Main

Group 4

```
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")
}

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

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

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

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

In here is the group's implementation of the baseline model

Step 0 set work directories, extract paths, summarize

```
set.seed(0)
#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: 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 2: 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 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

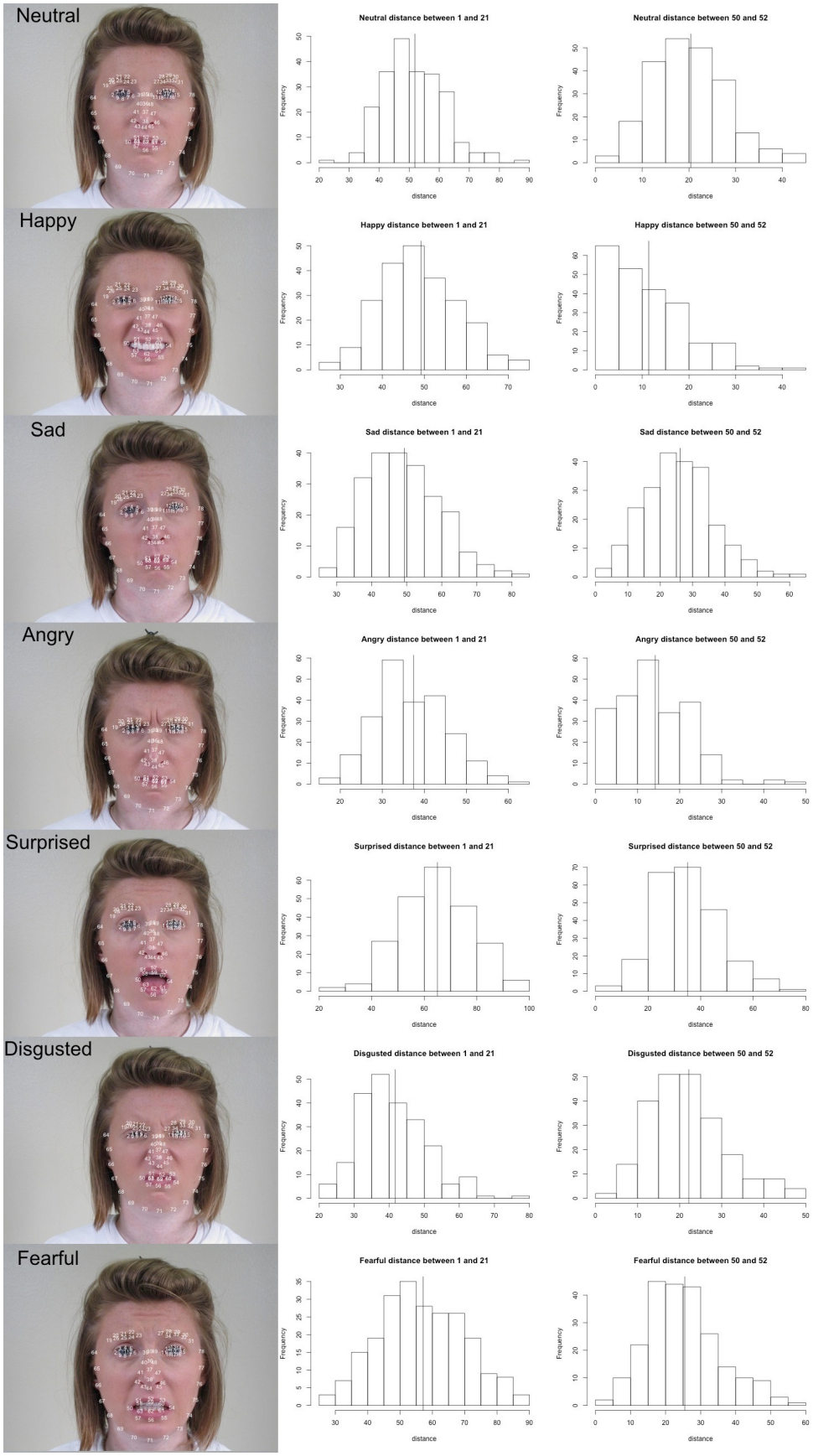


Figure 1: Figure1

- Output: an RData file that contains extracted features and corresponding responses

```
source("../lib/feature.R")
tm_feature_train <- NA
tm_feature_train <- system.time(dat_train <- feature(fiducial_pt_list, train_idx))

tm_feature_test <- NA
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 3: 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

```
#load the relevant files
source("../lib/train.R")
source("../lib/test.R")
load(file="../output/feature_train.RData")
load(file="../output/feature_test.RData")
```

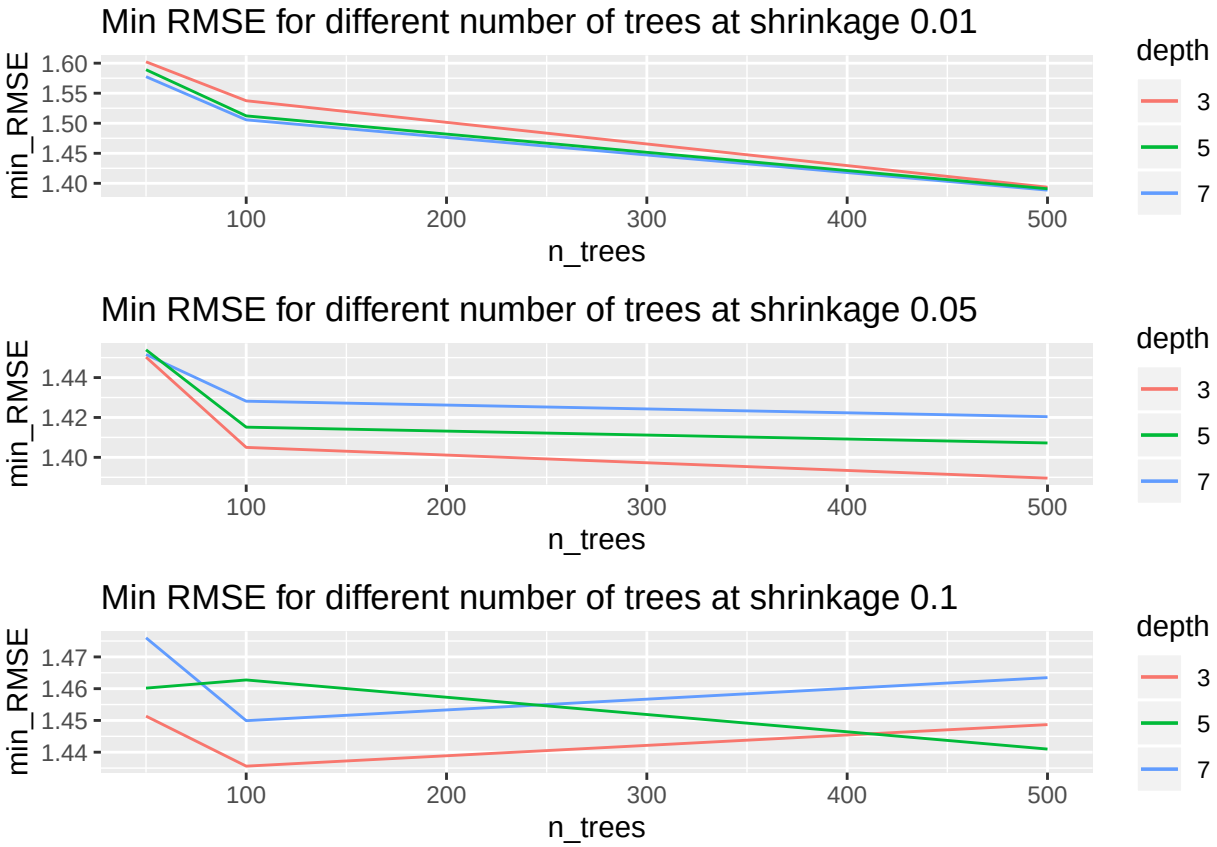
Step 4: Tune the model to get the best hyperparameters for the baseline model. Only ran this cell once, the hyperparameters are stored in output/hyper_grid.csv

```
#learn the optimal parameters for the gbm model
#create a grid of all the hyperparameters
#hyper_grid <- expand.grid(
  #shrinkage = c(.01, .05,.1), #the learning rate
  #interaction.depth = c(3, 5, 7), #maximum number of nodes per tree
  #n.minobsinnode = 0,
  #n_trees = c(50,100,500),#number of trees to try in the model
  #bag.fraction = 0,
  #optimal_trees = 0,
  #min_RMSE = 0
#)

#do a grid search
#for(i in 1:nrow(hyper_grid)) {
  # train model
  #cat(i, " of ", nrow(hyper_grid), " ")
  #params = c(hyper_grid$n_trees[i],hyper_grid$interaction.depth[i],hyper_grid$shrinkage[i],hyper_grid$
```

```
  #train.tune <- train(dat_train,  params)

  # add min training error and trees to grid add training time too
  #hyper_grid$optimal_trees[i] <- which.min(train.tune$valid.error)
  #hyper_grid$min_RMSE[i] <- sqrt(min(train.tune$valid.error))
#}
```

##Step 5: Visualize how the RMSE changes with the different parameters we tuned for

```
#read in csv with learned hyperparameters
hyper_params<-read.csv("../output/hyper_grid.csv", header=TRUE, sep = ",")

#split the table for different levels of shrinkage
data.01 <- hyper_params[hyper_params$shrinkage==0.01,]
data.05 <- hyper_params[hyper_params$shrinkage==0.05,]
data.1 <- hyper_params[hyper_params$shrinkage==0.1,]

#for each table, keep the different depths and their depths
plot.01 = ggplot() +
  geom_line(data = data.01[data.01$interaction.depth==7,], mapping = aes(x = n_trees, y = min_RMSE,colo
  geom_line(data = data.01[data.01$interaction.depth==3,], mapping = aes( x = n_trees, y = min_RMSE, co
  geom_line(data.01[data.01$interaction.depth==5,], mapping = aes(x = n_trees, y = min_RMSE,color="5"))

plot.05 = ggplot() +
  geom_line(data = data.05[data.05$interaction.depth==7,], mapping = aes(x = n_trees, y = min_RMSE,colo
  geom_line(data = data.05[data.05$interaction.depth==3,], mapping = aes( x = n_trees, y = min_RMSE, co
  geom_line(data.05[data.05$interaction.depth==5,], mapping = aes(x = n_trees, y = min_RMSE,color="5"))

plot.1 = ggplot() +
  geom_line(data = data.1[data.1$interaction.depth==7,], mapping = aes(x = n_trees, y = min_RMSE,color=
  geom_line(data = data.1[data.1$interaction.depth==3,], mapping = aes( x = n_trees, y = min_RMSE, colo
  geom_line(data.1[data.1$interaction.depth==5,], mapping = aes(x = n_trees, y = min_RMSE,color="5"))+


#plot the 3 plots
grid.arrange(plot.01, plot.05, plot.1,nrow=3)
```



Step 6: Train the model with the hyperparameters that resulted in the lowest RMSE

```
#using the best learned parameters, train the final model. Commented out for the sake of knitting
best_params = c(259,3,0.05,0,0)
tm_train=NA
tm_train <- system.time(fit_train <- train(dat_train, best_params))
```

```
## Distribution not specified, assuming multinomial ...
```

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

##Step 7: Test the final model

```
#test the final model: and print out graph
source("../lib/test.R")
load(file="../output/feature_test.RData")
tm_test=NA
load(file="../output/fit_train.RData")
tm_test <- system.time(pred <- test(fit_train, dat_test))

#the output is just probabilities,need to convert to categorical
pred <- apply(pred, 1, which.max)

accu <- mean(dat_test$emotion_idx == pred)
cat("The accuracy of model is", accu*100, "%.\n")
```

```
## The accuracy of model is 40.2 %.
```

##Step 4: Evaluate the final model

```
#evaluation
library(caret)
#this line below prints out the evaluation of the model
#confusionMatrix(table(pred, dat_test$emotion_idx))

### Summarize Running Time
#Prediction performance matters, so does the running times for constructing features and #for training
cat("Time for constructing training features=", tm_feature_train[1], "s \n")
```

```
## Time for constructing training features= 0.851 s
```

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

```
## Time for constructing testing features= 0.173 s
```

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

```
## Time for training model= 2010.708 s
```

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

```
## Time for testing model= 14.374 s
```

——»» **Implementation of the baseline model ends here**

###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. - http://uc-r.github.io/gbm_regression