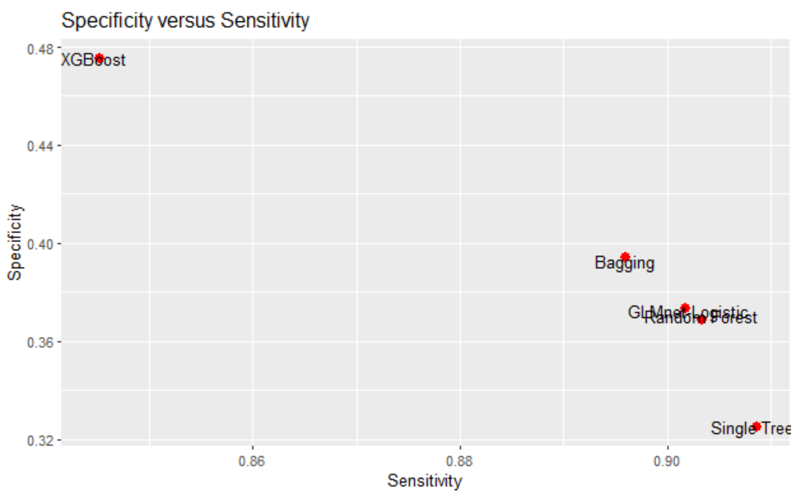


Figure 3: Random Forest Feature Importance

To sum up, different models perform roughly the same in terms of accuracy but not so same when correctly classifying different categories.

GLM Net and Logistic Regression

Using `cv.glmnet()` and follow the guideline of Andreis (2017), we found that $(\alpha, \lambda) = (1, 0.01421135)$ yield the lowest cross validation error, so we chose this tuning parameter pair to fit the final GLM Net. The final GLM Net with logistic regression has around 28.4% of test error rate, using 30% of the dataset as test set (2975 observations). The error rate is just below that of the best model we have fitted: XGboost, which will be discussed later. The sensitivity and specificity on the test set are 90.0% and 38.9%, we can see a trade-off between these two quantities: high sensitivity could imply a low specificity. The area under ROC curve is 0.7279, the second highest among the five models that we have tried. The results above indicate a satisfactory performance of GLM Net.

Random Forest and Bagging

Now, we fit random forests on the dataset. We let the number of trees in the random sequence vary and optimize for the best number of trees in the sequence.

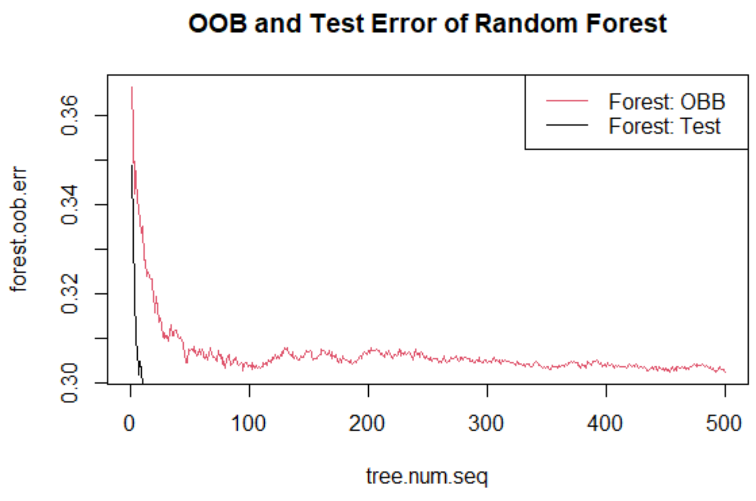


Figure 4: Error Rate vs. Number of Trees in Sequence

The plot suggests that using 100 random trees in the sequence minimizes the test error without overly complicate the model, so we proceed by fitting a random forest with a sequence of 100 trees. From the random forest, we can identify feature importance as well.

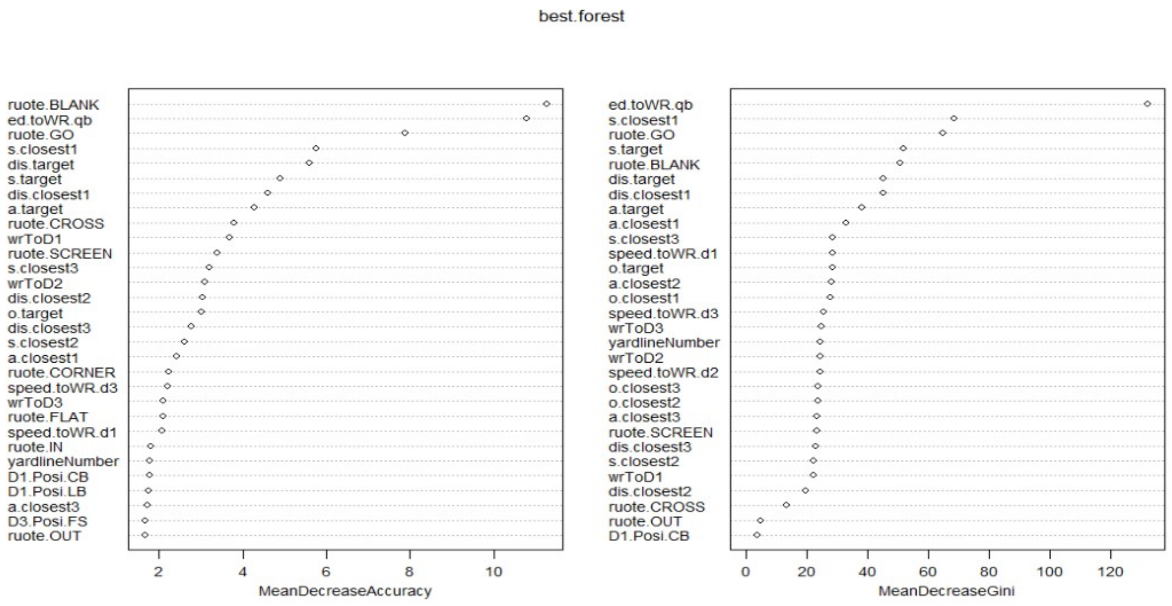


Figure 5: Random Forest Feature Importance

Bagging

For bagging, we also optimize the number of trees in the sequence to minimize the OOB error.

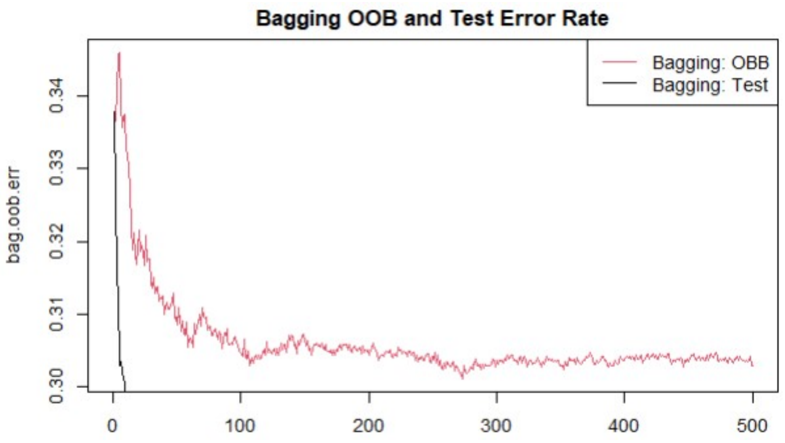


Figure 6: Bagging Feature Importance

The feature importance is as follows

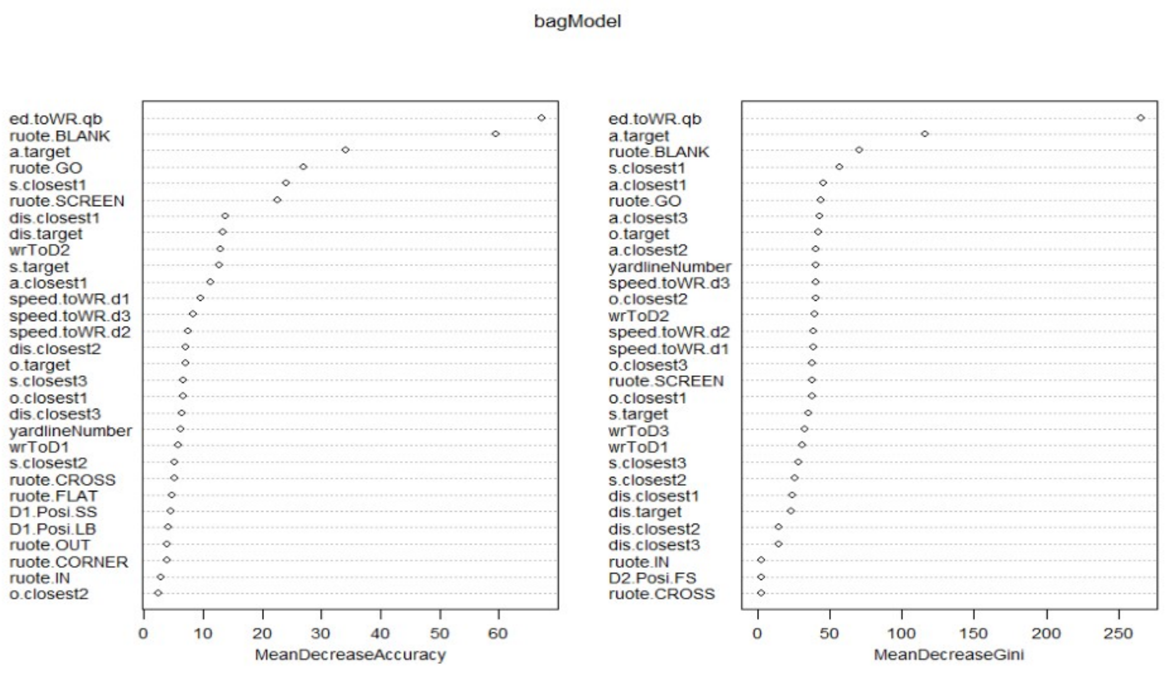
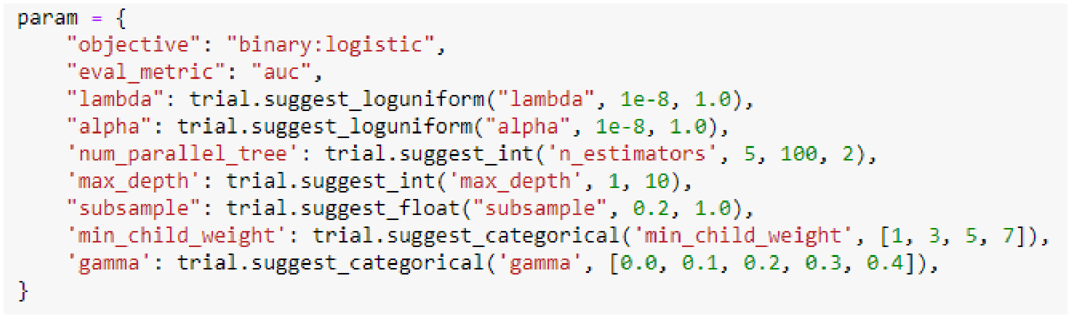


Figure 7: Bagging Feature Importance

As with the random forest, using 100 trees in the sequence is optimal.

XGBoost

Hyperparameter tunning over XGBoost was done using Optuna over these parameters:



```
param = {
    "objective": "binary:logistic",
    "eval_metric": "auc",
    "lambda": trial.suggest_loguniform("lambda", 1e-8, 1.0),
    "alpha": trial.suggest_loguniform("alpha", 1e-8, 1.0),
    'num_parallel_tree': trial.suggest_int('n_estimators', 5, 100, 2),
    'max_depth': trial.suggest_int('max_depth', 1, 10),
    "subsample": trial.suggest_float("subsample", 0.2, 1.0),
    'min_child_weight': trial.suggest_categorical('min_child_weight', [1, 3, 5, 7]),
    'gamma': trial.suggest_categorical('gamma', [0.0, 0.1, 0.2, 0.3, 0.4]),
}
```

Figure 8: XGBoost parameter-tuning with Optuna

- lambda: L2(ridge) regularization term on weights
- alpha: L1(Lasso) regularization term on weights
- num_parallel_tree(nround): number of parallel trees constructed during each boosting iteration

- max_depth: maximum depth of a single tree
- subsample: sample part of the training set prior to tree building, this is used for preventing overfitting
- min_child_weight: minimum sum of weight needed in a child node, if a node contains less weight than this, the tree will stop partitioning
- gamma: minimum loss reduction required to further partition a node

Conclusion

From the fitted models, we see that the different models optimize different metrics.To optimize prediction error rate on the test set, we should choose Bagging, which has a test error rate of 28.5% The XGBoost Model also has a very high specificity, of nearly 50%, much better than any of the other models.To optimize AUC of the ROC Curve, we should choose the XGBoost Model, which has an AUC of 0.738.

The Bagging and Random Forest Bagging also tell us which features are the most important. For instance, they both identify the distance between the WR and the nearest defender to be very important, which is reasonable. This can provide useful insights into features which may not seem important but actually are.

What is notable is that despite all the models having relatively similar test error rates and AUC values, some models are much better at classifying successful passes while other models are better at classifying unsuccessful passes. This suggests that the models are assigning different importance to the different features. However, they generally agree on the relative importance.

There is still some scope for improvement. We have not considered all the variables available to us in the dataset. This is due to us not having enough time, but also because it is not obvious how we can summarize all the remaining player movement information.

However, given that our models consistently perform much better than random chance, we have definitely picked up on key trends in the underlying data.

Contribution

Jiang Wei(David) Liu:

- Initial draft the project
- Data gather/cleaning
- XGBoost modelling

Jingxin (Owen) Wang:

- GLM Net, Single Tree, Random Forest, Bagging methods modelling and result analysis.
- Model performance summary Graphics

Rohit Gajendragadkar :

- Introduction, conclusion
- Report RMD formatting
- Video recording

Yicheng (Lucio) Qin :

- Topic Selection
- Data cleaning/Feature Engineering
- Report formatting

References

- Zou,H., Hastie, T. (2005). Regularization and variable selection via the elastic net. J.R Statistics. Soc. B. https://hastie.su.domains/Papers/B67.2%20(2005)%20301-320%20Zou%20&%20Hastie.pdf

- Andreis, F., & Milano,M. (2017). Shrinkage methods and variable selection: Ridge, Lasso, and Elastic Nets [Course Slides]. Università Commerciale Luigi Bocconi. https://www.researchgate.net/profile/Federico-Andreis/publication/321106005_Shrinkage_methods_ridge_lasso_elastic_nets/links/5a0da3d7aca2729b1f4eeabb/Shrinkage-methods-ridge-lasso-elastic-nets.pdf

- Hastie, T., James, G., Tibshirani, R., & Witten, D. (2021). Introduction to statistical Learning with Applications in R (Second Edition). https://hastie.su.domains/ISLR2/ISLRv2_website.pdf

- Kaggle(2021). 2021 Big Data Bowl. https://www.kaggle.com/competitions/nfl-big-data-bowl-2021/data

Appendix 1: Single Tree Method and Results

A full tree was grown based on all predictors in the dataset. The full tree was passed to `cv.tree()` choose the best size of tree and the multiplier k to the scalar. The tree size and scalar k that first settle the negative log likelihood to a stable level is chosen to build the final tree for testing. Gini index was used to make tree splits.

Next, we try to fit individual trees to the dataset. We let the size of the tree vary, and optimize for the ideal tree size.

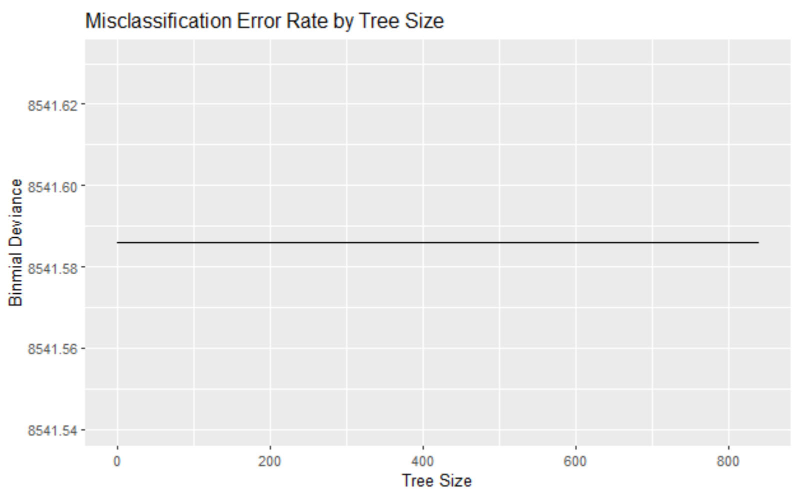


Figure 9: Binomial Deviance vs. Tree size

It seems like the size of the tree has no impact on the binomial deviance. The prediction power is also very weak. As a result, we will not be using individual trees to make predictions

Appendix 2: ROC Curve Inspection

We now compute take the best fitted models from each of the above categories, and compute the error rate on the test/validation set. We also construct the ROC plots and compare the AUC for each of the models.

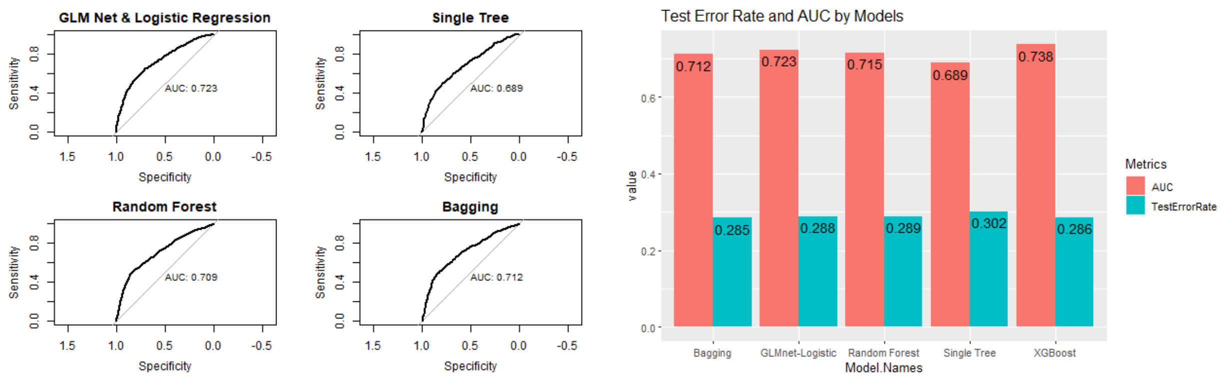
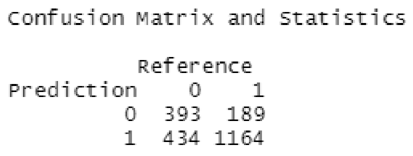


Figure 10: ROC Curves and Test Error Rates

The above plots suggest that the best model by test error rate is Bagging, while the best model by AUC of ROC is XGBoost.

The result accuracy of the XGBoost model is **71%** over the test data set, however, it is also useful to inspect the confusion matrix of the model.



```
Confusion Matrix and Statistics

          Reference
Prediction    0    1
         0  393  189
         1  434 1164
```

Figure 11: XGBoost confusion matrix

The model has a high Sensitivity, while the Specificity is not nearly as high, this is because it is easier to predict a successful pass attempt, while those unsuccessful ones can be caused by events that are not described in the data set(body movements/Linemen interactions).

Appendix 3: Code for data clearning

```
# Find all unique pair of (gameId, playId)
def find_gp_keys(original_dat,group_size=5):
    game_play=[i.values.tolist() for _,i in original_dat[['gameId','playId']].iterrows()
    gp_keys=[]
    gid=None
    pid=None
    count=None
    while game_play != []:
        gid_,pid_=game_play.pop(0)
        if gid_ != gid or pid_ !=pid:
            if count == group_size:
                gp_keys.append([gid,pid])
            count=1
    #          gp_keys.append([gid_,pid_])
            gid=gid_
            pid=pid_
        else:
            count+=1
    return gp_keys

#Convert data into long form
#input original data return df
def convert(original_dat):
    gp_keys=find_gp_keys(original_dat)

    dat_list=[]
    for gid,pid in gp_keys:
        sub_dat=original_dat[(original_dat['gameId']==gid) & (original_dat['playId']==pi

        wr_info=sub_dat[(sub_dat['position.x']=='WR')]\
        [[\
          'gameId','playId','yardlineNumber','passResult','targetNflId',\
          'x.x','y.x','s','a','dis','o','dir','route'\
         ]].values.reshape(-1) #15
#         if wr_info[-1]=='HITCH':
#             route_info=np.array([1,0,0])
#         elif wr_info[-1]=='IN':
#             route_info=np.array([0,1,0])
#         elif wr_info[-1]=='OUT':
#             route_info=np.array([0,0,1])
#         dat_per_play=np.append(wr_info[:-1],route_info)
        dat_per_play=np.append(np.array([]),wr_info)

        qb_info=sub_dat[(sub_dat['position.x']=='QB')]\
        [[\
          'x.x','y.x','x.toWR','y.toWR','ed.toWR'\
         ]].values.reshape(-1) #5
        dat_per_play=np.append(dat_per_play,qb_info)

        defender_info=sub_dat[(sub_dat['position.x']!='QB') & (sub_dat['position.x']!='W
        [[\
          'ed.toWR','x.x','y.x','s','a','dis','o','dir','position.x'\
         ]] #6*3
        sorted_by_ed=sorted(defender_info.iterrows(), key=lambda x:x[1][0])
        for _,defender in sorted_by_ed:
            dat_per_play=np.append(dat_per_play,defender.values[1:])

        dat_list.append(dat_per_play.reshape(1,-1))

#Return df version of long form data
    col_names=[\
        'gameId','playId','yardlineNumber','passResult','targetNflId',\
        'x.target','y.target','s.target','a.target','dis.target','o.target','dir.target'
        'route',\
        'x.qb','y.qb','x.toWR.qb','y.toWR.qb','ed.toWR.qb',\
        'x.closest1','y.closest1','s.closest1','a.closest1','dis.closest1','o.closest1',
        'x.closest2','y.closest2','s.closest2','a.closest2','dis.closest2','o.closest2',
        'x.closest3','y.closest3','s.closest3','a.closest3','dis.closest3','o.closest3',
    ]
    summarized=np.concatenate(dat_list,axis=0)
    summarized_df=pd.DataFrame(summarized,columns=col_names)

    return summarized_df
```

```
dat=pd.read_csv('...',header=0)

# Constructing Velocity projection
list_dat=[]
for _,play in dat.iterrows():
    game_info=play[['gameId','playId']]
    wr_coor=play[['x.target','y.target']].values.astype('float')
    d_coor=play[['x.closest1','y.closest1','x.closest2','y.closest2','x.closest3','y.clo

    d_coor=d_coor.reshape(3,-1)
    dist_vect=(wr_coor-d_coor)

    #Get direction of distance
    unit_vect=dist_vect/np.sqrt(np.sum(dist_vect**2,axis=1)).reshape(3,-1)

    #Mapping dirction degree into the commonly used version
    dir_wr=90-play[['dir.target']].values.astype('float')
    dir_d=90-play[['dir.closest1','dir.closest2','dir.closest3']].values.astype('float')

    #Get speed
    s_wr=play[['s.target']].values.astype('float')
    s_d=play[['s.closest1','s.closest2','s.closest3']].values.astype('float')

    #Vectorized velocity
    v_wr=np.append(np.cos(dir_wr),np.sin(dir_wr))*s_wr
    v_d=(np.append(np.cos(s_d),np.sin(s_d)).reshape(-1,3)).T*s_d.reshape(3,-1)

    #Making projection with inner product
    projection_d=np.sum(v_d*unit_vect,axis=1)
    projection_wr=np.sum(v_wr*unit_vect,axis=1)


    v_projection=np.append(game_info,projection_wr)
    v_projection=np.append(v_projection,projection_d)


    list_dat.append(v_projection.reshape(1,-1))


# Return df version of new variables
col_names=[\
    'gameId','playId','wrToD1','wrToD2','wrToD3','speed.toWR.d1','speed.toWR.d2','speed.
]
projection=np.concatenate(list_dat,axis=0)
projection_df=pd.DataFrame(projection,columns=col_names)
```

Load Data and Packages

```
library("tree")
library("ggplot2")
library("tidyverse")
library("randomForest")
library("glmnet")
library("pROC")
library(reshape2)
library(tidyverse)
library(xgboost)
library(xlsx)
library(caret)


target.com.inc <- target %>%
```

```r
  merge(players, by.x ="targetNflId", by.y = "nflId") %>%
  filter(position == "WR") %>%
  merge(plays, by =c("gameId", "playId")) %>%
  filter(passResult %in%  c('C','I', 'IN'))

getPassFrame.position.method <- function(path){
  passframe.total <- read_csv(path) %>%
    filter((event=="pass_forward") &
             (position %in% c("QB","WR","CB","SS","FS","LB"))) %>%
    select(x,y,s,a,dis,o,dir,nflId,position,gameId,playId,route) %>%

    #merge target df, filter out the WR that weren't targeted
    merge(target, c("gameId", "playId")) %>%
    filter(!((position =="WR")&(nflId != targetNflId)))



  passframe.target <- passframe.total %>%
    select(gameId,playId,x,y,nflId,position) %>%
    filter(position == "WR")



  passframe.calc <- passframe.total %>%
    merge(passframe.target, by.x = c("gameId", "playId", "targetNflId"),
          by.y = c("gameId", "playId", "nflId")) %>%

    # calculate distance to WR
    mutate(x.toWR = x.y - x.x, y.toWR = y.y - y.x, ed.toWR = sqrt(x.toWR^2 + y.toWR^2))
    distinct(.keep_all = TRUE)

  #passframe.calc
}

track.passframe <- list.files(path = "2021/tracking/",
              pattern="*.csv",
              full.names = T) %>%
    map_df(~getPassFrame.position.method(.))

track.closest3Defender <- track.passframe %>%
  group_by(gameId, playId) %>%
  filter(!(position.x %in% c("QB","WR"))) %>%
  top_n(-3,ed.toWR)

track.offence <- track.passframe %>%
  filter(position.x %in% c("QB","WR"))

target.playStat <- target.com.inc %>%

  # can include WR height weight here
  select(gameId, playId,yardlineNumber,passResult)

# add the final result
track.data <- rbind(track.closest3Defender,track.offence) %>%
  # merge the result later
  merge(target.playStat, c("gameId", "playId"))



NFL <- read.csv("221204-Lucio-imputed.csv")
NFL <- NFL[, -1]
```

```r
samp.size <- nrow(NFL)
train.size <- floor(samp.size * 0.7)
train <- sample(1:samp.size, train.size)

y.var <- "passResult"; x.var <- names(NFL)[-2]
model.formula <- formula(paste(y.var,"~",paste(x.var, collapse = "+")))

## Convert categorical to numeric
# NFL$passResult <- factor(NFL$passResult)
# for (column in NFL) {
#   if (class(column) == "character") {
#     NFL = NFL %>% mutate(colnames(column) = column %>% factor())
#   }
#   # print(class(column))
#
# }

for (var in names(NFL)) {
  if (class(NFL[, var]) == "character") {
    NFL[, var] <- NFL[, var] %>% factor()
  }
}

# GLM Net in Logistic Regression
NFL.mtrx <- model.matrix(passResult ~ .,NFL[train,])[,-1]
NFL.mtrx.y <- NFL$passResult[train]



### CV on Elastic Net
```

```r
alpha_list <- seq(0,1, 0.1)
tuning_frame <- data.frame("alpha" = c(), "lambda" = c(), "cost" = c())

for (alpha in alpha_list) {
    if (alpha == 0) {
      set.seed(1)
      logit.net.cv <- cv.glmnet(NFL.mtrx, NFL.mtrx.y, # function auto generate lambda
                                keep = T,                    # maintain foldid every itera
                                alpha = alpha, nfolds = 10, family = "binomial")
      foldid <- logit.net.cv$foldid
    } else {
      logit.net.cv <- cv.glmnet(NFL.mtrx, NFL.mtrx.y, # function auto generate lambda
                                keep = T, foldid = foldid,   # maintain foldid every iter
                                alpha = alpha, nfolds = 10, family = "binomial")
    }

    best.lambda <- logit.net.cv$lambda.1se ## 1-standard-error rule
    lowest.cost <- min(logit.net.cv$cvm)
    tuning_frame <- rbind(tuning_frame, c(alpha, best.lambda, lowest.cost))
}
colnames(tuning_frame) <- c("alpha", "lambda", "cost")
best_net_param <- tuning_frame[which.min(tuning_frame$cost), c("alpha", "lambda")]
best_net_param


## Elastic Net with the best alpha - lambda lambda
logit.net.best <- glmnet(NFL.mtrx, NFL$passResult[train], lambda = best_net_param$lambda
          alpha = best_net_param$alpha, family = "binomial")
```

```r
# Single Tree
full.tree<- tree(model.formula,
                data = NFL, subset = train,
                na.action = na.omit, split = "gini")

# Cross validation to choose tree size and the scalar for the tree size
k_list <- seq(0, 100, 2)
tree.cv.seq <- cv.tree(full.tree, K = 10, FUN = prune.tree, k = k_list)
tree.cv.seq
plot(tree.cv.seq)

best.param.indx <- which.min(tree.cv.seq$dev)
best.param <- list("alpha" = tree.cv.seq$k[best.param.indx],
                   "size" = tree.cv.seq$size[best.param.indx])
# best.tree <- prune.tree(full.tree, k = best.param$alpha, best = best.param$size)
best.tree <- prune.tree(full.tree, k = 10, best = 157)

# Plot by Tree Size
Error_by_treeSize <- data.frame(Size = tree.cv.seq$size, Loss = tree.cv.seq$dev)
ggplot(Error_by_treeSize) + geom_line(mapping = aes(x = Size, y = Loss)) +
  ggtitle("Misclassification Error Rate by Tree Size")

# Random Forest and Bagging
zero_One_loss <- function(pred, actual){
  mean(pred != actual)
}

forestModel <- randomForest(model.formula,
  data = NFL, subset = train, na.action = na.omit,
  mtry = 6, nodesize = 20, maxnodes = 157,
  xtest = NFL[-train,-2], ytest = NFL$passResult[-train],
  importance = TRUE)
# Error Decrease with ntree
tree.num.seq <- 1:500
forest.oob.err <- forestModel$err.rate[,1] #OOB error
forest.test.err <- forestModel$test$err.rate[,1] # Test error

plot(tree.num.seq, forest.oob.err, col = 2, type = "l")
lines(tree.num.seq, forest.test.err, col = 1)
legend("topright", col = c(2,1), legend = c("Forest: OBB", "Forest: Test"), lty = 1)



## Seems that ntree = 60 can achieve the satisfactory result
data.frame(
  "ntree" = 60,
  "Forest:OOB Error" = forest.oob.err[60],
  "Forest:Test Error" = forest.test.err[60]
)

best.forest <- randomForest(model.formula,
  data = NFL, subset = train, na.action = na.omit,
  mtry = 6, nodesize = 20, maxnodes = 157, ntree = 60,
  xtest = NFL[-train,-2], ytest = NFL$passResult[-train],
  importance = TRUE)

ncol = ncol(NFL) - 1
## Bagging
bagModel <- randomForest(model.formula,
  data = NFL, subset = train, na.action = na.omit,
  mtry = 44, nodesize = 20, maxnodes = 157,
  xtest = NFL[-train,-2], ytest = NFL$passResult[-train],
  importance = TRUE)

# Importance Plot
varImpPlot(bagModel)
# Confusion Matrix
bagModel$confusion

# Error Decrease with ntree
tree.num.seq <- 1:500
bag.oob.err <- bagModel$err.rate[,1] #OOB error
bag.test.err <- bagModel$test$err.rate[,1] # Test error

plot(tree.num.seq, bag.oob.err, col = 2, type = "l")
lines(tree.num.seq, bag.test.err, col = 1)
legend("topright", col = c(2,1), legend = c("bag: OBB", "bag: Test"), lty = 1)

## Seems that ntree = 100 can achieve the satisfactory result
data.frame(
  "ntree" = 100,
  "Bagging:OOB Error" = bag.oob.err[100],
  "Bagging:Test Error" = bag.test.err[100]
)

best.bagging <- randomForest(model.formula,
  data = NFL, subset = train, na.action = na.omit,
  mtry = 6, nodesize = 20, maxnodes = 157, ntree = 100,
  xtest = NFL[-train,-2], ytest = NFL$passResult[-train],
  importance = TRUE)

# Ada Boosting
library(gbm)
adaModel  <- gbm(as.numeric(passResult)-1 ~., data = NFL[train, ],
                 n.tree = 500, interaction.depth = 4,
                 shrinkage = 0.01, cv.folds = 10)

summary(adaModel, las = 1, plotit = T)

## Lasso, Logistic
NFL.test.mtrx <- model.matrix(NFL$passResult ~., NFL[-train])[,-1]
NFL.test.y <- NFL$passResult[-train]

net.fitted.type = predict(logit.net.best, newx = NFL.test.mtrx[train,], type = "class")
net.pred.type = predict(logit.net.best, newx = NFL.test.mtrx[-train,], type = "class")

# net.train.error <- mean(NFL$passResult[train] != net.fitted.type, na.rm = TRUE)
net.test.error <- mean(NFL$passResult[-train] != net.pred.type, na.rm = TRUE)

test.result <- data.frame(
    "Training Error" = net.train.error,
    "Test Error" = net.test.error
  )
test.result

## ROC curve
net.pred.prob = predict(logit.net.cv, newx = NFL.mtrx[-train,], type = "response")
net.ROC<- roc(NFL$passResult[-train], net.pred.prob, plot = T, print.auc = TRUE, directi

## Single Tree
tree.fitted.type = predict(best.tree, newdata = NFL[train,], type = "class")
tree.pred.type = predict(best.tree, newdata = NFL[-train,], type = "class")

tree.train.error <- mean(NFL$passResult[train] != tree.fitted.type, na.rm = TRUE)
tree.test.error <- mean(NFL$passResult[-train] != tree.pred.type, na.rm = TRUE)

test.result <- data.frame(
    "Training Error" = tree.train.error,
    "Test Error" = tree.test.error
  )
test.result

## ROC curve
tree.pred.prob = predict(best.tree, newdata = NFL[-train,], type = "vector")
tree.ROC<- roc(NFL$passResult[-train], tree.pred.prob[,1],
               plot = T, print.auc = TRUE, direction = "<")

# Worse than the random guess

## Random Forest
# Importance Plot
varImpPlot(best.forest)
# Confusion Matrix
best.forest$confusion

## Random Forest
```

```r
forest.fitted.type = predict(best.forest, newdata  = NFL[train,], type = "class")
forest.pred.type = predict(best.forest, newdata = NFL[-train,], type = "class")

forest.pred.prob = predict(best.forest, newdata = NFL[-train,], type = "prob")


# forest.train.error <- mean(NFL$passResult[train] != forest.fitted.type, na.rm = TRUE
forest.test.error <- mean(NFL$passResult[-train] != forest.pred.type, na.rm = TRUE)

# test.result <- data.frame(
#     "Training Error" = forest.train.error,
#     "Test Error" = forest.test.error
#   )
# test.result

# ROC curve
forest.pred.prob = predict(forestModel, newdata = NFL[-train,], type = "response")
forest.ROC<- roc(NFL$passResult[-train], forest.pred.prob,
                 plot = T, print.auc = TRUE, direction = "<") # ROC available for rando

## Bagging
bag.fitted.type = predict(baggingModel, newdata = NFL[train,], type = "class")
bag.pred.type = predict(baggingModel, newdata  = NFL[-train,], type = "class")

bag.train.error <- mean(NFL$passResult[train] != bag.fitted.type, na.rm = TRUE)
bag.test.error <- mean(NFL$passResult[-train] != bag.pred.type, na.rm = TRUE)

test.result <- data.frame(
    "Training Error" = bag.train.error,
    "Test Error" = bag.test.error
  )
test.result

## ROC curve
# bag.pred.prob = predict(baggingModel, newdata  = NFL[-train,], type = "class")
# bag.ROC<- roc(NFL$passResult[-train], bag.pred.prob,
               # plot = T, print.auc = TRUE, direction = "<") # ROC available for bagg


## Ada Boost
ntree_opt_cv = gbm.perf(adaModel, method = "cv")
pred.ada.prob <- predict(adaModel, n.trees = ntree_opt_cv, newdata = NFL[-train, ],
                         type = "response")
ada.pred.type = ifelse(pred.ada.prob >= 0.5, "C", "I")

table(ada.pred.type , NFL$passResult[-train])
mean(ada.pred.type != NFL$passResult[-train])

## Summary Table

data.frame(
  Model.Names = c("GLMnet-Logistic", "Single Tree", "Random Forest",
                  "Bagging", "AdaBoost"),
  TestErrorRate = c(logit.test.error, tree.test.error, forest.test.err, bag.test.error),
  Sensitivity = c(),
  Specificity = c()
)
```

```
# Area Under ROC
par(mfrow = c(2,3), mar = c(2,2,2,2))

roc(NFL$passResult[-train], logit.pred.prob,
                 plot = T, print.auc = TRUE, direction = "<") # ROC
roc(NFL$passResult[-train], tree.pred.prob,
                 plot = T, print.auc = TRUE, direction = "<") # ROC
roc(NFL$passResult[-train], forest.pred.prob,
                 plot = T, print.auc = TRUE, direction = "<") # ROC
roc(NFL$passResult[-train], forest.pred.prob,
                 plot = T, print.auc = TRUE, direction = "<") # ROC
roc(NFL$passResult[-train], ada.pred.prob,
                 plot = T, print.auc = TRUE, direction = "<") # ROC



# XGBoost model building

set.seed(441)

df <- read.csv("final_df_1205.csv")

df[is.na(df)] <- 0

train <- sample(seq_len(nrow(df)), size = 8000)

train.data <- data.matrix(df[train,] %>%
  subset(select=-c(gameId,playId,targetNflId)) %>%
  select(!passResult))
train.label <- df[train,]$passResult
```

```
test.view <- df[-train,] %>%
  subset(select=-c(gameId,playId,targetNflId)) %>%
  select(!passResult)
test.data <- data.matrix(df[-train,] %>%
  subset(select=-c(gameId,playId,targetNflId)) %>%
  select(!passResult))
test.label <- df[-train,]$passResult



model <-  xgboost(data = train.data, # the data
                  label = train.label,
                 max.depth = 4,
                 lambda = 0.4,
                 min_child_weight = 4,
                 nround = 27,
                 objective = "binary:logistic",
                 )

pred <- as.numeric(predict(model, test.data) > 0.5)
err <- mean(pred != test.label)
print(paste("test-error=", err))
```

```
# python codes for hyper tuning xgboost model
# package used: Optuna, sklearn
def Xgb_Objective(trial):

    train_x, test_x, train_y, test_y = train_test_split(df_x, df_y, test_size=0.25)
    dtrain = xgb.DMatrix(train_x, label=train_y)
    dtest = xgb.DMatrix(test_x, label=test_y)
    param = {
        "objective": "binary:logistic",
        "eval_metric": "auc",
        "lambda": trial.suggest_loguniform("lambda", 1e-8, 1.0),
        "alpha": trial.suggest_loguniform("alpha", 1e-8, 1.0),
        'n_estimators': trial.suggest_int('n_estimators', 5, 100, 2),
        'max_depth': trial.suggest_int('max_depth', 1, 10),
        "subsample": trial.suggest_float("subsample", 0.2, 1.0),
        'min_child_weight': trial.suggest_categorical('min_child_weight', [1, 3, 5, 7]),
        'gamma': trial.suggest_categorical('gamma', [0.0, 0.1, 0.2, 0.3, 0.4]),
    }

    # Add a callback for pruning.
    pruning_callback = optuna.integration.XGBoostPruningCallback(trial, "validation-auc"
    bst = xgb.train(param, dtrain, evals=[(dtest, "validation")], callbacks=[pruning_cal
    preds = bst.predict(dtest)
    pred_labels = np.rint(preds)
    accuracy = sklearn.metrics.accuracy_score(test_y, pred_labels)
    return accuracy

study = optuna.create_study(direction="maximize")
study.optimize(Xgb_Objective, n_trials=100, timeout=600)

print("Number of finished trials: ", len(study.trials))
print("Best trial:")
trial = study.best_trial

print("  Value: {}".format(trial.value))
print("  Params: ")
```

```python
for key, value in trial.params.items():
    print("{}: {}".format(key, value))
```