Part1.Rmd

Jen Johnson

11/11/2017

Part 1. HW2 with 30 eigen vectors

Function for reading in data.

```
read.files <- function(filenames, directory){
  dataset <- c()

  for (file in filenames){
    img <- readPNG(paste(directory, file, sep = "/"))
    v.img <- as.vector(img)
    dataset <- cbind(dataset, v.img)
  }
  return(dataset)
}
```

Read in the training data.

```
directory = "/Users/jen/Dropbox/CSCI 454/hw/trainingfaces2"
filenames <- list.files(path = directory)
training <- read.files(filenames, directory)
```

Funtion for centering the data using the mean of each row.

```
center.data <- function(training){
  training <- cbind(training, apply(training, 1, mean))
  X <- training[ , 1:dim(training)[2]-1]-training[ ,dim(training)[2]]
  return(X)
}
```

Center the data.

```
X <- center.data(training)
```

Function for calculating eigen vectors.

```
calculate.eigens <- function(X, num.vectors){
  trans.X <- t(X)
  C <- X %*% trans.X
  EV <- eigen(C)
  Top <- EV$vectors[ , 1:num.vectors]
  trans.EV <- t(Top)
  return(trans.EV)
}
```

Calculate eigen vectors.

```
trans.EV <- calculate.eigens(X, 30)
```

Read in testing data.

```r
directory = "/Users/jen/Dropbox/CSCI 454/hw/testingfaces2"
filenames <- list.files(path = directory)
testing <- read.files(filenames, directory)
```

Normalize testing data

```r
testing <- testing - training[ ,dim(training)[2]]
```

Function for making weight matrix.

```r
make.weight.matrix <- function(trans.EV, testing){
  weight.matrices <- c()

  for (i in 1:dim(testing)[2]){
    current.img <- testing[ ,i]
    current.weight <-  trans.EV %*% current.img
    weight.matrices <- cbind(weight.matrices, current.weight)
  }
  return(weight.matrices)
}
```

Make weight matrix.

```r
weight.matrices <- make.weight.matrix(trans.EV, testing)
```

Function for converting weight matrix into chart of comparisons that returns genuine and imposter.

```r
make.gen.imposter <- function(weight.matrices){
  genuine <- c()
  imposter <- c()
  all.data <- matrix(ncol = 3)
  colnames(all.data) <- c("image1", "image2", "score")

  for (i in 1:length(filenames)){
    image1 <- filenames[i]
    subject1 <- substr(image1, 2, 3)

    j <- i+1
    while (j <= length(filenames)){
      image2 <- filenames[j]
      subject2 <- substr(image2, 2, 3)

      weight1 <- weight.matrices[, i]
      weight2 <- weight.matrices[, j]
      weight.diff <- sum(abs(weight1-weight2))

      current.row <- cbind(image1, image2, weight.diff)
      all.data <- rbind(all.data, current.row)

      if (subject1 == subject2) {
        genuine <- c(genuine, weight.diff)
      } else {
        imposter <- c(imposter, weight.diff)
      }

      j <- j+1
    }
  }
  return(list("genuine" = genuine, "imposter" = imposter))
}
```

Make genuine and imposter datasets from weight matrix.

```
result <- make.gen.imposter(weight.matrices)
```

Plotting functions from HW1.

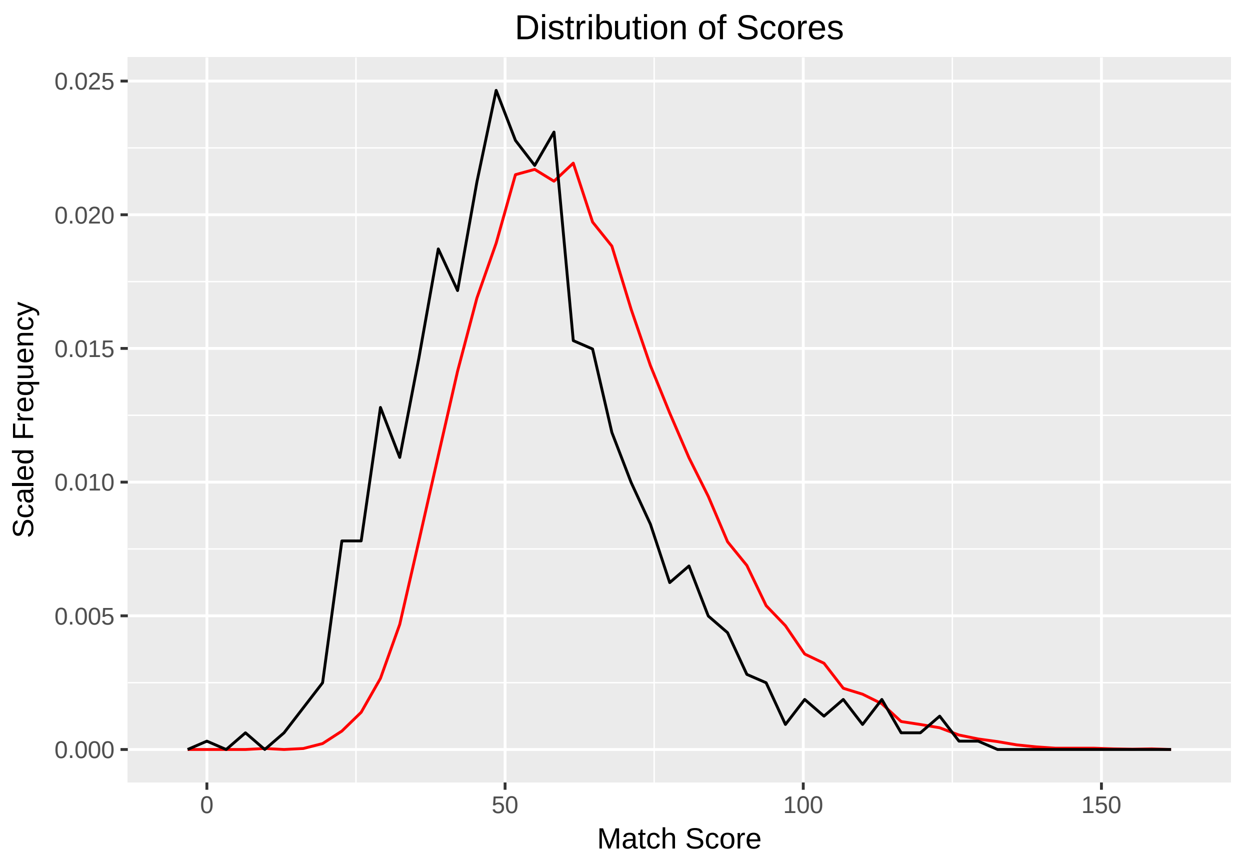
```
plot.scores <- function(result){
    imposter <- as.data.frame(result$imposter)
  genuine <- as.data.frame(result$genuine)

  #Scale using the density function and plot using ggplot's geom_freqpoly.
  plot <- ggplot() +
    geom_freqpoly(data = imposter, aes(x = imposter, y = ..density..), bins = 50, color = "red") +
    geom_freqpoly(data = genuine, aes(x = genuine, y = ..density..), bins = 50) +
    labs(title = "Distribution of Scores") +
    labs(x = "Match Score", y = "Scaled Frequency")
  print(plot)
}

plot.DET <- function(result){
   imposter <- as.data.frame(result$imposter)
  genuine <- as.data.frame(result$genuine)
  FAR_vs_FRR <- NULL

  #For each value of t, calculate FAR and FRR and add to dataset.

  for (t in seq(from = 0, to = 120.0, by = 2)){

    false_accept_count <- sum(imposter < t)
    false_accept_rate <- false_accept_count/dim(imposter)[1]
    false_reject_count <- sum(genuine > t)
    false_reject_rate <- false_reject_count/dim(genuine)[1]

    current_row <- c(false_accept_rate, false_reject_rate)
    FAR_vs_FRR<- rbind(FAR_vs_FRR, current_row)
  }

  rates_data_frame <- as.data.frame(FAR_vs_FRR)
  colnames(rates_data_frame) <- c("FAR", "FRR")
  plot <- ggplot(rates_data_frame, aes(x=FAR, y = FRR)) + geom_point() + geom_abline(slope = 1, intercep
  print(plot)
  return(rates_data_frame)
}

get.EER <- function(rates_data_frame){
  #Make new column containing boolean FAR > FRR.
rates_data_frame$larger <- rates_data_frame$FAR > rates_data_frame$FRR

#Find where FAR becomes less than FRR. Use these as upper and lower boundaries to estimate the EER.
far_is_smaller <- rates_data_frame[rates_data_frame$larger=="FALSE", ]
```

```r
lower_bound <- max(far_is_smaller$FAR)
far_is_larger <- rates_data_frame[rates_data_frame$larger=="TRUE", ]
upper_bound <- min(far_is_larger$FAR)

EER <- mean(lower_bound, upper_bound)
print(EER)
}
```

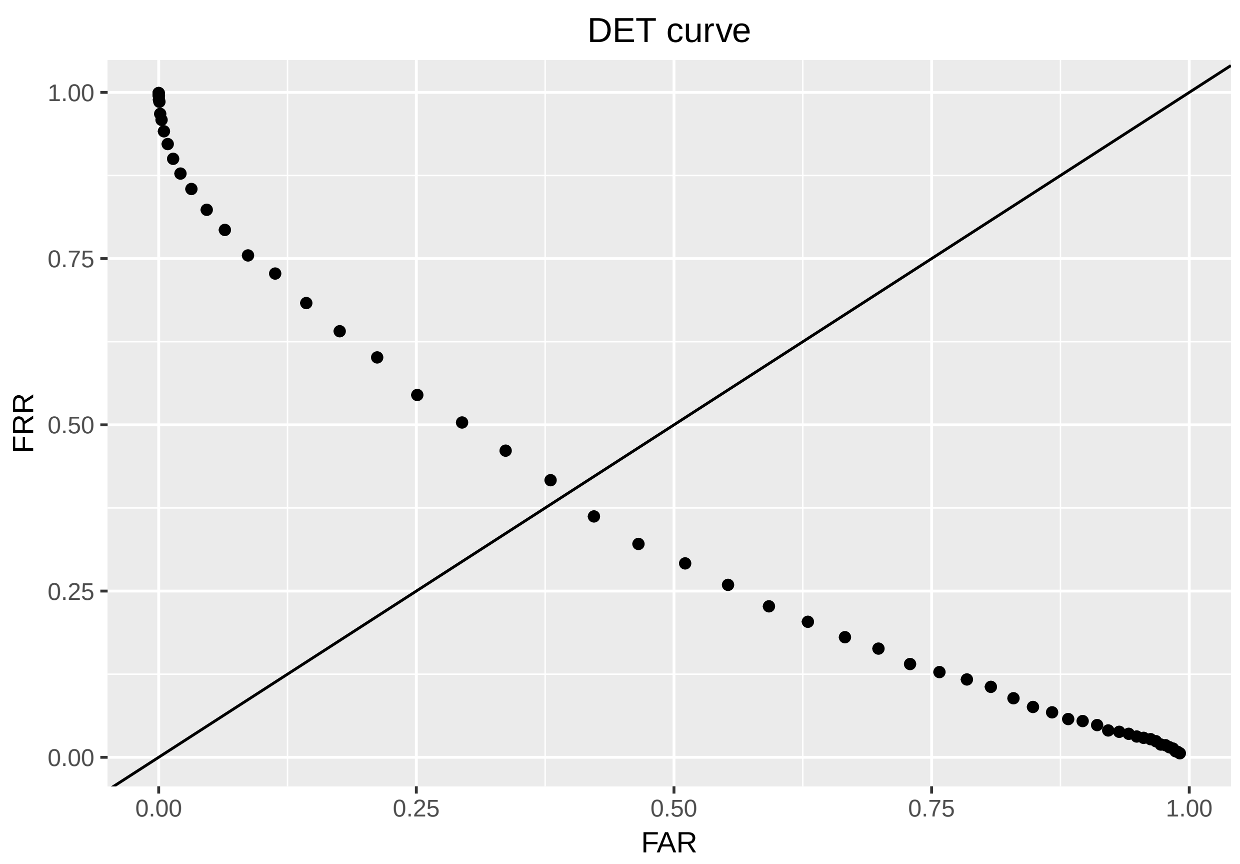
Use plotting functions.

```r
plot.scores(result)
```

```
## Don't know how to automatically pick scale for object of type data.frame. Defaulting to continuous.
```



```r
rates_data_frame <- plot.DET(result)
```



```
get.EER(rates_data_frame)
```

```
## [1] 0.3802508
```

Part 2 Min-Max Scaling

Read in the training data.

```
directory = "/Users/jen/Dropbox/CSCI 454/hw/trainingfaces2"
filenames <- list.files(path = directory)
training <- read.files(filenames, directory)
```

Function for Min-Max Scaling

```
MMScale <- function(dataset){
  col.mins <- apply(dataset, 2, min)
  intermediate.result <- t(t(dataset)-col.mins)
  col.maxes <- apply(dataset, 2, max)
  denominator <- col.maxes - col.mins
  dataset <- t(t(intermediate.result)/denominator)
  return(dataset)
}
```

Min-max scale.

```
training <- MMScale(training)
```

Center the data.

```r
X <- center.data(training)
```

Calculate eigen vectors.

```r
trans.EV <- calculate.eigens(X, 30)
```

Read in testing data.

```r
directory = "/Users/jen/Dropbox/CSCI 454/hw/testingfaces2"
filenames <- list.files(path = directory)
testing <- read.files(filenames, directory)
```

Min-max scale.

```r
testing <- MMScale(testing)
```

Normalize testing data by subtracting the mean of the training data.

```r
testing <- testing - training[ ,dim(training)[2]]
```

Make weight matrix.

```r
weight.matrices <- make.weight.matrix(trans.EV, testing)
```

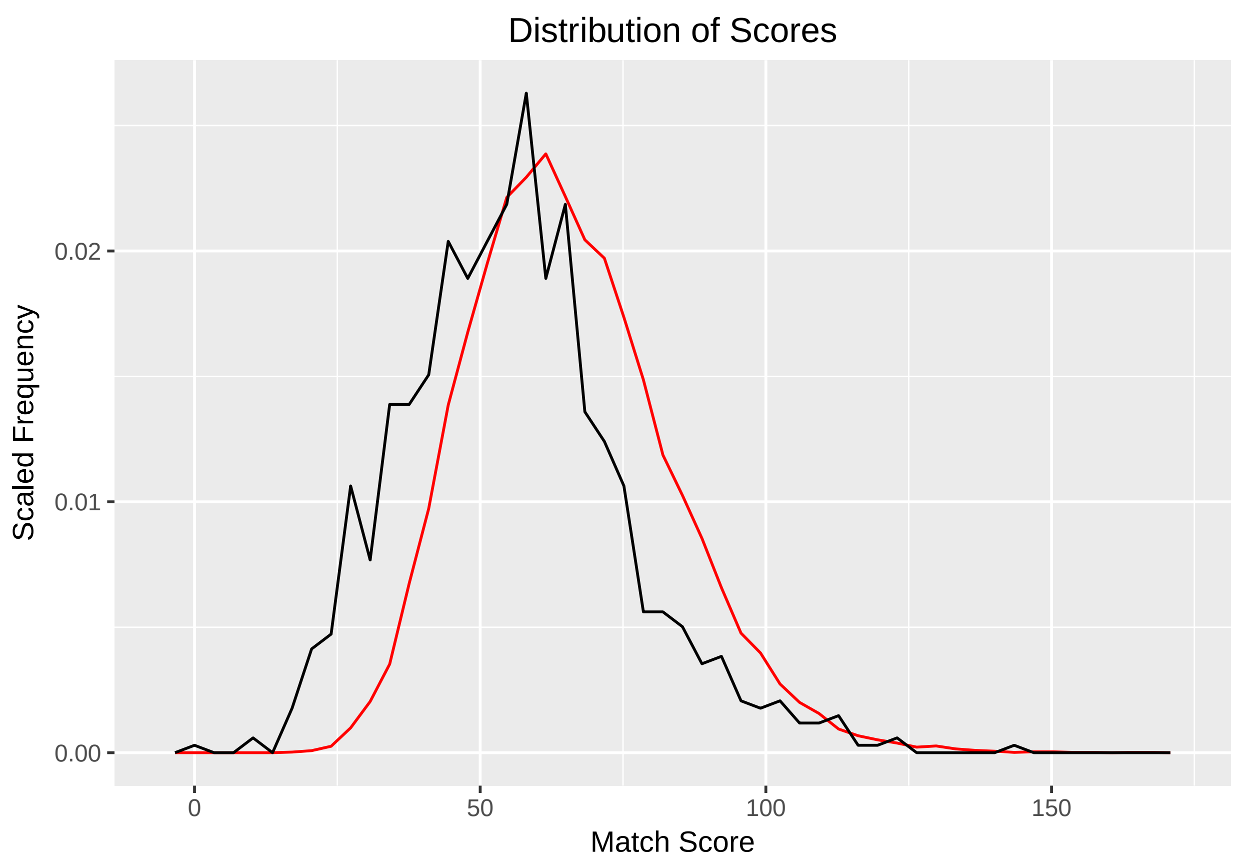
Make genuine and imposter datasets from weight matrix.

```r
result <- make.gen.imposter(weight.matrices)
```

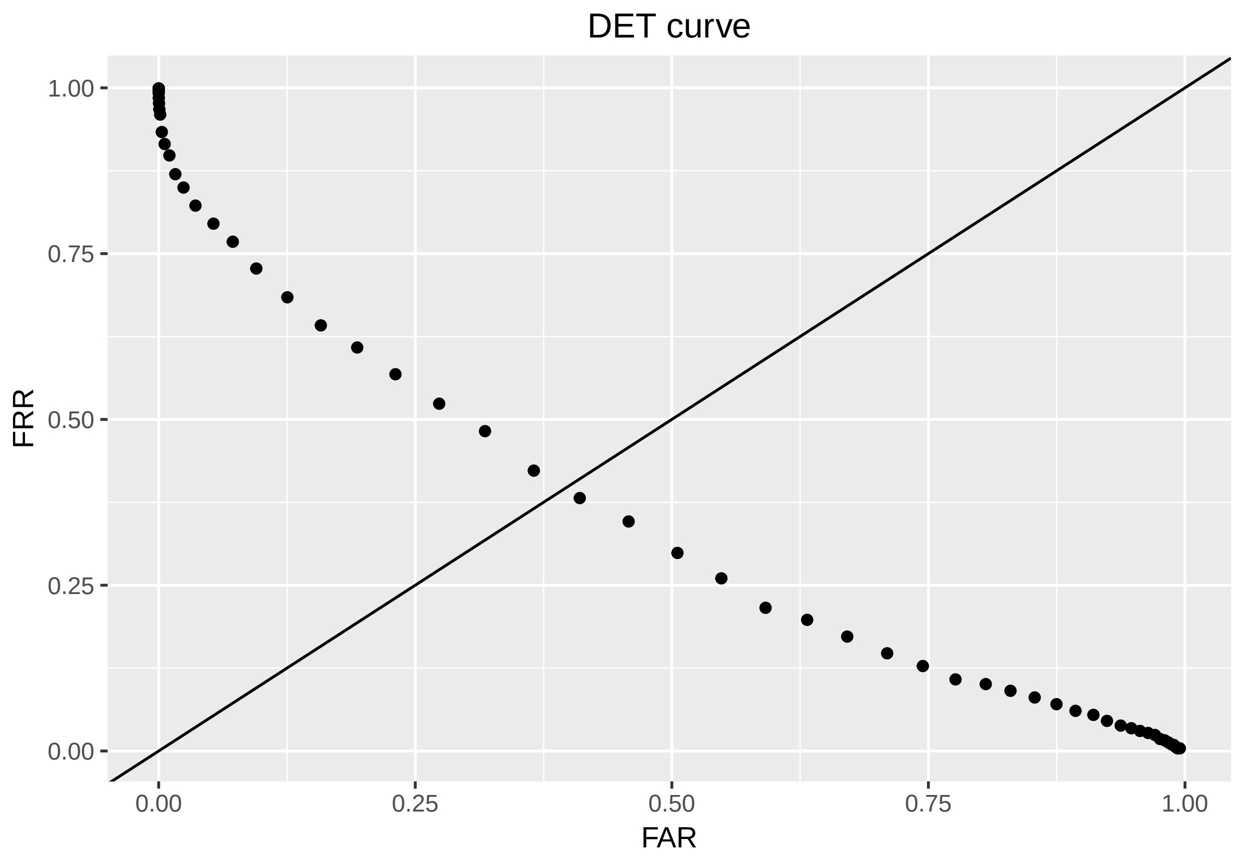
Use plotting functions.

```r
plot.scores(result)
```

```
## Don't know how to automatically pick scale for object of type data.frame. Defaulting to continuous.
```



```r
rates_data_frame <- plot.DET(result)
```



```
get.EER(rates_data_frame)
```

```
## [1] 0.3655186
```

Part 3 Aligned Data

Read in the training data.

```
directory = "/Users/jen/Dropbox/CSCI 454/hw/trainingrotated2"
filenames <- list.files(path = directory)
training <- read.files(filenames, directory)
```

Min-max scale.

```
training <- MMScale(training)
```

Center the data.

```
X <- center.data(training)
```

Calculate eigen vectors.

```
trans.EV <- calculate.eigens(X, 30)
```

Read in testing data.

```
directory = "/Users/jen/Dropbox/CSCI 454/hw/testingaligned2"
filenames <- list.files(path = directory)
testing <- read.files(filenames, directory)
```

Min-max scale.

```
testing <- MMScale(testing)
```

Normalize testing data by subtracting the mean of the training data.

```
testing <- testing - training[ ,dim(training)[2]]
```

Make weight matrix.

```
weight.matrices <- make.weight.matrix(trans.EV, testing)
```

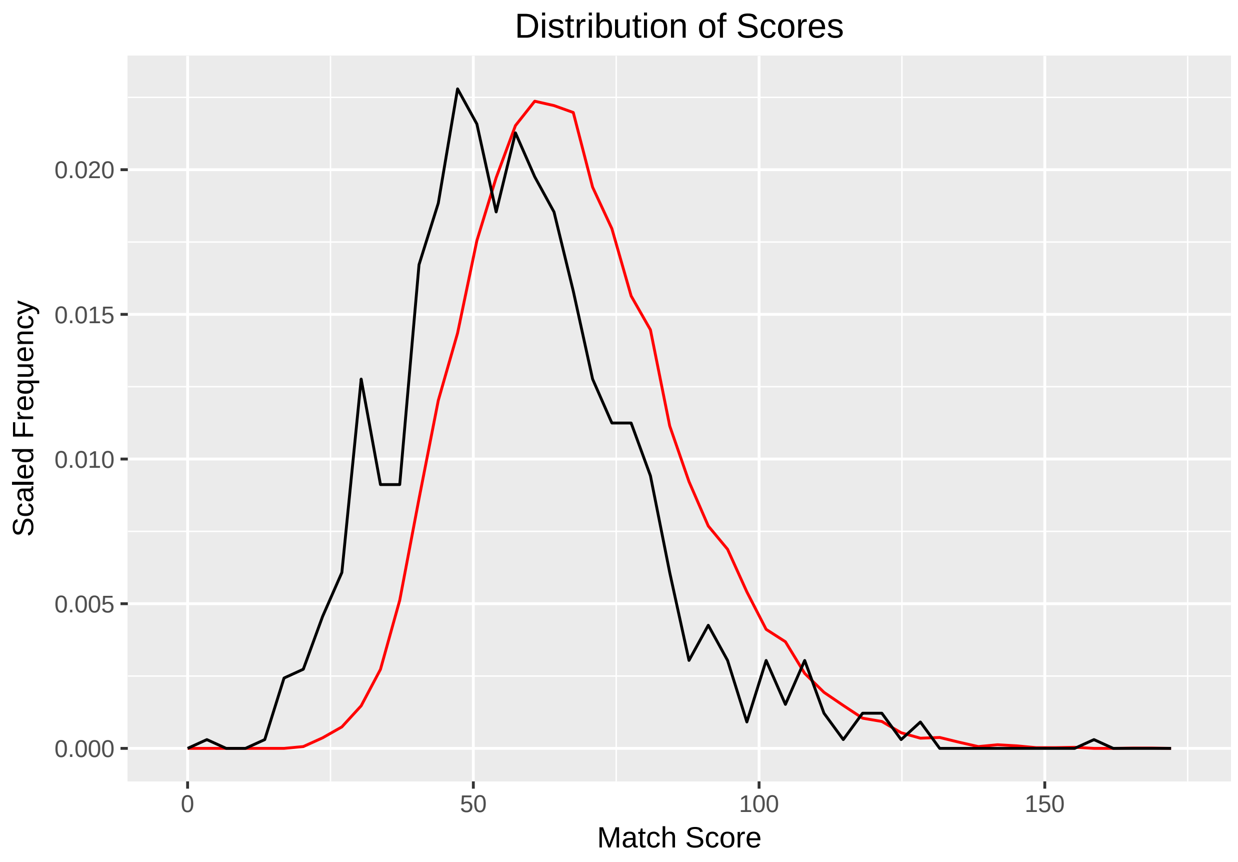
Make genuine and imposter datasets from weight matrix.

```
result <- make.gen.imposter(weight.matrices)
```

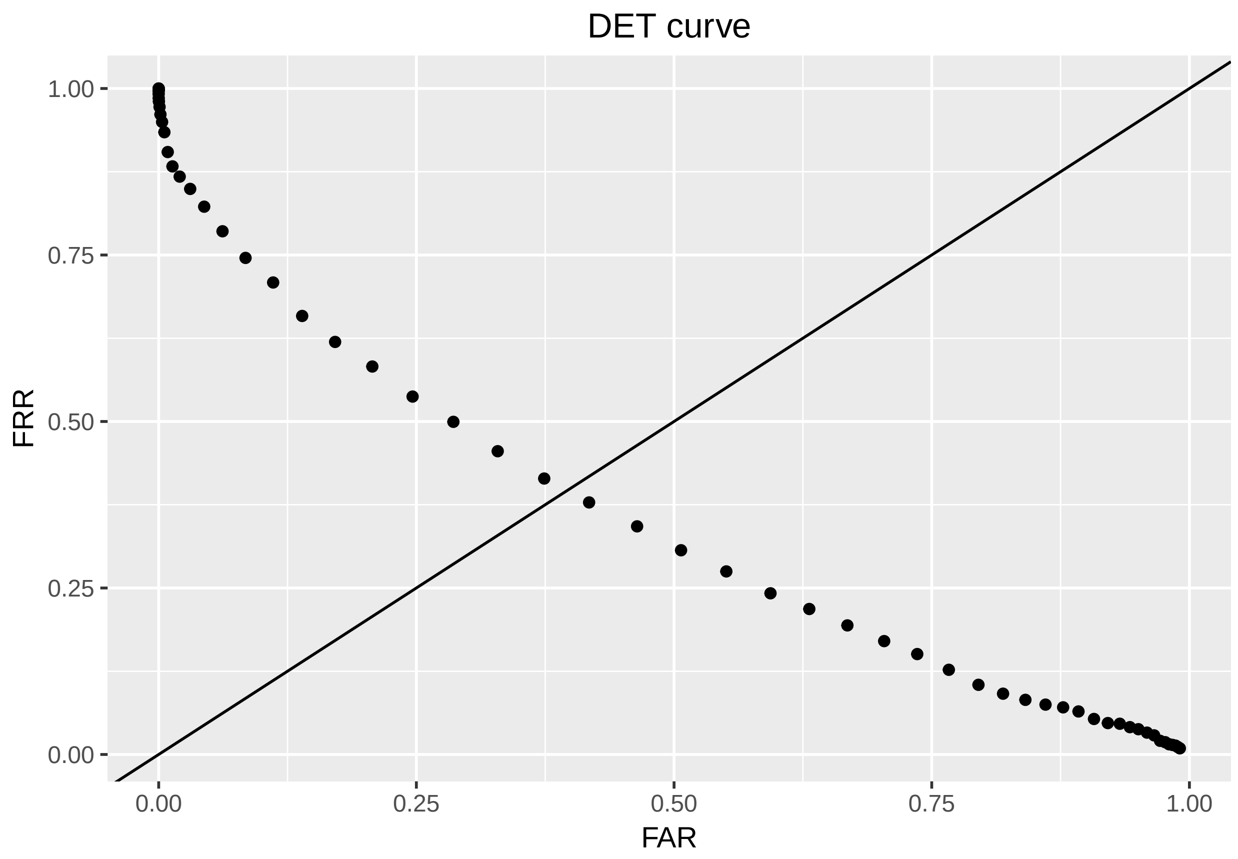
Use plotting functions.

```
plot.scores(result)
```

```
## Don't know how to automatically pick scale for object of type data.frame. Defaulting to continuous.
```



```
rates_data_frame <- plot.DET(result)
```



```
get.EER(rates_data_frame)
```

```
## [1] 0.3740448
```

Part 4

Helper Function for converting x and y rows/colums into sector numbers 1-9.

```
x.and.y.to.list.index <- function(x, y){
  if (x == 0) {
    if (y == 0) {
      final.index = 1
    } else if (y == 1) {
      final.index = 2
    } else {
      final.index = 3
    }
  } else if (x == 1){
    if (y == 0) {
      final.index = 4
    } else if (y == 1) {
      final.index = 5
    } else {
      final.index = 6
    }
  } else {
    if (y == 0) {
      final.index = 7
    } else if (y == 1) {
      final.index = 8
    } else {
      final.index = 9
    }
  }
  return(final.index)
}
```

Function for reading and dividing data into sectors.

```
dim.img <- 60
one.third <- dim.img[1]/3
len.array <- one.third ** 2

read.divide.files <- function(directory, filenames){

  sectors <- array(NA, c(len.array, length(filenames), 9))

  for (f in 1:length(filenames)) {
    img <- readPNG(paste(directory, filenames[f], sep = "/"))

    for (i in 0:2){
    min1 <- i*one.third +1
    max1 <- min1 + one.third - 1

      for (j in 0:2){
        min2 <- j * one.third + 1
        max2 <- min2 + one.third - 1

        current <- img[min1:max1, min2:max2]
        to.add <- as.vector(current)

        # get index from x and y using helper function
        index <- x.and.y.to.list.index(i, j)

        # assign using indices
        sectors[,f,index] <- to.add
      }
    }
  }
  return(sectors)
}
```

Read in the training data.

```
directory <- "/Users/jen/Dropbox/CSCI 454/hw/trainingrotated2"
filenames <- list.files(path = directory)
sectors <- read.divide.files(directory, filenames)
```

Normalize and center each sector. Store the mean of each row in each sector to normalize the testing later. Calculate the eigen vectors and keep the top 20.

```
weights <- array(NA, c(20, len.array, 9))
mean.rows.per.sector <- array(NA, c(len.array, 1 ,9))

for (i in 1:dim(sectors)[3]){
  training <- sectors[,,i]

  training <- MMScale(training)

  # Centering
  mean.rows <- apply(training, 1, mean)
  training <- cbind(training, mean.rows)
  mean.rows.per.sector[,,i] <- mean.rows

  X <- training[ , 1:dim(training)[2]-1]-training[ ,dim(training)[2]]

  trans.EV <- calculate.eigens(X, 20)

  weights[,,i] <- trans.EV
}
```

Read in the testing data.

```
directory <- "/Users/jen/Dropbox/CSCI 454/hw/testingaligned2"
filenames <- list.files(path = directory)
sectors <- read.divide.files(directory, filenames)
```

Normalize and center each sector. Normalize using the max of the column in the testing data. Center using the mean of the row in the training data.

```
for (i in 1:dim(sectors)[3]){
  testing <- sectors[,,i]

  testing <- MMScale(testing)

  # Subtract the mean of the training data for that sector
  sectors[,,i] <- testing - mean.rows.per.sector[,,i]
}
```

Multiply testing image sectors by the weights for those sectors and store.

```
weights.x.images <- array(NA, c(20, length(filenames), 9))

for (i in 1:dim(sectors)[3]){
  current.img <- sectors[,,i]
  current.weights <- weights[,,i]
  weights.x.images[,,i] <- current.weights %*% current.img
}
```

Make a boolean where matches == TRUE and imposters == FALSE for sorting later.

```
true.matches.false.imposters <- c()

for (i in 1:length(filenames)){
    j <- i+1
    while (j <= length(filenames)){
      image1 <- filenames[i]
      subject1 <- substr(image1, 2, 3)
      image2 <- filenames[j]
      subject2 <- substr(image2, 2, 3)

        if (subject1 == subject2) {
          true.matches.false.imposters <- c(true.matches.false.imposters, TRUE)
        } else {
          true.matches.false.imposters <- c(true.matches.false.imposters, FALSE)
        }
      j <- j+1
    }
}
```

For each image in each sector, compare to all other images and store in a matrix. The rows of the matrix will be each comparison. There will be 9 columns in the matrix that correspond to the difference obtained from each sector.

```
distance.scores.to.be.weighted <- c()

for (d in 1:dim(sectors)[3]){
  current.col.to.add <- c()
  current <- weights.x.images[,,d]
  for (i in 1:length(filenames)){
    j <- i+1
    while (j <= length(filenames)){
      weight1 <- current[, i]
      weight2 <- current[, j]
      weight.diff <- sum(abs(weight1-weight2))

      current.col.to.add <- c(current.col.to.add, weight.diff)
      j <- j+1
    }
  }
  distance.scores.to.be.weighted <- cbind(distance.scores.to.be.weighted, current.col.to.add)
  print(d)
}
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
## [1] 6
## [1] 7
## [1] 8
## [1] 9
```

For each sector, weight by the weight of the sector.

```
weights <- c(1, 2, 1, 2, 2, 2, 1, 2, 1)

weighted.scores <- t(t(distance.scores.to.be.weighted) * weights)
```

For each comparison, sum up the weighted differences.

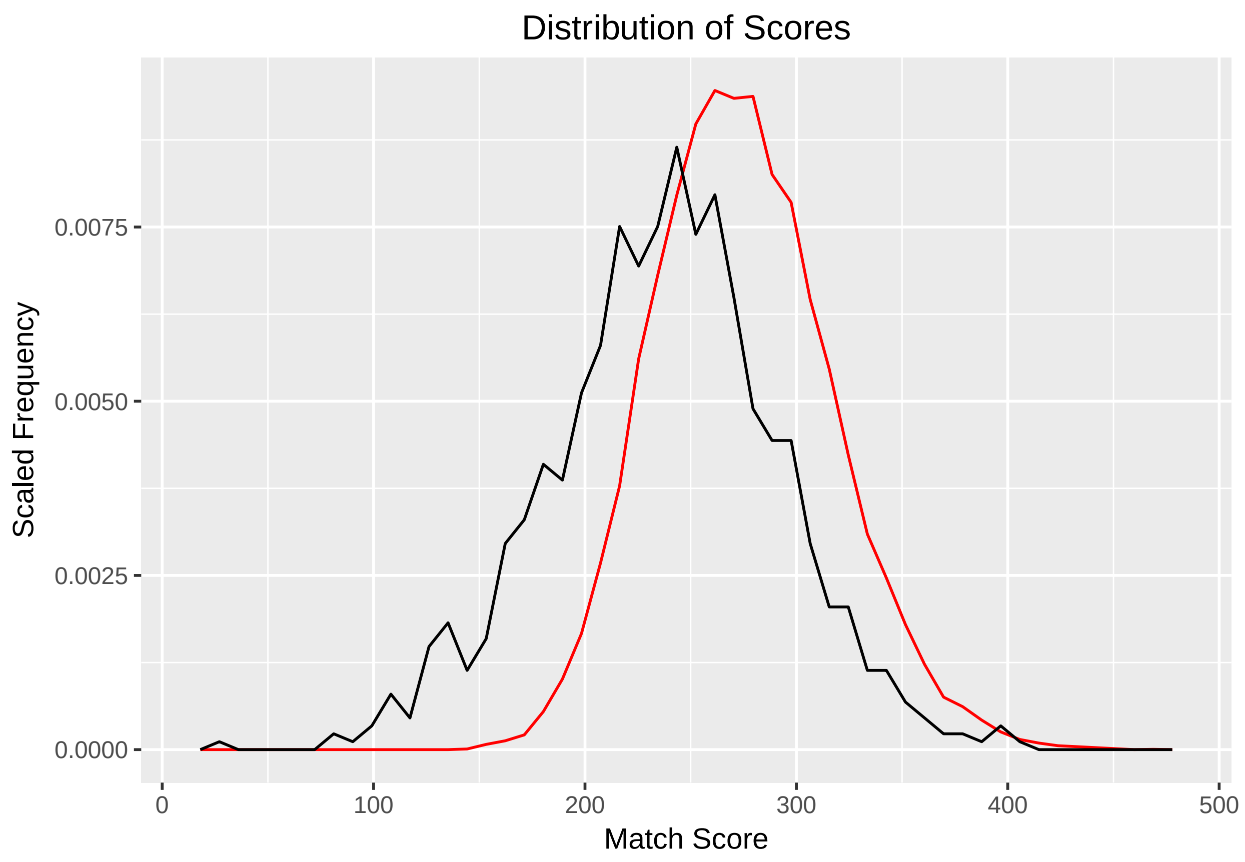
```
total.score <- apply(weighted.scores, 1, sum)
```

Use the boolean table to sort into imposter and genuine.

```
scores <- data.frame(cbind(true.matches.false.imposters, total.score))

genuine <- scores %>% filter(true.matches.false.imposters == 1) %>% select(total.score)

imposter <- scores %>% filter(true.matches.false.imposters == 0) %>% select(total.score)
```

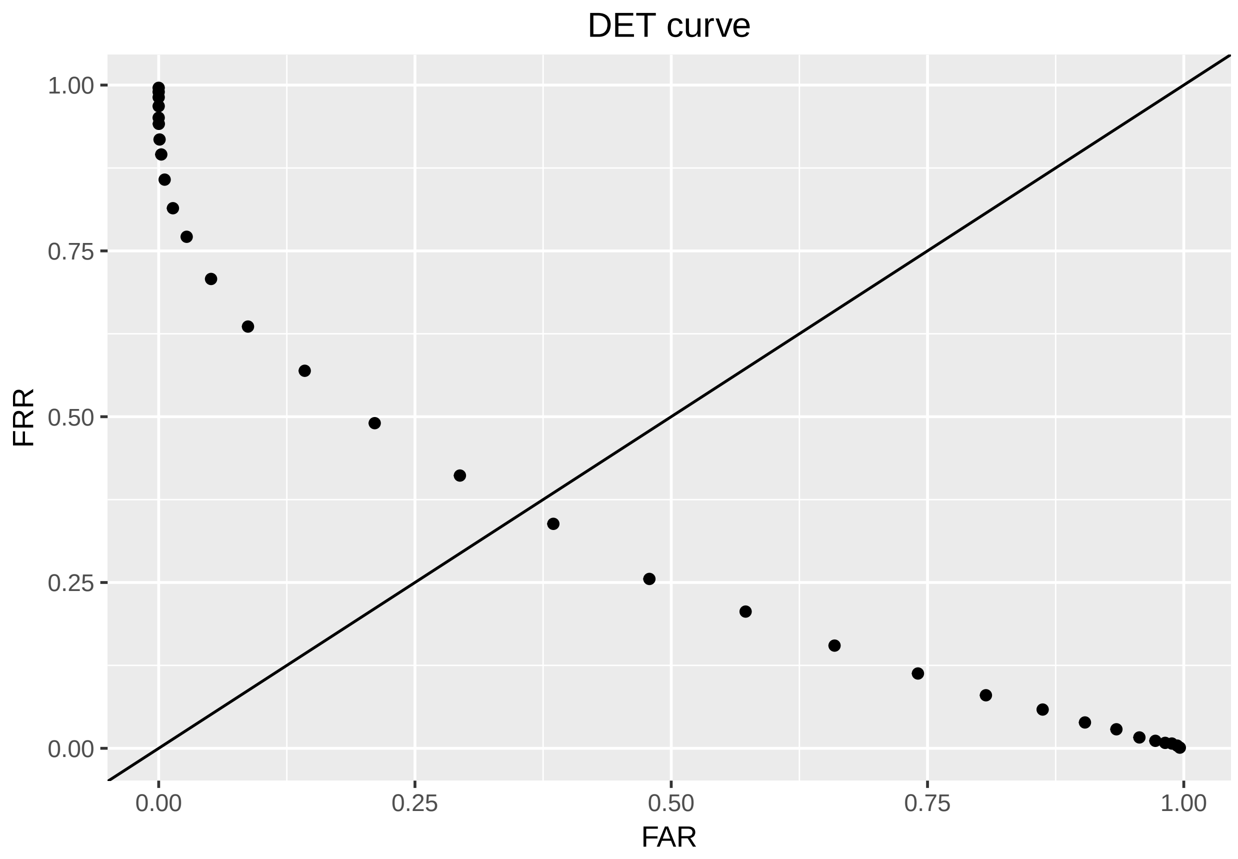
Plot distribution from HW1

```
theme_update(plot.title = element_text(hjust = 0.5))

imposter <- as.data.frame(imposter)
genuine <- as.data.frame(genuine)

#Scale using the density function and plot using ggplot's geom_freqpoly.
ggplot() + geom_freqpoly(data = imposter, aes(x = total.score, y = ..density..), bins = 50, color = "re
```



DET from HW1

```
FAR_vs_FRR <- NULL

for (t in seq(from = 100, to = 400, by = 10)){

  false_accept_count <- sum(imposter < t)
  false_accept_rate <- false_accept_count/dim(imposter)[1]
  false_reject_count <- sum(genuine > t)
  false_reject_rate <- false_reject_count/dim(genuine)[1]
  current_row <- c(false_accept_rate, false_reject_rate)
  FAR_vs_FRR<- rbind(FAR_vs_FRR, current_row)
}

rates_data_frame <- as.data.frame(FAR_vs_FRR)
colnames(rates_data_frame) <- c("FAR", "FRR")
ggplot(rates_data_frame, aes(x=FAR, y = FRR)) + geom_point() + geom_abline(slope = 1, intercept = 0) + 
```



EER from HW1

```
rates_data_frame$larger <- rates_data_frame$FAR > rates_data_frame$FRR

far_is_smaller <- rates_data_frame[rates_data_frame$larger=="FALSE", ]
lower_bound <- max(far_is_smaller$FAR)
far_is_larger <- rates_data_frame[rates_data_frame$larger=="TRUE", ]
upper_bound <- min(far_is_larger$FAR)

EER <- mean(lower_bound, upper_bound)
print(EER)
```

```
## [1] 0.2938529
```