Xgboost Tuning

Gordon Wang

December 3, 2018

Load R Environment

```
# Read in R environment (with training data already cleaned and feature engineering done)
load("temp_env.RData")
```

Import libraries/packages

```
library(dplyr)
```

```
##
## Attaching package: 'dplyr'
```

```
## The following objects are masked from 'package:stats':
##
##     filter, lag
```

```
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```
library(randomForest)
```

```
## randomForest 4.6-14
```

```
## Type rfNews() to see new features/changes/bug fixes.
```

```
##
## Attaching package: 'randomForest'
```

```
## The following object is masked from 'package:dplyr':
##
##     combine
```

```
library(chron)
library(pROC)
```

```
## Type 'citation("pROC")' for a citation.
```

```
##
## Attaching package: 'pROC'
```

```
## The following objects are masked from 'package:stats':
##
##     cov, smooth, var
```

```
library(foreach)
```

```
##
## Attaching package: 'foreach'
```

```
## The following object is masked from 'package:chron':
##
##     times
```

```
library(doParallel)
```

```
## Loading required package: iterators
```

```
## Loading required package: parallel
```

```
library(xgboost)
```

```
##
## Attaching package: 'xgboost'
```

```
## The following object is masked from 'package:dplyr':
##
##     slice
```

```
library(Matrix)
library(caret)
```

```
## Loading required package: lattice
```

```
## Loading required package: ggplot2
```

```
##
## Attaching package: 'ggplot2'
```

```
## The following object is masked from 'package:randomForest':
##
##     margin
```

Split Training Data (2015-2016 Flight Data) into Train/Validation Sets (Because CV Error can be too optimistic about model)

```
# Ensure results are repeatable
set.seed(9)

# Create indices for 80% train set and 20% validation set split using caret package
train_idxs =
  createDataPartition(    # Function preserves data's class imbalance in train and test sets
  departing_flights$DEP_DEL15, p = .80,
  list = FALSE,
  times = 1)

# Subset data into training and validation sets based on indices
train_df = departing_flights[train_idxs, ]
validation_df = departing_flights[-train_idxs, ]
```

Fitting Extreme Gradient Boosting (XGBoost) Models

xgb2: First Model

Preparing features and processing data and labels

```
# Choose features for model (variables we intuitively considered influential)
features = c("DEP_DEL15", "MONTH", "DAY_OF_WEEK", "AIRLINE_ID", "CRS_DEP_TIME",
             "AWND", "PRCP", "SNOW",
             "WT01", "WT02", "WT03", "WT04", "WT06", "WT08", "WT09",
             "NYE_PROX", "MLK_PROX", "WAS_PROX", "MEM_PROX", "IND_PROX",
             "LAB_PROX", "COL_PROX", "VET_PROX", "THX_PROX", "XMS_PROX",
             "PCT_DEL_1", "NUM_DEL_1", "NUM_BEF_1",
             "PCT_DEL_2", "NUM_DEL_2", "NUM_BEF_2",
             "PCT_DEL_3", "NUM_DEL_3", "NUM_BEF_3",
             "PCT_DEL_4", "NUM_DEL_4", "NUM_BEF_4")

# Select corresponding columns from train and validation sets for features
xgb2_train_df = train_df[, features]
xgb2_validation_df = validation_df[, features]

# Use One Hot Encoding to convert any categorical variables besides response into numeric
train_expanded = sparse.model.matrix(DEP_DEL15 ~ .-1, data=xgb2_train_df)
validation_expanded = sparse.model.matrix(DEP_DEL15 ~ .-1, data=xgb2_validation_df)

# Create labels for response variable
train_y = (train_df$DEP_DEL15 == 1)
validation_y = (validation_df$DEP_DEL15 == 1)

# Convert daa with xgb.DMatrix into matrix data structure used by XGBoost
dtrain = xgb.DMatrix(data=train_expanded, label=train_y)
dvalidation = xgb.DMatrix(data=validation_expanded, label=validation_y)
wgt = sum(train_y == FALSE) / sum(train_y == TRUE)  # To scale pos obs by, since high class imbalance
```

Train baseline, untuned xgb2 model with default parameters to examine how AUC is improved by tuning

```
# Default parameters
params = list(
         booster = "gbtree",
         objective = "binary:logistic",  # Because binary classification problem
         eta = 0.3,
         gamma = 0,
         max_depth = 6,
         min_child_weight = 1,
         subsample = 1,
         colsample_bytree=1,
         scale_pos_weight=wgt, # Scales by earlier-calculated wgt (in effect balances the classes)
         eval_metric = "auc"   # Evaluate predictions based on AUC (Under ROC)
)
```

```
# Train untuned xgb2 model with above default parameters
boost_out2_untuned = xgb.train(params = params, dtrain, nrounds = 100, verbose=0) #nrounds=100:default

# Make predictions
guess_xgb2_untuned = predict(boost_out2_untuned, validation_expanded)

# Check AUC performance
auc(validation_y, guess_xgb2_untuned)
```

```
## Area under the curve: 0.7029
```

AUC is below 0.70, which is low, meaning I am possibly overfitting the training data. Thus, I lower the shrinkage factor or learning rate (eta) from default value of 0.3 to 0.1 (a good rule of thumb). This slows down the learning of the model by applying a smaller weighting factor for corrections by new trees when added to the model.

When the learning rate is lower, the nrounds parameter (max number of iterations or number of trees to grow) needs to be higher since it takes more iterations for gradient descent to converge/get to the optimum number of trees. Thus, I raise this value from default of 100 to 200.

Now, with new eta = 0.1, I start tuning by determining via stratified 10-fold cross-validation the optimal number of trees for this learning rate. Number of trees is considered the main tuning parameter because too large of a value overfits training data and too small of a value leads to a bad classifier. I use stratified CV instead of CV to maintain in each fold the high class imbalance of the training data.

The optimal number of trees for this set of parameters is the iteration for which the model returns the lowest CV error. I use early_stopping_rounds to stop the boosting after CV error doesn't improve after 10 iterations of boosting.

Tuning xgb2 model

Step 1: Tune optimal number of trees for eta = 0.1 with CV (using nrounds and early_stopping_rounds)

```
# Use 10-fold CV to tune opt. tree num. with eta = 0.1, and defaults for other params
cv1 = xgb.cv(list(objective = 'binary:logistic',
             eta = 0.1,
             scale_pos_weight=wgt,
             eval_metric = "auc"), dtrain,
             watchlist = list(train =dtrain, test=dvalidation),
             early_stopping_rounds = 10,    # Stop if CV score doesn't improve after 10 iters.
             stratified = TRUE, nfold = 10, # Stratified CV to preserve in folds class imbalance
             metrics = list("auc"),
             maximize = TRUE,
             nrounds=200,    # Higher nrounds since lower eta = more iters. until optimal tree num.
             verbose=0)
cv1$best_iteration        # Optimal tree num is iter that returns lowest CV error
```

```
## [1] 46
```

Use random search with cross-validation to next tune variables that control tree complexity: max_depth, size of each new decision tree, and min_child_weight, the minimum number of observations that must be in the children after the split Smaller trees means less complexity and Smaller weight means more conservative model Tune these parameters together to find a good tradeoff between model bias and variance.

Step 2: Tune max_depth and min_child_weight to control tree complexity, (finding good tradeoff between model bias and variance), using newly tuned optimal number of trees as nrounds value.

```
# Use random grid search with cross-validation in caret package
## Set up grid for tuning max_depth and min_child_weight
xgb2_grid = expand.grid(
            nrounds = cv1$best_iteration,      # Use previously found optimal number of trees
            max_depth = c(1, 2, 3, 4, 5), # Size of new decision tree (how deep the tree can grow)
            eta = 0.1,
            gamma = 0,
            colsample_bytree = 1,
            min_child_weight = c(5, 6, 7, 8), # Min num obs req in child node after split
            subsample = 1                     # (blocks potential feature interactions)
)

# Fixes and stratifies folds to preserve class imbalance and ensure reproducibility
folds = createFolds(train_df$DEP_DEL15)

# Pack training control parameters
xgb2_trcontrol = trainControl(
                 method = "cv",
                 number = 10,           # Number of folds
                 search = "random",     # Random search to save time
                 verboseIter = FALSE,   # No training log
                 returnData = FALSE,
                 returnResamp = "all",  # Save losses across all models
                 classProbs = TRUE,     # Set to TRUE for computing AUC
                 summaryFunction = twoClassSummary,
                 allowParallel = TRUE,  # Ensures faster computation
                 index = folds)         # stratifed folds to preserve class imbalance

# Train models for each parameter comb. in grid and choose one with lowest error
xgb2_tune1 = train(
             x = dtrain,
             y = factor(train_df$DEP_DEL15, labels = c("NotDelayed","Delayed")),  # Refactor resp.
             trControl = xgb2_trcontrol,
             tuneGrid = xgb2_grid,
             method = "xgbTree",
             metric = "ROC",             # Use AUC under ROC as evaluation metric
             scale_pos_weight = wgt      # Bal. classes by weighting minority class (Non Delays) more
)

# Report the tuned min_child_Weight and max_depth values
xgb2_tune1$bestTune
```

```
##    nrounds max_depth eta gamma colsample_bytree min_child_weight subsample
## 12      46         3 0.1     0                1                8         1
```

Step 3: Tune sampling parameters: subsample and colsample_bytree, and retune opt tree num with early_stopping_rounds (These params adds randomness to make training robust to noise, to prevent overfitting)

```
# Set up tuning grid with updated parameters, and ranges for tuning targets
xgb2_grid2 = expand.grid(
             nrounds = 100,           # Need new num. of opt. trees grown, so raise max trees num.
             max_depth = xgb2_tune1$bestTune$max_depth,
             eta = 0.1,
             gamma = 0,
             colsample_bytree = c(0.5, 0.7, 0.8, 0.9), # Fract. of features/columns samp. each iter
             min_child_weight = xgb2_tune1$bestTune$min_child_weight,
             subsample = c(0.3, 0.4, 0.5, 0.6, 0.7, 0.8)  # Fract. of training set samp. each iter.
)

# Pack training control parameters
xgb2_trcontrol2 = trainControl(method = "cv",
                               number = 10,
                               search = "random",
                               verboseIter = FALSE,
                               returnData = FALSE,
                               returnResamp = "all",
                               classProbs = TRUE,
                               summaryFunction = twoClassSummary,
                               allowParallel = TRUE,
                               index = folds)     # Same stratifed folds for reproducibility

# Train models for each parameter combination in the grid and choose one with lowest error
xgb2_tune2 = train(x = dtrain,
                   y = factor(train_df$DEP_DEL15,
                              labels = c("NotDelayed","Delayed")),
                   trControl = xgb2_trcontrol2,
                   tuneGrid = xgb2_grid2,
                   method = "xgbTree",
                   metric = "ROC",
                   early_stopping_rounds = 10,       # stops at new optimal number of trees
                   scale_pos_weight = wgt,
                   watchlist = list(test = dvalidation),
                   verbose = 0
)

# Report the tuned subsampling and colsample_bytree values
xgb2_tune2$bestTune
```

```
##    nrounds max_depth eta gamma colsample_bytree min_child_weight subsample
## 18     100         3 0.1     0              0.8                8       0.8
```

Step 4: Tune Gamma, and use early_stopping_rounds to stop boosting at optimal number of trees for this parameter set

```
# Set up tuning grid with updated parameters, and ranges for tuning targets
xgb2_grid3 = expand.grid(
```

```
            nrounds = 150,                              # Raised max boosting rounds just in case
            max_depth = xgb2_tune1$bestTune$max_depth,
            eta = 0.1,
            gamma = c(0, 0.01, 0.02, 0.03, 0.04),
            colsample_bytree = xgb2_tune2$bestTune$colsample_bytree,
            min_child_weight = xgb2_tune1$bestTune$min_child_weight,
            subsample = xgb2_tune2$bestTune$subsample
)

# Pack training control parameters
xgb2_trcontrol3 = trainControl(method = "cv",
                               number = 10,
                               search = "random",
                               verboseIter = FALSE,
                               returnData = FALSE,
                               returnResamp = "all",
                               classProbs = TRUE,
                               summaryFunction = twoClassSummary,
                               allowParallel = TRUE,
                               index = folds)         # same stratifed folds as before

# Train models for each parameter combination in the grid and choose one with lowest error
xgb2_tune3 = train(x = dtrain,
                   y = factor(train_df$DEP_DEL15,
                              labels = c("NotDelayed","Delayed")),
                   trControl = xgb2_trcontrol3,
                   tuneGrid = xgb2_grid3,
                   method = "xgbTree",
                   metric = "ROC",
                   early_stopping_rounds = 10,      # Stops boosting at new optimal number of trees
                   scale_pos_weight = wgt,
                   watchlist = list(test=dvalidation),
                   verbose = 0
)

# Report the tuned gamma value
xgb2_tune3$bestTune
```

```
##   nrounds max_depth eta gamma colsample_bytree min_child_weight subsample
## 5     150         3 0.1  0.04              0.8                8       0.8
```

Step 5: Now tune learning rate (eta) lower and nrounds upwards (because optimal tree number increases)

```
# Set up tuning grid with updated parameters, and ranges for tuning targets
xgb2_grid4 = expand.grid(
            nrounds = c(50, 75, 90, 100, 200, 300, 400), # nrounds must inc as eta decr.
            max_depth = xgb2_tune1$bestTune$max_depth,
            eta = c(0.05, 0.07, 0.08, 0.09, 0.1),
            gamma = xgb2_tune3$bestTune$gamma,
            colsample_bytree = xgb2_tune2$bestTune$colsample_bytree,
            min_child_weight = xgb2_tune1$bestTune$min_child_weight,
            subsample = xgb2_tune2$bestTune$subsample
```

```
)

# Pack training control parameters
xgb2_trcontrol4 = trainControl(method = "cv",
                               number = 10,
                               search = "random",
                               verboseIter = FALSE,
                               returnData = FALSE,
                               returnResamp = "all",
                               classProbs = TRUE,
                               summaryFunction = twoClassSummary,
                               allowParallel = TRUE,
                               index = folds)         # same stratifed folds for reproducibility

# Train models for each parameter combination in the grid and choose one with lowest error
xgb2_tune4 = train(x = dtrain,
                   y = factor(train_df$DEP_DEL15,
                              labels = c("NotDelayed","Delayed")),
                   trControl = xgb2_trcontrol4,
                   tuneGrid = xgb2_grid4,
                   method = "xgbTree",
                   metric = "ROC",
                   scale_pos_weight = wgt,
                   early_stopping_rounds = 10,
                   watchlist = list(test = dvalidation),
                   verbose = 0
)

# Report the tuned eta value
xgb2_tune4$bestTune
```

```
##    nrounds max_depth eta gamma colsample_bytree min_child_weight subsample
## 29      50         3 0.1  0.04              0.8                8       0.8
```

Step 6: Fine-tune the optimal number of trees with updated parameters using CV (relying on early_stopping_rounds).

```
# Make list of Tuned parameters as input for xgb.cv()
xgb2_params_tuned = list(
         booster = "gbtree",
         objective = "binary:logistic",
         eta = xgb2_tune4$bestTune$eta,
         gamma = xgb2_tune4$bestTune$gamma,
         max_depth = xgb2_tune4$bestTune$max_depth,
         min_child_weight = xgb2_tune4$bestTune$min_child_weight,
         subsample = xgb2_tune4$bestTune$subsample,
         colsample_bytree=xgb2_tune4$bestTune$colsample_bytree,
         scale_pos_weight=wgt,
         eval_metric = "auc"
)

# Use stratified 10-fold cross-validation to retune optimal number of trees
```

```
cv_tuned = xgb.cv(params = xgb2_params_tuned,
             data = dtrain,
             early_stopping_rounds = 10,   # Boost. stops if CV score not improv. aft. 10 rounds
             watchlist = list(test = dvalidation),
             stratified = TRUE,
             nfold = 10,
             metrics = list("auc"),
             maximize = TRUE,
             nrounds = 800,    # Hig max tree num and hope opt tree numis smaller than this
             verbose=0)

cv_tuned$best_iteration    # New Optimal number of trees (Set as new nrounds for final training)
```

```
## [1] 98
```

Step 7: Train final tuned xgb2 model with nrounds = optimal number of trees and check AUC

```
# Train final tuned model with new nrounds value
boost_out2 = xgb.train(params = xgb2_params_tuned,
                       dtrain,
                       nrounds= cv_tuned$best_iteration,
                       verbose=0)

# Make predictions on hold-out test set
guess_xgb2 = predict(boost_out2, validation_expanded)

# Check AUC
auc(validation_y, guess_xgb2)
```

```
## Area under the curve: 0.7215
```

xgb3: Second Model with variables based on variable importance plot of variables used for xgb2 model

Preparing features and processing data and labels

```
# Choose features for model
features2 = c("DEP_DEL15", "MONTH", "DAY_OF_WEEK",
             "AIRLINE_ID", "CRS_DEP_TIME",
             "AWND", "PRCP", "SNOW",
             "HOLIDAY_PROX",
             "PCT_DEL_1", "NUM_BEF_1",
             "PCT_DEL_2", "NUM_BEF_2",
             "PCT_DEL_3", "NUM_DEL_3", "NUM_BEF_3",
             "PCT_DEL_4", "NUM_DEL_4", "NUM_BEF_4")

# Select corresponding data from train and validation sets
xgb3_train_df = train_df[, features2]
xgb3_validation_df = validation_df[, features2]

# Use One Hot Encoding to convert any categorical variables besides response into numeric
```

```
train_expanded2 = sparse.model.matrix(DEP_DEL15 ~ .-1, data=xgb3_train_df)
validation_expanded2 = sparse.model.matrix(DEP_DEL15 ~ .-1, data=xgb3_validation_df)

# Create labels for response variable
train_y2 = (train_df$DEP_DEL15 == 1)
validation_y2 = (validation_df$DEP_DEL15 == 1)

# Convert train and validation data tables with xgb.DMatrix into matrix data structure used by XGBoost
dtrain2 = xgb.DMatrix(data=train_expanded2, label=train_y2)
dvalidation2 = xgb.DMatrix(data=validation_expanded2, label=validation_y2)
wgt2 = sum(train_y == FALSE) / sum(train_y == TRUE)
```

Train baseline, untuned xgb3 model with default parameters to examine how AUC is improved by tuning

```
# Default parameters
params2 = list(
         booster = "gbtree",
         objective = "binary:logistic",
         eta = 0.3,
         gamma = 0,
         max_depth = 6,
         min_child_weight = 1,
         subsample = 1,
         colsample_bytree=1,
         scale_pos_weight=wgt2,    # USe earlier-calculated wgt (in effect balances the classes)
         eval_metric = "auc"
)

# Train untuned xgb3 model with above default parameters
boost_out3_untuned = xgb.train(params = params2, dtrain2, nrounds = 100, verbose=0)

# Make predictions
guess_xgb3_untuned = predict(boost_out3_untuned, validation_expanded2)

# Check AUC performance
auc(validation_y2, guess_xgb3_untuned)
```

```
## Area under the curve: 0.6966
```

Tuning xgb3 model

Step 1: Tune optimal number of trees for eta = 0.1 with CV

```
# Use 10-fold cv to tune opt tree num with eta = 0.1, and defaults for other param.
cv2 = xgb.cv(list(objective = 'binary:logistic',
             eta = 0.1,
             scale_pos_weight=wgt2,
             eval_metric = "auc"), dtrain2,
             watchlist = list(train =dtrain2, test=dvalidation2),
             early_stopping_rounds = 10,     # Stops boosting after 10 rounds of no CV score improv.
```

```
             stratified = TRUE, nfold = 10, # Strat. CV to keep train.data class imbal. in each fold
             metrics = list("auc"),
             maximize = TRUE,
             nrounds=200,    # Incr. nrounds since lower eta = more iter. to each opt. num trees
             verbose=0)
cv2$best_iteration           # Optimal number of trees
```

```
## [1] 29
```

Step 2: Tune max_depth and min_child_weight to control tree complexity

```
# Use random grid search with cross-validation in caret package
## Set up grid for tuning max_depth and min_child_weight
xgb3_grid = expand.grid(
             nrounds = cv2$best_iteration,      # Use opt tree num from previous tuning
             max_depth = c(1, 2, 3, 4, 5), # Size of each new decision tree
             eta = 0.1,
             gamma = 0,
             colsample_bytree = 1,
             min_child_weight = c(5, 6, 7, 8), # Min num of obs req in child node after split,
             subsample = 1
)

# Fixes and stratifies folds to preserve class imbalance and ensure reproducibility
folds2 = createFolds(train_df$DEP_DEL15)

# Pack training control parameters
xgb3_trcontrol = trainControl(
                 method = "cv",
                 number = 10,
                 search = "random",     # Random search to save time
                 verboseIter = FALSE,
                 returnData = FALSE,
                 returnResamp = "all",
                 classProbs = TRUE,
                 summaryFunction = twoClassSummary,
                 allowParallel = TRUE,
                 index = folds2)         # stratifed folds to preserve class imbalance

# Train models for each parameter combination in the grid and choose one with lowest error
xgb3_tune1 = train(
             x = dtrain2,
             y = factor(train_df$DEP_DEL15, labels = c("NotDelayed","Delayed")),
             trControl = xgb3_trcontrol,
             tuneGrid = xgb3_grid,
             method = "xgbTree",
             metric = "ROC",
             scale_pos_weight = wgt2
)

# Report the tuned min_child_weight and max_depth values
xgb3_tune1$bestTune
```

```
##   nrounds max_depth eta gamma colsample_bytree min_child_weight subsample
## 9      29         3 0.1     0                1                5         1
```

Step 3: Tune sampling parameters: subsample and colsample_bytree, and recalibrate optimal number of trees using early_stopping_rounds with nrounds (This step adds randomness to make training robust to noise, to prevent overfitting)

```
# Set up tuning grid with updated parameters, and ranges for tuning targets
xgb3_grid2 = expand.grid(
             nrounds = 100,       # Need to recalibrate tree num grown, (using early_stopping_rounds)
             max_depth = xgb3_tune1$bestTune$max_depth,
             eta = 0.1,
             gamma = 0,
             colsample_bytree = c(0.5, 0.6, 0.7, 0.8, 0.9), # Fract. of features samp. each iter.
             min_child_weight = xgb3_tune1$bestTune$min_child_weight,
             subsample = c(0.3, 0.4, 0.5, 0.6, 0.7, 0.8)     # Fract.of tr. set rows samp. each iter.
)

# Pack training control parameters
xgb3_trcontrol2 = trainControl(method = "cv",
                               number = 10,
                               search = "random",
                               verboseIter = FALSE,
                               returnData = FALSE,
                               returnResamp = "all",
                               classProbs = TRUE,
                               summaryFunction = twoClassSummary,
                               allowParallel = TRUE,
                               index = folds2)         # Same folds as before for reproducibility

# Train models for each parameter combination in the grid and choose one with lowest error
xgb3_tune2 = train(x = dtrain2,
                   y = factor(train_df$DEP_DEL15,
                              labels = c("NotDelayed","Delayed")),
                   trControl = xgb3_trcontrol2,
                   tuneGrid = xgb3_grid2,
                   method = "xgbTree",
                   metric = "ROC",
                   early_stopping_rounds = 10,    # Stops at new opt. tree num. for this param. set
                   scale_pos_weight = wgt2,
                   watchlist = list(test = dvalidation2),
                   verbose = 0
)

# Report the tuned subsampling and colsample_bytree values
xgb3_tune2$bestTune
```

```
##    nrounds max_depth eta gamma colsample_bytree min_child_weight subsample
## 30     100         3 0.1     0              0.9                5       0.8
```

Step 4: Tune Gamma, and use early_stopping_rounds to stop boosting at optimal number of trees for this parameter set

```
# Set up tuning grid with updated parameters, and ranges for tuning targets
xgb3_grid3 = expand.grid(
             nrounds = 150,                              # Raised max boosting rounds just in case
             max_depth = xgb3_tune1$bestTune$max_depth,
             eta = 0.1,
             gamma = c(0, 0.01, 0.02, 0.03, 0.04),
             colsample_bytree = xgb3_tune2$bestTune$colsample_bytree,
             min_child_weight = xgb3_tune1$bestTune$min_child_weight,
             subsample = xgb3_tune2$bestTune$subsample
)

# Pack training control parameters
xgb3_trcontrol3 = trainControl(method = "cv",
                               number = 10,
                               search = "random",
                               verboseIter = FALSE,
                               returnData = FALSE,
                               returnResamp = "all",
                               classProbs = TRUE,
                               summaryFunction = twoClassSummary,
                               allowParallel = TRUE,
                               index = folds2)         # same folds as before for reproducibility

# Train models for each parameter combination in the grid and choose one with lowest error
xgb3_tune3 = train(x = dtrain2,
                   y = factor(train_df$DEP_DEL15,
                              labels = c("NotDelayed","Delayed")),
                   trControl = xgb3_trcontrol3,
                   tuneGrid = xgb3_grid3,
                   method = "xgbTree",
                   metric = "ROC",
                   early_stopping_rounds = 10,       # Stops at new optimal number of trees
                   scale_pos_weight = wgt2,
                   watchlist = list(test=dvalidation2),
                   verbose = 0
)

# Report the tuned gamma value
xgb3_tune3$bestTune
```

```
##   nrounds max_depth eta gamma colsample_bytree min_child_weight subsample
## 4     150         3 0.1  0.03              0.9                5       0.8
```

Step 5: Now tune learning rate (eta) lower and nrounds upwards (because optimal tree number increases)

```
# Set up tuning grid with updated parameters, and ranges for tuning targets
xgb3_grid4 = expand.grid(
             nrounds = c(50, 75, 90, 100, 200, 300, 400),  # nrounds must inc as eta decreases
```

```
            max_depth = xgb3_tune1$bestTune$max_depth,
            eta = c(0.05, 0.07, 0.08, 0.09, 0.1),
            gamma = xgb3_tune3$bestTune$gamma,
            colsample_bytree = xgb3_tune2$bestTune$colsample_bytree,
            min_child_weight = xgb3_tune1$bestTune$min_child_weight,
            subsample = xgb3_tune2$bestTune$subsample
)

# Pack training control parameters
xgb3_trcontrol4 = trainControl(method = "cv",
                               number = 10,
                               search = "random",
                               verboseIter = FALSE,
                               returnData = FALSE,
                               returnResamp = "all",
                               classProbs = TRUE,
                               summaryFunction = twoClassSummary,
                               allowParallel = TRUE,
                               index = folds2)      # same folds as before for reproducibility

# Train models for each parameter combination in the grid and choose one with lowest error
xgb3_tune4 = train(x = dtrain2,
                   y = factor(train_df$DEP_DEL15,
                              labels = c("NotDelayed","Delayed")),
                   trControl = xgb3_trcontrol4,
                   tuneGrid = xgb3_grid4,
                   method = "xgbTree",
                   metric = "ROC",
                   scale_pos_weight = wgt2,
                   early_stopping_rounds = 10,
                   watchlist = list(test = dvalidation2),
                   verbose = 0
)

# Report the tuned eta value
xgb3_tune4$bestTune
```

```
##    nrounds max_depth eta gamma colsample_bytree min_child_weight subsample
## 29      50         3 0.1  0.03              0.9                5       0.8
```

Step 6: Fine-tune the optimal number of trees with updated parameters using CV (relying on early_stopping_rounds).

```
# Make list of Tuned parameters as input for xgb.cv()
xgb3_params_tuned = list(
         booster = "gbtree",
         objective = "binary:logistic",
         eta = xgb3_tune4$bestTune$eta,
         gamma = xgb3_tune4$bestTune$gamma,
         max_depth = xgb3_tune4$bestTune$max_depth,
         min_child_weight = xgb3_tune4$bestTune$min_child_weight,
         subsample = xgb3_tune4$bestTune$subsample,
```

```
          colsample_bytree=xgb3_tune4$bestTune$colsample_bytree,
          scale_pos_weight=wgt2,
          eval_metric = "auc"
)

# Use stratified 10-fold cross-validation to retune optimal number of trees
cv_tuned2 = xgb.cv(params = xgb3_params_tuned,
              data = dtrain2,
              early_stopping_rounds = 10,   # Boost. ends if CV error not improv. aft. 10 rds
              watchlist = list(test = dvalidation2),
              stratified = TRUE,
              nfold = 10,
              metrics = list("auc"),
              maximize = TRUE,
              nrounds = 800,    # Pick large max tree num and hope opt tree num is smaller
              verbose=0)

cv_tuned2$best_iteration    # New Optimal number of trees
```

```
## [1] 93
```

Step 7: Train final tuned xgb3 model with nrounds = optimal number of trees and check AUC

```
# Train final tuned model with new nrounds value
boost_out3 = xgb.train(params = xgb3_params_tuned,
                       dtrain2,
                       nrounds= cv_tuned2$best_iteration,
                       verbose=0)

# Make predictions on validation set
guess_xgb3 = predict(boost_out3, validation_expanded2)

# Check AUC
auc(validation_y2, guess_xgb3)
```

```
## Area under the curve: 0.7147
```

Evaluation of best tuned model (out of xgb2 and xgb3) based on performance on validation set.

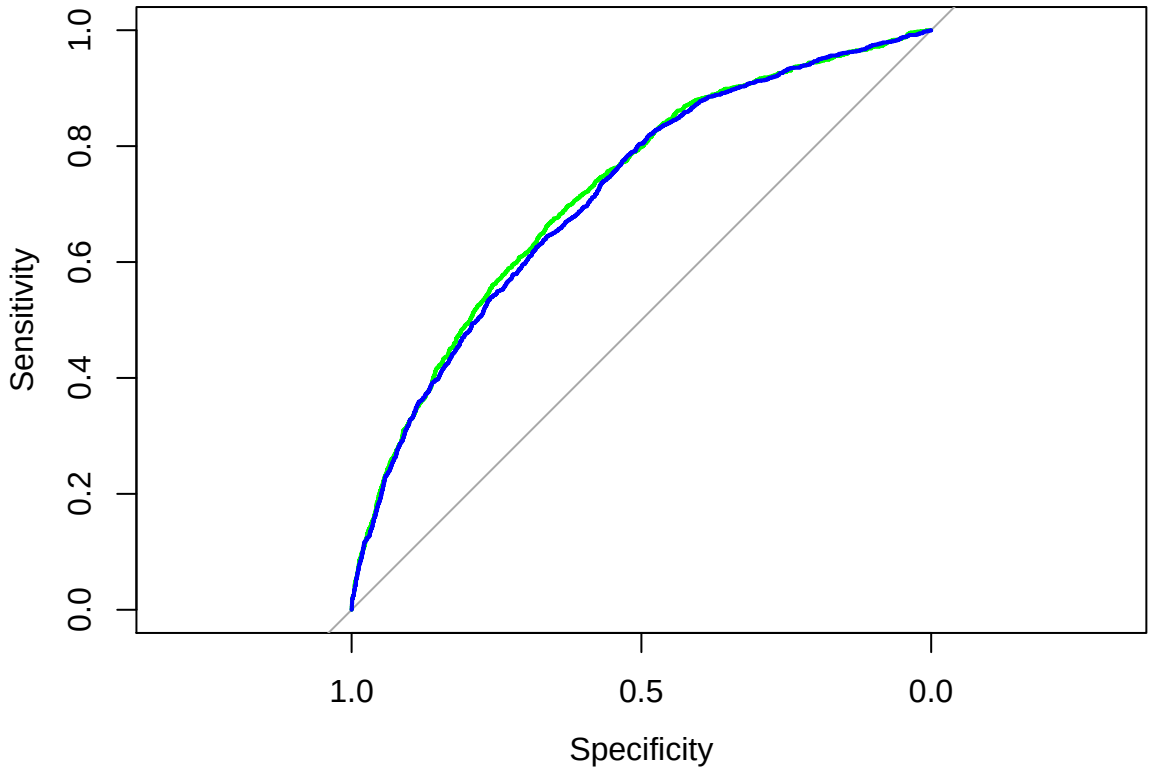
ROC Curves and AUC on Validation Set

```
# Use AUC of ROC curve to determine which of two tuned XGBoost models is better
roc(validation_y, guess_xgb2, plot=TRUE, col="green")        # This model has higher performance
```

```
##
## Call:
## roc.default(response = validation_y, predictor = guess_xgb2,     plot = TRUE, col = "green")
##
## Data: guess_xgb2 in 8855 controls (validation_y FALSE) < 1328 cases (validation_y TRUE).
```

```
## Area under the curve: 0.7215
```

```
roc(validation_y2, guess_xgb3, plot=TRUE, add=TRUE, col="blue")
```




```
##
## Call:
## roc.default(response = validation_y2, predictor = guess_xgb3,     plot = TRUE, add = TRUE, col = "blu
##
## Data: guess_xgb3 in 8855 controls (validation_y2 FALSE) < 1328 cases (validation_y2 TRUE).
## Area under the curve: 0.7147
```

Prepare hold-out test set (Flights from 2017 We have Departure Info On)

```
### Subset 2017 visible flights by departing flights and rename as test set
test_df = visible_df[visible_df$ORIGIN == "PIT", ]
```

Make sure test set and train set have same factored features with same levels

```
# Check NA values for DEP_DEL15 (response feature) in test set
sum(is.na(test_df$DEP_DEL15))/nrow(test_df)  # 1.286% of data  (can safely drop inst.)
```

```
## [1] 0.0128587
```

```
# Drop NA values for DEP_DEL15 in test set because we dropped NA values for train set too
test_df = test_df[complete.cases(test_df$DEP_DEL15), ]

# Factor test set's features to match features that were factored in full training set
test_df$AIRLINE_ID = factor(test_df$AIRLINE_ID)
test_df$MONTH = factor(test_df$MONTH)
test_df$DEP_DEL15 = factor(test_df$DEP_DEL15)

# Set the levels of factor features to be identical for full training set and test set
levels(test_df$MONTH) = levels(train_df$MONTH)
levels(test_df$AIRLINE_ID) = levels(train_df$AIRLINE_ID)
levels(test_df$DEP_DEL15) = levels(train_df$DEP_DEL15)
```

Create DATETIME column in test data set

```
# Extract departure times into string format for all rows of data
crs_dep_string = as.character(test_df$CRS_DEP_TIME)
for(i in 1:length(crs_dep_string)){
  if(nchar(crs_dep_string[i]) < 4){
    crs_dep_string[i] = paste("0", crs_dep_string[i], sep="")
  }
  crs_dep_string[i] = paste(substr(crs_dep_string[i], 1, 2),
                            substr(crs_dep_string[i], 3, 4),
                            sep=":")
}

# Combine date and departure times into one string for each row in data
datetime_strings = paste(as.character(test_df$FL_DATE), crs_dep_string, sep = " ")

# Make new col of datetime str conv into POSIXCT format so fl. dep. datetimes can be subtracted
test_df$DATETIME = as.POSIXct(datetime_strings, format = "%Y-%m-%d %H:%M")
```

Add to test data set features for holiday proximity and proximity to each individual holiday

```
# Add features of curr. flight's date's prox to each holiday in terms of days before and after
for(i in 1:nrow(test_df)){
  diffs = as.numeric(test_df$DATE[i] - NYE_dates)
  prox = diffs[which(abs(diffs) == min(abs(diffs)))]
  if(length(prox) != 1){
    prox = max(prox)
  }
  test_df$NYE_PROX[i] = prox

  diffs = as.numeric(test_df$DATE[i] - MLK_dates)
  prox = diffs[which(abs(diffs) == min(abs(diffs)))]
  if(length(prox) != 1){
    prox = max(prox)
  }
  test_df$MLK_PROX[i] = prox
```

```
diffs = as.numeric(test_df$DATE[i] - WAS_dates)
prox = diffs[which(abs(diffs) == min(abs(diffs)))]
if(length(prox) != 1){
  prox = max(prox)
}
test_df$WAS_PROX[i] = prox

diffs = as.numeric(test_df$DATE[i] - MEM_dates)
prox = diffs[which(abs(diffs) == min(abs(diffs)))]
if(length(prox) != 1){
  prox = max(prox)
}
test_df$MEM_PROX[i] = prox

diffs = as.numeric(test_df$DATE[i] - IND_dates)
prox = diffs[which(abs(diffs) == min(abs(diffs)))]
if(length(prox) != 1){
  prox = max(prox)
}
test_df$IND_PROX[i] = prox

diffs = as.numeric(test_df$DATE[i] - LAB_dates)
prox = diffs[which(abs(diffs) == min(abs(diffs)))]
if(length(prox) != 1){
  prox = max(prox)
}
test_df$LAB_PROX[i] = prox

diffs = as.numeric(test_df$DATE[i] - COL_dates)
prox = diffs[which(abs(diffs) == min(abs(diffs)))]
if(length(prox) != 1){
  prox = max(prox)
}
test_df$COL_PROX[i] = prox

diffs = as.numeric(test_df$DATE[i] - VET_dates)
prox = diffs[which(abs(diffs) == min(abs(diffs)))]
if(length(prox) != 1){
  prox = max(prox)
}
test_df$VET_PROX[i] = prox

diffs = as.numeric(test_df$DATE[i] - THX_dates)
prox = diffs[which(abs(diffs) == min(abs(diffs)))]
if(length(prox) != 1){
  prox = max(prox)
}
test_df$THX_PROX[i] = prox

diffs = as.numeric(test_df$DATE[i] - XMS_dates)
prox = diffs[which(abs(diffs) == min(abs(diffs)))]
if(length(prox) != 1){
  prox = max(prox)
```

```
  }
  test_df$XMS_PROX[i] = prox

  diffs = as.numeric(test_df$DATE[i] - us_holidays_df$Date)    # Calculate proximity to any US holiday
  prox = diffs[which(abs(diffs) == min(abs(diffs)))]
  if(length(prox) != 1){
    prox = max(prox)
  }
  test_df$HOLIDAY_PROX[i] = prox
}
```

Add to test set features for Number/Percentage of flight delays X hours or less prior to current flight and for Total number of flights departing X hours or less prior

```
# Create DATEIME feature in POSIX format for test set
datetime_strings = paste(as.character(test_df$FL_DATE), crs_dep_string, sep = " ")

# Add this new feature to test set
test_df$DATETIME = as.POSIXct(datetime_strings, format = "%Y-%m-%d %H:%M")

# Add to test data set Features for Percentage of flight Delays X hours prior to current flight
for(i in 1:nrow(test_df)) {           # For loop creates these features for each row in data set
  # later time - earlier time > 0
  hour_diff = difftime(test_df$DATETIME[i], test_df$DATETIME, units="hours") # Datetime difference

  is_depart_before = 0 < hour_diff & hour_diff < 1   # Cond.: flight dep. < 1hr prior to departure
  DEP_DEL15_before = test_df[is_depart_before, "DEP_DEL15"] # Bool. var: Fl. dep  <1hr bef. del.
  total = length(DEP_DEL15_before)   # tot. num flights dep. before flight in question
  delayed = sum(DEP_DEL15_before == 1)   # number of those flights that are delayed
  test_df[i, "PCT_DEL_1"] = ifelse(!is.na(total) & total > 0, delayed / total, 0)  # % of these delayed
  test_df[i, "NUM_DEL_1"] = ifelse(!is.na(delayed), delayed, 0)   # num of these flights delayed
  test_df[i, "NUM_BEF_1"] = total   # total number of flights departing 1 hour or less before

  is_depart_before = 0 < hour_diff & hour_diff < 2   #TRUE Cond. for if flight departing <2hr before
  DEP_DEL15_before = test_df[is_depart_before, "DEP_DEL15"]
  total = length(DEP_DEL15_before)
  delayed = sum(DEP_DEL15_before == 1)
  test_df[i, "PCT_DEL_2"] = ifelse(!is.na(total) & total > 0, delayed / total, 0)
  test_df[i, "NUM_DEL_2"] = ifelse(!is.na(delayed), delayed, 0)
  test_df[i, "NUM_BEF_2"] = total

  is_depart_before = 0 < hour_diff & hour_diff < 3         # Pattern continues
  DEP_DEL15_before = test_df[is_depart_before, "DEP_DEL15"]
  total = length(DEP_DEL15_before)
  delayed = sum(DEP_DEL15_before == 1)
  test_df[i, "PCT_DEL_3"] = ifelse(!is.na(total) & total > 0, delayed / total, 0)
  test_df[i, "NUM_DEL_3"] = ifelse(!is.na(delayed), delayed, 0)
  test_df[i, "NUM_BEF_3"] = total

  is_depart_before = 0 < hour_diff & hour_diff < 4
  DEP_DEL15_before = test_df[is_depart_before, "DEP_DEL15"]
```

```
  total = length(DEP_DEL15_before)
  delayed = sum(DEP_DEL15_before == 1)
  test_df[i, "PCT_DEL_4"] = ifelse(!is.na(total) & total > 0, delayed / total, 0)
  test_df[i, "NUM_DEL_4"] = ifelse(!is.na(delayed), delayed, 0)
  test_df[i, "NUM_BEF_4"] = total
}
```

Cleaning new features in test set

```
# Convert to numeric features for number of departing flights X hours or less before
test_df$NUM_BEF_1 = as.numeric(test_df$NUM_BEF_1)
test_df$NUM_BEF_2 = as.numeric(test_df$NUM_BEF_2)
test_df$NUM_BEF_3 = as.numeric(test_df$NUM_BEF_3)
test_df$NUM_BEF_4 = as.numeric(test_df$NUM_BEF_4)

# Set NA values to 0 for features for % departing flights delayed X hours or less before
test_df[is.na(test_df$PCT_DEL_1), "PCT_DEL_1"] = 0
test_df[is.na(test_df$PCT_DEL_2), "PCT_DEL_2"] = 0
test_df[is.na(test_df$PCT_DEL_3), "PCT_DEL_3"] = 0
test_df[is.na(test_df$PCT_DEL_4), "PCT_DEL_4"] = 0
```

Testing best tuned model (xgb2) on Hold-Out Test Set

```
### Subset train and test sets by features of xgb2 model
xgb_train_df_final = train_df[, features]    # Same features as xgb2 model
xgb_test_df_final = test_df[, features]

# Convert via One-Hot-Encoding test and train sets minus response feature into numeric matrix
train_expanded_final = sparse.model.matrix(DEP_DEL15 ~ .-1, data=xgb_train_df_final)
test_expanded_final = sparse.model.matrix(DEP_DEL15 ~ .-1, data=xgb_test_df_final)

# Set labels for test and train sets
train_y_final = (train_df$DEP_DEL15 == 1)
test_y_final = (test_df$DEP_DEL15 == 1)

# Convert training set and test set matrices into xgb.DMatrix format for XGBoost
dtrain_full_final = xgb.DMatrix(data=train_expanded_final, label=train_y_final)
dtest_full_final = xgb.DMatrix(data=test_expanded_final, label=test_y_final)

# Train final model on training set
boost_xgb2_test  = xgb.train(params = xgb2_params_tuned,  # Using xgb2's final tuned parameters
                       dtrain_full_final,
                       nrounds= cv_tuned$best_iteration,  # tuned number of trees
                       verbose=0)

# Predict on hold-out test set using final model
guess_xgb2 = predict(boost_xgb2_test, test_expanded_final)

# Check AUC of ROC on hold-out test set to evaluate realistic performance
auc(test_y_final, guess_xgb2, col="green")
```

```
## Area under the curve: 0.699
```

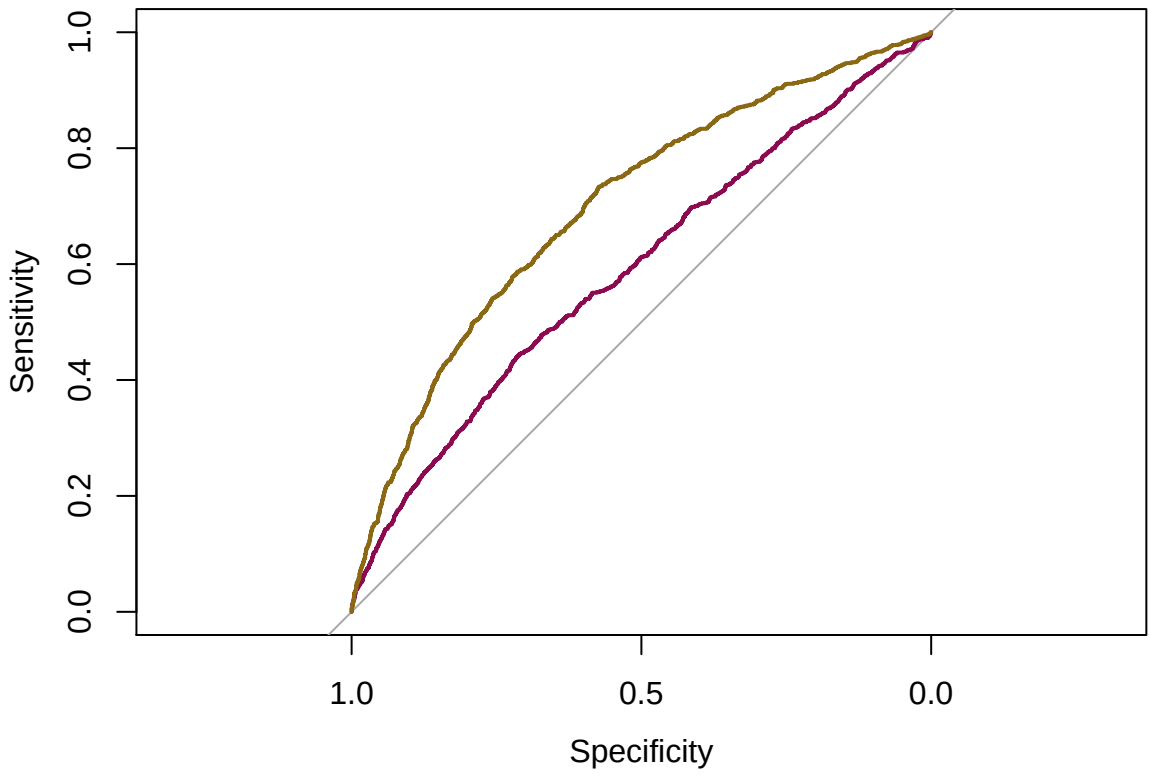
Compare untuned and tuned xgb2 models performance on Hold-Out Test Set using ROC curves

```
# Make predictions for untuned xgb2 model on test set
guess_xgb2_untuned_test = predict(boost_out2_untuned, test_expanded_final)

# Plot ROC curves for both
roc(test_df$DEP_DEL15, guess_xgb2_untuned_test, plot = TRUE, col='deeppink4')
```

```
##
## Call:
## roc.default(response = test_df$DEP_DEL15, predictor = guess_xgb2_untuned_test,     plot = TRUE, col =
##
## Data: guess_xgb2_untuned_test in 12273 controls (test_df$DEP_DEL15 0) < 1315 cases (test_df$DEP_DEL15
## Area under the curve: 0.5907
```

```
roc(test_df$DEP_DEL15, guess_xgb2, plot = TRUE, add=TRUE, col='goldenrod4')
```




```
##
## Call:
## roc.default(response = test_df$DEP_DEL15, predictor = guess_xgb2,     plot = TRUE, add = TRUE, col =
##
## Data: guess_xgb2 in 12273 controls (test_df$DEP_DEL15 0) < 1315 cases (test_df$DEP_DEL15 1).
## Area under the curve: 0.699
```

Testing Tuned xgb3 model on Hold-Out Test Set

```
### Subset train and test sets by features of xgb3 model
xgb_train_df_final2 = train_df[, features2]    # Same features as xgb3 model
xgb_test_df_final2 = test_df[, features2]

# Convert via One-Hot-Encoding test and train sets minus response feature into numeric matrix
train_expanded_final2 = sparse.model.matrix(DEP_DEL15 ~ .-1, data=xgb_train_df_final2)
test_expanded_final2 = sparse.model.matrix(DEP_DEL15 ~ .-1, data=xgb_test_df_final2)

# Set labels for test and train sets
train_y_final2 = (train_df$DEP_DEL15 == 1)
test_y_final2 = (test_df$DEP_DEL15 == 1)

# Convert training set and test set matrices into xgb.DMatrix format for XGBoost
dtrain_full_final2 = xgb.DMatrix(data=train_expanded_final2, label=train_y_final2)
dtest_full_final2 = xgb.DMatrix(data=test_expanded_final2, label=test_y_final2)

# Train final model on training set
boost_xgb3_tuned_test  = xgb.train(params = xgb3_params_tuned, # Tuned parameters from xgb3
                       dtrain_full_final2,
                       nrounds= cv_tuned2$best_iteration,     # Tuned number of trees
                       verbose=0)

# Predict on hold-out test set using final model
guess_xgb3 = predict(boost_xgb3_tuned_test, test_expanded_final2)

# Check AUC of ROC on hold-out test set to evaluate realistic performance
auc(test_y_final2, guess_xgb3, col="green")
```

```
## Area under the curve: 0.6946
```

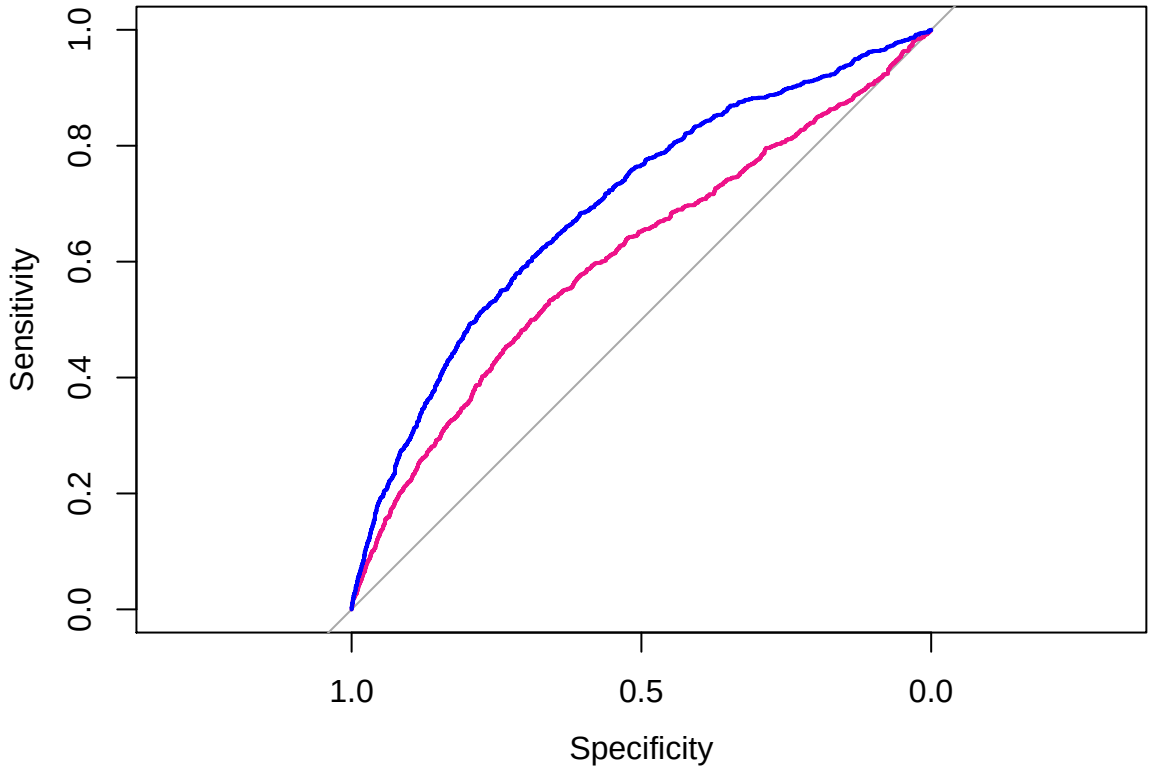
Compare untuned and tuned xgb3 models on Hold-Out Test Set using ROC Curves

```
# Make predictions for untuned xgb3 model on test set
guess_xgb3_untuned_test = predict(boost_out3_untuned, test_expanded_final2)

# Plot ROC curves for both
roc(test_df$DEP_DEL15, guess_xgb3_untuned_test, plot = TRUE, col='deeppink2')
```

```
##
## Call:
## roc.default(response = test_df$DEP_DEL15, predictor = guess_xgb3_untuned_test,     plot = TRUE, col =
##
## Data: guess_xgb3_untuned_test in 12273 controls (test_df$DEP_DEL15 0) < 1315 cases (test_df$DEP_DEL15
## Area under the curve: 0.6058
```

```
roc(test_df$DEP_DEL15, guess_xgb3, plot = TRUE, add=TRUE, col='blue')
```



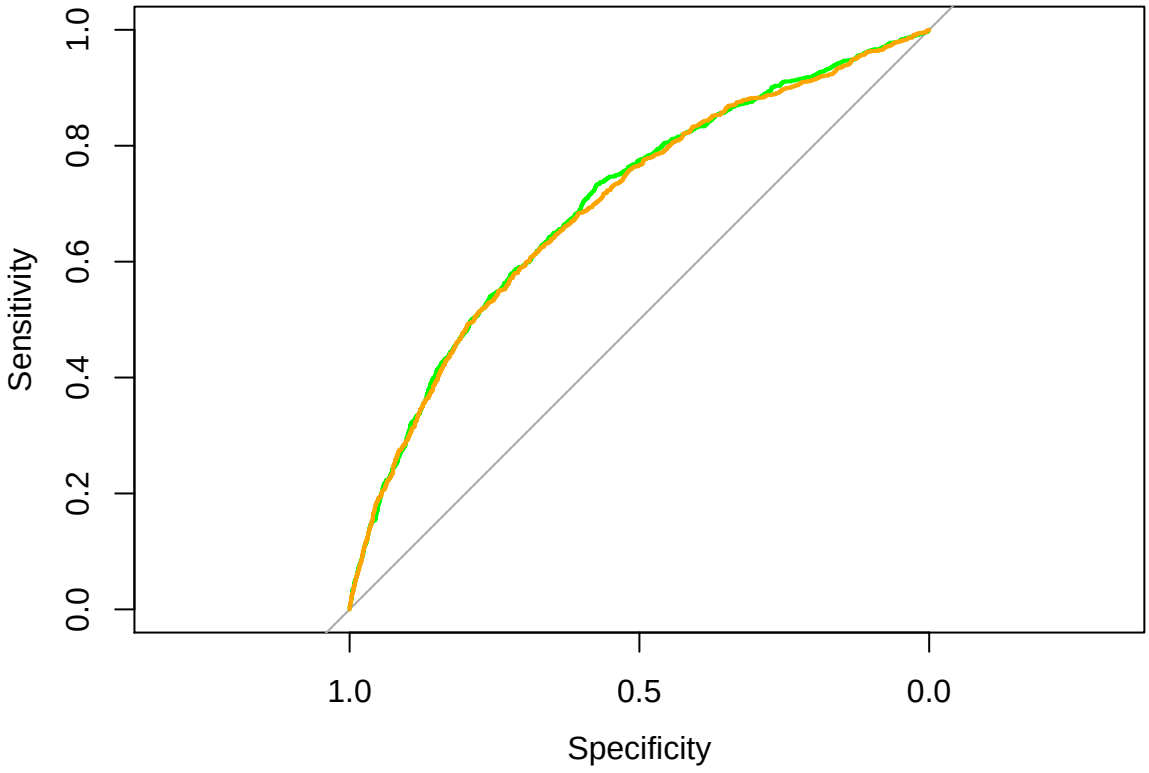

```
##
## Call:
## roc.default(response = test_df$DEP_DEL15, predictor = guess_xgb3,     plot = TRUE, add = TRUE, col =
##
## Data: guess_xgb3 in 12273 controls (test_df$DEP_DEL15 0) < 1315 cases (test_df$DEP_DEL15 1).
## Area under the curve: 0.6946
```

Finally, compare tuned xgb2 and xgb3 model performance on hold-out test data

```
# Plot ROC curves for tuned xgb2 and xgb3 models
roc(test_df$DEP_DEL15, guess_xgb2, plot = TRUE, col='green')
```

```
##
## Call:
## roc.default(response = test_df$DEP_DEL15, predictor = guess_xgb2,     plot = TRUE, col = "green")
##
## Data: guess_xgb2 in 12273 controls (test_df$DEP_DEL15 0) < 1315 cases (test_df$DEP_DEL15 1).
## Area under the curve: 0.699
```

```
roc(test_df$DEP_DEL15, guess_xgb3, plot = TRUE, add=TRUE, col='orange')
```




```
##
## Call:
## roc.default(response = test_df$DEP_DEL15, predictor = guess_xgb3,     plot = TRUE, add = TRUE, col =
##
## Data: guess_xgb3 in 12273 controls (test_df$DEP_DEL15 0) < 1315 cases (test_df$DEP_DEL15 1).
## Area under the curve: 0.6946
```