**XGBoost Tuning Methodology and Background Gordon Wang**

**What is this project?**

Before tuning methodology explanation, here is some brief context/background. The project’s objective was to predict flight departure delays at Pittsburgh International Airport only using information available before departure. Specifically, given flight data from 2015 and 2016, my team needed to predict for 2017 departures the DEP\_DEL15 variable, a binary variable that indicates whether a flight was delayed at least 15 minutes. Predictions were evaluated based on AUC.

For each day in 2017, we received data on flights only up to a random point in time, and then we needed to predict delays for departing flights that occurred after that cutoff. So, we had visible 2017 flights for which we knew if they were delayed, and 2017 flights we needed to guess on.

**Summary of Overall Project Procedure**

2015-2016 flight data was used as training set and 2017 visible flight data was used as test set. We further split that training set into 80% training set and 20% validation set. We built predictive models on that latter training set and validated and tuned them with the validation set. Then, we tested those models on our hold-out 2017 visible flight data test set and evaluated them based on AUC. Finally, we chose the best ensemble of our models to make predictions on flight delays for 2017 departing flights after the cutoff.

**Important Note About Severe Class Imbalance**

The data set we worked with showed severe class imbalance; over 80% of departing flights aren’t delayed. This can cause problems. To ensure enough presence of minority class observations in training and validation sets, I stratified the training set and validation set according to class distribution of the original training set. Also, later, I used stratified cross-validation instead of regular cross-validation to preserve class imbalance in each fold.

Also, because this affects model training, I artificially balanced class weights by using XGBoost’s scale\_pos\_weight argument to increase the minority class’s weight, since we only care about evaluating based on AUC rather than predicting the right probability.

**What is XGBoost?**

XGBoost or Extreme Gradient Boosting is an ensemble machine learning algorithm used to combine many weak estimators into a powerful estimator. It builds an additive sequence of simple prediction functions (decision trees), where each function tries to do well on observations that previous functions did poorly on. Boosting uses shallow trees, which have low variance and high bias, but bias can be reduced because this method adjusts for previous error. Shallow trees ensure boosting is faster than deep tree methods like bagging or random forests. Thus, boosting is very powerful; however, it requires careful tuning to perform well.

**What are XGBoost’s parameters?**

In this project, using xgboost package in R, I built two models, each with a different set of features/variables. For each model, I tuned 6 hyper-parameters, the only ones R’s caret package offered as options for tuning. I’ll explain these tuning parameters and their purpose.

The main tuning parameter is total **number of trees to build** or number of boosting rounds/iterations in the additive sequence/model. Too many trees is the main cause of overfitting. However, too few trees will cause bad underfitting/classification. The optimal number of trees is very sensitive and depends on all the other parameters’ values. Thus, the number of trees needs to be re-tuned each time a parameter is tuned.

An important thing to note is that the nrounds parameter in R’s XGBoost isn’t the same as model’s optimal number of trees. Nrounds is the model’s max tree number while actual optimal tree number could be lower or higher. XGBoost provides a nice way to find the optimal tree number. Because of boosting’s additive nature, instead of fixing tree number to grow at the start, we test at each iteration to see if an additional tree improves the model. To do that, we use stratified 10-fold cross-validation with AUC as metric to measure performance at each iteration. If performance hasn’t improved for X rounds (where is X is set by early\_stopping\_rounds argument) boosting ends and iteration number X rounds ago with best performance is optimal tree number. We set nrounds as a large number to ideally obtain the optimal tree number before reaching nrounds iterations.

Then, we have two tree-specific parameters that directly control model complexity: **max\_depth** and **min\_child\_weight**. Max\_depth controls how deep each iteration’s additional tree can grow. Deeper trees can model more complex relationships, but eventually more splits cause over overfitting. Min\_child\_weight is minimum weight required to create a new node in a tree. A smaller value allows boosting to create children nodes more often, leading to overfitting.

Also, we have two sampling parameters: **subsample** and **colsample\_bytree**. In essence, these parameters control data sampling done at each boosting iteration. Subsample is fraction of training data rows randomly selected in each iteration. Colsample\_bytree is fraction of the features/variables in subsample that the algorithm randomly selects for splitting new trees in each iteration. Thus, these parameters allow for tree building on slightly variant data at each iteration, preventing likelihood of overfitting to one particular feature/sample.

Another important parameter is **eta** or learning rate. This one sets amount of shrinkage for weights given to newly added trees after each boosting iteration. This controls how much correction we make to the model at each iteration. When eta is lower, boosting learns at slower rate due to the lower weight of each additional tree, which means the model’s optimal tree number is correspondingly higher since it takes many more boosting iterations to reach the optimum.

Finally, the last parameter is **gamma**. Gamma is a regularization parameter that controls overfitting by specifying the minimum reduction in the binary logistic loss function required to make a split in the tree node. The higher the gamma value, the fewer the number of splits that occur when growing a tree, meaning less complexity in the model.

**What were my tuning steps?**

Prior to tuning, for both of my two models, I trained a baseline model using default parameters to check the initial validation set AUC performance, for comparison to validation set AUC performance after tuning.

My first tuning step was to lower eta, learning rate, to 0.1, and then correspondingly raise nrounds to 200. I used stratified 10-fold cross-validation and early\_stopping\_rounds = 10 to determine the optimal number of trees and set that as my new nrounds value. Second, I used random search with cross-validation to tune tree complexity parameters max\_depth and min\_child\_weight. I set a vector of several possible values for each parameters and caret package’s random search trains models for each random combination of these values and finds the optimal values that maximizes cross-validated AUC. Third, I tune the tree sampling parameters subsample and colsample\_bytree; fourth, I tune gamma; fifth, I tune eta even lower and also tune nrounds higher (to support the possible decrease in eta). For each of these 3 steps, I’m using cross-validation/random search to tune, and also retuning optimal tree number with help of early\_stopping\_rounds each time I update a parameter. Finally, I retune for the last time optimal tree number with cross-validation and early\_stopping\_rounds, and train the final model with that optimal tree number as new nrounds value and all of the other tuned parameter values.