**Methodology & Results**

After feature engineering the next step is to develop a learner using the training data. As the training data is very complex we need to use complex learners to achieve high accuracy. We chose to use ensemble learning methods. The methods we chose to implement are *‘Random Forest’* and *‘Gradient Boosting’.*

*Random Forest:*

Random Forest is a combination of Bootstrapping and Random attribute selection. This methods basically creates decision trees using random attributes at different levels of the learning process. And now because of bootstrapping we now have more data and in machine learning more data is always good.

The reason for selecting this method is to avoid over fitting which was the problem with decision trees. To avoid over fitting instead of taking all the available attributes into consideration at any given point, random forest randomly selects a set of attributes. The decrease in accuracy of training data is very minute compared to decision trees but the increase in accuracy of testing data will be much higher compared to decision trees.

We now implement this method using R. We use a package called ‘randomForest’ to implement our methodology. Implementing this is pretty straight forward. We have a function called randomForest()to implement random forest. We also implement cross validation to get MSE over the training data.

cvresults <- rfcv(traindata, traindata$status\_group, cv.fold = 10)

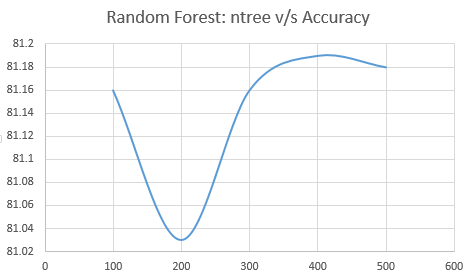
rf\_model<-randomForest(traindata$status\_group~.,data = traindata, ntree = 100)

We then use the predict function to predict the classes for testdata

predtemp <- predict(rf\_model, testdata)

For the initial testing we fixed ntree to ‘100’ and used feature engineering to get the best attribute set. We now report accuracies for both training and testing data with varied ntree values. The training data set is divided into two parts: 90% for training the data and 10% for the validation data set.

|  |  |  |
| --- | --- | --- |
| **ntree =** | **Validation Data Accuracy** | **Test Data Accuracy** |
| 100 | 81.00 | 81.16 |
| 200 | 81.15 | 81.03 |
| 300 | 81.32 | 81.16 |
| 400 | 81.38 | 81.19 |
| 500 | 81.27 | 81.18 |

**

*Fig. : Random Forest Accuracy Curve*

*Gradient Boosting:*

Random Forest method gave a best accuracy of 81.19% but generating the model took a lot of time(~20 minutes) and we also wanted to improve our accuracy. So, we choose a new technique called Gradient Boosting which is an extension to the traditional boosting technique by combining it with gradient descent algorithm.

The general concept in boosting is to introduce a weak learner at every stage of learning. In the normal adaboost we use the weights of the data weights to introduce a weak learner and compensate for weak learner. The method in gradient boosting is different where we use the negative gradients of each class to overcome the shortcomings.

Similar to Random Forest we divide the dataset into training set and validation set with 90:10 ratio. We present the accuracies of our model on validation set. To use the training data to train the model, gradient boosting package in R requires that the data is in xgb.Dmatrix format. The following commands converts the data set to xgb.Dmatrix format.

newtrain <- xgb.DMatrix(train.xgb, label = TL)

The above line only works if train.xgb is in either dense or sparse matrix format.

Gradient Boosting takes a lot of parameters to model the training dataset.

param.xgb <- list(objective = "multi:softmax",eval\_metric = "merror",num\_class = 3, booster = "gbtree",eta = 0.2,subsample = 0.7,colsample\_bytree = 0.4, max\_depth = 14)

Cross validation is performed using the following line

results <- xgb.cv(params = param.xgb, newtrain, nrounds = 200, nfold = 10, early.stop.round = 20,maximize = FALSE, print.every.n = 10)

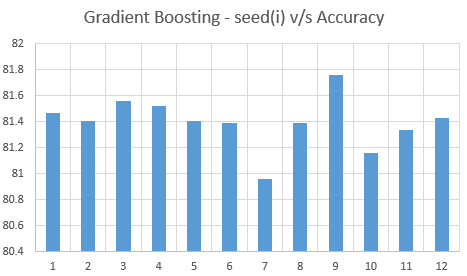
The main parameters that affect the accuracy are ‘eta’ and ‘maxdepth’. After numerous experiments the best value for eta is 0.2 and the best value of max\_depth = 14. Now we train the data using the following line.

model.xgb <- xgb.train(data = newtrain, param.xgb, nrounds = 200, watchlist = list(valid = newvalid, train = newtrain), nfold = 10, early.stop.round = 20, print.every.n = 10,maximize = FALSE,save\_name = "model.xgb")

The above line performs the gradient boosting for ‘200’ rounds. The drawback for performing these many rounds is that the model may overfit. So, we need to stop training once the accuracy on validation set starts increasing. Thanks to the very customizable xgb.train() function which outputs the best round we can easily avoid overfitting. One interesting observation we found is that setting the seed before training affected the model. We now present the results for different seed values for their best rounds.

|  |  |  |  |
| --- | --- | --- | --- |
| **set.seed(i);**  **i =** | **No. of rounds (Best Iteration)** | **Validation Set Accuracy** | **Testing Set Accuracy** |
| 1 | 46 | 81.86 | 81.47 |
| 2 | 42 | 81.83 | 81.41 |
| 3 | 51 | 81.71 | 81.56 |
| 4 | 31 | 81.73 | 81.52 |
| **5** | **56** | **82.52** | **81.41** |
| 6 | 35 | 81.81 | 81.39 |
| 7 | 28 | 81.85 | 80.96 |
| 8 | 28 | 81.75 | 81.39 |
| **9** | **38** | **81.61** | **81.76** |
| 10 | 38 | 81.71 | 81.16 |
| 11 | 77 | 82.08 | 81.34 |
| 12 | 71 | 82.00 | 81.43 |

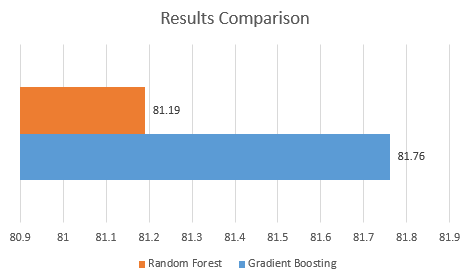
To fing the no. of rounds (best iteration) we first run the model for all seed values with nrounds = 200 in xgb.train() function. Then xgb.train() function outputs the best iteration based on the error rate for validation set. This best iteration value is used as the no. of rounds to model the data for testing data set.



*Fig. : Gradient Boosting Accuracy Curve*

**Comparison of two models:**

Now that we have the results of two of our best methods, we now compare the accuracies on testing set using the following bar graph.



*Fig. : Results Comparison*