Code ▾

GBM model

project3 group4

Hide

```
if(!require("EBImage")){
  source("https://bioconductor.org/biocLite.R")
  biocLite("EBImage")
}
if(!require("R.matlab")){
  install.packages("R.matlab")
}
if(!require("readxl")){
  install.packages("readxl")
}
```

```
package ‘readxl’ was built under R version 3.5.2
```

Hide

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

```
package ‘dplyr’ was built under R version 3.5.2
```

Hide

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

```
package ‘caret’ was built under R version 3.5.2
```

Hide

```
library(R.matlab)
library(readxl)
library(dplyr)
library(EBImage)
library(ggplot2)
library(caret)
library(gbm)
```

```
package ‘gbm’ was built under R version 3.5.2
```

Step 0: set work directories

Hide

```
set.seed(0)
setwd("~/Documents/GitHub/Spring2020-Project3-group4/doc")
```

Hide

```
train_dir <- "../data/" # 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
- (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

Hide

```
run.cv=TRUE # run cross-validation on the training set
K <- 5  # number of CV folds
run.feature.train=TRUE # process features for training set
run.test=TRUE # run evaluation on an independent test set
run.feature.test=TRUE # process features for test set
```

Using cross-validation or independent test set evaluation, we compare the performance of models with different specifications. In this Starter Code, we tune parameter k (number of neighbours) for KNN.

Step 2: import data and train-test split

Hide

```
#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.

Hide

```
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.

Hide

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

`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

Hide

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

Step 4: Train a classification model with training features and responses

Call the train model and test model from library.

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

- `train_gbm.R`
- Input: a data frame containing features and labels and a parameter list.
- 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

Hide

```
shrink = c(0.10,0.05,0.01)
model_labels = paste("GBM with Shrink =", shrink)
```

cross-validation to choose shrinkage parameter

Hide

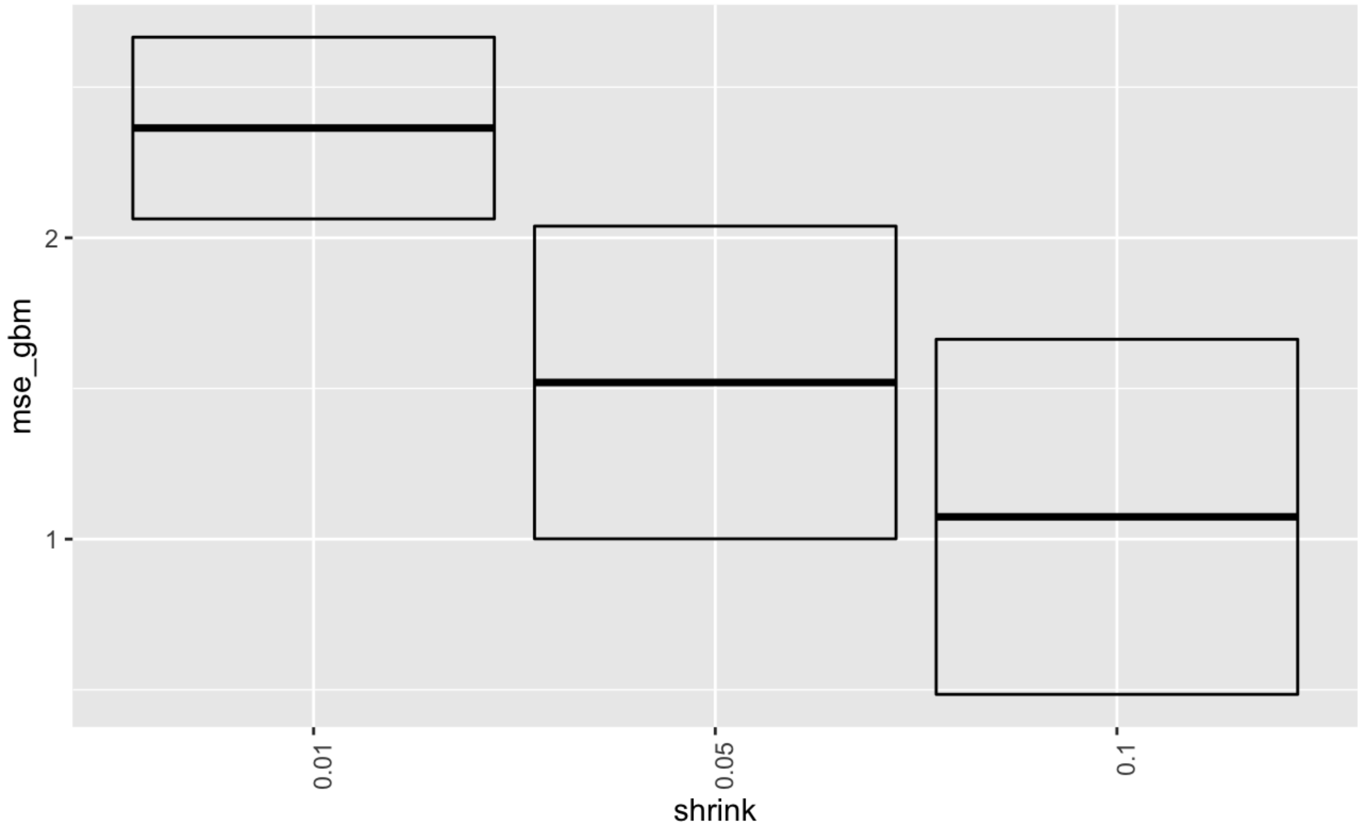
```
source("../lib/tuning_parameter_gbm.R")
if(run.cv){
     err_cv_gbm <- matrix(0, nrow = length(shrink), ncol = 2)
     for(i in 1:length(shrink)){
         cat("Shrink =", shrink[i], "\n")
         err_cv_gbm[i,] <- cv.function.gbm(dat_train, shrink[i])
#         save(err_cv_gbm, file="../output/err_cv_gbm.RData")
     }
 }
```

```
Shrink = 0.1
[1] 1.0776051 0.5865022
Shrink = 0.05
[1] 1.5213593 0.5193543
Shrink = 0.01
[1] 2.3673650 0.2991708
```

Visualize cross-validation results.

Hide

```
if(run.cv){
  load("../output/err_cv_gbm.RData")
  mse_cv_gbm <- as.data.frame(err_cv_gbm)
  colnames(mse_cv_gbm) <- c("mse_gbm", "sd_gbm")
  mse_cv_gbm$shrink = as.factor(shrink)
  mse_cv_gbm %>%
    ggplot(aes(x = shrink, y = mse_gbm,
               ymin = mse_gbm - sd_gbm, ymax = mse_gbm + sd_gbm)) +
    geom_crossbar() +
    theme(axis.text.x = element_text(angle = 90, hjust = 1))
}
```



- Choose the “best” parameter value

Hide

```
if(run.cv){
  model_best_gbm <- shrink[which.min(err_cv_gbm[,1])]
}
par_best_gbm <- list(shrink = model_best_gbm)
```

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

Hide

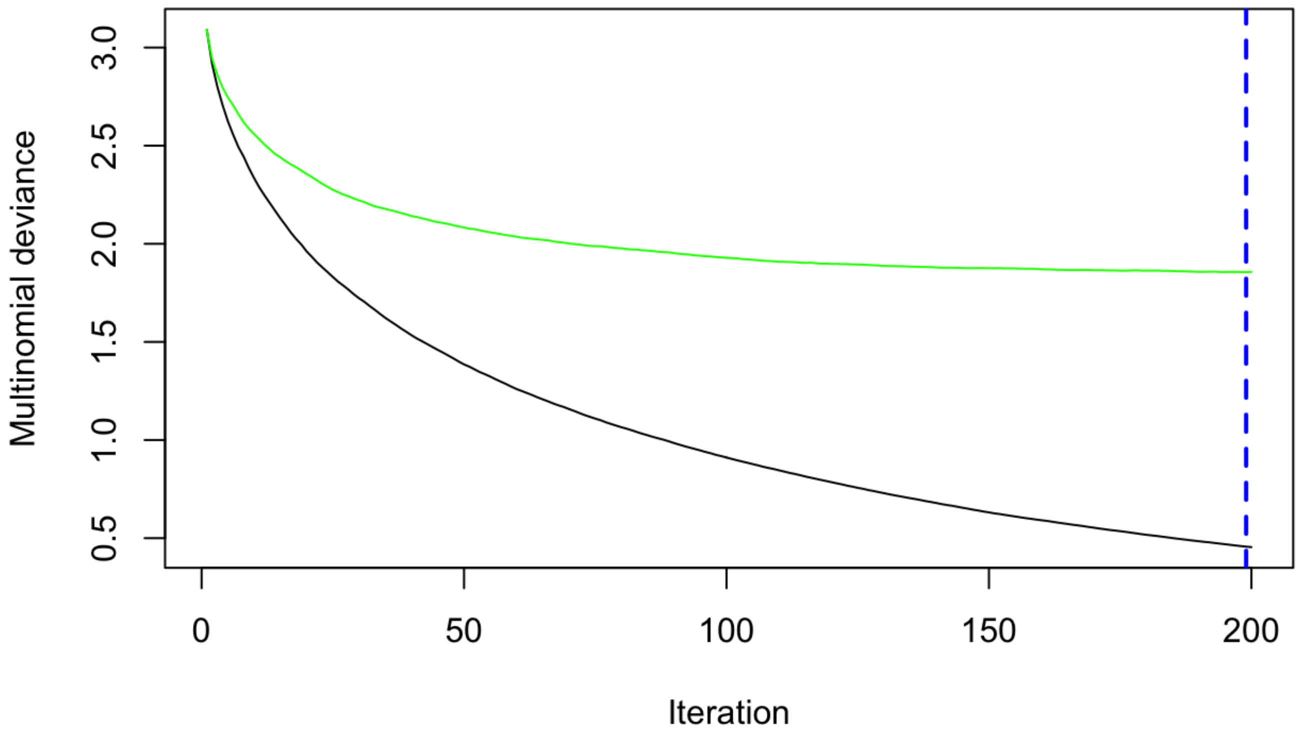
```
source("../lib/train_gbm.R")
###Traing
#gbm.fit<-gbm_train(dat_train)
#Save model
#saveRDS(gbm.fit, "../output/gbm.RDS")

#Google Drive link: https://drive.google.com/file/d/16ZQ-hkR1sJURZNX_NIpXcOsRSNsXgwy
C/view?usp=sharing
#load model
gbm.fit<-readRDS("../output/gbm.RDS")
```

Step 5: Run test on test images

Hide

```
source("../lib/test_gbm.R")
pred_gbm<-gbm_test(gbm.fit[[1]],dat_test)
```



Evaluation

Hide

```
pred.class<-apply(pred_gbm[[1]],1,which.max)
confusionMatrix(dat_test$emotion_idx,as.factor(pred.class))
```

```
Confusion Matrix and Statistics

          Reference
Prediction  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22
        1  14  0  4  1  0  0  0  0  0  1  0  0  0  0  0  0  0  0  0  0  0  0
        2   0 18  0  0  0  0  0  2  4  0  0  0  0  0  0  0  0  0  0  0  0  1
        3   5  0 18  0  0  0  1  0  0  3  0  1  0  0  0  1  0  1  0  0  0  0
        4   0  0  1 10  0  5  1  0  0  3  5  1  3  0  0  0  0  0  0  0  0  1
        5   0  0  0  0 11  0  1  0  0  0  0  0  0  4  6  0  1  1  0  0  0  0
        6   0  0  0  2  0 11  0  0  0  1  2  2  1  0  0  1  0  0  0  0  1  0
        7   0  0  0  0  0  0 16  0  0  0  1  0  0  0  0  1  1  1  5  0  1  1
        8   0  3  0  0  0  0  0 18  0  0  0  0  0  0  0  0  4  0  1  0  0  0
        9   0  2  0  0  0  1  0  1 14  0  2  0  0  0  1  0  0  0  0  1  1  0
        10  1  1  4  0  0  0  0  0  0  7  2  1  4  0  0  0  0  0  0  0  0  2
        11  0  0  0  1  0  1  0  0  0  4  5  8  1  1  0  0  0  0  0  0  0  0
        12  0  0  0  1  0  0  0  0  0  1  1  6  4  1  0  0  0  0  0  0  0  3
        13  0  0  2  3  0  0  0  0  0  0  3  4  3  1  0  0  0  0  0  0  0  3
        14  0  0  0  0  0  0  1  0  0  0  0  1  0 17  4  0  0  0  0  1  2  0
        15  1  0  0  0  1  0  0  0  0  0  0  0  2  4  5  1  1  1  3  1  3  0
        16  0  1  3  0  0  0  0  0  0  0  0  2  0  0  1 13  0  0  0  0  0  0
        17  0  0  0  0  1  0  1  0  0  0  0  0  1  1  0  0  6  9  0  3  0  0
        18  0  0  1  0  3  0  0  1  0  0  0  0  0  2  0  0  4  6  0  0  0  0
        19  0  0  0  0  1  0  3  0  0  0  0  0  0  0  0  0  1  1  4  2  3  3
        20  0  1  0  0  0  0  0  2  1  0  0  1  0  0  0  1  3  2  2  4  3  3
        21  0  0  0  0  0  1  3  1  0  0  2  0  1  3  0  1  2  0  1  3  4  5
        22  0  0  2  1  0  0  1  0  1  1  2  1  2  0  0  0  0  0  2  2  2  2

Overall Statistics

               Accuracy : 0.424
                 95% CI : (0.3802, 0.4687)
    No Information Rate : 0.07
    P-Value [Acc > NIR] : < 2.2e-16

                  Kappa : 0.3963

 Mcnemar's Test P-Value : NA

Statistics by Class:

                     Class: 1 Class: 2 Class: 3 Class: 4 Class: 5 Class: 6 Class: 7
Sensitivity            0.6667   0.6923   0.5143   0.5263   0.6471   0.5789   0.5714
Specificity            0.9875   0.9852   0.9742   0.9584   0.9731   0.9792   0.9767
Pos Pred Value         0.7000   0.7200   0.6000   0.3333   0.4583   0.5238   0.5926
Neg Pred Value         0.9854   0.9832   0.9638   0.9809   0.9874   0.9833   0.9746
Prevalence             0.0420   0.0520   0.0700   0.0380   0.0340   0.0380   0.0560
Detection Rate         0.0280   0.0360   0.0360   0.0200   0.0220   0.0220   0.0320
Detection Prevalence   0.0400   0.0500   0.0600   0.0600   0.0480   0.0420   0.0540
Balanced Accuracy      0.8271   0.8388   0.7442   0.7424   0.8101   0.7791   0.7741
                     Class: 8 Class: 9 Class: 10 Class: 11 Class: 12 Class: 13
Sensitivity            0.7200   0.7000    0.3333    0.2000    0.2143    0.1364
Specificity            0.9832   0.9812    0.9687    0.9663    0.9767    0.9665
Pos Pred Value         0.6923   0.6087    0.3182    0.2381    0.3529    0.1579
Neg Pred Value         0.9852   0.9874    0.9707    0.9582    0.9545    0.9605
Prevalence             0.0500   0.0400    0.0420    0.0500    0.0560    0.0440
Detection Rate         0.0360   0.0280    0.0140    0.0100    0.0120    0.0060
Detection Prevalence   0.0520   0.0460    0.0440    0.0420    0.0340    0.0380
```

```
Balanced Accuracy      0.8516   0.8406    0.6510    0.5832    0.5955    0.5514
                     Class: 14 Class: 15 Class: 16 Class: 17 Class: 18 Class: 19
Sensitivity             0.5000    0.2941    0.6842    0.2609    0.2727    0.2222
Specificity             0.9807    0.9627    0.9854    0.9665    0.9770    0.9710
Pos Pred Value          0.6538    0.2174    0.6500    0.2727    0.3529    0.2222
Neg Pred Value          0.9641    0.9748    0.9875    0.9644    0.9669    0.9710
Prevalence              0.0680    0.0340    0.0380    0.0460    0.0440    0.0360
Detection Rate          0.0340    0.0100    0.0260    0.0120    0.0120    0.0080
Detection Prevalence    0.0520    0.0460    0.0400    0.0440    0.0340    0.0360
Balanced Accuracy       0.7403    0.6284    0.8348    0.6137    0.6249    0.5966
                     Class: 20 Class: 21 Class: 22
Sensitivity             0.2353    0.2000   0.08333
Specificity             0.9607    0.9521   0.96429
Pos Pred Value          0.1739    0.1481   0.10526
Neg Pred Value          0.9727    0.9662   0.95426
Prevalence              0.0340    0.0400   0.04800
Detection Rate          0.0080    0.0080   0.00400
Detection Prevalence    0.0460    0.0540   0.03800
Balanced Accuracy       0.5980    0.5760   0.52381
```

Hide

```
cat("The accuracy for gbm model is", mean(dat_test$emotion_idx==pred.class)*100,
"%.\n")
```

```
The accuracy for gbm model is 42.4 %.
```

Summarize Running Time

Hide

```
tm_test_gbm<-system.time(pred_gbm)
cat("Time for constructing training features=", tm_feature_train[1], "s \n")
```

```
Time for constructing training features= 0.891 s
```

Hide

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

```
Time for constructing testing features= 0.167 s
```

Hide

```
cat("Time for training model=", gbm.fit[[2]][1], "s \n")
```

```
Time for training model= 805.146 s
```

Hide

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

```
Time for testing model= 0 s
```

```
!pip install pyreadr
!pip install PyDrive
```

```
import numpy as np
import os
import pandas as pd
import time
import xgboost as xgb
import pyreadr
import scipy.io as scio
from collections import OrderedDict
from google.colab import auth
from oauth2client.client import GoogleCredentials
from pydrive.auth import GoogleAuth
from pydrive.drive import GoogleDrive
from scipy.io import loadmat
from scipy.spatial.distance import cdist
from sklearn import datasets
from sklearn import metrics
from sklearn.decomposition import PCA
from sklearn.svm import SVC
from sklearn.model_selection import train_test_split,GridSearchCV
from sklearn.metrics import accuracy_score,classification_report
from sklearn.ensemble import BaggingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import scale
```

ADVANCED MODEL

Instruction

1. Upload training data file in the google drive
2. Get shareable link of the data file
3. Get file ID (the file ID can be obstained from the link.)
4. replace the file ID in corresponding code. (Detailed instruction also come with the code throughout the file.)

Part 0: set up control and work directories, extract paths.

```
####Authenticate the google drive account
auth.authenticate_user()
gauth = GoogleAuth()
gauth.credentials = GoogleCredentials.get_application_default()
drive = GoogleDrive(gauth)
```

```
from sklearn.model_selection import train_test_split, GridSearchCV #Perforing grid se
import matplotlib.pylab as plt
%matplotlib inline
from matplotlib.pylab import rcParams
rcParams['figure.figsize'] = 12, 4
```

Part 1: Import Data

```
#####get the file shareable link = https://drive.google.com/open?id=1oliwM2-sH8CD_3q3
#####The file ID is the letter after "id=".

#####please replace the id of your file
download = drive.CreateFile({'id': '1oliwM2-sH8CD_3q3yUbLF836U9A1-Lc0'})
download.GetContentFile('train_set.zip')
!unzip train_set.zip

#####Run the code, and go to the URL in th output, enter the authorization code, done
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call dr:
```

I. Our Advanced Model

We are using PCA with Bagging-SVM as our Advacned Model

Notation on These functions:

1. extract_mat():

   TAKES IN a list returned by a loadmat function.

   RETURN an array that have all the points in the mat file.

2. get_f():

   TAKES IN a direction that contains a *single* .mat file.

   RETURN an ndarray contains the pairwise euclidian distance between the coordinate contains i

3. feature_extraction():

   TAKES IN a direction that contains the direction that contains *all* the .mat file for the train.

RETURNS a ndarray contains the train_x with features set as pairwise euclidian distance betwe
contains in the .mat file.

4. f_pca():

   TAKES IN a ndarray contains all the train_x.

   RETURNS a ndarray contains decomposed x and the decompositon model.

5. BaggingSVM_w_pca():

   TAKES IN two ndarrays as train_x(without decomposition) and train_y.

   RETURNS the SVM-Bagging model trained with decomposed-train_x and train_y.

6. claim_possible_acc_BSVM():

   TAKES IN three arguments which is the direction that contains _all_the .mat file for x, the directi
   named *label.csv* that have a column named as emotion_idx as the train_y.

   RETURNS the possible accuracy of the Logistic-Bagging model.

```
def extract_mat(x):
    v = list(x.keys())[-1]
    return x[v]

def get_f(file_dir):
    '''Argument:
        file_dir: The whole direction contain the exact mat file

       Return:
        a np.array contains the featrues of single X'''
    a = extract_mat(loadmat(file_dir))
    b = cdist(a, a)
    r = b[np.triu_indices(b.shape[1], 1)].flatten()
    return r

def f_pca(x):
    my_pca = PCA(n_components = 130)
    new_X = my_pca.fit_transform(x)
    compo = sum(my_pca.explained_variance_ratio_)*100
    print(f'The Decomposition take up {compo: 0.4f}% Information of original Data')

    return new_X, my_pca

def feature_extraction(dir_x):
    if (dir_x[-1] != '/'):
        dir_x = dir_x + '/'

    fea_start = time.time()
```

```
    filenames = list(os.listdir(dir_x))
    filenames.sort()
    X = np.array(list(map(get_f, ((dir_x + i) for i in filenames))))

    fea_end = time.time()
    fea_time = fea_end - fea_start

    print('Feature Extraction Completed!')
    print(f'Feature Extraction Cost: {fea_time: 0.2f} Seconds')
    return X

def BaggingSVM_w_pca(train_X, train_y):

    train_X, pca_mode = f_pca(train_X)

    start_SVM = time.time()
    S_svm = SVC(C = 0.1,
                kernel = 'linear',
                shrinking = True,
                decision_function_shape = 'ovo')
    Bagg_SVM = BaggingClassifier(S_svm,
                                 n_estimators = 80,
                                 n_jobs = 5,
                                 bootstrap_features = True)
    Bagg_SVM.fit(train_X, train_y)
    end_SVM = time.time()

    Train_time = end_SVM - start_SVM
    print(f'The Time for train is: {Train_time: 0.2f} Seconds')
    return Bagg_SVM, pca_mode


def claim_possible_acc_BSVM(X_path, y_path, n_iter = 1):
    X = feature_extraction(X_path)
    y = pd.read_csv(y_path).emotion_idx

    accs = []
    for i in range(n_iter):
        trainx, testx, trainy, testy = train_test_split(X, y, test_size = .2)
        model, pca_mode= BaggingSVM_w_pca(trainx, trainy)
        new_testx = pca_mode.transform(testx)
        testy_hat = model.predict(new_testx)
        accs.append(accuracy_score(testy, testy_hat))
    ret = np.mean(accs)*100
    return print(f'Our model should have about {ret: 0.4f}% accuracy')


# This line can output the Claimed Accuracy
# You don't really need run it
claim_possible_acc_BSVM('train_set/points', 'train_set/label.csv', 15)
```

```
Feature Extraction Completed!
Feature Extraction Cost:  0.89 Seconds
The Decomposition take up  99.9079% Information of original Data
The Time for train is:  132.73 Seconds
The Decomposition take up  99.9084% Information of original Data
The Time for train is:  133.39 Seconds
The Decomposition take up  99.9082% Information of original Data
The Time for train is:  150.10 Seconds
The Decomposition take up  99.9053% Information of original Data
The Time for train is:  150.06 Seconds
The Decomposition take up  99.9091% Information of original Data
The Time for train is:  141.05 Seconds
The Decomposition take up  99.9090% Information of original Data
The Time for train is:  126.75 Seconds
The Decomposition take up  99.9083% Information of original Data
The Time for train is:  145.79 Seconds
The Decomposition take up  99.9084% Information of original Data
The Time for train is:  148.31 Seconds
The Decomposition take up  99.9078% Information of original Data
The Time for train is:  139.24 Seconds
The Decomposition take up  99.9133% Information of original Data
The Time for train is:  142.79 Seconds
The Decomposition take up  99.9082% Information of original Data
The Time for train is:  125.05 Seconds
The Decomposition take up  99.9081% Information of original Data
The Time for train is:  125.84 Seconds
The Decomposition take up  99.9074% Information of original Data
The Time for train is:  148.81 Seconds
The Decomposition take up  99.9096% Information of original Data
The Time for train is:  133.39 Seconds
The Decomposition take up  99.9078% Information of original Data
The Time for train is:  147.61 Seconds
Our model should have about  52.2533% accuracy
```

After doing some test, we can claim that Our Advanced Model would have 52.25% Accuracy.

The train time for our Advanced Model is about 110 seconds

You can use The Code below to train the model on the whole train data set

```
X = feature_extraction('train_set/points')
y = pd.read_csv('train_set/label.csv').emotion_idx
advanced_model, pca_sub_model = BaggingSVM_w_pca(X, y)
```

```
Feature Extraction Completed!
Feature Extraction Cost:  0.82 Seconds
The Decomposition take up  1.00% Information of original Data
The Time for train is:  322.94 Seconds
```

Then, you can use the code below to test on the test set

```
# X_test = feature_extraction('''your test set direction''')
```

```
# X_test = feature_extraction(' your test set direction ')
# X_test_decomp = pca_sub_model.transform(X_test)
# y_predict = advanced_model.predict(X_test_decomp)
```

II. XGBOOST Model

Part 0: Feature Extration and Train/Test Split

```
##### Importing the fidusial points
import scipy.io as scio
from collections import OrderedDict
points_path = 'train_set/points'
points = [p for p in sorted(os.listdir(points_path))]
all_points = []
for p in points:
  poiFile = os.path.join(points_path, p)
  poi = scio.loadmat(poiFile)
  poi = OrderedDict(poi)
  all_points.append(poi.popitem()[1])
y = pd.read_csv('train_set/label.csv')['emotion_idx']

print('success')
```

```
success
```

```
##### Calculating pairwise distance
pair_dist = []
for i in range(len(all_points)):
  pair_dist.append(metrics.pairwise_distances(all_points[i])[np.triu_indices(78)])

##### Split train_set & test_set
points_train, points_test, y_train, y_test = train_test_split(pair_dist, y, random_st
print('success')
```

```
success
```

```
##### Feature Extration/Calculating pairwise distance and the time for feature extrat
import time

allpoints_train, allpoints_test, y_train, y_test = train_test_split(all_points, y, ra
print('success')


train_pair_dist = []
for i in range(len(allpoints_train)):
  pair_dist.append(metrics.pairwise_distances(allpoints_train[i])[np.triu_indices(78)
```

```
test_pair_dist = []
for i in range(len(allpoints_test)):
  pair_dist.append(metrics.pairwise_distances(allpoints_test[i])[np.triu_indices(78)]


start = time.time()
pair_dist = []
for i in range(len(all_points)):
  pair_dist.append(metrics.pairwise_distances(all_points[i])[np.triu_indices(78)])
finish = time.time()
print("Time on feature selection done in %0.3fs" % (finish-start))

start = time.time()
train_pair_dist = []
for i in range(len(allpoints_train)):
  pair_dist.append(metrics.pairwise_distances(allpoints_train[i])[np.triu_indices(78)
finish = time.time()
print("Time on feature selection training set done in %0.3fs" % (finish-start))

start = time.time()
test_pair_dist = []
for i in range(len(allpoints_test)):
  pair_dist.append(metrics.pairwise_distances(allpoints_test[i])[np.triu_indices(78)]
finish = time.time()
print("Time on feature selection test set done in %0.3fs" % (finish-start))
```

```
success
Time on feature selection done in 1.098s
Time on feature selection training set done in 0.922s
Time on feature selection test set done in 0.277s
```

Part 1: XGBoost Training

```
import xgboost as xgb
from sklearn.model_selection import GridSearchCV
from xgboost.sklearn import XGBClassifier
import time
import numpy as np


def modelfit(alg, dtrain, predictors, cv_folds=10):
  #Fit the algorithm on the data
  alg.fit(dtrain, predictors)

  #Predict training set:
  dtrain_predictions = alg.predict(dtrain)
  dtrain_predprob = alg.predict_proba(dtrain)[:,1]

  #Print model report:
  print("\nModel Report")
```

```
    print("Accuracy : %.4g" % metrics.accuracy_score(predictors, dtrain_predictions))
```

Part 2:XGBoost Default setting

```
####XGBOOST base model with default setting
start = time.time()
xgb_base = XGBClassifier(
 objective= 'multi:softmax',
 num_class= 22,
 seed=1000)

modelfit(xgb_base, np.array(points_train), np.array(y_train))
finish = time.time()
print("Prediction on train_set done in %0.3fs" % (finish-start))
```

```
Model Report
Accuracy : 1
Prediction on train_set done in 807.874s
```

```
start = time.time()
preds = xgb_base.predict(points_test)
acc_preds = metrics.accuracy_score(preds, y_test)
finish = time.time()
print("Prediction on test_set done in %0.3fs" % (finish - start))
print("Test_set accurarcy is %0.3f" %acc_preds)
```

```
Prediction on test_set done in 0.740s
Test_set accurarcy is 0.482
```

Tuning Process(comment it out because the process is time comsuming)

```
####tune hyperparameter

## tune the max_depth and min_child_weight parameter
# param_test1 = {
#  'max_depth': range(4,5,6),
#  'min_child_weight': range(4,5,6)
# }
# gsearch1 = GridSearchCV(xgb_base, param_grid = param_test1, scoring ='accuracy', cv
# gsearch1.fit(np.array(points_train), np.array(y_train))
# best_parameters1 = gsearch1.best_estimator_.get_params()
# for param_name in sorted(param_test1.keys()):
#     print("\t%s: %r" % (param_name, best_parameters1[param_name]))

## use the best_parameter above to xgb2
# start = time.time()
```

```
# xgb2 = XGBClassifier(
#  objective= 'multi:softmax',
#  num_class= 22,
#  max_depth=4,
#  min_child_weight=4,
#  seed=1000)
# modelfit(xgb2, np.array(points_train), np.array(y_train))
# finish = time.time()
# print("Prediction on train_set done in %0.3fs" % (finish-start))
# start = time.time()
# preds = xgb2.predict(points_test)
# acc_pred = metrics.accuracy_score(preds, y_test)
# finish = time.time()
# print("Prediction on test_set done in %0.3fs" % (finish - start))
# print("Test_set accurarcy is %0.3f" %acc_pred)

## However, the accuracy is lower than that of the base model, so we keep the same pa
## as before, and tune other parameters.

## tune the gamma parameter
# param_test2 = {
#  'gamma':[i/10.0 for i in range(0,5)]
# }
# gsearch2 = GridSearchCV(xgb1, param_grid = param_test2, scoring ='accuracy', cv = 5
# gsearch2.fit(np.array(points_train), np.array(y_train))
# best_parameters2 = gsearch2.best_estimator_.get_params()
# for param_name in sorted(param_test2.keys()):
#     print("\t%s: %r" % (param_name, best_parameters2[param_name]))

# start = time.time()
# xgb3 = XGBClassifier(
#  objective= 'multi:softmax',
#  num_class= 22,
#  gamma=0.4,
#  seed=1000)

# modelfit(xgb3, np.array(points_train), np.array(y_train))
# finish = time.time()
# print("Prediction on train_set done in %0.3fs" % (finish-start))

# start = time.time()
# preds = xgb3.predict(points_test)
# acc_pred = metrics.accuracy_score(preds, y_test)
# finish = time.time()
# print("Prediction on test_set done in %0.3fs" % (finish - start))
# print("Test_set accurarcy is %0.3f" %acc_pred)

## However, the accuracy is lower than that of the base model, so we keep the same pa
## as before, and tune other parameters.

## tune the subsample and colsample_bytree parameters
#param_test = {
```

```
     #'subsample':[i/10.0 for i in range(6,10)],
     #'colsample_bytree':[i/10.0 for i in range(6,10)]
    #}
    #gsearch = GridSearchCV(xgb_base, param_grid = param_test, scoring ='accuracy', cv =
    #gsearch.fit(np.array(points_train), np.array(y_train))
    #best_parameters = gsearch.best_estimator_.get_params()
    #for param_name in sorted(param_test.keys()):
        #print("\t%s: %r" % (param_name, best_parameters[param_name]))

    # start = time.time()
    # xgb4 = XGBClassifier(
    #  objective = 'multi:softmax',
    #  num_class = 22,
    #  seed = 1000,
    #  colsample_bytree=0.6,
    #  subsample=0.7)

    # modelfit(xgb4, np.array(points_train), np.array(y_train))
    # finish = time.time()
    # print("Prediction on train_set done in %0.3fs" % (finish-start))

    # start = time.time()
    # preds = xgb4.predict(points_test)
    # acc_pred = metrics.accuracy_score(preds, y_test)
    # finish = time.time()
    # print("Prediction on test_set done in %0.3fs" % (finish - start))
    # print("Test_set accurarcy is %0.3f" %acc_pred)

    ## We use the best parameters above because the accuracy increases and the prediction
    ## decreases. Then, we tune other parameter based on the xgb4.

    ##tune reg_alpha parameter
    #param_test4 = {
    #  'reg_alpha':[1e-5, 1e-2, 0.1, 1, 100]
    # }
    # gsearch4 = GridSearchCV(xgb4, param_grid = param_test4, scoring ='accuracy', cv = 5
    # gsearch4.fit(np.array(points_train), np.array(y_train))
    # best_parameters4 = gsearch4.best_estimator_.get_params()
    # for param_name in sorted(param_test4.keys()):
    #     print("\t%s: %r" % (param_name, best_parameters4[param_name]))

    # start = time.time()
    # xgb5 = XGBClassifier(
    #  objective= 'multi:softmax',
    #  num_class= 22,
    #  seed=1000,
    #  colsample_bytree=0.7,
    #  subsample=0.6,
    #  reg_alpha=1)

    # modelfit(xgb5, np.array(points_train), np.array(y_train))
    # finish = time.time()
```

```
# print("Prediction on train_set done in %0.3fs" % (finish-start))

# start = time.time()
# preds = xgb5.predict(points_test)
# acc_pred = metrics.accuracy_score(preds, y_test)
# finish = time.time()
# print("Prediction on test_set done in %0.3fs" % (finish - start))
# print("Test_set accurarcy is %0.3f" %acc_pred)

##Since the best parameter of reg_alpha=1e-05, and the accuracy is so close to that o
##model, we decide to use the xgb5 as our final model.
```

Part 3: The improved XGboost model after tuning the parameters

```
####XGBOOSTING improved model
start = time.time()
xgb5 = XGBClassifier(
 objective= 'multi:softmax',
 num_class= 22,
 seed=1000,
 colsample_bytree=0.6,
 subsample=0.7,
 reg_alpha=1)

modelfit(xgb5, np.array(points_train), np.array(y_train))
finish = time.time()
print("Prediction on train_set done in %0.3fs" % (finish-start))
```

```
Model Report
Accuracy : 0.9995
Prediction on train_set done in 483.013s
```

```
start = time.time()
preds = xgb5.predict(points_test)
acc_pred = metrics.accuracy_score(preds, y_test)
finish = time.time()
print("Prediction on test_set done in %0.3fs" % (finish - start))
print("Test_set accurarcy is %0.3f" %acc_pred)
```

```
Prediction on test_set done in 0.770s
Test_set accurarcy is 0.502
```

III. BAGGING-LOG MODEL

Notation on These functions:

1. extract_mat():

   TAKES IN a list returned by a loadmat function.

   RETURN an array that have all the points in the mat file.

2. get_f():

   TAKES IN a direction that contains a *single* .mat file.

   RETURN an ndarray contains the pairwise euclidian distance between the coordinate contains i

3. feature_extraction():

   TAKES IN a direction that contains the direction that contains *all* the .mat file for the train.

   RETURNS a ndarray contains the train_x with features set as pairwise euclidian distance betwe contains in the .mat file.

4. f_pca():

   TAKES IN a ndarray contains all the train_x.

   RETURNS a ndarray contains decomposed x and the decompositon model.

5. BaggingLR_w_pca():

   TAKES IN two ndarrays as train_x(without decomposition) and train_y.

   RETURNS the Logistic-Bagging model trained with decomposed-train_x and train_y.

6. claim_possible_acc_BL():

   TAKES IN three arguments which is the direction that contains _all_the .mat file for x, the directi named *label.csv* that have a column named as emotion_idx as the train_y.

   RETURNS the possible accuracy of the Logistic-Bagging model.

```
def extract_mat(x):
    v = list(x.keys())[-1]
    return x[v]

def get_f(file_dir):
    '''Argument:
        file_dir: The whole direction contain the exact mat file

       Return:
        a np.array contains the featrues of single X'''
    a = extract_mat(loadmat(file_dir))
    b = cdist(a, a)
    r = b[np.triu_indices(b.shape[1], 1)].flatten()
    return r
```

```
def feature_extraction(dir_x):
    if (dir_x[-1] != '/'):
        dir_x = dir_x + '/'

    fea_start = time.time()

    filenames = list(os.listdir(dir_x))
    filenames.sort()
    X = np.array(list(map(get_f, ((dir_x + i) for i in filenames))))

    fea_end = time.time()
    fea_time = fea_end - fea_start

    print('Feature Extraction Completed!')
    print(f'Feature Extraction Cost: {fea_time: 0.2f} Seconds')
    return X

def f_pca(x):
    my_pca = PCA(n_components = 130)
    new_X = my_pca.fit_transform(x)
    compo = sum(my_pca.explained_variance_ratio_)*100
    print(f'The Decomposition take up {compo: 0.2f}% Information of original Data')

    return new_X, my_pca

def BaggingLR_w_pca(train_X, train_y):

    train_X, pca_mode = f_pca(train_X)

    start_lr = time.time()
    lr = LogisticRegression(C = 1,
                            penalty = 'l2',
                            fit_intercept = False)
    Bag_lr = BaggingClassifier(lr,
                               n_estimators = 70,
                               n_jobs = 5,
                               bootstrap_features = True,
                               verbose = 7)
    Bag_lr.fit(train_X, train_y)
    end_lr = time.time()

    Train_time = end_lr - start_lr
    print(f'The Time for train is: {Train_time: 0.2f} Seconds')
    return Bag_lr, pca_mode


def claim_possible_acc_BL(X_path, y_path, n_iter = 1):
    X = feature_extraction(X_path)
    y = pd.read_csv(y_path).emotion_idx

    accs = []
    for i in range(n_iter):
```

```
        trainx, testx, trainy, testy = train_test_split(X, y, test_size = .2)
        model, pca_mode= BaggingLR_w_pca(trainx, trainy)
        new_testx = pca_mode.transform(testx)
        testy_hat = model.predict(new_testx)
        accs.append(accuracy_score(testy, testy_hat))
    ret = np.mean(accs)*100
    return print(f'The Bagging-Logistic model should have about {ret: 0.4f}% accuracy

claim_possible_acc_BL('train_set/points', 'train_set/label.csv',10)
```

```
Feature Extraction Completed!
Feature Extraction Cost:  0.88 Seconds
The Decomposition take up  99.91% Information of original Data
[Parallel(n_jobs=5)]: Using backend LokyBackend with 5 concurrent workers.
[Parallel(n_jobs=5)]: Done   2 out of   5 | elapsed:   25.1s remaining:   37.7s
[Parallel(n_jobs=5)]: Done   3 out of   5 | elapsed:   25.1s remaining:   16.8s
[Parallel(n_jobs=5)]: Done   5 out of   5 | elapsed:   25.8s remaining:    0.0s
[Parallel(n_jobs=5)]: Done   5 out of   5 | elapsed:   25.8s finished
[Parallel(n_jobs=5)]: Using backend LokyBackend with 5 concurrent workers.
[Parallel(n_jobs=5)]: Done   2 out of   5 | elapsed:    0.1s remaining:    0.1s
[Parallel(n_jobs=5)]: Done   3 out of   5 | elapsed:    0.1s remaining:    0.1s
[Parallel(n_jobs=5)]: Done   5 out of   5 | elapsed:    0.1s remaining:    0.0s
[Parallel(n_jobs=5)]: Done   5 out of   5 | elapsed:    0.1s finished
The Time for train is:  25.76 Seconds
The Decomposition take up  99.91% Information of original Data
[Parallel(n_jobs=5)]: Using backend LokyBackend with 5 concurrent workers.
[Parallel(n_jobs=5)]: Done   2 out of   5 | elapsed:   25.5s remaining:   38.3s
[Parallel(n_jobs=5)]: Done   3 out of   5 | elapsed:   25.5s remaining:   17.0s
[Parallel(n_jobs=5)]: Done   5 out of   5 | elapsed:   25.7s remaining:    0.0s
[Parallel(n_jobs=5)]: Done   5 out of   5 | elapsed:   25.7s finished
[Parallel(n_jobs=5)]: Using backend LokyBackend with 5 concurrent workers.
The Time for train is:  25.66 Seconds
[Parallel(n_jobs=5)]: Done   2 out of   5 | elapsed:    0.1s remaining:    0.1s
[Parallel(n_jobs=5)]: Done   3 out of   5 | elapsed:    0.1s remaining:    0.1s
[Parallel(n_jobs=5)]: Done   5 out of   5 | elapsed:    0.1s remaining:    0.0s
[Parallel(n_jobs=5)]: Done   5 out of   5 | elapsed:    0.1s finished
The Decomposition take up  99.91% Information of original Data
[Parallel(n_jobs=5)]: Using backend LokyBackend with 5 concurrent workers.
[Parallel(n_jobs=5)]: Done   2 out of   5 | elapsed:   24.9s remaining:   37.4s
[Parallel(n_jobs=5)]: Done   3 out of   5 | elapsed:   25.0s remaining:   16.7s
[Parallel(n_jobs=5)]: Done   5 out of   5 | elapsed:   25.3s remaining:    0.0s
[Parallel(n_jobs=5)]: Done   5 out of   5 | elapsed:   25.3s finished
[Parallel(n_jobs=5)]: Using backend LokyBackend with 5 concurrent workers.
[Parallel(n_jobs=5)]: Done   2 out of   5 | elapsed:    0.1s remaining:    0.1s
[Parallel(n_jobs=5)]: Done   3 out of   5 | elapsed:    0.1s remaining:    0.1s
[Parallel(n_jobs=5)]: Done   5 out of   5 | elapsed:    0.1s remaining:    0.0s
[Parallel(n_jobs=5)]: Done   5 out of   5 | elapsed:    0.1s finished
The Time for train is:  25.30 Seconds
The Decomposition take up  99.91% Information of original Data
[Parallel(n_jobs=5)]: Using backend LokyBackend with 5 concurrent workers.
[Parallel(n_jobs=5)]: Done   2 out of   5 | elapsed:   25.3s remaining:   38.0s
[Parallel(n_jobs=5)]: Done   3 out of   5 | elapsed:   25.3s remaining:   16.9s
[Parallel(n_jobs=5)]: Done   5 out of   5 | elapsed:   25.8s finished
[Parallel(n_jobs=5)]: Done   5 out of   5 | elapsed:   25.8s remaining:    0.0s
[Parallel(n_jobs=5)]: Using backend LokyBackend with 5 concurrent workers.
[Parallel(n_jobs=5)]: Done   2 out of   5 | elapsed:    0.1s remaining:    0.2s
[Parallel(n_jobs=5)]: Done   3 out of   5 | elapsed:    0.1s remaining:    0.1s
[Parallel(n_jobs=5)]: Done   5 out of   5 | elapsed:    0.1s remaining:    0.0s
[Parallel(n_jobs=5)]: Done   5 out of   5 | elapsed:    0.1s finished
The Time for train is:  25.76 Seconds
The Decomposition take up  99.91% Information of original Data
[Parallel(n_jobs=5)]: Using backend LokyBackend with 5 concurrent workers.
[Parallel(n_jobs=5)]: Done   2 out of   5 | elapsed:   25.0s remaining:   37.5s
[Parallel(n_jobs=5)]: Done   3 out of   5 | elapsed:   25.1s remaining:   16.7s
[Parallel(n_jobs=5)]: Done   5 out of   5 | elapsed:   25.3s finished
[Parallel(n_jobs=5)]: Done   5 out of   5 | elapsed:   25.3s remaining:    0.0s
[Parallel(n_jobs=5)]: Using backend LokyBackend with 5 concurrent workers.
```

```
[Parallel(n_jobs=5)]: Done   2 out of   5 | elapsed:    0.1s remaining:    0.1s
[Parallel(n_jobs=5)]: Done   3 out of   5 | elapsed:    0.1s remaining:    0.1s
[Parallel(n_jobs=5)]: Done   5 out of   5 | elapsed:    0.1s remaining:    0.0s
[Parallel(n_jobs=5)]: Done   5 out of   5 | elapsed:    0.1s finished
The Time for train is:  25.27 Seconds
The Decomposition take up  99.91% Information of original Data
[Parallel(n_jobs=5)]: Using backend LokyBackend with 5 concurrent workers.
[Parallel(n_jobs=5)]: Done   2 out of   5 | elapsed:   24.7s remaining:   37.0s
[Parallel(n_jobs=5)]: Done   3 out of   5 | elapsed:   24.8s remaining:   16.5s
[Parallel(n_jobs=5)]: Done   5 out of   5 | elapsed:   25.4s finished
[Parallel(n_jobs=5)]: Done   5 out of   5 | elapsed:   25.4s remaining:    0.0s
[Parallel(n_jobs=5)]: Using backend LokyBackend with 5 concurrent workers.
[Parallel(n_jobs=5)]: Done   2 out of   5 | elapsed:    0.1s remaining:    0.1s
[Parallel(n_jobs=5)]: Done   3 out of   5 | elapsed:    0.1s remaining:    0.1s
[Parallel(n_jobs=5)]: Done   5 out of   5 | elapsed:    0.1s remaining:    0.0s
[Parallel(n_jobs=5)]: Done   5 out of   5 | elapsed:    0.1s finished
The Time for train is:  25.36 Seconds
The Decomposition take up  99.91% Information of original Data
[Parallel(n_jobs=5)]: Using backend LokyBackend with 5 concurrent workers.
[Parallel(n_jobs=5)]: Done   2 out of   5 | elapsed:   24.9s remaining:   37.3s
[Parallel(n_jobs=5)]: Done   3 out of   5 | elapsed:   25.0s remaining:   16.6s
[Parallel(n_jobs=5)]: Done   5 out of   5 | elapsed:   25.2s finished
[Parallel(n_jobs=5)]: Done   5 out of   5 | elapsed:   25.2s remaining:    0.0s
[Parallel(n_jobs=5)]: Using backend LokyBackend with 5 concurrent workers.
[Parallel(n_jobs=5)]: Done   2 out of   5 | elapsed:    0.1s remaining:    0.2s
[Parallel(n_jobs=5)]: Done   3 out of   5 | elapsed:    0.1s remaining:    0.1s
[Parallel(n_jobs=5)]: Done   5 out of   5 | elapsed:    0.1s finished
[Parallel(n_jobs=5)]: Done   5 out of   5 | elapsed:    0.1s remaining:    0.0s
The Time for train is:  25.17 Seconds
The Decomposition take up  99.91% Information of original Data
[Parallel(n_jobs=5)]: Using backend LokyBackend with 5 concurrent workers.
[Parallel(n_jobs=5)]: Done   2 out of   5 | elapsed:   24.6s remaining:   36.9s
[Parallel(n_jobs=5)]: Done   3 out of   5 | elapsed:   24.7s remaining:   16.5s
[Parallel(n_jobs=5)]: Done   5 out of   5 | elapsed:   25.3s finished
[Parallel(n_jobs=5)]: Done   5 out of   5 | elapsed:   25.3s remaining:    0.0s
[Parallel(n_jobs=5)]: Using backend LokyBackend with 5 concurrent workers.
[Parallel(n_jobs=5)]: Done   2 out of   5 | elapsed:    0.1s remaining:    0.1s
[Parallel(n_jobs=5)]: Done   3 out of   5 | elapsed:    0.1s remaining:    0.1s
[Parallel(n_jobs=5)]: Done   5 out of   5 | elapsed:    0.1s finished
[Parallel(n_jobs=5)]: Done   5 out of   5 | elapsed:    0.1s remaining:    0.0s
The Time for train is:  25.29 Seconds
The Decomposition take up  99.91% Information of original Data
[Parallel(n_jobs=5)]: Using backend LokyBackend with 5 concurrent workers.
[Parallel(n_jobs=5)]: Done   2 out of   5 | elapsed:   24.2s remaining:   36.3s
[Parallel(n_jobs=5)]: Done   3 out of   5 | elapsed:   24.3s remaining:   16.2s
[Parallel(n_jobs=5)]: Done   5 out of   5 | elapsed:   25.3s finished
[Parallel(n_jobs=5)]: Done   5 out of   5 | elapsed:   25.3s remaining:    0.0s
[Parallel(n_jobs=5)]: Using backend LokyBackend with 5 concurrent workers.
[Parallel(n_jobs=5)]: Done   2 out of   5 | elapsed:    0.1s remaining:    0.1s
[Parallel(n_jobs=5)]: Done   3 out of   5 | elapsed:    0.1s remaining:    0.1s
[Parallel(n_jobs=5)]: Done   5 out of   5 | elapsed:    0.1s remaining:    0.0s
[Parallel(n_jobs=5)]: Done   5 out of   5 | elapsed:    0.1s finished
The Time for train is:  25.28 Seconds
The Decomposition take up  99.91% Information of original Data
[Parallel(n_jobs=5)]: Using backend LokyBackend with 5 concurrent workers.
[Parallel(n_jobs=5)]: Done   2 out of   5 | elapsed:   24.8s remaining:   37.2s
[Parallel(n_jobs=5)]: Done   3 out of   5 | elapsed:   25.1s remaining:   16.7s
The Time for train is:  25.42 Seconds
```

```
The Bagging-Logistic model should have about  53.7200% accuracy
[Parallel(n_jobs=5)]: Done   5 out of   5 | elapsed:   25.4s remaining:    0.0s
[Parallel(n_jobs=5)]: Done   5 out of   5 | elapsed:   25.4s finished
[Parallel(n_jobs=5)]: Using backend LokyBackend with 5 concurrent workers.
[Parallel(n_jobs=5)]: Done   2 out of   5 | elapsed:    0.1s remaining:    0.1s
[Parallel(n_jobs=5)]: Done   3 out of   5 | elapsed:    0.1s remaining:    0.1s
[Parallel(n_jobs=5)]: Done   5 out of   5 | elapsed:    0.1s finished
[Parallel(n_jobs=5)]: Done   5 out of   5 | elapsed:    0.1s remaining:    0.0s
```

The accuracy of Bagging-Logistic model may be a bit higher(53.6%) than our advanced model but aft model is not as stable as our advanced model.

reference: https://www.cnblogs.com/wj-1314/p/10422159.html