Main Group 8

Amir Idris, Catherine Gao, Eve Washington, Ruize Yu, Yiwen Fang

Baseline model: GBM

Advanced models: SVM, RandomForest, DNN(best), CNN, Xgboost

```
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("xgboost")){
  install.packages("xgboost")
}
```

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

# Install Miniconda (https://docs.conda.io/en/latest/miniconda.html)
if(!require("keras")){
  install.packages("keras")
}

if(!require("tensorflow")){
  install.packages("tensorflow")
  install_tensorflow()
}

use_condaenv("r-tensorflow")
library(keras)
library(tensorflow)

library(R.matlab)
library(readxl)
library(dplyr)
library(EBImage)
library(ggplot2)
library(caret)
library(glmnet)
library(WeightedROC)
library(gbm)
require(xgboost)
library(caret)
```

Step 0 set work directories

```
set.seed(2030)
# 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

```
# run.cv <- TRUE # run cross-validation on the training set
# sample.reweight <- TRUE # run sample reweighting in model training
# 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

sample.reweight <- TRUE # run sample reweighting in model training
K <- 5  # number of CV folds
run.feature.train <- TRUE # process features for training set
run.feature.test <- TRUE # process features for test set

run.cv_gbm <- FALSE # run GBM cross-validation on the training set
run.test_gbm <- TRUE # run GBM evaluation on an independent test set

run.cv_dnn <- FALSE # run DNN cross-validation on the training set
run.test_dnn <- TRUE # run DNN evaluation on an independent test set
```

Using cross-validation or independent test set evaluation, we compare the performance of models with different specifications. In this part, we tune parameter n.trees and shrinkage for GBM.

```
# GBM parameters
n.trees <- c(500, 1000)
shrinkage <- c(0.01, 0.05)

# DNN parameters
# lrs <- c(0.0002, 0.0005, 0.001)
lrs <- c(0.0002)
```

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.

```
#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")
```

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

```
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{
```

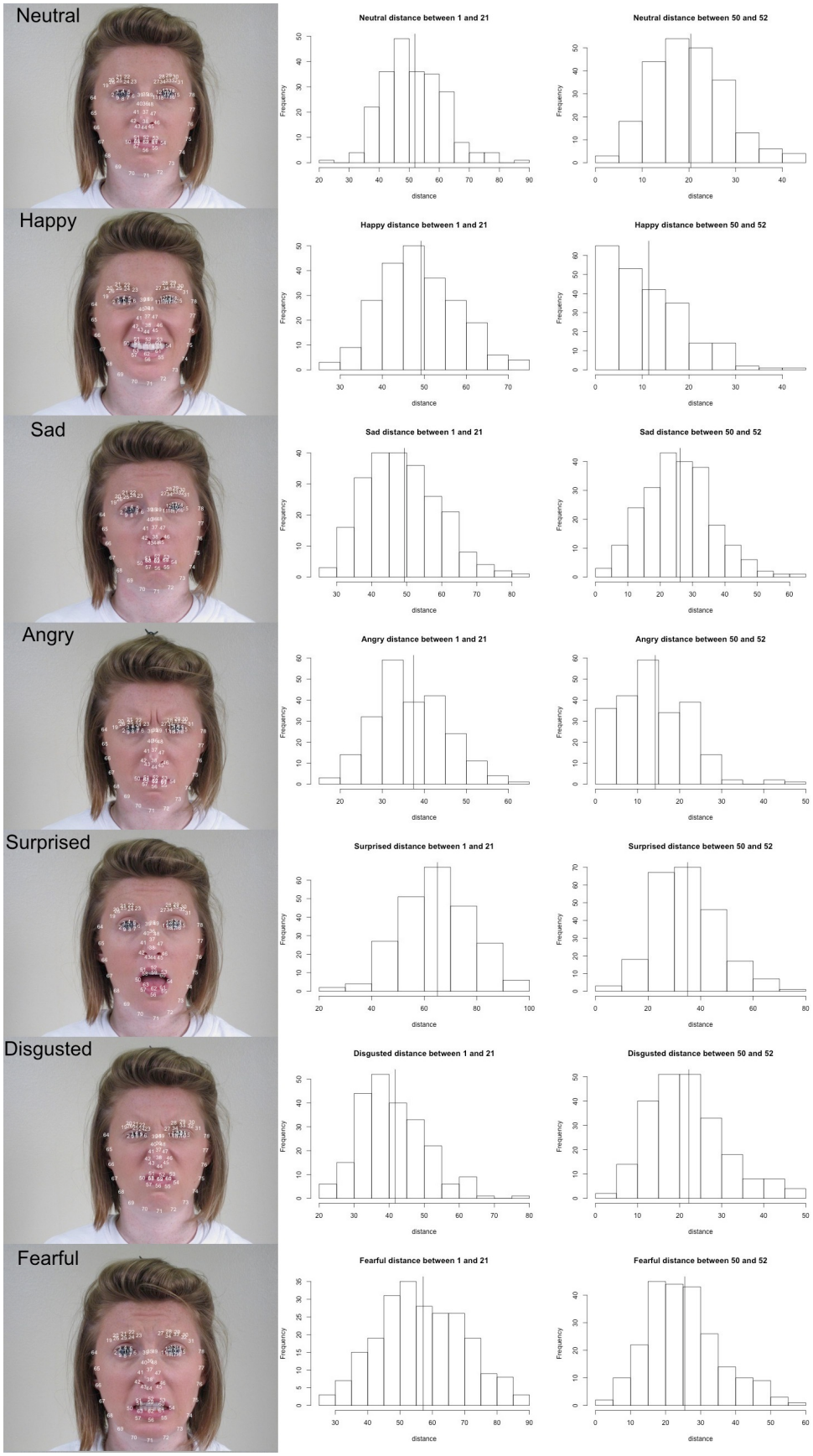

Figure 1: Figure1

```
  load(file="../output/feature_test.RData")
}
```

GBM (Baseline Model)

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

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 part, we use GBM (baseline model) to do classification.

```
source("../lib/train_gbm.R")
source("../lib/test_gbm.R")
```

Model selection with cross-validation

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

```
source("../lib/cross_validation_gbm.R")
feature_train = as.matrix(dat_train[, -6007])
label_train = as.integer(dat_train$label)

if(run.cv_gbm){
  res_cv_gbm <- matrix(0, nrow = length(n.trees) * length(shrinkage), ncol = 6)
  count = 0
  for(i in 1:length(n.trees)){
    for(j in 1:length(shrinkage)){
      count = count + 1
      cat("n.trees =", n.trees[i], "\n")
      cat("shrinkage =", shrinkage[j], "\n")

      res_cv <- cv.function_gbm(features = feature_train, labels = label_train, K,
                                n.trees[i], shrinkage[j], reweight = sample.reweight)

      res_cv_gbm[count,] <- c(n.trees[i], shrinkage[j], res_cv[1], res_cv[2], res_cv[3], res_cv[4])
    }
  }
```

```
  colnames(res_cv_gbm) <- c("n.trees","shrinkage","mean_error", "sd_error", "mean_AUC", "sd_AUC")
  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)

if(run.cv_gbm){
  p1 <- res_cv_gbm %>%
    ggplot(aes(x = n.trees, y = mean_error,
               ymin = mean_error - sd_error, ymax = mean_error + sd_error)) +
    geom_crossbar() +
    facet_wrap(~shrinkage)+
    theme(axis.text.x = element_text(angle = 90, hjust = 1))

  p2 <- res_cv_gbm %>%
    ggplot(aes(x = n.trees, y = mean_AUC,
               ymin = mean_AUC - sd_AUC, ymax = mean_AUC + sd_AUC)) +
    geom_crossbar() +
    facet_wrap(~shrinkage)+
    theme(axis.text.x = element_text(angle = 90, hjust = 1))

  print(p1)
  print(p2)
}
```

- Choose the "best" parameter value

```
# par_n.trees_best <- as.numeric(res_cv_gbm[which.min(res_cv_gbm$mean_error), 1])
# par_shrinkage_best <- as.numeric(res_cv_gbm[which.min(res_cv_gbm$mean_error), 2])

par_n.trees_best <- 500
par_shrinkage_best <- 0.05
```

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

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

# training weights
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])
}

tm_train <- NA
if (sample.reweight){
```

```
  tm_train <- system.time(fit_train_gbm <- train_gbm(feature_train, label_train, w = weight_train, par_
} else {
  tm_train <- system.time(fit_train_gbm <- train_gbm(feature_train, label_train, w = NULL, par_n.trees_
}
save(fit_train_gbm, file="../output/fit_train_gbm.RData")
```

Step 5: Run test on test images

```
tm_test = NA
feature_test <- as.matrix(dat_test[, -6007])
if(run.test_gbm){
  load(file="../output/fit_train_gbm.RData")
  tm_test <- system.time({prob_pred <- test_gbm(fit_train_gbm, feature_test, par_n.trees_best, pred.typ
}
```

- 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])
}

label_pred <- ifelse(prob_pred > 0.5, 1, 0)
label_test <- ifelse(label_test == 2, 1, 0)

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 model:", "GBM with n.trees =" , par_n.trees_best, "and shrinkage =", par_shrinkage
```

```
## The accuracy of model: GBM with n.trees = 500 and shrinkage = 0.05 is 69.07131 %.
```

```
cat("The AUC of model:", "GBM with n.trees =" , par_n.trees_best, "and shrinkage =", par_shrinkage_best
```

```
## The AUC of model: GBM with n.trees = 500 and shrinkage = 0.05 is 0.7874499 .
```

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[3], "s \n")
```

```
## Time for constructing training features= 0.78 s
```

```
cat("Time for constructing testing features=", tm_feature_test[3], "s \n")
```

```
## Time for constructing testing features= 0.22 s
```

```
cat("Time for training model=", tm_train[3], "s \n")
```

```
## Time for training model= 71.25 s
```

```
cat("Time for testing model=", tm_test[3], "s \n")
```

```
## Time for testing model= 0.07 s
```

DNN (Dense Neural Network) (Advanced Model)

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

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 part, we use DNN (advanced model) to do classification.

```
source("../lib/train_dnn.R")
source("../lib/test_dnn.R")
```

Model selection with cross-validation

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

DNN training takes long time. There are almost numberless parameters, layers, regulation functions, to play with. It is unrealistic to use a simple and straightforward cross-validation to illustrate. Instead we put our best DNN model here after tuning.

Visualize cross-validation results. N/A

- Choose the "best" parameter value

```
# par_lr_best <- as.numeric(res_cv_dnn[which.min(res_cv_dnn$mean_error), 1])
par_lr_best = 0.0002
```

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

```
source("../lib/train_dnn.R")
source("../lib/test_dnn.R")

feature_train = as.matrix(dat_train[, -6007])
label_train = as.integer(dat_train$label)
feature_test <- as.matrix(dat_test[, -6007])
label_test <- as.integer(dat_test$label)
label_test <- ifelse(label_test == 2, 1, 0)
weight_test <- rep(NA, length(label_test))

# training weights
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])
}

for (v in unique(label_test)){
  weight_test[label_test == v] = 0.5 * length(label_test) / length(label_test[label_test == v])
}

tm_train <- NA
if (sample.reweight){
  tm_train <- system.time(fit_train_dnn <- train_dnn(feature_train, label_train, w = weight_train, featu
} else {
  tm_train <- system.time(fit_train_dnn <- train_dnn(feature_train, label_train, w = NULL, feature_test
}
save_model_hdf5(fit_train_dnn, "../output/fit_train_dnn.h5")
```

Step 5: Run test on test images

```
tm_test = NA
feature_test <- as.matrix(dat_test[, -6007])
if(run.test_dnn){
  fit_train_dnn <- load_model_hdf5("../output/fit_train_dnn.h5")
  tm_test <- system.time({label_pred <- test_dnn(fit_train_dnn, feature_test, type = "predict_classes")
  prob_pred <- test_dnn(fit_train_dnn, feature_test, type = "predict_proba")
}
```

- 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])
}

label_test <- ifelse(label_test == 2, 1, 0)

accu <- sum(weight_test * (label_pred == label_test)) / sum(weight_test)

# prob_pred <- apply(prob_pred, 1, max)
prob_pred <- prob_pred[, 2]

tpr.fpr <- WeightedROC(prob_pred, label_test, weight_test)
auc <- WeightedAUC(tpr.fpr)


cat("The accuracy of model:", "DNN with lr =", par_lr_best, "is", accu*100, "%.\n")
```

```
## The accuracy of model: DNN with lr = 2e-04 is 74.22463 %.
```

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

```
## The AUC of model: DNN is 0.8543674 .
```

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[3], "s \n")
```

```
## Time for constructing training features= 0.78 s
```

```
cat("Time for constructing testing features=", tm_feature_test[3], "s \n")
```

```
## Time for constructing testing features= 0.22 s
```

```
cat("Time for training model=", tm_train[3], "s \n")
```

```
## Time for training model= 301.55 s
```

```
cat("Time for testing model=", tm_test[3], "s \n")
```

```
## Time for testing model= 0.55 s
```

###Reference - Du, S., Tao, Y., & Martinez, A. M. (2014). Compound facial expressions of emotion. Proceedings of the National Academy of Sciences, 111(15), E1454-E1462.