# Main

Aurore Gosmant, Changhao He, James Smiley, Weiwei Song, Zi Fang

In your final repo, there should be an R markdown file that organizes **all computational steps** for evaluating your proposed Facial Expression Recognition framework.

This file is currently a template for running evaluation experiments. You should update it according to your codes but following precisely the same structure.

```
if(!require("EBImage")){
  install.packages("BiocManager")
  BiocManager::install("EBImage")
}
if(!require("R.matlab")){
  install.packages("R.matlab")
}
if(!require("readxl")){
  install.packages("readxl")
}

if(!require("dplyr")){
  install.packages("dplyr")
}
if(!require("readxl")){
  install.packages("readxl")
}

if(!require("ggplot2")){
  install.packages("ggplot2")
}

if(!require("caret")){
  install.packages("caret")
}

if(!require("glmnet")){
  install.packages("glmnet")
}

if(!require("WeightedROC")){
  install.packages("WeightedROC")
}

if(!require("gbm")){
  install.packages("gbm")
}

if(!require("randomForest")){
```

```
  install.packages("randomForest")
}
if(!require("grid")){
  install.packages("grid")
}
if(!require("gridExtra")){
  install.packages("gridExtra")
}
if(!require("xgboost")){
  install.packages("xgboost")
}
if(!require("DMwR")){
  install.packages("DMwR")
}
library(grid)
library(gridExtra)
library(R.matlab)
library(readxl)
library(dplyr)
library(EBImage)
library(ggplot2)
library(caret)
library(glmnet)
library(WeightedROC)
library(gbm)
library(randomForest)
library(xgboost)
library(DMwR)
library(pROC)
```

## Step 0 set work directories

```
set.seed(2020)
# setwd("~/Project3-FacialEmotionRecognition/doc")
# here replace it with your own path or manually set it in RStudio to where this rmd file is located.
# use relative path for reproducibility
```

Provide directories for training images. Training images and Training fiducial points will be in different subfolders.

```
train_dir <- "../data/train_set/" # This will be modified for different data sets.
train_image_dir <- paste(train_dir, "images/", sep="")
train_pt_dir <- paste(train_dir,  "points/", sep="")
train_label_path <- paste(train_dir, "label.csv", sep="")
```

# Step 1: set up controls for evaluation experiments.

In this chunk, we have a set of controls for the evaluation experiments.

- (T/F) cross-validation on the training set
- (T/F) reweighting the samples for training set
- (number) K, the number of CV folds
- (T/F) process features for training set
- (T/F) run evaluation on an independent test set
- (T/F) process features for test set

```
K <- 5  # number of CV folds

run.feature.train <- TRUE # process features for training set
run.feature.test <- TRUE # process features for test set

sample.reweight <- TRUE # run sample reweighting in model training

run.cv.GBM<-FALSE #run cross-validation on the training set applying GBM mod
run.train.GBM<-TRUE
run.test.GBM<-TRUE

run.cv.XGS<-FALSE
fit.train.XGS<-TRUE
run.test.XGS<-TRUE
```

Using cross-validation or independent test set evaluation, we compare the performance of models with different specifications. In this Starter Code, we tune parameter lambda (the amount of shrinkage) for logistic regression with LASSO penalty.

```
##hyper parameters GBM
hyper_grid_GBM<- expand.grid(
  shrinkage=c(0.05, 0.1,.15),
  interaction.depth=c(1,2),
  n.minobsinnode=c(10),
  bag.fraction=c(1),
  n.trees=c(60,100,200,400)
)
hyper_grid_GBM
```

| shrinkage <dbl> | interaction.depth <dbl> | n.minobsinnode <dbl> | bag.fraction <dbl> | n.trees <dbl> |
|---|---|---|---|---|
| 0.05 | 1 | 10 | 1 | 60 |
| 0.10 | 1 | 10 | 1 | 60 |
| 0.15 | 1 | 10 | 1 | 60 |
| 0.05 | 2 | 10 | 1 | 60 |
| 0.10 | 2 | 10 | 1 | 60 |
| 0.15 | 2 | 10 | 1 | 60 |
| 0.05 | 1 | 10 | 1 | 100 |
| 0.10 | 1 | 10 | 1 | 100 |
| 0.15 | 1 | 10 | 1 | 100 |
| 0.05 | 2 | 10 | 1 | 100 |

1-10 of 24 rows Previous **1** 2 3 Next

```
##hyper parameters XGboost + Smote
hyper_grid_XGS<- expand.grid(
 max_depth= c(1,3,5,7,9,10,11,13,15,17,19)
                    )
hyper_grid_XGS
```

| max_depth<br><dbl> |
|---|
| 1 |
| 3 |
| 5 |
| 7 |
| 9 |
| 10 |
| 11 |
| 13 |
| 15 |
| 17 |

1-10 of 11 rows Previous **1** 2 Next

## Step 2: import data and train-test split

```
#train-test split
info <- read.csv(train_label_path)
n <- nrow(info)
n_train <- round(n*(4/5), 0)
train_idx <- sample(info$Index, n_train, replace = F)
test_idx <- setdiff(info$Index, train_idx)
```

If you choose to extract features from images, such as using Gabor filter, R memory will exhaust all images are read together. The solution is to repeat reading a smaller batch(e.g 100) and process them.

```
n_files <- length(list.files(train_image_dir))

image_list <- list()
for(i in 1:100){
   image_list[[i]] <- readImage(paste0(train_image_dir, sprintf("%04d", i), ".jpg"))
}
```

Fiducial points are stored in matlab format. In this step, we read them and store them in a list.0

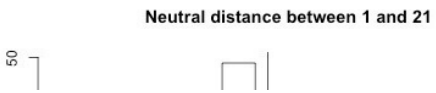
```
#function to read fiducial points
#input: index
#output: matrix of fiducial points corresponding to the index
readMat.matrix <- function(index){
     return(round(readMat(paste0(train_pt_dir, sprintf("%04d", index), ".mat"))[[1]],0))
}

#load fiducial points
fiducial_pt_list <- lapply(1:n_files, readMat.matrix)
```

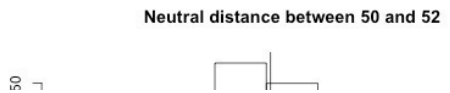
```
## Warning in readMat(paste0(train_pt_dir, sprintf("%04d", index), ".mat")):
## strings not representable in native encoding will be translated to UTF-8
```

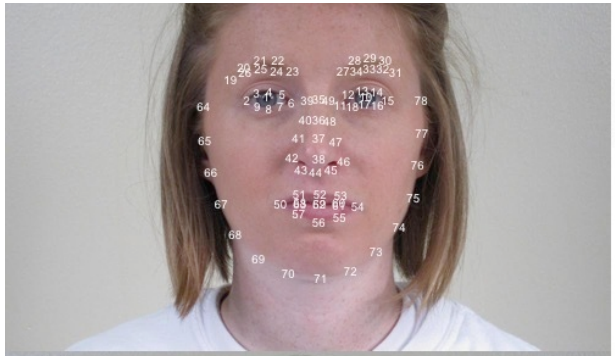
```
save(fiducial_pt_list, file="../output/fiducial_pt_list.RData")
```

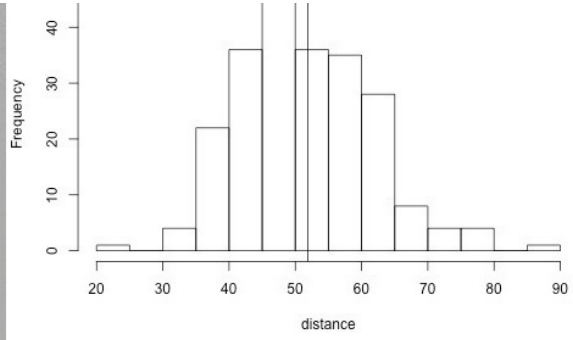
# Step 3: construct features and responses

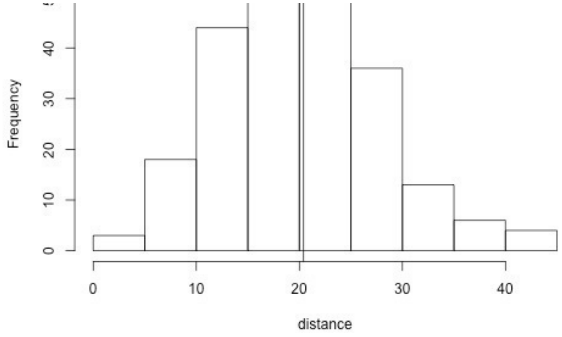
- The follow plots show how pairwise distance between fiducial points can work as feature for facial emotion recognition.
    - In the first column, 78 fiducials points of each emotion are marked in order.
    - In the second column distributions of vertical distance between right pupil(1) and right brow peak(21) are shown in histograms. For example, the distance of an angry face tends to be shorter than that of a surprised face.
    - The third column is the distributions of vertical distances between right mouth corner(50) and the midpoint of the upper lip(52). For example, the distance of an happy face tends to be shorter than that of a sad face.

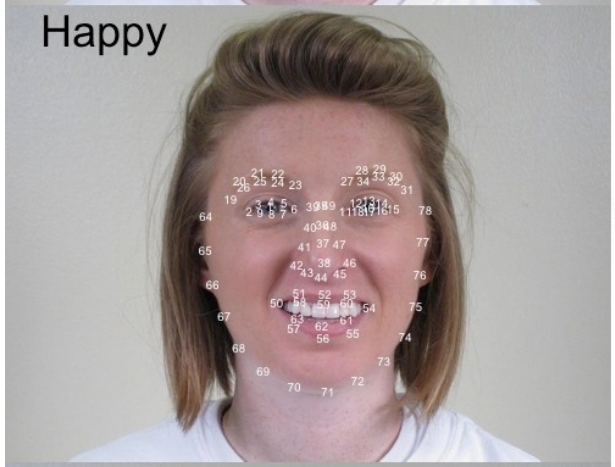


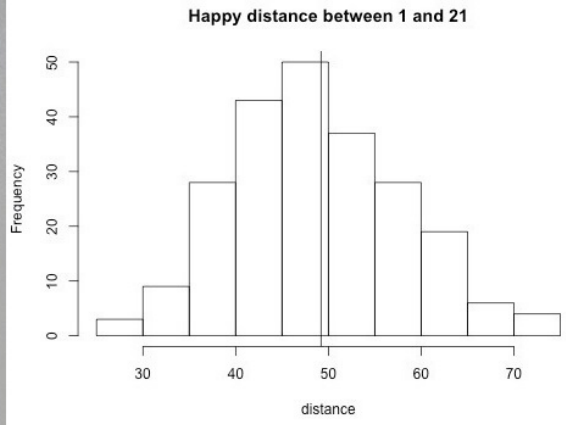


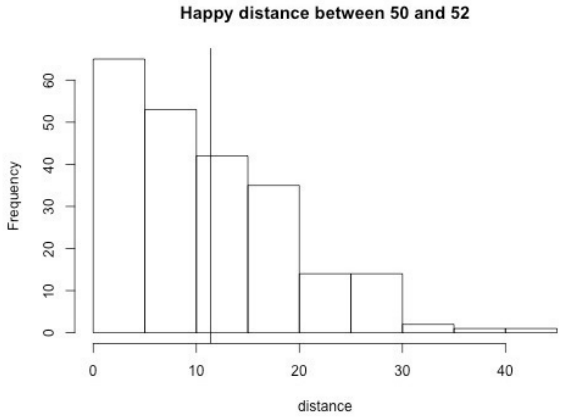


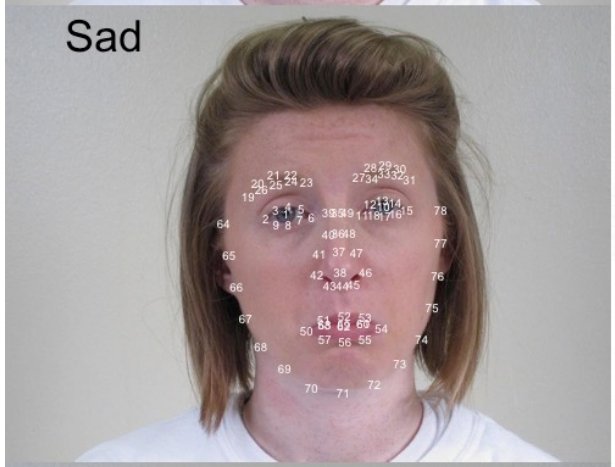

Frequency
distance

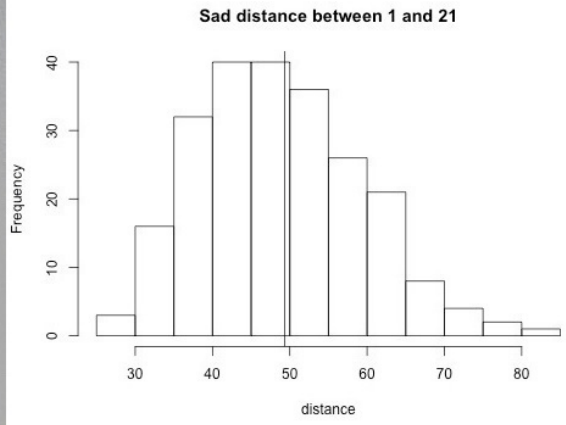

Frequency
distance

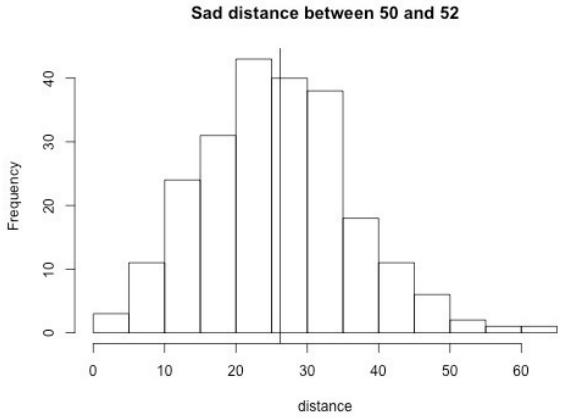

Happy

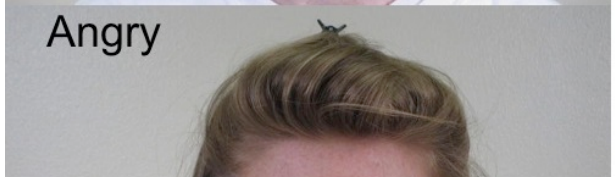

Happy distance between 1 and 21
Frequency
distance

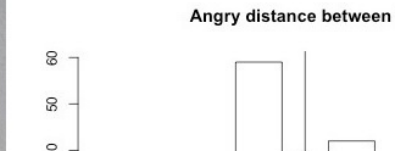

Happy distance between 50 and 52
Frequency
distance

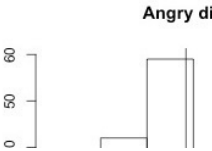

Sad

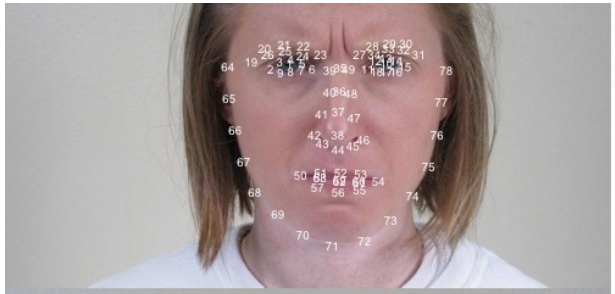

Sad distance between 1 and 21
Frequency
distance

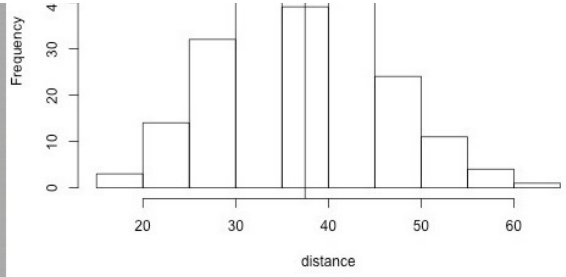

Sad distance between 50 and 52
Frequency
distance

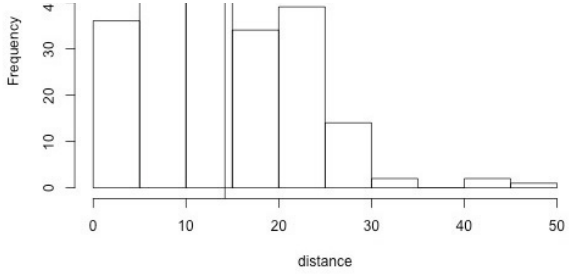

Angry


Angry distance between 1 and 21

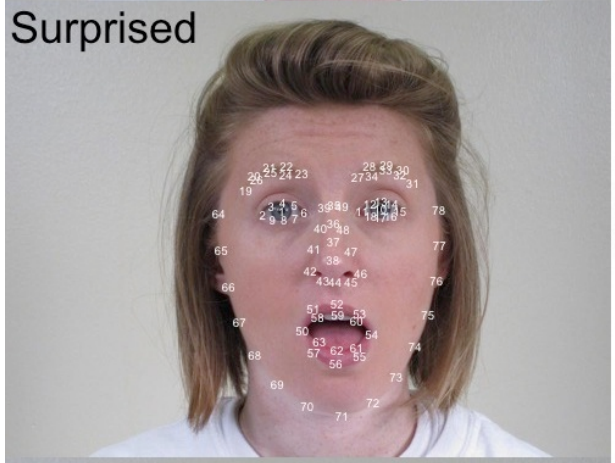

Angry distance between 50 and 52



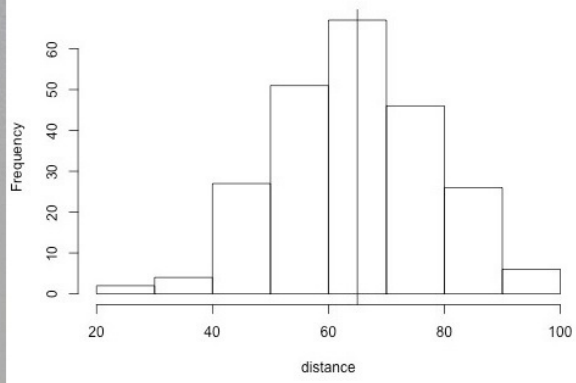


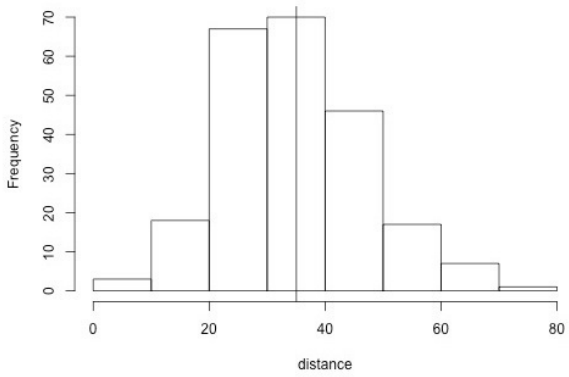

Surprised



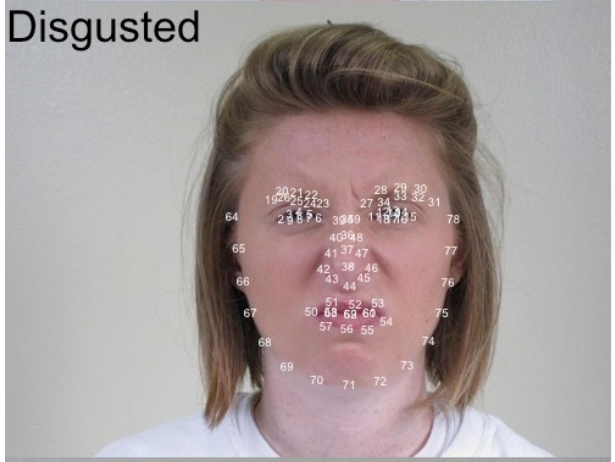
**Surprised distance between 1 and 21**



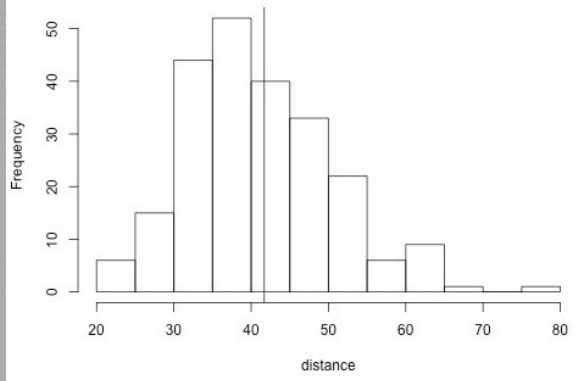

**Surprised distance between 50 and 52**



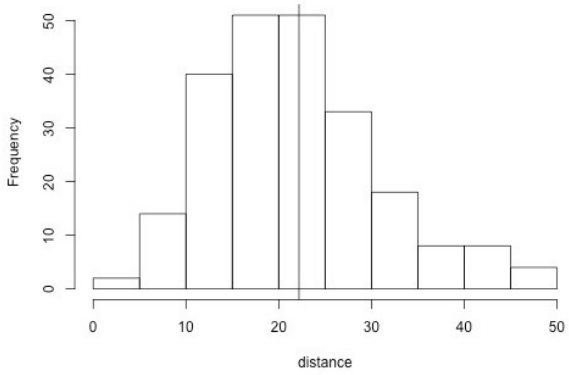

Disgusted



**Disgusted distance between 1 and 21**

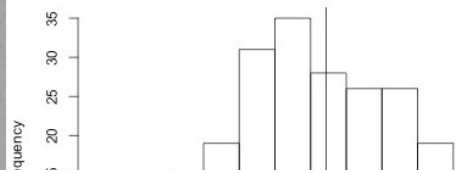



**Disgusted distance between 50 and 52**

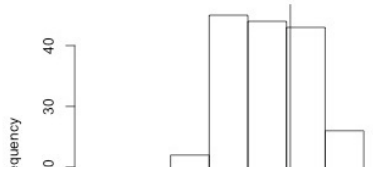



Fearful

**Fearful distance between 1 and 21**



**Fearful distance between 50 and 52**



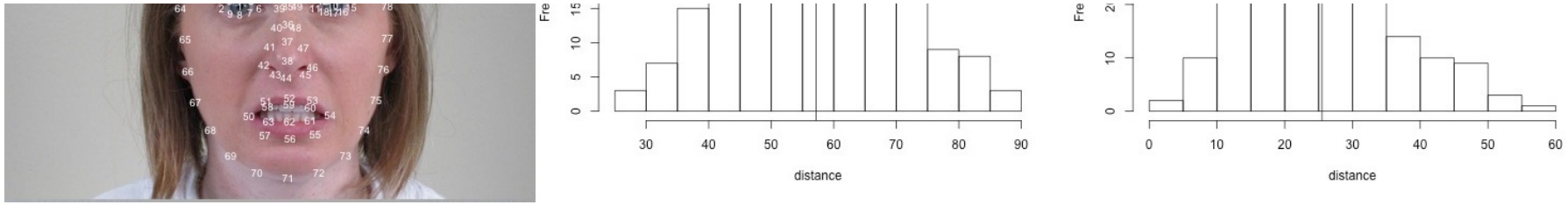


Figure1

`feature.R` should be the wrapper for all your feature engineering functions and options. The function `feature( )` should have options that correspond to different scenarios for your project and produces an R object that contains features and responses that are required by all the models you are going to evaluate later.

- `feature.R`
- Input: list of images or fiducial point
- Output: an RData file that contains extracted features and corresponding responses

```
source("../lib/feature.R")
tm_feature_train <- NA
if(run.feature.train){
  tm_feature_train <- system.time(dat_train <- feature(fiducial_pt_list, train_idx))
  save(dat_train, file="../output/feature_train.RData")
}else{
  load(file="../output/feature_train.RData")
}

tm_feature_test <- NA
if(run.feature.test){
  tm_feature_test <- system.time(dat_test <- feature(fiducial_pt_list, test_idx))
  save(dat_test, file="../output/feature_test.RData")
}else{
  load(file="../output/feature_test.RData")
}
```

# Step 4: Train a classification model with training features and responses(BGM)

Call the train model and test model from library. + `train_BGM.R` + Input: 1. a data frame containing features and labels(be careful labels should be either 0 or 1 when using packages gbm)
2.a parameter list: distribution:choose bernoulli because this is a binary classification problem. n.trees: number of trees(the number of gradient

boosting iteration) i.e Increasing N reduce the errir on the training set,but setting it too big will lead overfitting, too small will leads too many iterations, which is time consuming interation.depth: Maximum nodes per tree - number of nodes split it has to perform on a tree(starting from a single node) shrinkage: learning rate, big shrinkage is potentially a reason for overfitting, small shrinkage give better result, but time consuming n.minobsinnode: the minimum number of observations in tree's terminal nodes, Set n.minobsinnode = 10. When working with small training samples it may be vital to lower this setting to five or even three. bag.fraction: subsampling fraction the fraction of the training set observations randomly selected to propose the next tree in the expansion. In this case, it adopts stochastic gradient boosting strategy. By default, it is 0.5. That is half of the training sample at each iteration. You can use fraction greater than 0.5 if training sample is small + Output:a trained model + `test_GBM.R` + Input: the fitted classification model using training data and processed features from testing images + Input: an R object that contains a trained classifier. + Output: training model specification

```
source("../lib/train_GBM.R")
source("../lib/test_GBM.R")
source("../lib/cross_validation_GBM.R")
```

## Model selection with cross-validation

- Do model selection by choosing among different values of training model parameters.

```
feature_train = as.matrix(dat_train[, -6007])
label_train = as.integer(dat_train$label)

if(run.cv.GBM){
  res_cv_GBM <- matrix(0, nrow = nrow(hyper_grid_GBM), ncol = 4)
  for(i in 1:nrow(hyper_grid_GBM)){
    print(i)
    res_cv_GBM[i,] <- cv.function_GBM(traindf = dat_train, K,
                              num.trees = hyper_grid_GBM$n.trees[i],
                              depth = hyper_grid_GBM$interaction.depth[i],
                              rate = hyper_grid_GBM$shrinkage[i],
                              bag_f = hyper_grid_GBM$bag.fraction[i],
                              n_m = hyper_grid_GBM$n.minobsinnode[i],
                              reweight = TRUE)
  save(res_cv_GBM, file="../output/res_cv_GBM.RData")
  }
}else{
  load("../output/res_cv_GBM.RData")
}
```

Visualize cross-validation results.

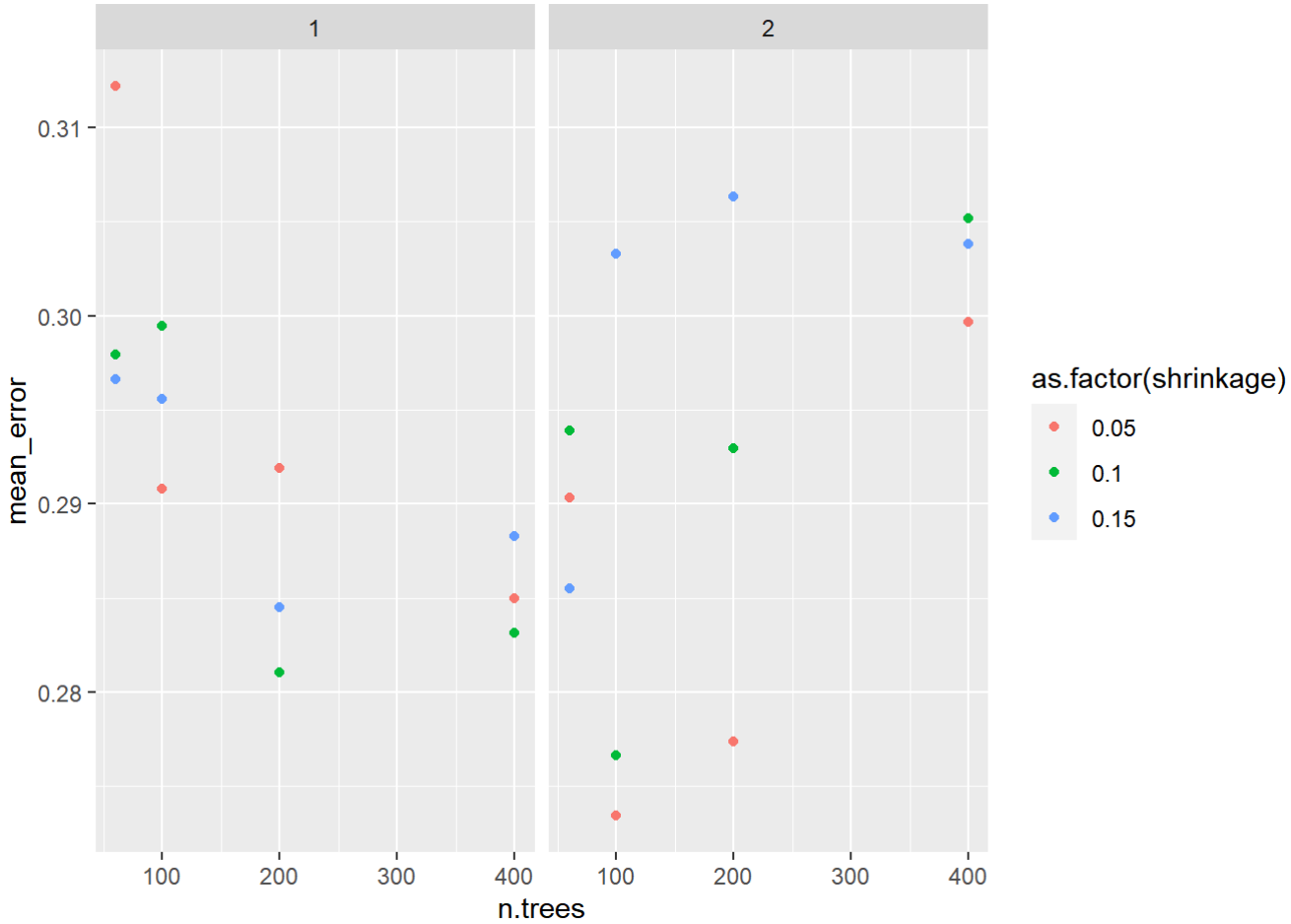
```
res_cv_GBM <- as.data.frame(res_cv_GBM)
colnames(res_cv_GBM) <- c("mean_error", "sd_error", "mean_AUC", "sd_AUC")
GBM_cv_results = data.frame(hyper_grid_GBM, res_cv_GBM)
GBM_cv_results[order(GBM_cv_results$mean_error), ]
```

| | shrinkage <dbl> | interaction.depth <dbl> | n.minobsinnode <dbl> | bag.fraction <dbl> | n.trees <dbl> | mean_error <dbl> | sd_error <dbl> |
|---|---|---|---|---|---|---|---|
| 10 | 0.05 | 2 | 10 | 1 | 100 | 0.2734100 | 0.02345315 |
| 11 | 0.10 | 2 | 10 | 1 | 100 | 0.2766518 | 0.01890666 |
| 16 | 0.05 | 2 | 10 | 1 | 200 | 0.2773706 | 0.02389684 |
| 14 | 0.10 | 1 | 10 | 1 | 200 | 0.2810600 | 0.02749531 |
| 20 | 0.10 | 1 | 10 | 1 | 400 | 0.2831496 | 0.03166271 |
| 15 | 0.15 | 1 | 10 | 1 | 200 | 0.2845108 | 0.03550026 |
| 19 | 0.05 | 1 | 10 | 1 | 400 | 0.2849841 | 0.03240151 |
| 6 | 0.15 | 2 | 10 | 1 | 60 | 0.2854899 | 0.03202788 |
| 21 | 0.15 | 1 | 10 | 1 | 400 | 0.2882771 | 0.02707742 |
| 4 | 0.05 | 2 | 10 | 1 | 60 | 0.2903504 | 0.03607912 |

1-10 of 24 rows | 1-8 of 10 columns

Previous **1** 2 3 Next

```
ggplot(GBM_cv_results, aes(x = n.trees, y= mean_error))+
  geom_point(aes(color = as.factor(shrinkage)))+
  facet_wrap(~interaction.depth)
```




- Choose the “best” parameter value

```
#Choose the hyper parameters which minimize cross validation error. Because the AUC this model is similar to other good models
 and it has the smalles model complexity.
par_best_GBM <- GBM_cv_results[which.min(GBM_cv_results$mean_error),]
par_best_GBM <- par_best_GBM[1:5]
par_best_GBM
```

| shrinkage | interaction.depth | n.minobsinnode | bag.fraction | n.trees |
|---|---|---|---|---|
| <dbl> | <dbl> | <dbl> | <dbl> | <dbl> |

| | shrinkage<br><dbl> | interaction.depth<br><dbl> | n.minobsinnode<br><dbl> | bag.fraction<br><dbl> | n.trees<br><dbl> |
|---|---|---|---|---|---|
| 10 | 0.05 | 2 | 10 | 1 | 100 |

1 row

- Train the model with the entire training set using the selected model (model parameter) via cross-validation.

```
# training weights
if(run.train.GBM){
weight_train <- rep(NA, length(label_train))
for (v in unique(label_train)){
  weight_train[label_train == v] = 0.5 * length(label_train) / length(label_train[label_train == v])
}
if (sample.reweight){
  tm_train_GBM <- system.time(fit_train_GBM <- train_GBM(dat_train, w = weight_train,
                              num.trees = par_best_GBM$n.trees,
                              depth = par_best_GBM$interaction.depth,
                              rate = par_best_GBM$shrinkage,
                              bag_f = par_best_GBM$bag.fraction,
                              n_m = par_best_GBM$n.minobsinnode))
} else {
  tm_train_GBM <- system.time(fit_train_GBM <- train_GBM(dat_train, w = NULL,
                                                         num.trees = par_best_GBM$n.trees,
                              depth = par_best_GBM$interaction.depth,
                              rate = par_best_GBM$shrinkage,
                              bag_f = par_best_GBM$bag.fraction,
                              n_m = par_best_GBM$n.minobsinnode))
}
save(fit_train_GBM, file="../output/fit_train_GBM.RData")
}else{
    load(file="../output/fit_train_GBM.RData")
}
```

# Step 5: Run test on test images

```
tm_test_GBM = NA
if(run.test.GBM){
  tm_test_GBM <- system.time({pred_GBM <- test_GBM(fit_train_GBM, dat_test)
                              prob_pred_GBM <- pred_GBM[[2]];
                              label_pred_GBM <- as.integer(pred_GBM[[1]])})
}
```

```
## Using 100 trees...
```

- evaluation

```
## reweight the test data to represent a balanced label distribution
label_test <- as.integer(dat_test$label)

weight_test <- rep(NA, length(label_test))
for (v in unique(label_test)){
  weight_test[label_test == v] = 0.5 * length(label_test) / length(label_test[label_test == v])
}

accu_GBM <- sum(weight_test * (label_pred_GBM == label_test)) / sum(weight_test)
tpr.fpr.GBM <- WeightedROC(prob_pred_GBM, label_test, weight_test)
auc_GBM <- WeightedAUC(tpr.fpr.GBM)


cat("The accuracy of weighted GBM model:", "is", accu_GBM*100, "%.\n")
```

```
## The accuracy of weighted GBM model: is 71.50696 %.
```

```
cat("The AUC of GBM  model:", "is", auc_GBM, ".\n")
```

```
## The AUC of GBM  model: is 0.7669593 .
```

# Summarize Running Time

Prediction performance matters, so does the running times for constructing features and for training the model, especially when the computation resource is limited.

```
cat("Time for constructing training features=", tm_feature_train[1], "s \n")
```

```
## Time for constructing training features= 0.48 s
```

```
cat("Time for constructing testing features=", tm_feature_test[1], "s \n")
```

```
## Time for constructing testing features= 0.13 s
```

```
cat("Time for training GBM model=", tm_train_GBM[1], "s \n")
```

```
## Time for training GBM model= 69.67 s
```

```
cat("Time for testing GBM model=", tm_test_GBM[1], "s \n")
```

```
## Time for testing GBM model= 10.77 s
```

# Step 6: Train a classification model with training features and responses(Advanced Model: XGboost + Smote)

Call the train model and test model from library.

`train.R` and `test.R` should be wrappers for all your model training steps and your classification/prediction steps.

- `train.R`
  - Input: a data frame containing features and labels and a parameter list.
  - Output:a trained model
- `test.R`
  - Input: the fitted classification model using training data and processed features from testing images
  - Input: an R object that contains a trained classifier.
  - Output: training model specification

- In this Starter Code, we use logistic regression with LASSO penalty to do classification.

```
source("../lib/train_XGS.R")
```

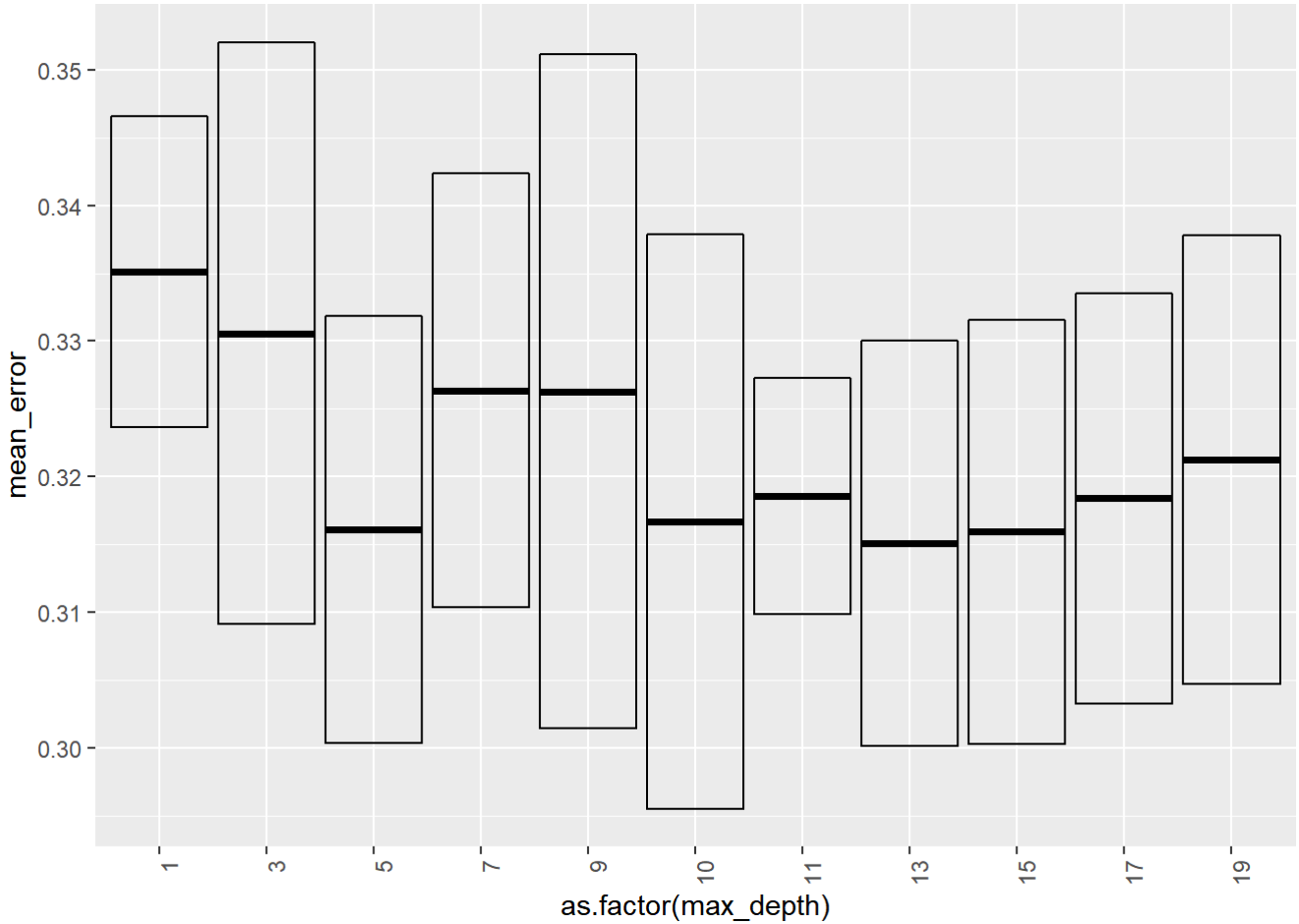
# Step 7: Model selection with cross-validation on XGboost + Smote

- Do model selection by choosing among different values of training model parameters.

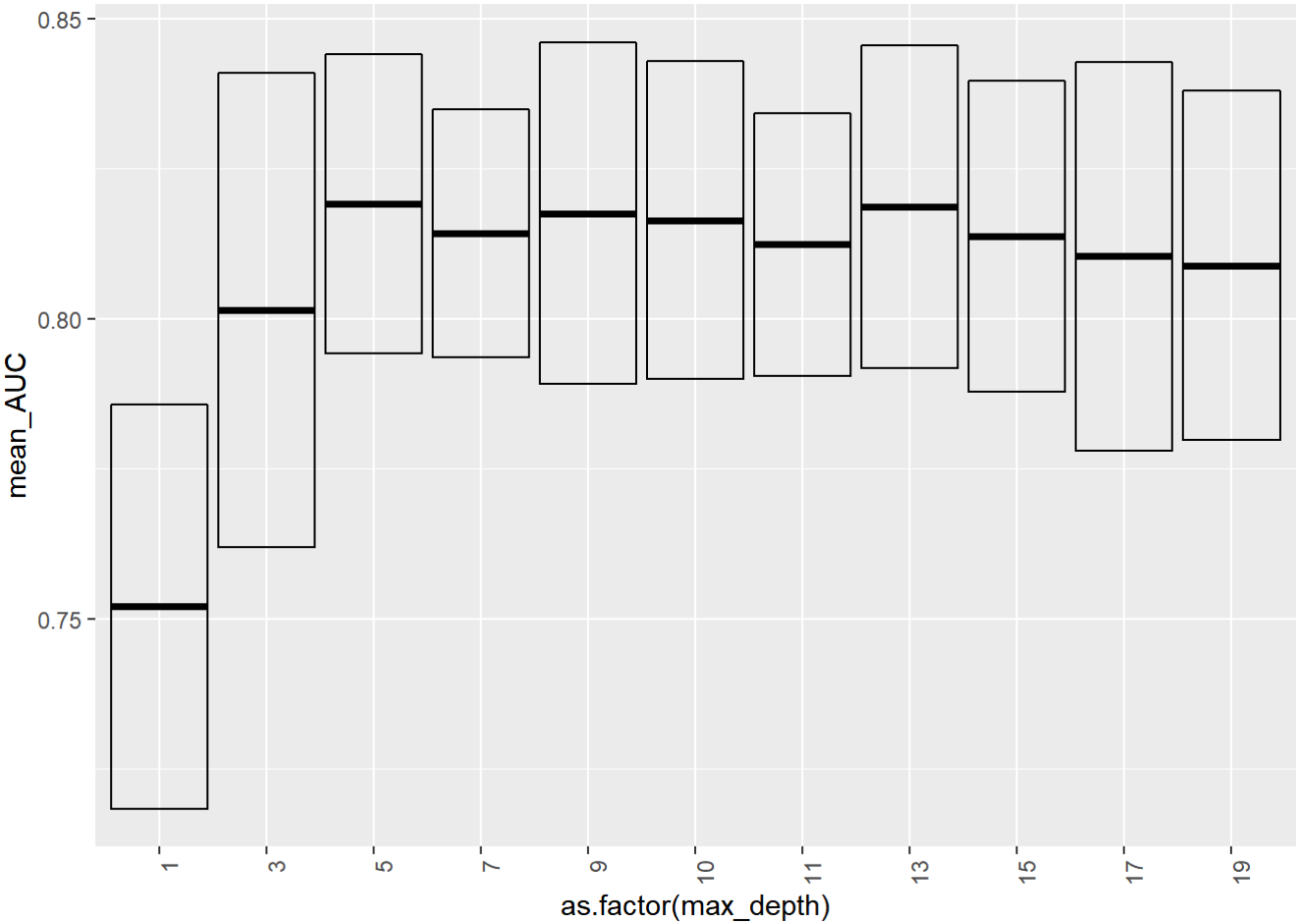
```
source("../lib/cross_validation_XGS.R")

if(run.cv.XGS){
  res_cv_XGS <- matrix(0, nrow = nrow(hyper_grid_XGS), ncol = 4)
  for(i in 1:nrow(hyper_grid_XGS)){
    print(i)
    res_cv_XGS[i,] <- cv.function_XGS(data=dat_train,
                                      K,
                                l = list(
      max_depth= hyper_grid_XGS$max_depth[i] ))

  save(res_cv_XGS, file="../output/res_cv_XGS.RData")}
 }else{
  load("../output/res_cv_XGS.RData")
}
```

Visualize cross-validation results.

```
res_cv_XGS <- as.data.frame(res_cv_XGS)
colnames(res_cv_XGS) <- c("mean_error", "sd_error", "mean_AUC", "sd_AUC")
result_cv_XGS<- cbind(hyper_grid_XGS,res_cv_XGS)


  p1 <- result_cv_XGS %>%
    ggplot(aes(x = as.factor(max_depth), y = mean_error,
               ymin = mean_error - sd_error, ymax = mean_error + sd_error)) +
    geom_crossbar() +
    theme(axis.text.x = element_text(angle = 90, hjust = 1))

  p2 <- result_cv_XGS %>%
    ggplot(aes(x = as.factor(max_depth), y = mean_AUC,
               ymin = mean_AUC - sd_AUC, ymax = mean_AUC + sd_AUC)) +
    geom_crossbar() +
    theme(axis.text.x = element_text(angle = 90, hjust = 1))

  print(p1)
```


0.35
0.34
0.33
0.32
0.31
0.30
mean_error
1
3
5
7
9
10
11
13
15
17
19
as.factor(max_depth)

```
print(p2)
```




- Choose the “best” parameter value

```
depth_best <- hyper_grid_XGS$max_depth[which.max(res_cv_XGS$mean_AUC)]
```

- Train the model with the entire training set using the selected model (model parameter) via cross-validation.

```
#SMOTE the training set
data_train.SMOTE<- SMOTE(label~.,dat_train)
table(data_train.SMOTE$label)
```

```
##
##    0    1
## 1904 1428
```

```
feature_train.S<-as.matrix(data_train.SMOTE[,-6007])
label_train.S<-as.integer(data_train.SMOTE$label)-1
table(label_train.S)
```

```
## label_train.S
##    0    1
## 1904 1428
```

```
fit.train.XGS<-TRUE
if(fit.train.XGS){
tm_train_XGS<- system.time(
  fit_train_XGS<- train_XGS(l=list(max_depth=depth_best),
                feature=feature_train.S,
                label= label_train.S))
save(fit_train_XGS,file="../output/fit_train_XGS.RData")
}else{
  load(file="../output/fit_train_XGS.RData")
}
```

## Step 8: Run test on test images based on XGboost + Smote

```
tm_test_XGS = NA
feature_test <- as.matrix(dat_test[, -6007])
if(run.test.XGS){
  load(file="../output/fit_train_XGS.RData")
  tm_test_XGS <- system.time({prob_pred <- predict(fit_train_XGS, feature_test);
                          label_pred <- ifelse(prob_pred>0.5,1,0)})
}
```

- evaluation

```
## reweight the test data to represent a balanced label distribution
label_test <- as.integer(dat_test$label)-1
weight_test <- rep(NA, length(label_test))
for (v in unique(label_test)){
  weight_test[label_test == v] = 0.5 * length(label_test) / length(label_test[label_test == v])
}

accu <- sum(weight_test * (label_pred == label_test)) / sum(weight_test)
tpr.fpr <- WeightedROC(prob_pred, label_test, weight_test)
auc <- WeightedAUC(tpr.fpr)


cat("The accuracy of XGboost + Smote model:", "is", accu*100, "%.\n")
```

```
## The accuracy of XGboost + Smote model: is 71.44351 %.
```

```
cat("The AUC of XGboost + Smote model:", "is", auc, ".\n")
```

```
## The AUC of XGboost + Smote model: is 0.8111325 .
```

# Summarize Running Time

Prediction performance matters, so does the running times for constructing features and for training the model, especially when the computation resource is limited.

```
cat("Time for constructing training features=", tm_feature_train[1], "s \n")
```

```
## Time for constructing training features= 0.48 s
```

```
cat("Time for constructing testing features=", tm_feature_test[1], "s \n")
```

```
## Time for constructing testing features= 0.13 s
```

```
cat("Time for training XGboost + Smote model =", tm_train_XGS[1], "s \n")
```

```
## Time for training XGboost + Smote model = 232.83 s
```

```
cat("Time for testing XGboost + Smote model=", tm_test_XGS[1], "s \n")
```

```
## Time for testing XGboost + Smote model= 0.37 s
```

The XGboost + smote model are very similar as our baseline model in the accuracy. But it has a relatively higher AUC and much smaller running time for testing.