Main

Group 10

```r
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("readxl")){
  install.packages("readxl")
}

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

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

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

if(!require("h2o")){
  install.packages("h2o")
}

library(R.matlab)
library(readxl)
library(dplyr)
library(EBImage)
library(ggplot2)
library(caret)
library(gbm)
library(h2o)
h2o.init()
```

```
##  Connection successful!
##
## R is connected to the H2O cluster:
##     H2O cluster uptime:         1 hours 10 minutes
##     H2O cluster timezone:       America/New_York
##     H2O data parsing timezone:  UTC
##     H2O cluster version:        3.28.0.4
##     H2O cluster version age:    1 month and 8 days
##     H2O cluster name:           H2O_started_from_R_59446_qfd075
##     H2O cluster total nodes:    1
##     H2O cluster total memory:   2.66 GB
##     H2O cluster total cores:    8
##     H2O cluster allowed cores:  8
##     H2O cluster healthy:        TRUE
##     H2O Connection ip:          localhost
##     H2O Connection port:        54321
##     H2O Connection proxy:       NA
##     H2O Internal Security:      FALSE
##     H2O API Extensions:         Amazon S3, Algos, AutoML, Core V3, TargetEncoder, Core V4
##     R Version:                  R version 3.6.3 (2020-02-29)
```

Step 0 set work directories

.

```
set.seed(0)
setwd(dirname(rstudioapi::getActiveDocumentContext()$path))
# 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="")
```

Part1 : Baseline model

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

```
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, we compare the performance of models with different specifications. In this case, we tune parameter t (number of trees) for GBM.

```
t = c(50,100,150,200,250)
model_labels = paste("Number of trees in GBM function:", t)
```

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)
```

Fiducial points are stored in matlab format. In this case, we select inner face points as fiducial points, 33 points in total. 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
n_files <- length(list.files(train_image_dir))
fiducial_pt_list <- lapply(1:n_files, readMat.matrix)
```

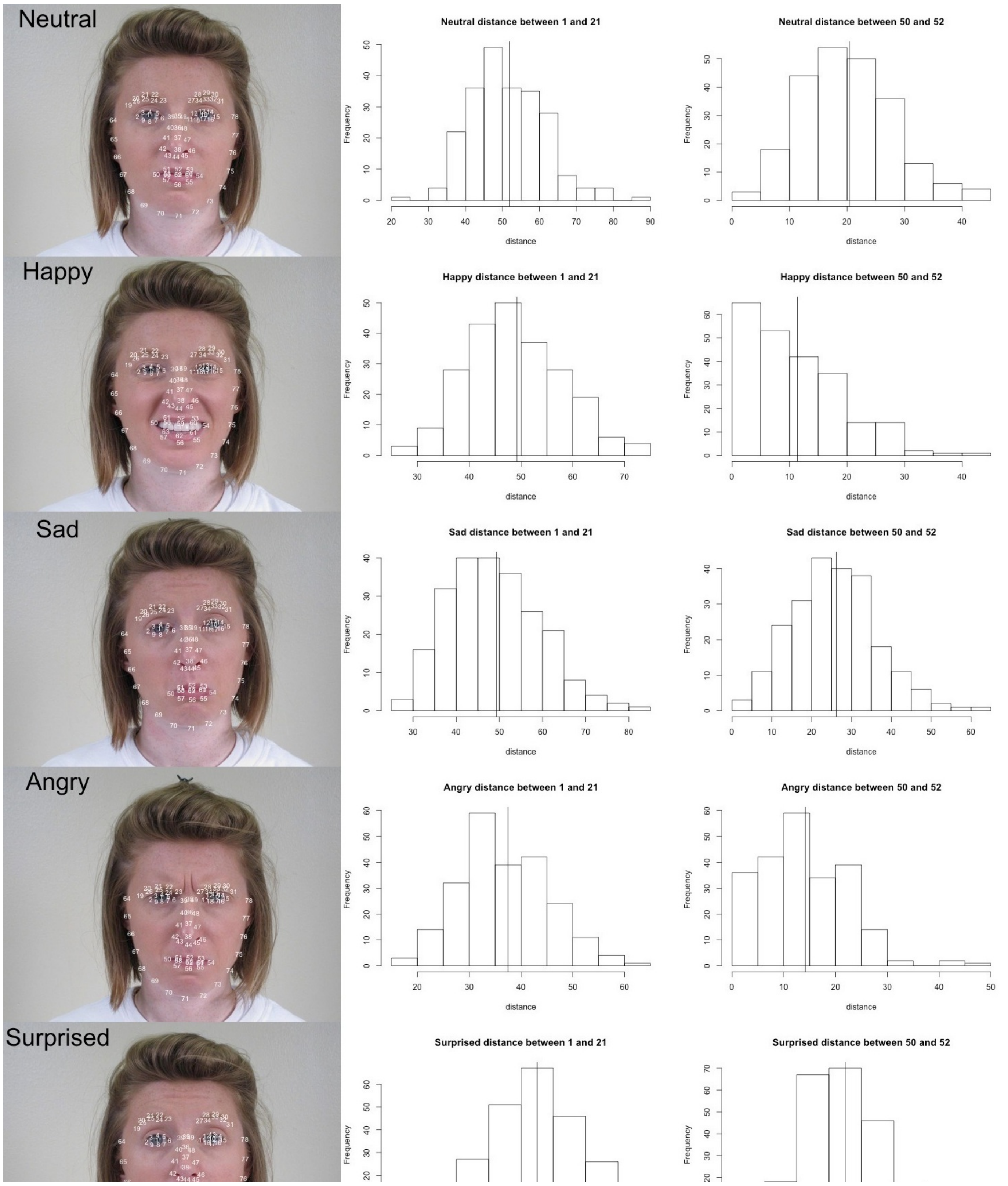
```
## Warning in readMat(paste0(train_pt_dir, sprintf("%04d", index), ".mat")):
## strings not representable in native encoding will be translated to UTF-8
```

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

```r
#inner points
for (i in 1:2500){
  fiducial_pt_list[[i]] = fiducial_pt_list[[i]][c(4,8,2,6,13,17,11,15,43,44,45,19,23,25,31,27,33,37,50,54,
59,62,71,41,47,1,10,57,63,55),]
}
```

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.



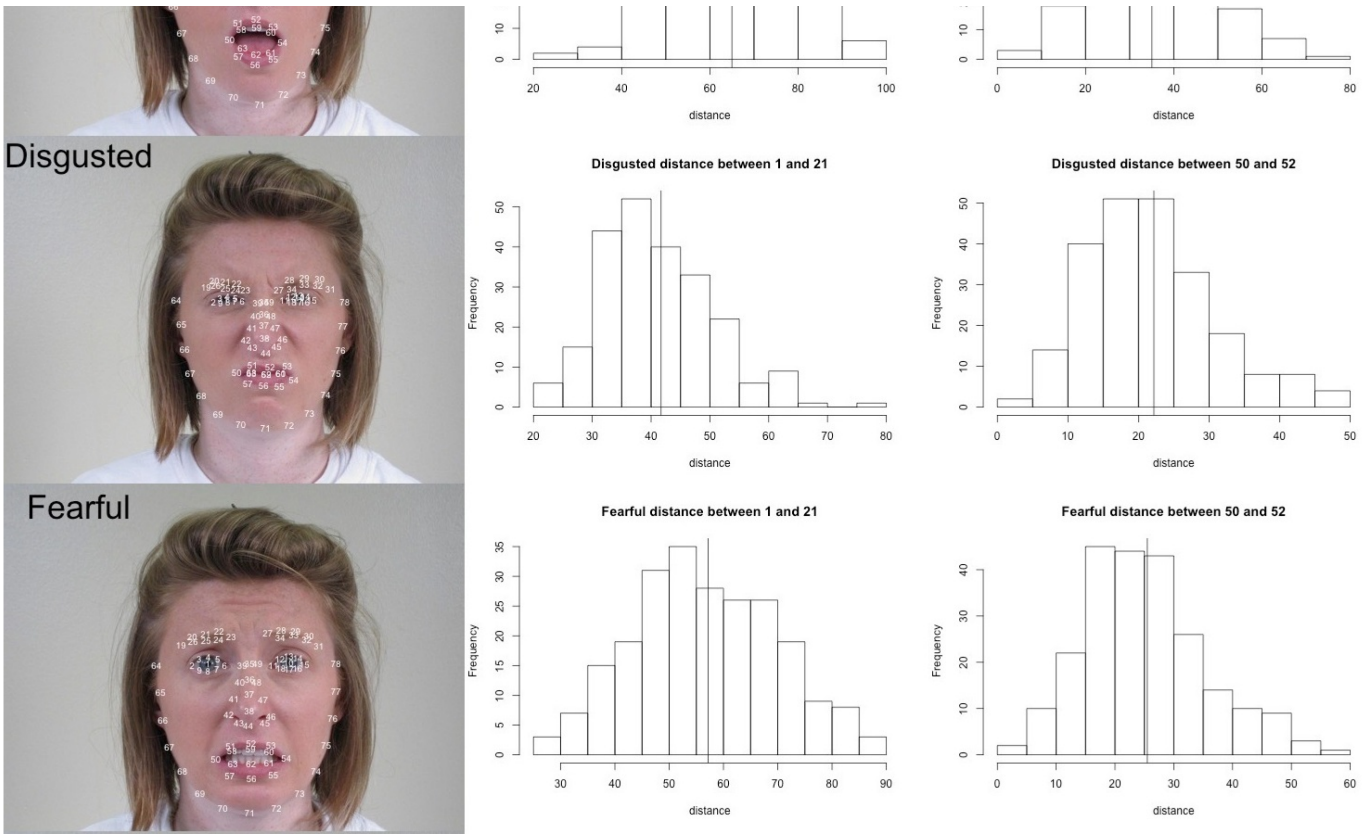


Figure1

`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

```r
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.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 GBM to do classification.

```r
source("../lib/train.R")
source("../lib/test.R")
```

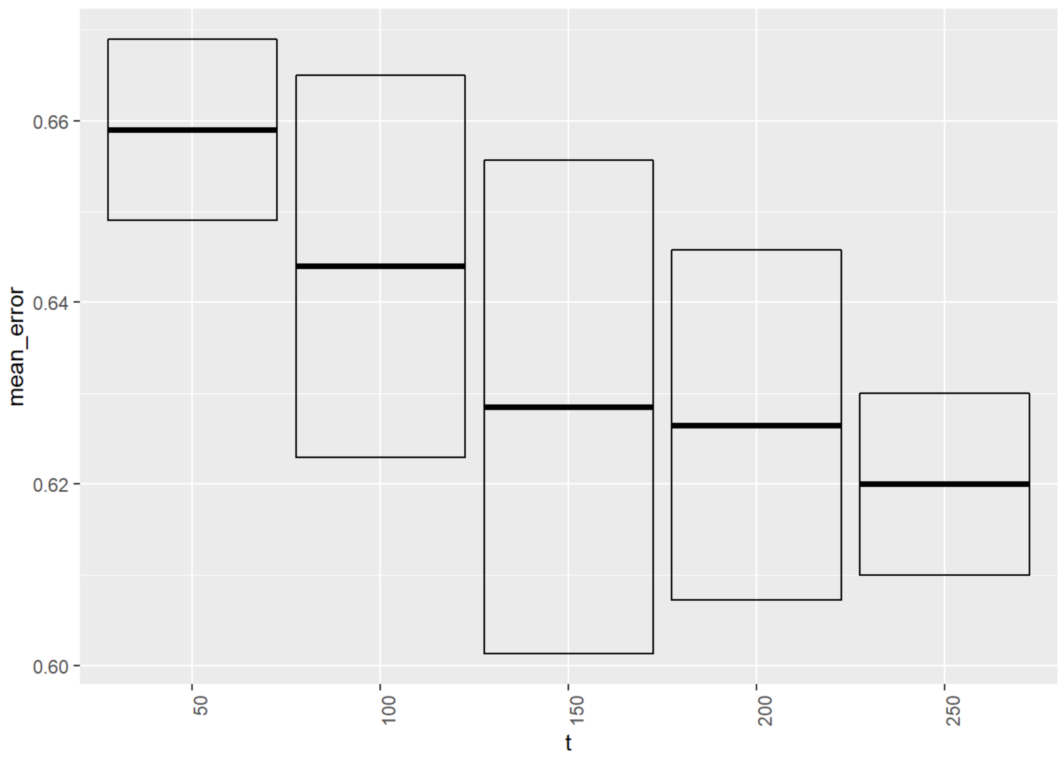
Model selection with cross-validation

- Do model selection by choosing among different values of training model parameters. In this case, we use different number of trees, t.

```r
source("../lib/cross_validation.R")
if(run.cv){
  err_cv <- matrix(0, nrow = length(t), ncol = 2)
  for(i in 1:length(t)){
    cat("tree number=", t[i], "\n")
    err_cv[i,] <- cv.function(dat_train, K, t[i])
  save(err_cv, file="../output/err_cv.RData")
  }
}
```

Visualize cross-validation results.

```r
if(run.cv){
  load("../output/err_cv.RData")
  err_cv <- as.data.frame(err_cv)
  colnames(err_cv) <- c("mean_error", "sd_error")
  err_cv$t = as.factor(t)
  err_cv %>%
    ggplot(aes(x = t, 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))
}
```



- Choose the “best” parameter value

```r
if(run.cv){
  model_best <- t[which.min(err_cv[,1])]
}
par_best <- list(t = model_best)
```

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

```r
tm_train=NA
tm_train <- system.time(fit_train <- train(dat_train, par_best))
save(fit_train, file="../output/fit_train.RData")
```

Step 5: Run test on test images

```
tm_test=NA
if(run.test){
  load(file="../output/fit_train.RData")
  tm_test <- system.time(pred <- test(fit_train, dat_test))
}
```

```
## [1] 0.31995744 0.25029022 0.29549673 0.64217352 0.22295356 0.08156265
```

- evaluation

```
pred <- factor(pred, levels = levels(dat_test$emotion_idx))
confusionMatrix(pred, dat_test$emotion_idx)
```

```
## 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   9  0  6  2  0  0  0  0  0  1  0  0  0  0  0  1  0  0  0  0  0  0
##         2   0 16  0  0  0  0  0  0  4  0  0  0  0  0  0  0  1  0  1  0  0  0
##         3   5  0 13  1  0  0  0  0  0  5  1  0  1  0  0  0  0  1  0  0  1  1
##         4   4  0  1 16  1  1  0  0  0  4  1  1  1  0  0  1  0  0  0  0  0  2
##         5   0  0  0  0 12  0  2  2  0  1  0  0  0  0  2  0  1  3  1  1  0  1
##         6   0  0  1  5  0  9  0  0  0  1  4  5  0  0  0  0  0  0  0  0  1  0
##         7   0  0  0  0  0  1  8  0  0  0  0  0  0  1  0  0  0  0  0  0  2  0
##         8   0  3  0  0  1  0  0 17  0  0  0  0  0  0  0  0  1  0  1  2  0  0
##         9   0  5  0  0  0  1  0  0  8  0  0  0  0  0  1  1  0  0  0  1  0  0
##         10  0  0  2  1  0  0  1  0  0  3  3  2  3  0  0  1  0  0  0  1  0  0
##         11  0  0  0  1  0  3  0  0  0  1 11  3  1  1  0  0  0  0  0  0  0  1
##         12  1  0  0  0  0  3  0  0  0  3  5  6  3  0  0  0  0  0  0  0  1  1
##         13  1  0  0  0  0  1  0  0  0  2  0  3  7  0  0  1  0  0  0  0  1  0
##         14  0  0  0  0  2  0  0  0  0  1  0  0  0 15  6  0  1  0  1  3  3  0
##         15  0  0  0  0  2  1  0  0  1  0  0  0  0  2  5  0  0  0  1  1  1  0
##         16  1  0  3  0  0  1  1  0  0  0  1  1  1  0  0 16  0  0  0  0  0  1
##         17  0  0  0  0  2  0  2  0  0  0  0  0  0  1  2  1 13  9  0  4  0  0
##         18  1  0  0  0  2  0  4  0  0  0  0  0  0  1  0  0  3  7  1  3  0  0
##         19  0  0  2  0  1  0  1  0  0  0  0  0  1  0  1  0  1  3  9  3  2  2
##         20  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  1  2  1  1  5  2  3
##         21  0  1  0  0  0  2  2  0  1  0  0  0  0  3  2  0  0  0  4  6  7  2
##         22  0  0  2  0  0  0  0  0  0  2  0  0  0  0  1  0  0  0  3  2  0  4
## 
## Overall Statistics
##                                           
##                Accuracy : 0.432           
##                  95% CI : (0.3881, 0.4767)
##     No Information Rate : 0.064           
##     P-Value [Acc > NIR] : < 2.2e-16       
##                                           
##                   Kappa : 0.4046          
##                                           
##  Mcnemar's Test P-Value : NA              
## 
## Statistics by Class:
## 
##                      Class: 1 Class: 2 Class: 3 Class: 4 Class: 5 Class: 6
## Sensitivity            0.4091   0.6400   0.4333   0.6154   0.5217   0.3913
## Specificity            0.9791   0.9874   0.9660   0.9641   0.9706   0.9644
## Pos Pred Value         0.4737   0.7273   0.4483   0.4848   0.4615   0.3462
## Neg Pred Value         0.9730   0.9812   0.9639   0.9786   0.9768   0.9705
## Prevalence             0.0440   0.0500   0.0600   0.0520   0.0460   0.0460
## Detection Rate         0.0180   0.0320   0.0260   0.0320   0.0240   0.0180
## Detection Prevalence   0.0380   0.0440   0.0580   0.0660   0.0520   0.0520
## Balanced Accuracy      0.6941   0.8137   0.6996   0.7898   0.7462   0.6778
##                      Class: 7 Class: 8 Class: 9 Class: 10 Class: 11 Class: 12
## Sensitivity            0.3810   0.8947   0.5714    0.1250    0.4231    0.2857
## Specificity            0.9916   0.9834   0.9815    0.9706    0.9768    0.9645
## Pos Pred Value         0.6667   0.6800   0.4706    0.1765    0.5000    0.2609
## Neg Pred Value         0.9734   0.9958   0.9876    0.9565    0.9686    0.9686
## Prevalence             0.0420   0.0380   0.0280    0.0480    0.0520    0.0420
## Detection Rate         0.0160   0.0340   0.0160    0.0060    0.0220    0.0120
## Detection Prevalence   0.0240   0.0500   0.0340    0.0340    0.0440    0.0460
## Balanced Accuracy      0.6863   0.9391   0.7765    0.5478    0.6999    0.6251
## Balanced Accuracy       0.6865    0.9391    0.7765    0.5478    0.6999    0.6231
##                      Class: 13 Class: 14 Class: 15 Class: 16 Class: 17
## Sensitivity             0.3889    0.6250    0.2500    0.6957    0.5652
## Specificity             0.9813    0.9643    0.9812    0.9790    0.9560
## Pos Pred Value          0.4375    0.4688    0.3571    0.6154    0.3824
## Neg Pred Value          0.9773    0.9808    0.9691    0.9852    0.9785
## Prevalence              0.0360    0.0480    0.0400    0.0460    0.0460
## Detection Rate          0.0140    0.0300    0.0100    0.0320    0.0260
## Detection Prevalence    0.0320    0.0640    0.0280    0.0520    0.0680
## Balanced Accuracy       0.6851    0.7946    0.6156    0.8373    0.7606
##                      Class: 18 Class: 19 Class: 20 Class: 21 Class: 22
## Sensitivity             0.2917    0.3913    0.1562    0.3333    0.2222
## Specificity             0.9685    0.9644    0.9786    0.9520    0.9793
## Pos Pred Value          0.3182    0.3462    0.3333    0.2333    0.2857
## Neg Pred Value          0.9644    0.9705    0.9443    0.9702    0.9712
## Prevalence              0.0480    0.0460    0.0640    0.0420    0.0360
## Detection Rate          0.0140    0.0180    0.0100    0.0140    0.0080
## Detection Prevalence    0.0440    0.0520    0.0300    0.0600    0.0280
## Balanced Accuracy       0.6301    0.6778    0.5674    0.6427    0.6007
```

In this case, we have 42 % accuracy when we use 250 as number of trees in our GBM function.

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.

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

```
## Time for constructing training features= 0.25 s
```

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

```
## Time for constructing testing features= 0.09 s
```

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

```
## Time for training model= 233.54 s
```

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

```
## Time for testing model= 0.41 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.

Part2 : Advanced Model

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 Starter Code, we use a stacking of GBM and neural network to do classification.

```
h2o_train<-as.h2o(dat_train)
```

```
##
  |
  |                                                                      |   0%
  |
  |======================================================================| 100%
```

```
h2o_test<-as.h2o(dat_test)
```

```
##
  |
  |                                                                      |   0%
  |
  |======================================================================| 100%
```

```
source("../lib/train_adv.R")
source("../lib/test_adv.R")
```

Model selection with cross-validation

- Do model selection by choosing among different values of training model parameters. In this case, we mainly focus on the hidden layer size, because we already examine how number of tree will influence the performance of GBM model.

```
#source("../lib/cross_validation_adv.R")
#t<-c(50, 100, 200)
#if(run.cv){
 # err_cv <- matrix(0, nrow = length(t), ncol = 1)
 # for(i in 1:length(t)){
 #   cat("layer size=", t[i], "\n")
 #   err_cv[i,] <- cv.function(dat_train, K, t[i])
 # save(err_cv, file="../output/err_cv_adv.RData")
 # }
#}
```

- Choose the “best” parameter value

```
#if(run.cv){
  #model_best <- t[which.min(err_cv[,1])]
#}
par_best <- list(t = 200,nfolds = 5)
```

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

```
tm_train2<-540
#delete my_ensemble in "../output/" if you want to run the following code (to save the new model)

#time_start<-Sys.time()
#fit_train_adv <- train_adv(h2o_train, par_best)
#time_end<-Sys.time()
#tm_train<-as.numeric(difftime(time_end, time_start))*60
#h2o.saveModel(fit_train_adv$model, path ="../output/")
```

Run test on test images

```
tm_test=NA
if(run.test){
  model<-h2o.loadModel(path ="../output/my_ensemble")
  tm_test <- system.time(pred <- test_adv(model, h2o_test))
}
```

```
##
  |
  |                                                                      |   0%
  |
  |======================================================================| 100%
```

- evaluation

After running the following code, one can see that the accuracy of advanced model is around 49% and the accuracy differs with different data split.

```
#p<-as.data.frame(pred[[1]])
#confusion_matrix<-as.data.frame(h2o.confusionMatrix(model, newdata = h2o_test))
#1-confusion_matrix[['Error']][23]
```

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= 0.25 s
```

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

```
## Time for constructing testing features= 0.09 s
```

```
cat("Time for training model=", tm_train2, "s \n")
```

```
## Time for training model= 540 s
```

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

```
## Time for testing model= 0.1 s
```