Project 3: Facial Expression Predictive Modeling

Group 5

Jingbin Cao, Chuanchuan Liu, Dennis Shpits, Yingyao Wu, Zikun Zhuang

Step 0 set work directories

```
set.seed(2020)
```

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) whether it is on presentation day for producing the csv file
- (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
- (T/F) run gbm baseline model
- (T/F) evaluate performance on the test set
- (number) gbm.numtrees, the number of trees to use in GBM baseline
- (T/F) return polynomial features matrix only
- (T/F) add polynomial features to starter code features matrix
- (T/F) run svm model
- (T/F) train svm model
- (T/F) perform model selection over a list of svm models
- (T/F) evaluate performance on the test set
- (T/F) run random forest model
- (T/F) train random forest model
- (T/F) run evaluation on the test set

- (T/F) run random forest model with old features
- (T/F) train random forest model with old features
- (T/F) run evaluation on the test set
- (T/F) run ridge model
- (0/1) alpha, alpha=0 for ridge regression, alpha=1 for lasso regression
- (T/F) train ridge model
- (T/F) run PCA+LDA model
- (T/F) run different principal components
- (T/F) run LDA on training set
- (T/F) run evaluation on the test set

```
run.presentation.day <- FALSE #presentation day flag. No training. Generate a csv file
run.cv <- TRUE # run cross-validation on the training set
sample.reweight <- FALSE # run sample reweighting in model training
K <- 5  # number of CV folds
run.feature.train <- FALSE # process features for training set
run.test <- TRUE # run evaluation on an independent test set
run.feature.test <- TRUE # process features for test set

# gbm
run.gbm.train <- FALSE # gbm(imroved) is the chosen advanced model
run.gbm.test <- TRUE # gbm(imroved) is the chosen advanced model
gbm.numtrees <- 1000 #number of trees to use in gbm

#features options
run.poly.feature <- TRUE # process poly features
run.add.poly.feature <- TRUE # and poly features to features matrix

# svm
run.svm <- TRUE # svm is the chosen advanced model
run.svm.train <- FALSE # train svm model
model.selection <- FALSE # perform model selection on svm models
run.svm.test <- TRUE # evaluate performance on the test set

# Random Forest Model with new feature
run.rf <- TRUE
train.rf <- FALSE # Train Random Forest Model
test.rf <- TRUE # Test Random Forest Model

# Random Forest Model with old feature
run.rf.old.feature <- TRUE # run random forest using old features
train.rf.old.feature <- FALSE # train random forest with old features
test.rf.old.feature <- TRUE # evaluate performance on test set

# ridge
run.ridge <- TRUE # ridge is the chosen advanced model
alpha <- 0 # ridge regression alpha
lambda <- 10^seq(10, -2, length = 100) # lambda
train.ridge <- FALSE # train ridge model
```

```
# PCA + LDA
run.pca_lda <- TRUE # PCA + LDA is the chosen adcanced model
run.select_PC <- FALSE #run different PCs
run.lda.train <- FALSE # run lda on the training set
run.pca_lad.test <- TRUE # evaluate performance on the test set
```

Using cross-validation or independent test set evaluation, we compare the performance of models with different specifications.

Step 2: import data and train-test split

```
#train-test split
info <- read.csv(train_label_path)
n <- nrow(info) #get number of rows from csv
n_train <- round(n*(4/5), 0) #use 4/5 amount of data for training
train_idx <- sample(info$Index, n_train, replace = F) #grab indexes used for training
test_idx <- setdiff(info$Index, train_idx) # get indexes not used for training
```

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
n_files <- length(list.files(train_image_dir,'*jpg'))
readMat.matrix <- function(index){
     return(round(readMat(paste0(train_pt_dir, sprintf("%04d", index), ".mat"))[[1]],0))
}

if (run.presentation.day){
  test_idx <- c(1:n_files) #sample(n_files, n_files, replace = F)
  run.gbm.train <- FALSE
  run.feature.train <- FALSE
  run.gbm.test <- TRUE

  run.rf <- TRUE
  train.rf <- FALSE
  test.rf <- TRUE
}

#load fiducial points
fiducial_pt_list <- lapply(1:n_files, readMat.matrix)
save(fiducial_pt_list, file="../output/fiducial_pt_list.RData")
```

If on presentation day, we will run our baseline and advanced model, and produce a csv file containing label predictions.

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` is the wrapper for all feature engineering functions and options. The function `feature( )` have options that correspond to different scenarios for the project and produces an R object that contains features and responses that are required by all the models that are going to be evaluated 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
gbm_tm_feature_train <- NA
if(run.feature.train){
  tm_feature_train <- system.time(dat_train<-feature(fiducial_pt_list,train_idx,
                                                     run.poly.feature, run.add.poly.feature))
  gbm_tm_feature_train <- system.time(gbm_dat_train<-feature(fiducial_pt_list,train_idx,
                                                            FALSE, FALSE))
  save(gbm_dat_train, file="../output/gbm_feature_train.RData")
  save(dat_train, file="../output/feature_train.RData")
}else{
  load(file="../output/feature_train.RData")
  load(file="../output/gbm_feature_train.RData")
}

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

Step 4: train classification models with training features and responses; run test on test images

Call the train model and test model from library.

`train.R` and `test.R` are wrappers for all model training steps and 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.

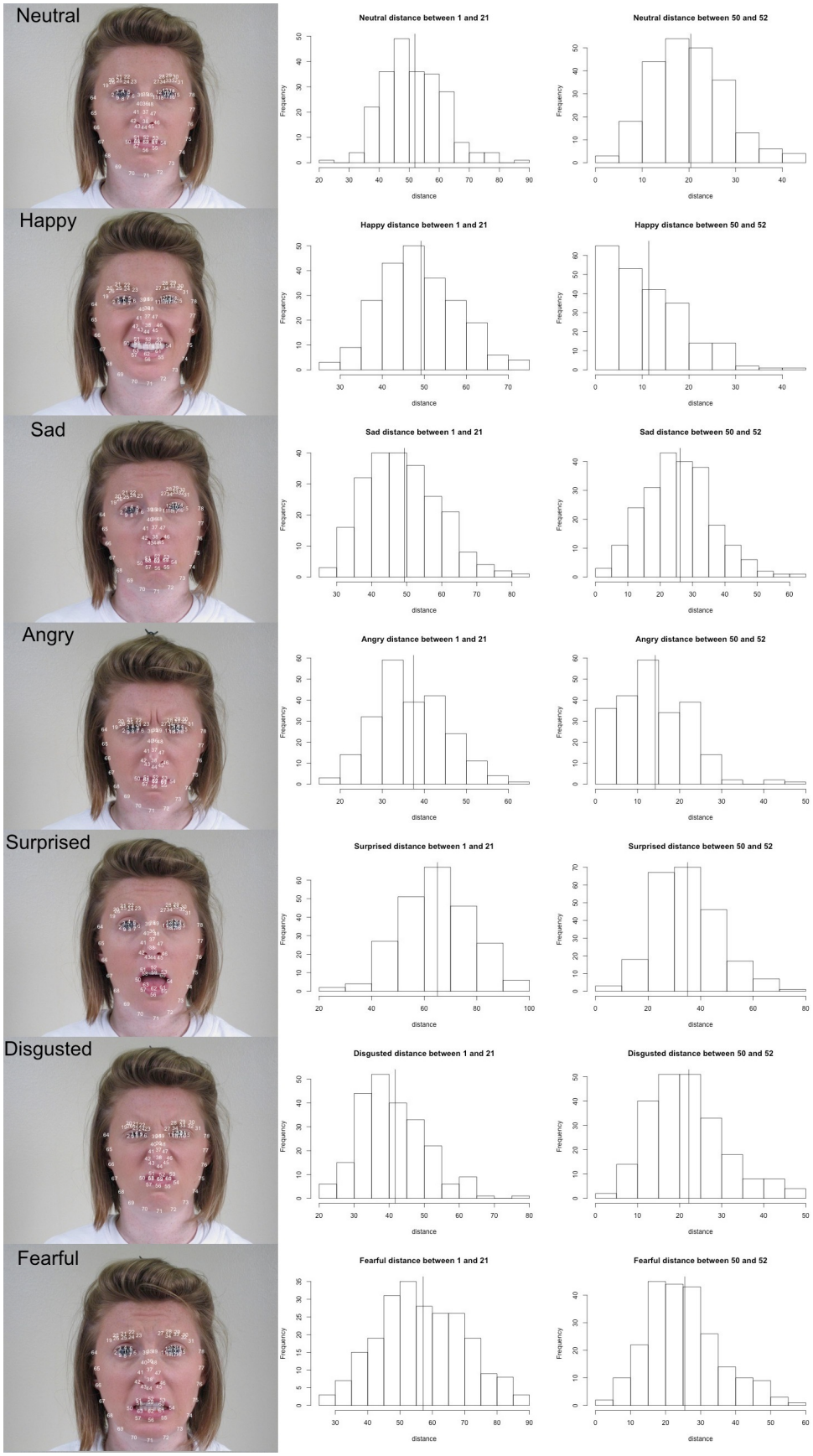

Figure 1: Figure1

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

Baseline GBM Model

- Model Training

```
if (run.gbm.train){
  if (sample.reweight){

    gbm_dat_train$label <- as.factor(gbm_dat_train$label)
    dat_train_balanced_gbm <- SMOTE(label ~ ., gbm_dat_train, perc.over = 100, perc.under=200)
    table(dat_train_balanced_gbm$label)

    gbm_tm_train <- system.time(gbm_train <- train_gbm(dat_train_balanced_gbm, s=0.1,
                                                       K=K, n=gbm.numtrees,w = NULL))

  } else {
    gbm_tm_train <- system.time(gbm_train <- train_gbm(gbm_dat_train, s=0.1,
                                                       K=K, n=gbm.numtrees,w = NULL))
  }

  # plot the performance
  best.iter.oob <- gbm.perf(gbm_train,method="OOB") # returns out-of-bag estimated best number of trees
  print(best.iter.oob)
  best.iter.cv <- gbm.perf(gbm_train,method="cv") # returns K-fold cv estimate of best number of trees
  print(best.iter.cv)

  saveRDS(gbm_train, "../output/gbm_model.rds")
  save(gbm_tm_train, best.iter.cv, file="../output/gbm_outputs.RData")
}
```

- Evaluation on Test Set

```
if(run.gbm.test){
  load(file="../output/gbm_outputs.RData")
  gbm_tm_test = NA
  feature_test <- as.matrix(gbm_dat_test[, 1:ncol(gbm_dat_test)-1])

  gbm_train <- readRDS("../output/gbm_model.rds")
  gbm_tm_test <- system.time(prob_pred_baseline<-test_gbm(gbm_train,as.data.frame(feature_test),
                                                          n=best.iter.cv,pred.type = 'response'))

  label_pred_baseline <- colnames(prob_pred_baseline)[apply(prob_pred_baseline, 1, which.max)]
}
```

- Show GBM Accuracy and AUC

```
if (run.gbm.test){
  load(file="../output/gbm_outputs.RData")
  gbm_accu <- mean(gbm_dat_test$label == label_pred_baseline)
  gbm.auc <- WeightedROC(as.numeric(label_pred_baseline), gbm_dat_test$label)
  gbm_auc = WeightedAUC(gbm.auc)
  cat("Time for constructing gbm training features=", gbm_tm_feature_train[1], "s \n")
  cat("Time for constructing gbm testing features=", gbm_tm_feature_test[1], "s \n")
```

```
  cat("The AUC of gbm model is", gbm_auc, ".\n")
  cat("The accuracy of GBM baseline model is", gbm_accu*100, "%.\n")
  cat("Time for training gbm model=", gbm_tm_train[1], "s \n")
  cat("Time for testing model=", gbm_tm_test[1], "s \n")
}
```

```
## Time for constructing gbm training features= NA s
## Time for constructing gbm testing features= 0.181 s
## The AUC of gbm model is 0.8819249 .
## The accuracy of GBM baseline model is 88.16667 %.
## Time for training gbm model= 451.15 s
## Time for testing model= 14.44 s
```

Advanced Model: Random Forest

The second advanced model is random forest. Here we use the datasets that are extracted by new feature functions. We used two models trained by both imbalanced and balanced dataset. We used SMOTE function to balance both training and testing data. For tuning the model, we set mtry = 308, tree number = 1000, and node size = 30 for the RF model using balanced data (SMOTE), and we set mtry = 308, tree number = 1500, and node size = 30 for the RF model using imbalanced data. The evaluation of the model is shown at the end of this section.

The tuning part is in a separate file named "appendix_tune_rf.rmd" in doc folder. Please feel free to check that to see the tuning process.

We also trained random forest model with the datasets that are extracted by old feature functions. That is in part 6. Thank you for reading!

```
if(run.rf){
  ## Training RF
  if(train.rf){
    rf_dat_train <- dat_train
    rf_dat_train$label <- as.factor(rf_dat_train$label)
    dat_train_balanced_SMOTE <- SMOTE(label ~ ., rf_dat_train, perc.over = 100, perc.under=200)
    # Train RF by balanced data
    time.rf.train.final.balanced <- system.time(
      random_forest_fit_final_balanced <- random_forest_train(dat_train_balanced_SMOTE,
                                                              mtry = 308, tree_number = 1000,
                                                              node_size = 30))
    save(random_forest_fit_final_balanced, file = "../output/rf_train_final_balanced.RData")
    save(time.rf.train.final.balanced, file = "../output/rf_train_final_time_balanced.RData")
    # Train RF by imbalanced data
    time.rf.train.final.imbalanced <- system.time(
      random_forest_fit_final_imbalanced <- random_forest_train(dat_train,
                                                                mtry = 308, tree_number = 1000,
                                                                node_size = 30))
    save(time.rf.train.final.imbalanced, file = "../output/rf_train_final_time_imbalanced.RData")
    save(random_forest_fit_final_imbalanced, file = "../output/rf_train_final_imbalanced.RData")
  }else{
    load("../output/rf_train_final_balanced.RData")
    load("../output/rf_train_final_time_balanced.RData")
  }
  # Evaluation:
  if(test.rf){
    rf_dat_test <- dat_test
```

```
    rf_dat_test$label <- as.numeric(rf_dat_test$label)

    time.rf.test.final.balanced <- system.time(
      rf_predicted_balanced <- as.numeric(as.vector(random_forest_test(random_forest_fit_final_balanced
                                                                       rf_dat_test))))
    rf_accuracy_balanced <- mean(round(rf_predicted_balanced == rf_dat_test$label))
    tpr.fpr.balanced <- WeightedROC(as.numeric(rf_predicted_balanced),rf_dat_test$label)
    rf_AUC_balanced <- WeightedAUC(tpr.fpr.balanced)

    cat("AUC for tuned Random Forest(balanced): ", rf_AUC_balanced,".\n")
    cat("Accuracy for tuned Random Forest(balanced)", rf_accuracy_balanced*100,"%.\n")
    cat("Training time for tuned Random Forest: ", time.rf.train.final.balanced[1], "s.\n")
    cat("Testing time for tuned Random Forest: ", time.rf.test.final.balanced[1], "s.\n")
    cat("   ","\n")
  }
}
```

```
## AUC for tuned Random Forest(balanced):  0.9037981 .
## Accuracy for tuned Random Forest(balanced) 91.16667 %.
## Training time for tuned Random Forest:  810.32 s.
## Testing time for tuned Random Forest:  0.592 s.
##
```

We think RF model do not need Cross-Validation. Here is a snippet from Breiman's official documentation: In random forests, there is no need for cross-validation or a separate test set to get an unbiased estimate of the test set error. It is estimated internally, during the run, as follows: Each tree is constructed using a different bootstrap sample from the original data. About one-third of the cases are left out of the bootstrap sample and not used in the construction of the kth tree. Put each case left out in the construction of the kth tree down the kth tree to get a classification. In this way, a test set classification is obtained for each case in about one-third of the trees. At the end of the run, take j to be the class that got most of the votes every time case n was oob. The proportion of times that j is not equal to the true class of n averaged over all cases is the oob error estimate. This has proven to be unbiased in many tests.

Alternative Model 1: SVM Model

- Balance the Training Set

If the given data set is imbalanced, we apply SMOTE to rebalance the data and use it to train our svm models.

```
if(run.svm){
  if(run.svm.train){
    tm_svm_rebalanced_train <- NA
    if(sample.reweight){
      dat_train$label = as.factor(dat_train$label)
      tm_svm_rebalanced_train <- system.time(svm_training_data <- SMOTE(label ~ ., data = dat_train))
      save(tm_svm_rebalanced_train, file="../output/tm_svm_rebalanced_train.RData")
    } else {
      svm_training_data <- dat_train
      tm_svm_rebalanced_train <- tm_feature_train
    }
  }
}
```

- Model Selection

Tuning hyper-parameters for both linear and radial basis kernel and select the kernel method that produces the highest AUC and accuracy among the two methods.

```
if(run.svm){
  set.seed(2020)
  if(model.selection){
    svm_model_auc <- rep(NA, 2)
    svm_model_accu <- rep(NA, 2)
    ### linear kernel
    if(run.cv){
      best.linear.cost <- svm_linear_cost_tune(svm_training_data)
      cat("The best cost for svm model with linear kernel is: ",
          best.linear.cost$best.parameters$cost) # best cost is 0.01
      svm.linear.train.start = proc.time()
      svm_linear_mod <- svm_linear_train(svm_training_data, 0.01, K)
      svm.linear.train.end = proc.time()
      svm.linear.tm = svm.linear.train.end - svm.linear.train.start
      save(svm_linear_mod, file="../output/svm_linear_mod.RData")
      save(svm.linear.tm, file="../output/tm_svm_linear_mod.RData")
    } else {
      load(file="../output/svm_linear_mod.RData")
    }
    svm_linear_pred <- svm_test(svm_linear_mod, svm_training_data, FALSE)
    # evaluate performance on linear kernel
    svm_model_accu[1] <- mean(round(svm_linear_pred == svm_training_data$label))
    tpr.fpr_linear <- WeightedROC(as.numeric(svm_linear_pred), svm_training_data$label)
    svm_model_auc[1] <- WeightedAUC(tpr.fpr_linear)

    #### radial basis kernel
    if(run.cv){
      best.radial.cost <- svm_radial_cost_tune(svm_training_data)
      svm.rbf.train.start = proc.time()
      svm_radial_mod <- svm_radial_train(svm_training_data, best.radial.cost, K)
      svm.rbf.train.end = proc.time()
      svm.rbf.tm = svm.rbf.train.end - svm.rbf.train.start
      save(svm_radial_mod, file="../output/svm_radial_mod.RData")
      save(svm.rbf.tm, file="../output/tm_svm_radial_mod.RData")
    } else {
      load(file="../output/svm_radial_mod.RData")
    }
    svm_radial_pred <- svm_test(svm_radial_mod, svm_training_data, FALSE)
    # evaluate performance on rbf kernel
    svm_model_accu[2] <- mean(round(svm_radial_pred == svm_training_data$label))
    tpr.fpr_rbf <- WeightedROC(as.numeric(svm_radial_pred), svm_training_data$label)
    svm_model_auc[2] <- WeightedAUC(tpr.fpr_rbf)

    # table to display results for the two kernel methods
    svm_res = matrix(rep(NA,6),ncol=3)
    svm_res[,1] = svm_model_accu
    svm_res[,2] = svm_model_auc
    svm_res[,3] = c(svm.linear.tm[[3]], svm.rbf.tm[[3]])
    colnames(svm_res) = c("Accuracy", "AUC","Running Time")
    rownames(svm_res) = c("linear","radial basis")
    save(svm_res, file="../output/svm_model_selection.RData")
```

```
  } else {
    if(run.presentation.day){
      tm_svm_radial_mod < system.time(svm_radial_mod <- svm_radial_train(svm_training_data,
                                                                         1, K))
    } else {
      load(file="../output/svm_radial_mod.RData")
      load(file="../output/svm_model_selection.RData")
      svm_res
    }
  }
}
```

```
##                Accuracy       AUC Running Time
## linear       0.9224402 0.9216738      423.341
## radial basis 0.9573881 0.9557403      523.557
```

Since radial basis kernel has higher accuracy and AUC than linear kernel, we will choose radial basis as our kernel method for training the svm model.

- Evaluation on Testing Data

```
if(run.svm){
  tm_svm_rbf_test <- NA
  svm_testing_data <- dat_test
  if(run.svm.test){
    if(!run.presentation.day) {
      tm_svm_rbf_test <- system.time(svm_rbf_pred <- svm_test(svm_radial_mod, svm_testing_data, FALSE))
      svm_test_accu = mean(round(svm_rbf_pred == svm_testing_data$label))
      tpr.fpr.rbf <- WeightedROC(as.numeric(svm_rbf_pred), svm_testing_data$label)
      svm_test_auc = WeightedAUC(tpr.fpr.rbf)
      save(tm_svm_rbf_test, file="../output/tm_svm_rbf_test.RData")

      cat("The accuracy of svm model is", svm_test_accu*100, "%.\n")
      cat("The AUC of svm model is", svm_test_auc, ".\n")
    } else {
      tm_svm_rbf_test <- system.time(svm_rbf_pred <- svm_test(svm_radial_mod, svm_testing_data))
      svm_test_accu = mean(round(svm_rbf_pred == svm_testing_data$label))
      tpr.fpr.rbf <- WeightedROC(as.numeric(svm_rbf_pred), svm_testing_data$label)
      svm_test_auc = WeightedAUC(tpr.fpr.rbf)
      save(tm_svm_rbf_test, file="../output/tm_svm_rbf_test.RData")

      cat("The accuracy of svm model is", svm_test_accu*100, "%.\n")
      cat("The AUC of svm model is", svm_test_auc, ".\n")
    }
  }
}
```

```
## The accuracy of svm model is 91.16667 %.
## The AUC of svm model is 0.865852 .
```

- Summarize Running Time

```
if(run.svm){
  cat("Time for construct training features =", tm_feature_train[1], "s \n")
  cat("Time for construct testing features =", tm_feature_test[1], "s \n")
  cat("Time for training svm model =", svm_res[2,3], "s \n")
```

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

```
## Time for construct training features = NA s
## Time for construct testing features = 7.382 s
## Time for training svm model = 523.557 s
## Time for testing svm model= 13.892 s
```

Alternative Model 2: Ridge Model

- Apply Constructed Ridge Model to the Training Set

```
if(run.ridge){
  tm_ridge_train <- NA
  if (train.ridge){
    dat_train_rebalanced <- ROSE(label ~ ., data = dat_train, seed=2020)$data
    tm_ridge_train <- system.time(ridge_cv_model<-ridge_train(train_data=dat_train_rebalanced,
                                                              alpha=alpha, K=K, lambda=lambda))
    save(ridge_cv_model, file="../output/ridge_cv_model.RData")
    save(tm_ridge_train, file="../output/ridge_train_time.RData")
  } else {
    load(file="../output/ridge_cv_model.RData")
    load(file="../output/ridge_train_time.RData")
  }
}
```

- Use Cross-Validation to Choose the Optimal Lambda with Smallest MSE

```
if(run.ridge){
  if (train.ridge){
    set.seed(2020)
    dat_train_rebalanced <- ROSE(label ~ ., data = dat_train, seed=2020)$data
    feature_train = as.matrix(dat_train_rebalanced[,-dim(dat_train_rebalanced)[2]])
    label_train = as.integer(dat_train_rebalanced$label)
    ridge_model = cv.glmnet(x=feature_train, y=label_train, alpha=alpha, nfolds=K, lambda=lambda)
    opt_lambda = ridge_model$lambda.min
    save(opt_lambda, file="../output/ridge_optimal_lambda.RData")
  } else {
    load(file="../output/ridge_optimal_lambda.RData")
  }
}
```

- Predict on Testing Set with the Optimal Lambda

```
if(run.ridge){
  tm_ridge_test = NA
  if(run.test){
    load("../output/ridge_cv_model.RData")
    dat_test_rebalanced <- dat_test
    feature_test <- as.matrix(dat_test_rebalanced[, -dim(dat_test_rebalanced)[2]])
    tm_ridge_test <- system.time(label_pred<-as.integer(ridge_test(model=ridge_cv_model,
                                                                   features=feature_test,
                                                                   pred.type = 'class')))
    save(tm_ridge_test, file="../output/ridge_test_time.RData")
  } else{
```

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

- Summarize Running Time

```
if(run.ridge){
  cat("Time for constructing training features=", tm_feature_train[1], "s \n")
  cat("Time for constructing testing features=", tm_feature_test[1], "s \n")
  cat("Time for training ridge model=", tm_ridge_train[1], "s \n")
  cat("Time for testing ridge model=", tm_ridge_test[1], "s \n")
}
```

```
## Time for constructing training features= NA s
## Time for constructing testing features= 7.382 s
## Time for training ridge model= 27.49 s
## Time for testing ridge model= 0.078 s
```

- Evaluation on Independent Testing Data

```
if(run.ridge){
  load("../output/ridge_cv_model.RData")
  dat_test_rebalanced <- dat_test
  feature_test <- as.matrix(dat_test_rebalanced[, -dim(dat_test_rebalanced)[2]])
  label_pred = as.integer(predict(ridge_cv_model, s=opt_lambda, newx=feature_test,
                                  type='class'))
  label_test = as.integer(dat_test_rebalanced$label)
  ridge_accuracy = mean(round(label_test== label_pred))
  cat("The accuracy of the ridge model is", ridge_accuracy*100, "%.\n")
  ridge_AUC = auc(roc(label_pred,label_test))
  cat("The AUC of the ridge model is", ridge_AUC, ".\n")
}
```

```
## The accuracy of the ridge model is 80.5 %.
## The AUC of the ridge model is 0.9021739 .
```

Alternative Model 3: PCA + LDA

- Rebalance Training Set

```
if(run.pca_lda){
  if(run.lda.train){
    if(sample.reweight){
      dat_train$label <- as.factor(dat_train$label)
      balanced_train_data <- SMOTE(label~.,data = dat_train)
      save(balanced_train_data, file="../output/feature_balanced_train.RData")
    } else {
      load(balanced_train_data, file="../output/feature_train.RData")
    }
  }
}
```

- Perform PCA for Dimension Reduction

Since there are over 6000 features, we implement the PCA method to reduce dimension according to the covariance matrix. We only retain PCs with large variance.

```
if(run.pca_lda){
  balanced_test_data <- dat_test
  if(run.select_PC){
    #separate the features from label
    dat_train_new <- balanced_train_data[,-dim(balanced_train_data)[2]]
    dat_test_new <- balanced_test_data[,-dim(balanced_test_data)[2]]
    #create a vector contain target number of PCs
    num.pca <- c(10,50,500,1000)
    train_pca <- function(num.pca){
      for(i in 1:length(num.pca)){
        #start time for training the model
        train.model.start = proc.time()
        #run PCA
        pca <- prcomp(dat_train_new)
        #store for each potential PC
        train_pca <- data.frame(pca$x[,1:num.pca[i]],
                                label = balanced_train_data[dim(balanced_train_data)[2]])
        pred_pca <- predict(pca,dat_test_new)
        test_pca <- data.frame(pred_pca[,1:num.pca[i]],
                               label = balanced_test_data[dim(balanced_test_data)[2]])
        #fitting the lda model
        lda_pca <- lda(label ~ ., data = train_pca)
        #stop time for training the model
        train.model.end = proc.time()
        #start time for testing the model
        test.model.start = proc.time()
        #predict lda model
        lda_pred_pca = predict(lda_pca,test_pca[-dim(test_pca)[2]])
        #end time for testing the model
        test.model.end = proc.time()
        #test accuracy
        test_accuracy=confusionMatrix(lda_pred_pca$class, test_pca$label)$overall[1]
        print(list(l1=train.model.end - train.model.start,
                   l2=test.model.end - test.model.start,
                   l3=test_accuracy))
      }
    }
    train_pca(num.pca)
  }
}
```

By comparing the training time, test time and accuracy, we use model with 500 PCs.

- Model Training

```
if(run.pca_lda){
  train.model.start = proc.time()
  if(run.lda.train){
    pca_500 <- prcomp(balanced_train_data[,-dim(balanced_train_data)[2]])
    train_pca_500 <- data.frame(pca_500$x[,1:500],
                                label = balanced_train_data[dim(balanced_train_data)[2]])
    pred_pca_500 <- predict(pca_500,balanced_test_data[,-dim(balanced_test_data)[2]])
    test_pca_500 <- data.frame(pred_pca_500[,1:500],
                               label = balanced_test_data[dim(balanced_test_data)[2]])
    save(train_pca_500, file="../output/feature_pca_train.RData")
```

```
    save(test_pca_500, file="../output/feature_pca_test.RData")
  } else {
    load(file="../output/feature_pca_train.RData")
    load(file="../output/feature_pca_test.RData")
  }
  #calculate the training time
  lda_pca_500 <- lda(label ~ ., data = train_pca_500, cv = TRUE)
  train.model.end = proc.time()
}
```

- Calculate the Training and Testing Accuracy of LDA Model

```
if(run.pca_lda){
  test.model.start = proc.time()
  pred_train_lda <- predict(lda_pca_500, train_pca_500[-dim(train_pca_500)[2]])
  accu_train_lda <- mean(pred_train_lda$class == train_pca_500$label)
  cat("The trainig accuracy of model: LDA", "is", accu_train_lda*100, "%.\n")
  #calculating the test time
  if(run.test){
    pred_test_lda <- predict(lda_pca_500, test_pca_500)
  }
  test.model.end = proc.time()
  save(pred_test_lda, file="../output/fit_train.RData")
  accu_test_lda <- mean(pred_test_lda$class == test_pca_500$label)
  cat("The accuracy of model: LDA", "is", accu_test_lda*100, "%.\n")
  tpr.fpr <- WeightedROC(as.numeric(pred_test_lda$class), test_pca_500$label)
  lda_auc = WeightedAUC(tpr.fpr)
  cat("The AUC of model: LDA is", lda_auc, ".\n")
}
```

```
## The trainig accuracy of model: LDA is 89.94048 %.
## The accuracy of model: LDA is 74 %.
## The AUC of model: LDA is 0.6973767 .
```

- 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.

```
if(run.pca_lda){
  tm_train <- train.model.end - train.model.start
  tm_test <- test.model.end - test.model.start
  cat("Time for constructing training features =", tm_feature_train[1], "s \n")
  cat("Time for constructing testing features =", tm_feature_test[1], "s \n")
  cat("Time for training model =", tm_train[1], "s \n")
  cat("Time for testing model =", tm_test[1], "s \n")
}
```

```
## Time for constructing training features = NA s
## Time for constructing testing features = 7.382 s
## Time for training model = 4.396 s
## Time for testing model = 0.101 s
```

Alternative Model 4: Random Forest (RF) Model with old features:

The fourth alternative model is random forest using old features given in the starter code. Here we use the datasets that are extracted by old feature functions. We used two models trained by both imbalanced and balanced dataset. We used ROSE function to balance both training and testing data. For tuning the model, we set mtry = 308, tree number = 500, and node size = 10 for the RF model using balanced data, and we set mtry = 308, tree number = 1500, and node size = 30 for the RF model using imbalanced data. The evaluation of the model is shown at the end of this section, and we will compare this model with the RF model trained by new features.

```
if(run.rf.old.feature){
  old_feature <- function(input_list = fiducial_pt_list, index){
    old_pairwise_dist <- function(vec){
      return(as.vector(dist(vec)))
    }
    old_pairwise_dist_result <-function(mat){
      return(as.vector(apply(mat, 2, old_pairwise_dist)))
    }
    old_pairwise_dist_feature <- t(sapply(input_list[index], old_pairwise_dist_result))
    dim(old_pairwise_dist_feature)
    old_pairwise_data <- cbind(old_pairwise_dist_feature, info$label[index])
    colnames(old_pairwise_data) <- c(paste("feature", 1:(ncol(old_pairwise_data)-1), sep = ""),
                                     "label")
    old_pairwise_data <- as.data.frame(old_pairwise_data)
    old_pairwise_data$label <- as.factor(old_pairwise_data$label)
    return(feature_df = old_pairwise_data)
  }

  old_tm_feature_train <- NA
  if(run.feature.train){
    old_tm_feature_train <- system.time(old_dat_train <- old_feature(fiducial_pt_list, train_idx))
    save(old_dat_train, file="../output/feature_train_old.RData")
  }else{
    load(file="../output/feature_train_old.RData")
  }
  old_tm_feature_test <- NA
  if(run.feature.test){
    old_tm_feature_test <- system.time(old_dat_test <- old_feature(fiducial_pt_list, test_idx))
    save(old_dat_test, file="../output/feature_test_old.RData")
  }else{
    load(file="../output/feature_test_old.RData")
  }

  # Train Model
  if(train.rf.old.feature){
    # transfer label column from factor to numeric
    old_dat_train$label <- as.numeric(old_dat_train$label)
    old_dat_test$label <- as.numeric(old_dat_test$label)

    # Balance data
    dat_train$label <- as.factor(dat_train$label)
    old_dat_train_balanced_SMOTE <- SMOTE(label ~ ., dat_train, perc.over = 100, perc.under=200)
    # Balanced
    old_time.rf.train.final.balanced <- system.time(
```

```
      old_random_forest_fit_final_balanced <- old_random_forest_train(old_dat_train_balanced_SMOTE,
                                                                      mtry = 308, tree_number = 1000,
                                                                      node_size = 30))
    save(old_random_forest_fit_final_balanced,
         file = "../output/rf_train_final_balanced_old_feature.RData")
    save(old_time.rf.train.final.balanced,
         file = "../output/rf_train_final_time_balanced_old_feature.RData")

  }else{
    load("../output/rf_train_final_balanced_old_feature.RData")
    load("../output/rf_train_final_time_balanced_old_feature.RData")
  }


  # Evaluate Model
  old_rf_dat_test <- old_dat_test
  old_rf_dat_test$label <- as.numeric(old_rf_dat_test$label)
  if(test.rf.old.feature){
    old_time.rf.test.final.balanced <- system.time(
      rf_predicted_balanced <- as.numeric(as.vector(old_random_forest_test(old_random_forest_fit_final_
    old_rf_accuracy_balanced <- mean(round(rf_predicted_balanced == old_rf_dat_test$label))
    old_tpr.fpr.balanced <- WeightedROC(as.numeric(rf_predicted_balanced),old_rf_dat_test$label)
    old_rf_AUC_balanced <- WeightedAUC(old_tpr.fpr.balanced)

    cat("AUC(balanced) for Random Forest with old feature: ",
        old_rf_AUC_balanced,".\n")
    cat("Accuracy(balanced) for Random Forest with old feature",
        old_rf_accuracy_balanced*100,"%.\n")
    cat("Training time (balanced) for Random Forest with old feature: ",
        old_time.rf.train.final.balanced[1], "s.\n")
    cat("Testing time (balanced) for Random Forest with old feature: ",
        old_time.rf.test.final.balanced[1], "s.\n")
    cat("    ","\n")
  }
}
```

```
## AUC(balanced) for Random Forest with old feature:  0.892582 .
## Accuracy(balanced) for Random Forest with old feature 86.83333 %.
## Training time (balanced) for Random Forest with old feature:  462.69 s.
## Testing time (balanced) for Random Forest with old feature:  0.31 s.
##
```

Generate a csv on presentation day

```
if (run.presentation.day){
  csvfileoutput<-"../output/label_prediction.csv"
  Advanced<-rf_predicted_balanced
  Baseline<-label_pred_baseline
  Index<-test_idx
  csvdata <- data.frame(Index, Baseline, Advanced)

  write.csv(csvdata,csvfileoutput, row.names=FALSE,quote = FALSE)
}
```

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.