Formatting Data:

I used one-hot-encoder for DNA-genetic sequence dataset. As I already demonstrated the whole process of formatting dataset in homework 1 and 2, I would not demonstrate each process in this document. These are code used for initial uploading and formatting of the dataset.

```
library(e1071)

library(caret)

library(caTools)

library(ROCR)

require(nnet)

require(randomForest)

require(parallel)

library(doParallel) ## for parallel computing

require(gbm)

require(ROCR)

require(xgboost)

require(Matrix)


cl <- makePSOCKcluster(5) ##for parallel processing

registerDoParallel(cl)


sdf<-read.csv('/home/2021/nyu/fall/gl1858/splice.csv')

my_cols <- c(names(sdf[2:61]))  ## take instance column names

mdf <-sdf[my_cols]

head(mdf)


output <- sdf[c(62)]

head(output)


onehotencoder <- function(df_orig) {
```

```
 df<-cbind(df_orig)
 df_clmtyp<-data.frame(clmtyp=sapply(df,class))
 df_col_typ<-data.frame(clmnm=colnames(df),clmtyp=df_clmtyp$clmtyp)
 for (rownm in 1:nrow(df_col_typ)) {
   if (df_col_typ[rownm,"clmtyp"]=="factor") {
     clmn_obj<-df[toString(df_col_typ[rownm,"clmnm"])]
     dummy_matx<-data.frame(model.matrix( ~.-1, data = clmn_obj))
     dummy_matx<-dummy_matx[,c(1,3:ncol(dummy_matx))]
     df[toString(df_col_typ[rownm,"clmnm"])]<-NULL
     df<-cbind(df,dummy_matx)
     df[toString(df_col_typ[rownm,"clmnm"])]<-NULL
   } }
 return(df)
}



cdf <- onehotencoder(mdf)
data <-cbind(cdf,output)
#head(data)

set.seed(43)
randomized=data[sample(1:nrow(data),nrow(data)),] # Shuffle
tridx=sample(1:nrow(data),0.7*nrow(data),replace=F) #Get indices for 70% of the total number of samples
trdf=randomized[tridx,] # Define training data set
tstdf=randomized[-tridx,] # Define testing data set
table(data$Class)/nrow(data) # Check if class distribution is similar
table(trdf$Class)/nrow(trdf)
table(tstdf$Class)/nrow(tstdf)
```

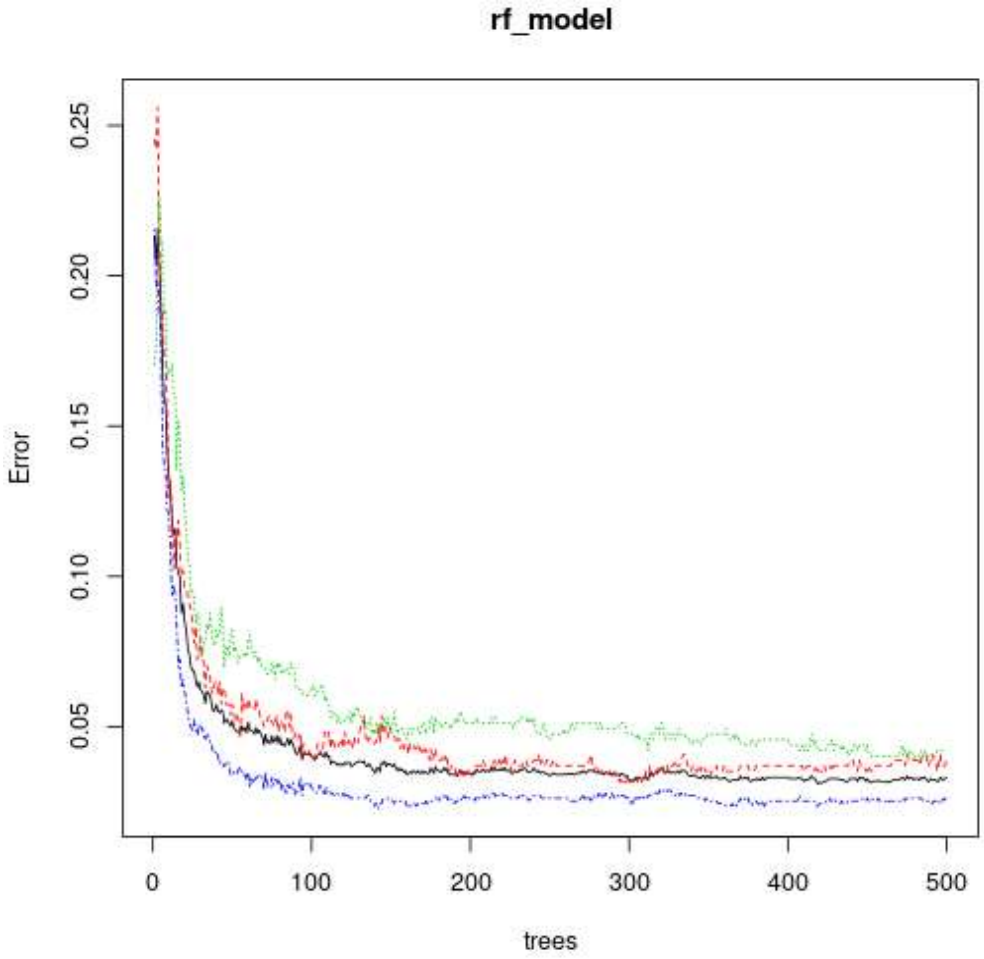
A. Random Forest

I choose random forest as I used decision tree for hw2 and I think it is important to see how pruned decision tree, which is random forest can be beneficial to balance out bias and variance problem. I choose 500 trees to run the model.

```
require(randomForest)

trdf_RF=trdf  #Take train dataset

trdf_RF$Class=as.factor(trdf_RF$Class)  #Take Y from the train dataset


rf_model=randomForest(Class~.,trdf_RF, ntree=500)

ry_pred = predict(rf_model, newdata = tstdf[-228])

cm = table(tstdf[, 228], ry_pred)

cfm<-confusionMatrix(cm)

cfm   ##to plot confusion matrix
```

```
Confusion Matrix and Statistics

    y_pred
      EI  IE   N
  EI 210   8   9
  IE   4 211   6
  N    4  10 495

Overall Statistics

               Accuracy : 0.9572
                 95% CI : (0.9423, 0.9691)
    No Information Rate : 0.5329
    P-Value [Acc > NIR] : <2e-16

                  Kappa : 0.9295

 Mcnemar's Test P-Value : 0.2351

Statistics by Class:

                     Class: EI Class: IE Class: N
Sensitivity             0.9633    0.9214   0.9706
Specificity             0.9770    0.9863   0.9687
Pos Pred Value          0.9251    0.9548   0.9725
Neg Pred Value          0.9890    0.9755   0.9665
Prevalence              0.2278    0.2393   0.5329
Detection Rate          0.2194    0.2205   0.5172
Detection Prevalence    0.2372    0.2309   0.5319
Balanced Accuracy       0.9701    0.9538   0.9696
```

plot(rf_model) ## To plot error graph by the tree number



We can find that the increase of the number of trees is decreasing the error rate. After increasing the tree number over 100, the error remains same.

B. Boosting

Initially - I tried with generalized boosting classification with GBM and as the boosting is iterative method, I demonstrated how train deviance decrease per each iteration.

I set the distribution as multinomial as it is multi classification data. To balance out bias and variance I set tree number as 500 and made the cross-validation fold =10.

```
gbm_model<-gbm(Class~., data=trdf,
distribution="multinomial" ,n.trees=500,shrinkage=0.01,interaction.depth=3, n.minobsinnode=10,
verbose=T, keep.data=T)
```

Iter	TrainDeviance	ValidDeviance	StepSize	Improve
1	1.0986	nan	0.0100	0.0326
2	1.0782	nan	0.0100	0.0311
3	1.0587	nan	0.0100	0.0303
4	1.0396	nan	0.0100	0.0295
5	1.0211	nan	0.0100	0.0286
6	1.0031	nan	0.0100	0.0279
7	0.9856	nan	0.0100	0.0271
8	0.9687	nan	0.0100	0.0261
9	0.9523	nan	0.0100	0.0256
10	0.9363	nan	0.0100	0.0245
20	0.7990	nan	0.0100	0.0189
40	0.6074	nan	0.0100	0.0122
60	0.4832	nan	0.0100	0.0083
80	0.3980	nan	0.0100	0.0057
100	0.3385	nan	0.0100	0.0036
120	0.2950	nan	0.0100	0.0026
140	0.2646	nan	0.0100	0.0023
160	0.2397	nan	0.0100	0.0019
180	0.2196	nan	0.0100	0.0016
200	0.2028	nan	0.0100	0.0010
220	0.1888	nan	0.0100	0.0008
240	0.1767	nan	0.0100	0.0007
260	0.1660	nan	0.0100	0.0006
280	0.1573	nan	0.0100	0.0004
300	0.1499	nan	0.0100	0.0006
320	0.1429	nan	0.0100	0.0002
340	0.1367	nan	0.0100	0.0002
360	0.1311	nan	0.0100	0.0002
380	0.1260	nan	0.0100	0.0003
400	0.1214	nan	0.0100	0.0003
420	0.1168	nan	0.0100	0.0001
440	0.1129	nan	0.0100	0.0002

As the iteration got closer to 500, I could see train deviance, and improvement number goes down accordingly.

```
gbm_predict<-predict(gbm_model, tstdf[,-c(228)], gbm_model$n.trees, type="response")
gbm_predicted<-round(gbm_predict)
labels = colnames(gbm_predicted)[apply(gbm_predicted, 1, which.max)]
#gbm_prediction<-prediction(labels,tstdf$Class)
p.gbm_predict=apply(gbm_predicted, 1, which.max)
result = data.frame(tstdf$Class, labels)
#print(result)
cm = confusionMatrix(tstdf$Class, as.factor(labels))
print(cm)
```

```
Confusion Matrix and Statistics

          Reference
Prediction  EI  IE   N
        EI 215   8   4
        IE   4 212   5
        N    7  13 489

Overall Statistics

               Accuracy : 0.9572
                 95% CI : (0.9423, 0.9691)
    No Information Rate : 0.5204
    P-Value [Acc > NIR] : <2e-16

                  Kappa : 0.9299

 Mcnemar's Test P-Value : 0.1268

Statistics by Class:

                     Class: EI Class: IE Class: N
Sensitivity             0.9513    0.9099   0.9819
Specificity             0.9836    0.9876   0.9564
Pos Pred Value          0.9471    0.9593   0.9607
Neg Pred Value          0.9849    0.9715   0.9799
Prevalence              0.2362    0.2435   0.5204
Detection Rate          0.2247    0.2215   0.5110
Detection Prevalence    0.2372    0.2309   0.5319
Balanced Accuracy       0.9675    0.9487   0.9692
```

For next step – I used XGBoost model to reduce the overfitting. It is typically faster in execution speed and performs well in a prediction of classifications. I set up the max depth as 3 and n round up to 50.

```
##Xgboost

#To create matrix for Xgboost model

train_x = data.matrix(trdf[,-228])

train_y = trdf[,228]

test_x = data.matrix(tstdf[,-228])

test_y = tstdf[,228]

xgb_train = xgb.DMatrix(data=train_x, label=train_y)

xgb_test = xgb.DMatrix(data=test_x, label=test_y)

xgbc = xgboost(data=xgb_train, max.depth=3, nrounds=50)

print(xgbc)
```

```
> xgbc = xgboost(data=xgb_train, max.depth=3, nrounds=50)
[1]      train-rmse:1.394636
[2]      train-rmse:1.010855
[3]      train-rmse:0.749773
[4]      train-rmse:0.577606
[5]      train-rmse:0.466829
[6]      train-rmse:0.396866
[7]      train-rmse:0.355331
[8]      train-rmse:0.327898
[9]      train-rmse:0.314734
[10]     train-rmse:0.304898
[11]     train-rmse:0.298873
[12]     train-rmse:0.294441
[13]     train-rmse:0.290463
[14]     train-rmse:0.286555
[15]     train-rmse:0.284126
[16]     train-rmse:0.281708
[17]     train-rmse:0.279169
[18]     train-rmse:0.277204
[19]     train-rmse:0.275191
[20]     train-rmse:0.273085
[21]     train-rmse:0.271235
[22]     train-rmse:0.269615
[23]     train-rmse:0.268176
[24]     train-rmse:0.266544
[25]     train-rmse:0.264973
[26]     train-rmse:0.263553
[27]     train-rmse:0.262299
[28]     train-rmse:0.260401
[29]     train-rmse:0.259193
[30]     train-rmse:0.258592
[31]     train-rmse:0.257244
[32]     train-rmse:0.255941
[33]     train-rmse:0.254098
[34]     train-rmse:0.253050
[35]     train-rmse:0.251948
[36]     train-rmse:0.251398
[37]     train-rmse:0.250340
[38]     train-rmse:0.248851
[39]     train-rmse:0.247765
[40]     train-rmse:0.246507
[41]     train-rmse:0.245348
[42]     train-rmse:0.243856
[43]     train-rmse:0.242691
[44]     train-rmse:0.241453
[45]     train-rmse:0.241019
[46]     train-rmse:0.239942
```

```
> print(xgbc)
##### xgb.Booster
raw: 35.8 Kb
call:
  xgb.train(params = params, data = dtrain, nrounds = nrounds,
    watchlist = watchlist, verbose = verbose, print_every_n = print_every_n,
    early_stopping_rounds = early_stopping_rounds, maximize = maximize,
    save_period = save_period, save_name = save_name, xgb_model = xgb_model,
    callbacks = callbacks, max.depth = 3)
params (as set within xgb.train):
  max_depth = "3", validate_parameters = "1"
xgb.attributes:
  niter
callbacks:
  cb.print.evaluation(period = print_every_n)
  cb.evaluation.log()
# of features: 227
niter: 50
nfeatures : 227
evaluation_log:
    iter train_rmse
       1   1.394636
       2   1.010855
---
      49   0.237125
      50   0.236849
```

pred = predict(xgbc, xgb_test)

#print(pred)

pred[(pred>3)] = 3 ##as the model prediction gives probabilities we have to convert into factor type to ##run in a confusion matrix

pred_y = as.factor((levels(test_y))[round(pred)])

cm = confusionMatrix(test_y, pred_y)

print(cm)

```
Confusion Matrix and Statistics

          Reference
Prediction  EI  IE   N
        EI 207  19   1
        IE   6 212   3
        N    1  48 460

Overall Statistics

               Accuracy : 0.9185
                 95% CI : (0.8993, 0.935)
    No Information Rate : 0.4848
    P-Value [Acc > NIR] : < 2.2e-16

                  Kappa : 0.8689

 Mcnemar's Test P-Value : 4.515e-10

Statistics by Class:

                     Class: EI Class: IE Class: N
Sensitivity             0.9673    0.7599   0.9914
Specificity             0.9731    0.9867   0.9006
Pos Pred Value          0.9119    0.9593   0.9037
Neg Pred Value          0.9904    0.9090   0.9911
Prevalence              0.2236    0.2915   0.4848
Detection Rate          0.2163    0.2215   0.4807
Detection Prevalence    0.2372    0.2309   0.5319
Balanced Accuracy       0.9702    0.8733   0.9460
```

I was rather surprised to find out that XGBoost method had lower accuracy than Generalized-Boosting method. From my understanding – XGBoost method uses effective regularization method to negate bias and variance while training the gradient more efficiently by using parallelized computation. I am going to further explore later in this document by comparing AUC and Accuracy of these two model directly.

C. Stacking

As previously I ran random forest, Generalized Boosting classifier, and Xgbooster, I decided to use on Stacking techniques. I opted out Naïve Bayes model from this as I wanted to compare only three model that specifically intended to reduce Bias or Variance and then compared with Naïve Bayes that was used in homework 2.

The predictive table contained categorical values. I decided to convert them all as the numeric value first between 1 and 3 to make majority selection. I rounded up the mean value instead as it seemed more probabilistic approach when we getting mean value for each output of the three model.

```
stackeddf<-data.frame(actual=tstdf$Class,

rfpred=ry_pred, #random forest

gbmpred=as.factor(labels), ##gbm_pred

xgbpred=pred_y) ##xgb_pred
```

```
head(stackeddf)
```

	actual	rfpred	gbmpred	xgbpred
3116	N	N	N	N
1173	IE	IE	IE	IE
66	EI	EI	EI	IE
440	EI	EI	EI	EI
2887	N	N	N	IE
1513	IE	IE	IE	IE

```
tail(stackeddf)
```

	actual	rfpred	gbmpred	xgbpred
363	EI	EI	EI	EI
2837	N	N	N	N
2675	N	N	N	N
448	EI	EI	EI	EI
2510	N	N	N	N
377	EI	EI	EI	EI

I realized I need to convert them into numerical data in order to compare

```
data_new <- sapply(stackeddf, unclass)          # Convert categorical variables

#data_new
```

```
stacked_mean<-round(unlist(apply(data_new [,2 :4],1,mean))) #rounded so that it does not have to choose value between and it is more probabilistic.

data_new<-cbind(data_new,stacked=stacked_mean)

testing<-as.data.frame(data_new) #to convert it back into

(stbl<-table(testing$actual,testing$stacked))

(scfm<-caret::confusionMatrix(stbl))
```

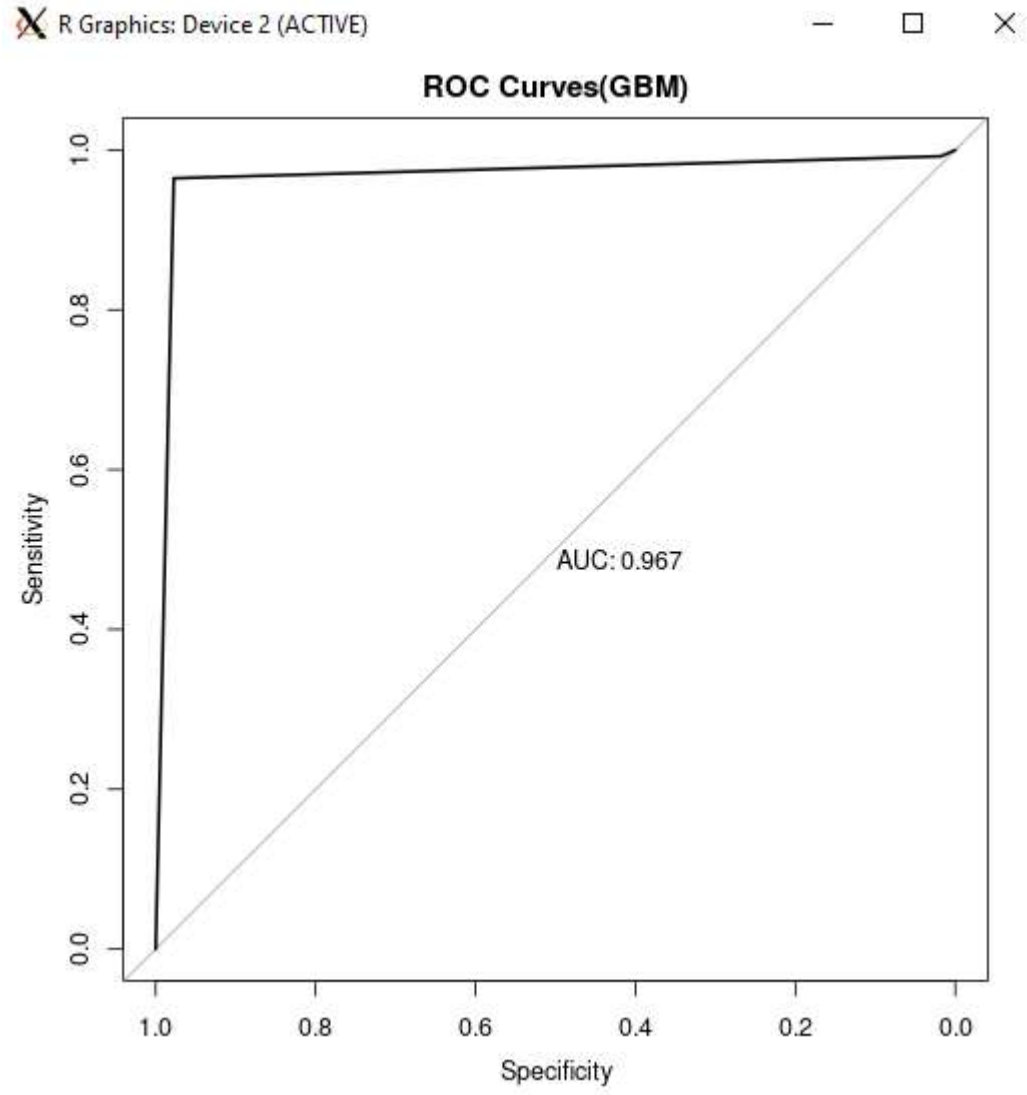
```
> (scfm<-caret::confusionMatrix(stbl))
Confusion Matrix and Statistics


     1   2   3
  1 213  12   2
  2   4 213   4
  3   2  16 491

Overall Statistics

               Accuracy : 0.9582
                 95% CI : (0.9435, 0.97)
    No Information Rate : 0.5193
    P-Value [Acc > NIR] : < 2e-16

                  Kappa : 0.9316

 Mcnemar's Test P-Value : 0.01069

Statistics by Class:

                     Class: 1 Class: 2 Class: 3
Sensitivity            0.9726   0.8838   0.9879
Specificity            0.9810   0.9888   0.9609
Pos Pred Value         0.9383   0.9638   0.9646
Neg Pred Value         0.9918   0.9620   0.9866
Prevalence             0.2288   0.2518   0.5193
Detection Rate         0.2226   0.2226   0.5131
Detection Prevalence   0.2372   0.2309   0.5319
Balanced Accuracy      0.9768   0.9363   0.9744
```

Comparative Analysis

To see how those models are balancing out bias and variance, I used AUC and Accuracy comparison by plotting multi-class AUC graph and comparison table.

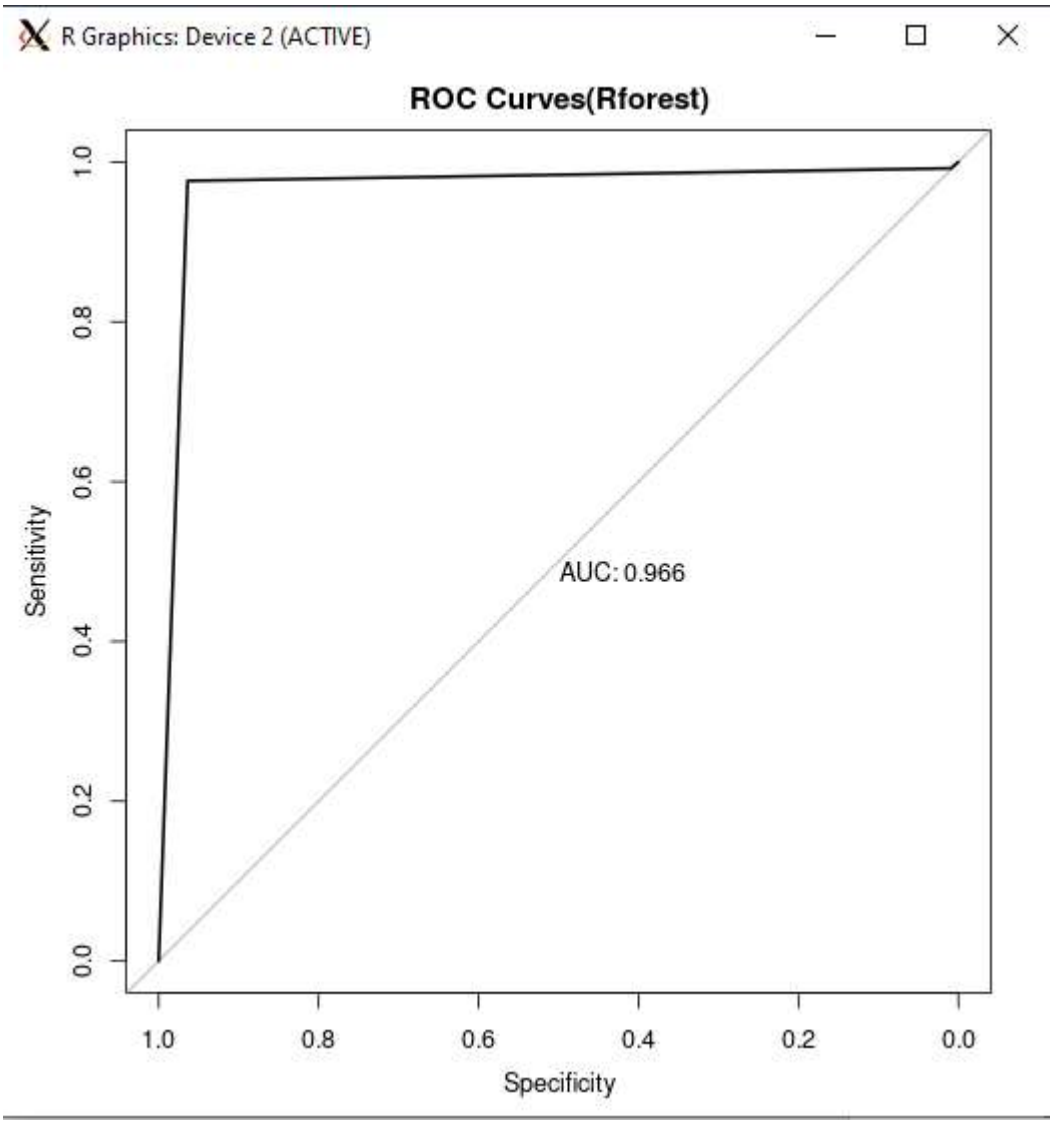
```
accuracy<-function(xt)sum(diag(xt))/sum(xt)


xgbtbl <-table(tstdf$Class,pred_y)

gbmtbl <-table(tstdf$Class,as.factor(labels))

rfmtbl <-table(tstdf$Class,ry_pred)


gbm_acc<-accuracy(gbmtbl)

rfm_acc<-accuracy(rfmtbl)

xgb_acc<-accuracy(xgbtbl)

stk_acc<-accuracy(stbl)


detach("package:caret", unload=TRUE)  ##To avoid conflict with pRoc library

require(pRoc) ##To call the multiclass ROC
```

```
gy_pred<-as.factor(labels)
##GBM
mqa<-multiclass.roc(response=tstdf$Class, predictor=factor(gy_pred, ordered=TRUE),  plot=TRUE
,print.auc=TRUE, main='ROC Curves(GBM)')
```



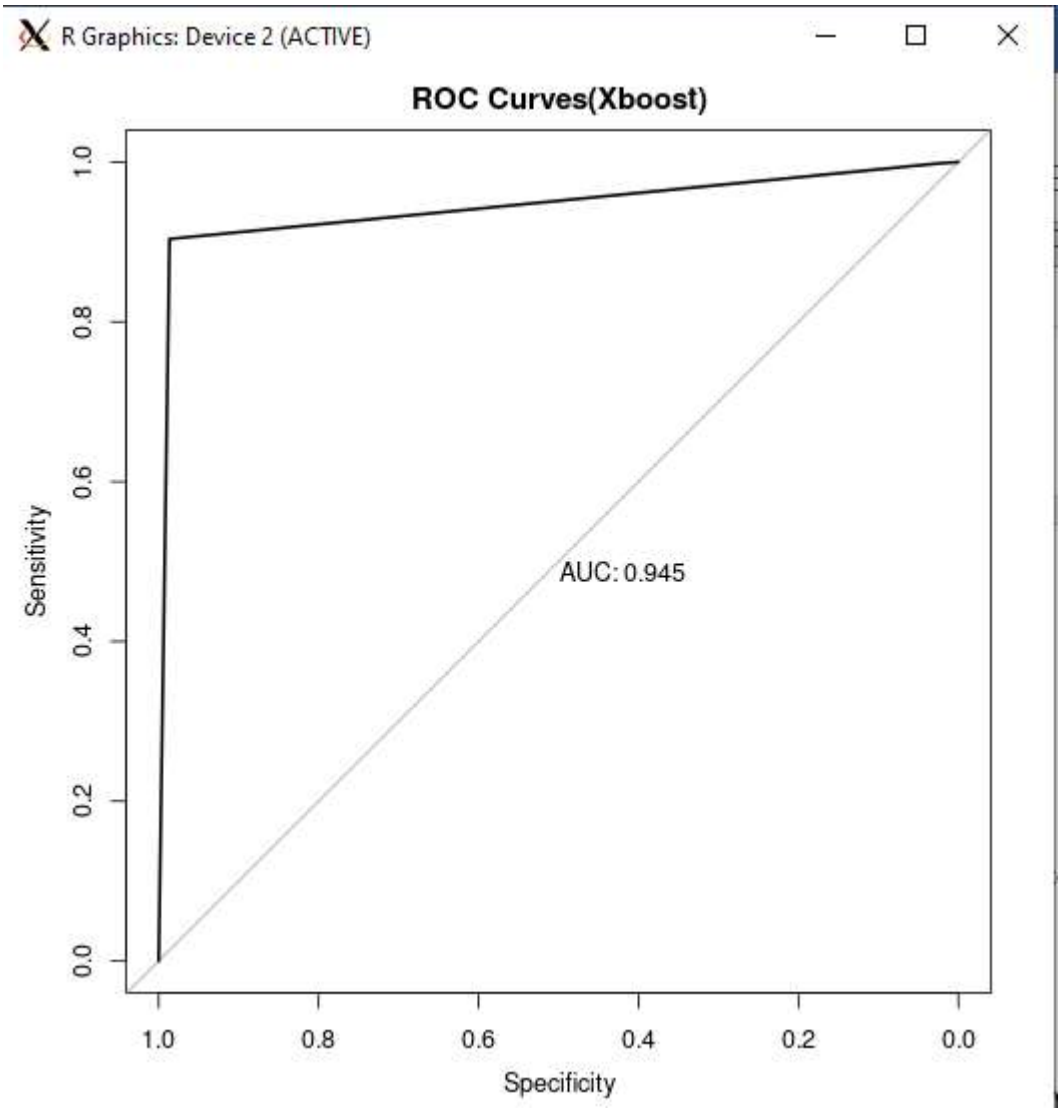
##RFM

rqa<-multiclass.roc(response=tstdf$Class, predictor=factor(ry_pred, ordered=TRUE), plot=TRUE ,print.auc=TRUE, main='ROC Curves(Rforest)')



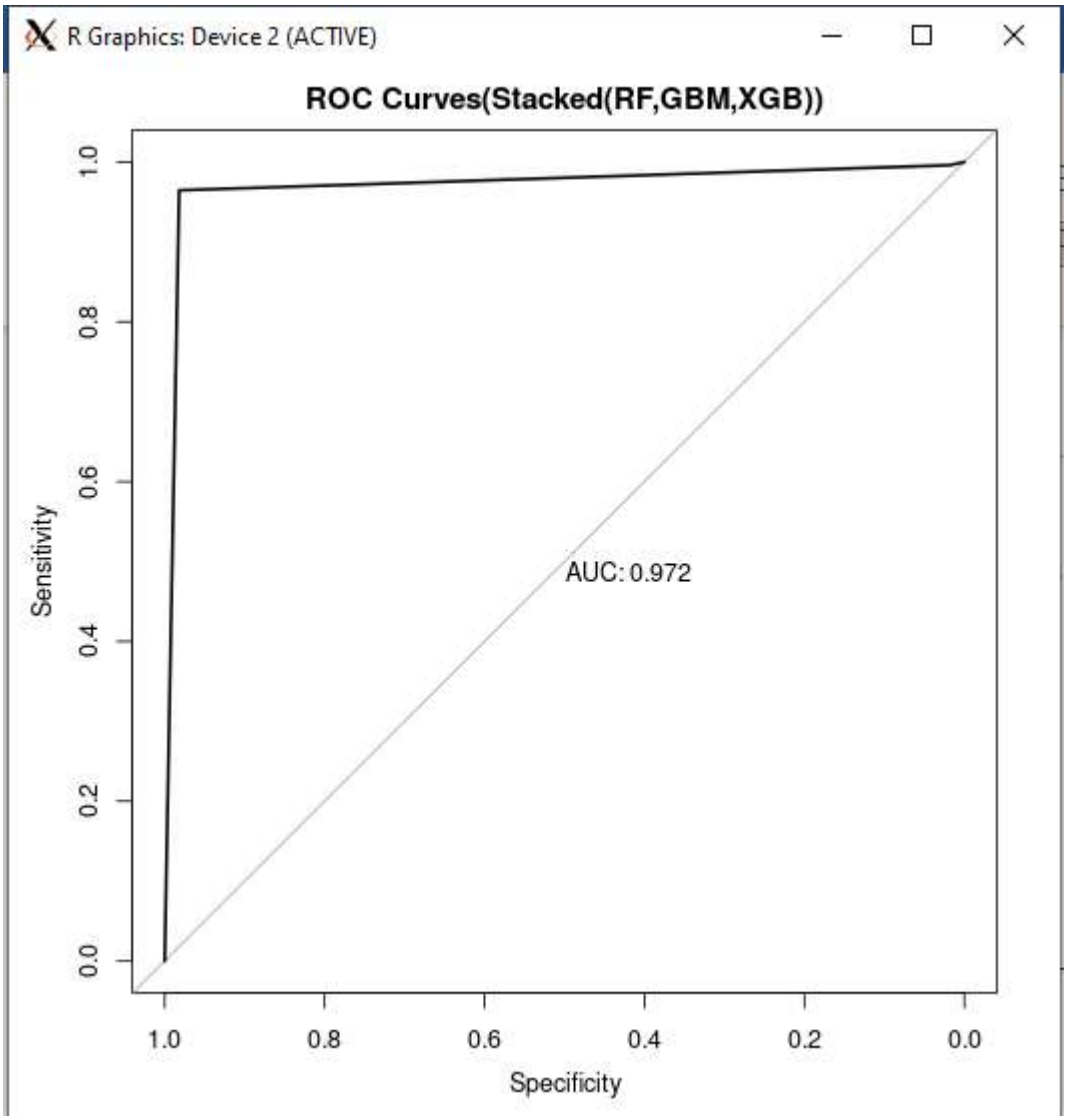
##Xboost

```
xqa<-multiclass.roc(response=tstdf$Class, predictor=factor(pred_y, ordered=TRUE), plot=TRUE
,print.auc=TRUE, main='ROC Curves(Xboost)')
```



##Stacking

```
sqa<-multiclass.roc(response=tstdf$Class, predictor=factor(testing$stacked, ordered=TRUE), plot=TRUE
,print.auc=TRUE, main='ROC Curves(Stacked(RF,GBM,XGB))')
```



To obtain accuracy for each model

```
gbauc<-auc(mqa)

rtauc<-auc(rqa)

xbauc<-auc(xqa)

stauc<-auc(sqa)
```

##To create a table for accuracy and AUC

```
perfdf<-data.frame(Algo=c("gbm","rfm","xgb","Stk"), Acc=c(gbm_acc, rfm_acc,xgb_acc,stk_acc),
AUC=c(gbauc,rtauc,xbauc,stauc))
```

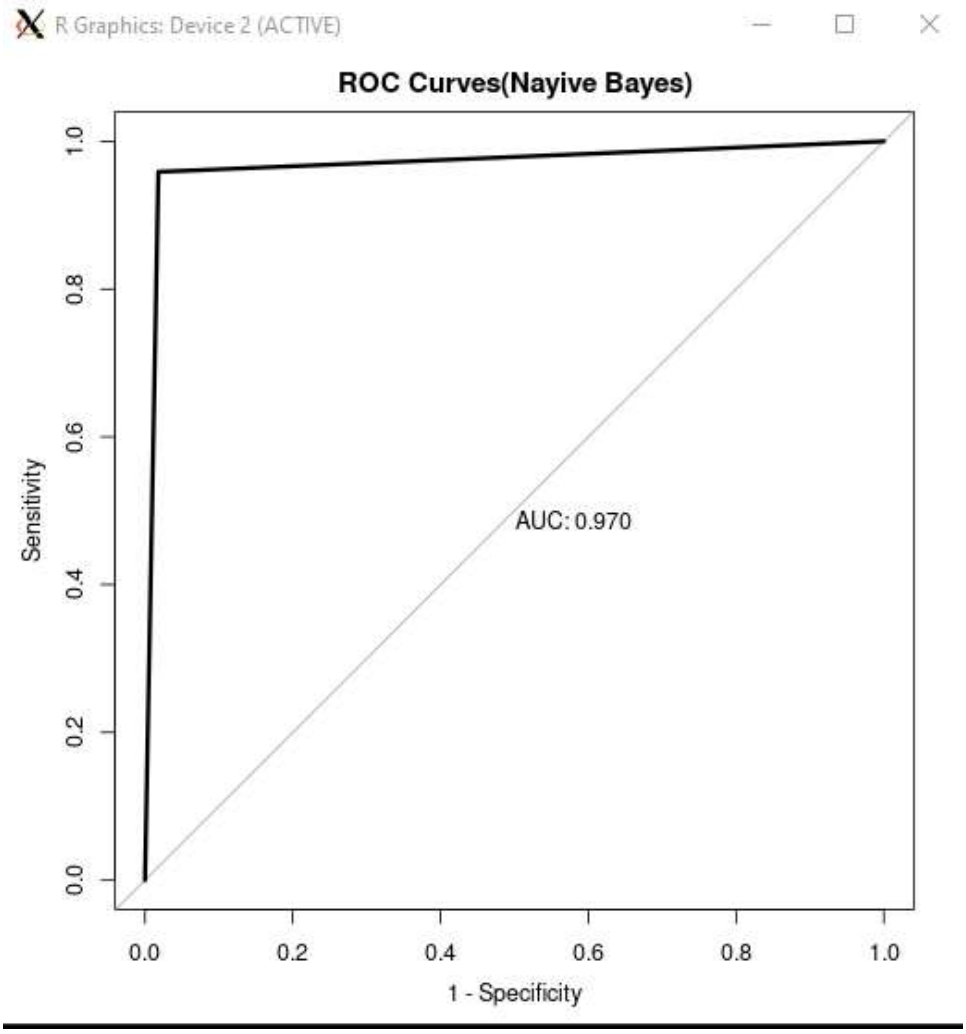
print(perfdf)

```
  Algo       Acc       AUC
1  gbm 0.9613375 0.9730518
2  rfm 0.9634274 0.9678323
3  xgb 0.9184953 0.9596077
4  Stk 0.9582027 0.9735207
```

The best performing model was Generalized-Boosting Model. It had 2nd most highest AUC and the most highest accuracy among the all the models. The higher AUC is a good measure to see how the classifier distinguish classes. I believe the stacking method could be improved better if I could choose a mode of three classifier's prediction or something that allows me to pick out the best selection of those three.

Naïve Bayes VS The methods to reduce bias or variance

```
nbmodel <- naiveBayes(Class ~., data=trdf)

nbpred <- predict(nbmodel, newdata=tstdf, type="class")

NBA<-multiclass.roc(response=tstdf$Class, predictor=factor(nbpred, ordered=TRUE),
plot=TRUE,lwd=3,

legacy.axes=TRUE,print.auc=TRUE, main='ROC Curves(Nayive Bayes)')
```



```
nbtbl <-table(tstdf$Class,nbpred)

nb_acc<-accuracy(nbtbl )

nbauc<-auc(NBA)

perfdf<-data.frame(Algo=c("gbm","rfm","xgb","Stk","NaiB"), Acc=c(gbm_acc,
rfm_acc,xgb_acc,stk_acc,nb_acc), AUC=c(gbauc,rtauc,xbauc,stauc,nbauc))

print(perfdf)
```

```
> print(perfdf)
  Algo       Acc       AUC
1  gbm 0.9613375 0.9730518
2  rfm 0.9634274 0.9678323
3  xgb 0.9184953 0.9596077
4  Stk 0.9582027 0.9735207
5 NaiB 0.7366771 0.8135477
```

Naïve Bayes did not perform well for multi-classification model. Although 0.81 is still high score, compared to other methods, the difference was rather significant. Generally Naïve Bayes are bad for high correlated features, but during EDA phase I did not see highly correlated relationship between each alphabet. I discovered that this topic could be more advanced topic such as natural-language-processing so I decided to step back from exploring further.

Unfortunately, using one-hot-encoded data made the data leakage during cross-validation, but I performed cross-validation folding in Generalized-boosting model and this could be why it performed better than XGBoost as the cross validation is very effective on reducing bias as the model use the most of the data for fitting, while reduces variance by using the most data for validation set. So it balances out bias and variances in an useful way.