Using SVM and Random Forests to learn thresholds from FCM density data

Code idea

The main idea is: see the problem of learn thresholds as a classification problem. Each sample has 'n' features (128 for example), a set of features, and each feature has x and y values. This set of features generates the density graph for the sample. So, see the problem as a classification problem means that each feature can be classified.

Since for each sample we have four thresholds (two for channel A and two for channel B), we are going to deal with three possible classes:

- **left**:features that are before from the low threshold and also (consequently) before from the high threshold
- **inside**: features that are between the low and high thresholds
- **right**: features that are after low and high thresholds

Considering this, we use SVM/RandomForest to learn a model using some samples for training and then use the learned model to predict the classes of each feature from test samples. Once we have the predicted classes, the process to obtain the low and high thresholds is simple: just save the greatest and the lowest **x** value from the features that were classified as **inside**. The lowest value is going to be our predicted low threshold and the greatest is going to be our predicted high threshold. Having all this information we are able to compute precision, recall and f-measure values, comparing the predicted thresholds with the actual thresholds.

Code structure

The code consists of a big loop to run using all the specified cell populations and plot all the results in one boxplot. A version that receives the run options as parameters is also implemented, including a python script to run everything automatically. However, using this version there will be one boxplot for each cell population (the boxplots will not be in the same plot). Code phases:

1. Run configuration

1.1 Libraries and parameters

Libraries for SVM and Random Forest; functions.R for some useful functions.

```
library(e1071)
library(randomForest)
source('functions.R')
```

1.2 Cell population list to use in the experiment

Add the cell populations that you want to predict thresholds in this list.

```
cell_population_list <- c(
"NOT(Granulocyte Pre)", #easy
"CD3 T-cell", #intermediate
"Plasma" #difficult
)
```

1.3 Learning technique to use

Choose **"svm"** to run support vector machines or **"rf"** to run random forest

```
learn_technique <- "rf"
```

1.4 Kernel to use in the experiment (if SVM)

Choose the kernel for the SVM: linear, polynomial, radial or sigmoid

```
kernel <- "linear"
```

1.5 Lists to store f_measure related information

```
f_measure_list <- list()
precision_list <- list()
recall_list <- list()
counter_fmeasure <- 1 # counter for f_measure: sum one for each population used in the experiment
```

1.6 Variables for loop control (how many runs for each experiment)

```
run_counter <- 1 # counter
run_times <- 2 # limit
```

1.7 Variables for experiment control

Configure the run options

```
numFeatures <- 128 # number of features for each sample
numTrain <- 10 # number of samples to use for training

# use this line if you want to test with the all other data not used for training
#numTest <- length(files) - numTrain -1
# use this line if you want to especify exactly how many samples to use for test
numTest <- 2

numFiles <- numTrain + numTest # number of files to read
# flag for plot each sample with the thresholds (one pdf for each group of test samples).
# TRUE if you want to plot
plot_flag <- TRUE
```

2. Get data from files

Read the .rds data at the "~/trainingFiles" directory. Save the density and threshold information in tables.

```
# List of files to read
files <- list.files(path = "trainingFiles", full.names = T, recursive = F, pattern = '*.rds')

# Matrix for channel densities
densXchanA <- matrix(NaN, ncol = numFeatures, nrow = length(files))
densYchanA <- matrix(NaN, ncol = numFeatures, nrow = length(files))
densXchanB <- matrix(NaN, ncol = numFeatures, nrow = length(files))
densYchanB <- matrix(NaN, ncol = numFeatures, nrow = length(files))
# Matrix for threhsolds (high and low)
```

```
threshA <- matrix(NaN, ncol = 2, nrow = length(files))
threshB <- matrix(NaN, ncol = 2, nrow = length(files))

# sample matrix in the format to use SVM/RF algorithm
# 2 columns (one for X value and one for Y value) and each row is a feature (with X and Y values)
sample_matrixA <- matrix(NaN, ncol= 2, nrow = numFeatures * numTrain)
sample_matrixB <- matrix(NaN, ncol= 2, nrow = numFeatures * numTrain)

# List of samples
samples <- list()
# List of samples file names (used in the function to compute f-measure)
samples_names <- list()

# ========================= GET DATA FROM FILES ================================
i <- 1
for (f in files){
  print(i)
  s <- readRDS(f)
  samples[[f]] <- s

  samples_names[[i]] <- sub("trainingFiles","",f[1])
  densXchanA[i, ] <- s[[cell_population]]@densitiesA[[as.character(numFeatures)]]$x
  densYchanA[i, ] <- s[[cell_population]]@densitiesA[[as.character(numFeatures)]]$y
  densXchanB[i, ] <- s[[cell_population]]@densitiesB[[as.character(numFeatures)]]$x
  densYchanB[i, ] <- s[[cell_population]]@densitiesB[[as.character(numFeatures)]]$y

  lowA <- getLowThresholdA(s[[cell_population]], numFeatures)
  highA <- getHighThresholdA(s[[cell_population]], numFeatures)
  lowB <- getLowThresholdB(s[[cell_population]], numFeatures)
  highB <- getHighThresholdB(s[[cell_population]], numFeatures)
  threshA[i, ] <- c(lowA, highA)
  threshB[i, ] <- c(lowB, highB)

  i <- i + 1
  if (i > numFiles){
    break
  }
}
```

3. Random shuffle

```
perm <- sample(length(samples))

samples <- samples[perm]
samples_names <- samples_names[perm]
densXchanA <- densXchanA[perm, ]
densYchanA <- densYchanA[perm, ]
densXchanB <- densXchanB[perm, ]
densYchanB <- densYchanB[perm, ]
threshA <- threshA[perm, ]
threshB <- threshB[perm, ]
```

4. Classification

Modify the data representation in order to use the SVM/RandomForest functions. **4.1 Dealing with training set**

Modifying the data representation for the test set. Assigning a class for each set of X and Y values (feature) and saving this infomation in a vector (set the response and predictors for the learn function).

```
# ---------------------- Dealing with Training set -------------------------
i <- 1
j <- 1
k <- 1
class_vectorA <- c()
class_vectorB <- c()
while (i <= numTrain){ # (row)
  j <- 1
  flag_leftB <- FALSE
  flag_rightB <- FALSE
  while(j <= numFeatures){ # x,y value for each sample
    sample_matrixA[k, ] <- c(densXchanA[i,j],densYchanA[i,j])
    sample_matrixB[k, ] <- c(densXchanB[i,j],densYchanB[i,j])

    #dealWith sampleA: classify each sample_matrix row as left, right or inside and save into a vector
    # each element in the class_vector is a classifier for each row in the sample_matrix
    if(densXchanA[i,j] < threshA[i,1]){
      # left
      class_vectorA <- c(class_vectorA, "left")
    }
    else if(densXchanA[i,j] > threshA[i,2]){
      # right
      class_vectorA <- c(class_vectorA, "right")
    }
    else{
      # inside
      class_vectorA <- c(class_vectorA, "inside")
    }

    #dealWith sampleB: same
    if(densXchanB[i,j] < threshB[i,1]){
      # left
      flag_leftB <- TRUE
      class_vectorB <- c(class_vectorB, "left")
    }
    else if(densXchanB[i,j] > threshB[i,2]){
      # right
      flag_rightB <- TRUE
      class_vectorB <- c(class_vectorB, "right")
    }
    else{
      # inside
      if (flag_leftB == FALSE){
        flag_leftB <- TRUE
        class_vectorB <- c(class_vectorB, "left")
      }
      else{
        class_vectorB <- c(class_vectorB, "inside")
      }
```

```
    }
    j <- j + 1
    k <- k + 1
  }
  if (flag_rightB == FALSE){
    class_vectorB <- class_vectorB[1:length(class_vectorB)-1]
    class_vectorB <- c(class_vectorB, "right")
  }
  i <- i + 1
}
```

4.2 Dealing with test set

Populate the test set (predictors matrix) in the right format. Assign a class is not necessary for the test set.

```
# just populating the test set into a format to use the SVM function
# class_vector is not necessary once we are going to predict the classification for each row of this ma
sample_matrix_testA <- matrix(ncol= 2, nrow = 0)
sample_matrix_testB <- matrix(ncol= 2, nrow = 0)
sample_matrix_aux <- matrix(NaN, ncol=2,nrow = numFeatures)
while (i <= numFiles){
  sample_matrix_aux[,1] <- densXchanA[i,]
  sample_matrix_aux[,2] <- densYchanA[i,]
  sample_matrix_testA <- rbind(sample_matrix_testA,sample_matrix_aux)
  sample_matrix_aux[,1] <- densXchanB[i,]
  sample_matrix_aux[,2] <- densYchanB[i,]
  sample_matrix_testB <- rbind(sample_matrix_testB,sample_matrix_aux)
  i <- i + 1
}
```

4.3 Transform the response vector

Transform the response vector: from string to factor

```
class_vectorA <- factor(class_vectorA)
class_vectorB <- factor(class_vectorB)
```

4.4 Learn

Learn a model for channel A and a model for channel B

```
# getPredModel: functions.R
model <- getPredModel(learn_technique, sample_matrixA, class_vectorA, kernel)
modelB <- getPredModel(learn_technique, sample_matrixB, class_vectorB, kernel)
```

The function getPredModel (use SVM or Random Forest):

```
getPredModel <- function(learn_technique, sample_matrix, class_vector, kernel){
  # learn one model for channel A and one model for channel B
  if (learn_technique == "svm"){
    model <- svm(sample_matrix, class_vector,  probability = TRUE, cross = 10, kernel = kernel)
  }
  else{
```

```
    model <- randomForest(sample_matrix, class_vector, importance=TRUE, proximity=TRUE)
  }
  print(model)
  summary(model)

  return (model)
}
```

4.5 Classify

Predict using the learned models and the test data

```
# predict using the test data
pred_A <- predict(model, sample_matrix_testA)
pred_B <- predict(modelB, sample_matrix_testB)
```

5. Compute the thresholds based on the predicted classes

Find thresholds based on the predicted values (the model classifies each row from the sample_matrix_testA/B) based on the predicted classification and save the predicted thresholds

```
# threshold matrix for channel A and channel B: two columns (for low and high thresholds values) and on
pred_threshA <- matrix(NaN, nrow = length(pred_A)/numFeatures, ncol = 2)
pred_threshB <- matrix(NaN, nrow = length(pred_B)/numFeatures, ncol = 2)

# compute the thresholds based on the classification prediction
i <- 1
k <- 1
while(k <= numTest){
  j <- 1
  highestA <- -1000
  lowestA <- 1000
  highestB <- -1000
  lowestB <- 1000
  #loop to infer the high and low thresholds: save the lowest and highest Y values ("inside" feature) f
  # in other words: take the highest and lowest Y values from the features classified as "inside"
  #                 the highest value is going to be the HIGH THRESHOLD
  #                 the lowest value is going to be the LOW THRESHOLD
  while (j <= numFeatures){
    # Dealing with Channel A
    if (as.character(pred_A[i]) == "inside"){
      if (sample_matrix_testA[i,1] < lowestA){
        lowestA <- sample_matrix_testA[i,1]
        lowestA_index <- i
      }
      else if (sample_matrix_testA[i,1] > highestA){
        highestA <- sample_matrix_testA[i,1]
        highestA_index <- i
      }
    }

    # Dealing with Channel B
    if (as.character(pred_B[i]) == "inside"){
      if (sample_matrix_testB[i,1] < lowestB){
```

```
        lowestB <- sample_matrix_testB[i,1]
        lowestB_index <- i+(numFeatures*numTest)
      }
      else if (sample_matrix_testB[i,1] > highestB){
        highestB <- sample_matrix_testB[i,1]
        highestB_index <- i+(numFeatures*numTest)
      }
    }
    j <- j + 1
    i <- i + 1
  }
  # once the highest and lowest ("inside") were found, save the threshold values
  pred_threshA[k, ] <- c(lowestA, highestA)
  pred_threshB[k, ] <- c(lowestB, highestB)

  k <- k + 1
}
```

6. Plot gates

Plot the gates with the predicted and actual thresholds.

```
# if plot_flag is TRUE, plot all the training samples with the actual thresholds and the predicted thres
if (plot_flag){
  plotPredThresh(learn_technique, numTrain, numTest, samples, cell_population, numFeatures, threshA, pr
}
```

The plotPredThresh function:

```
plotPredThresh <- function(learn_technique, numTrain, numTest, samples, cell_population, numFeatures, t
  if(learn_technique == "svm"){
    # output_plot: .pdf file names for the plots
    output_plot <- paste(cell_population,kernel,sep="_k-")
    output_plot <- paste(output_plot,"pdf",sep=".")
  }else{ # random forest
    # output_plot: .pdf file names for the plots
    output_plot <- paste(cell_population,"randomForest.pdf",sep="_")
  }
  # set pdf output name
  pdf(output_plot)
  print("Plotting:")
  print(output_plot)

  par(mfrow = c(4,5))
  i <- numTrain + 1
  j <- 1
  k <- 1
  #i <- 1 #plot everything
  #while(j < 1+numFiles*2){
  while(j < numTest*2+1){ # plot test data
    print(j)
    if(j %% 2 != 0){ # A
      plot(
```

```
        samples[[i]][[cell_population]]@densitiesA[[as.character(numFeatures)]]$x,
        samples[[i]][[cell_population]]@densitiesA[[as.character(numFeatures)]]$y,
        type = 'o',
        xlim=c(samples[[i]][[cell_population]]@densitiesA[[as.character(numFeatures)]]$x[1]-0.5,samples
      );
      abline(v = threshA[i, 1], col = 'red', lwd = 2, lty = 2)
      abline(v = threshA[i, 2], col = 'red', lwd = 2, lty = 2)
      abline(v = pred_threshA[k, 1], col = 'blue', lwd = 2, lty = 2)
      abline(v = pred_threshA[k, 2], col = 'blue', lwd = 2, lty = 2)
    }
    else{ # B
      print ("b")
      plot(
        samples[[i]][[cell_population]]@densitiesB[[as.character(numFeatures)]]$x,
        samples[[i]][[cell_population]]@densitiesB[[as.character(numFeatures)]]$y,
        type = 'o',
        xlim=c(samples[[i]][[cell_population]]@densitiesB[[as.character(numFeatures)]]$x[1]-1,samples[[
      );
      abline(v = threshB[i, 1], col = 'red', lwd = 2, lty = 2)
      abline(v = threshB[i, 2], col = 'red', lwd = 2, lty = 2)
      abline(v = pred_threshB[k, 1], col = 'blue', lwd = 2, lty = 2)
      abline(v = pred_threshB[k, 2], col = 'blue', lwd = 2, lty = 2)
      i <- i + 1
      k <- k + 1
    }
    j <- j + 1
  }
  dev.off()
}
```

7. Compute f-measure

```
# ============================ computing f-measure ==============================
# Vectors to store all the information to plot the boxplot
f_measures <- c()
precisions <- c()
recalls <- c()

# loop to compute the f_measure of each sample predicted thresholds (4 thresholds)
i <- 1
while (i <= nrow(pred_threshA)){
  #pred_threshA[i, 1] (low)
  #pred_threshA[i, 2] (high)
  #pred_threshB[i, 1] (low)
  #pred_threshB[i, 2] (high)

  #f_values returns c(precision, recall, f1)
  f_values <- f1.score(samples_names[[length(samples_names)-i+1]], cell_population, c(pred_threshA[i,1]

  print("precision, recall, f-measure")
  print(f_values)

  precisions <- c(precisions, f_values[1])
```

```
  recalls <- c(recalls, f_values[2])
  f_measures <- c(f_measures, f_values[3])
  i <- i + 1
}
```

The f1.score function:

```
f1.score <- function(sample, population, predicted.threshA, predicted.threshB)
{
  tg <- readRDS(paste('trainingFiles/', sample , sep = ''))
  gate.assignments <- tg[[population]]@gateAssignments
  population.norm <- tolower(str_replace_all(population, '[^a-zA-Z0-9]+', ''))
  fcs.name <- str_replace(sample, '\\.rds', '')
  exprs <- readRDS(paste('trainingFiles/fcs/', fcs.name, '.', population.norm, '.parent.fcs.rds', sep =
  predicted.gate.assignments <- matrix(T, nrow(exprs), 1)
  if (!is.nan(predicted.threshA[1]))
  {
    predicted.gate.assignments <- predicted.gate.assignments & exprs[,1] > predicted.threshA[1]
  }
  if (!is.nan(predicted.threshA[2]))
  {
    predicted.gate.assignments <- predicted.gate.assignments & exprs[,1] < predicted.threshA[2]
  }
  if (!is.nan(predicted.threshB[1]))
  {
    predicted.gate.assignments <- predicted.gate.assignments & exprs[,2] > predicted.threshB[1]
  }
  if (!is.nan(predicted.threshB[2]))
  {
    predicted.gate.assignments <- predicted.gate.assignments & exprs[,2] < predicted.threshB[2]
  }

  precision <- sum(gate.assignments & predicted.gate.assignments) / sum(predicted.gate.assignments)
  recall <- sum(gate.assignments & predicted.gate.assignments) / sum(gate.assignments)
  f1 <- 2 * precision * recall / (precision + recall)

  return (c(precision, recall, f1))

}
```

8. Boxplot

```
# ===================== plot box plot (all cell populations into one plot) ========================
print("printing boxplot")
#pdf(boxplot_output)

boxplot(xlab="Cell Population", ylab="f-measure",
      f_measure_list[[1]], # cell population
      f_measure_list[[2]],
      f_measure_list[[3]]
      )
title(main="f_measures", sub="", xlab="", ylab="")
```

```
boxplot(xlab="Cell Population", ylab="precision value",
      precision_list[[1]],
      precision_list[[2]],
      precision_list[[3]]
)
title(main="precision", sub="", xlab="", ylab="")

boxplot(xlab="Cell Population", ylab="recall value",
      recall_list[[1]],
      recall_list[[2]],
      recall_list[[3]]
      )
title(main="recall", sub="", xlab="", ylab="")

# clean all the lists for the next iteration (using another kernel for SVM)
f_measures_list <- list()
recall_list <- list()
precision_list <- list()
```

9. Save experiment run information

Save training/test data info and running info (kernel, cell population): save all the running parameters used in the experiment in order to be possilble to reproduce if necessary.

```
run_data <- matrix(NaN, ncol = 2, nrow = 3 + length(cell_population_list) + length(samples_names))

run_data[1,] = c("kernel", kernel)
run_data[2,] = c("training", toString(numTrain))
run_data[3,] = c("test", toString(numTest))

i <- 4
j <- 1
for (cell_population in cell_population_list){
run_data[i,] = c(paste("cellPopulation",toString(j),sep="-"),cell_population)
i <- i + 1
j <- j + 1
}

j <- 1
while (j < length(samples_names)-numTest){
run_data[i,] = c(paste("trainingFile",toString(j),sep="-"), samples_names[[j]])
j <- j + 1
i <- i + 1
}
test_file_counter<-1
while (j <= length(samples_names)){
run_data[i,] = c(paste("testFile",toString(test_file_counter),sep="-"), samples_names[[j]])
j <- j + 1
i <- i + 1
test_file_counter <- test_file_counter + 1
}

write.table(run_data, file=run_info_output, sep = ":", row.names=FALSE, col.names=FALSE)
```

10. Complete code # Libraries and parameters

```
library(e1071)
library(randomForest)
```

```
## randomForest 4.6-12

## Type rfNews() to see new features/changes/bug fixes.
```

```
source('functions.R')
```

```
## Loading required package: proxy

##
## Attaching package: 'proxy'

## The following objects are masked from 'package:stats':
##
##     as.dist, dist

## The following object is masked from 'package:base':
##
##     as.matrix

## Loaded dtw v1.18-1. See ?dtw for help, citation("dtw") for use in publication.
```

Cell population list to use in the experiment

```
cell_population_list <- c(
"NOT(Granulocyte Pre)", #easy
"CD3 T-cell", #intermediate
"Plasma" #difficult
)
```

Learning technique to use

Choose “svm” to run support vector machines or “rf” to run random forest

```
learn_technique <- "rf"
```

Kernel to use in the experiment (if SVM)

linear, polynomial, radial or sigmoid

```
kernel <- "linear"
```

Lists to store f_measure related information

```
f_measure_list <- list()
precision_list <- list()
recall_list <- list()
counter_fmeasure <- 1 # counter for f_measure: sum one for each population used in the experiment
```

Variables for loop control (how many runs for each experiment)

```
run_counter <- 1 # counter
run_times <- 2 # limit
```

Variables for experiment control

```
numFeatures <- 128 # number of features for each sample
numTrain <- 10 # number of samples to use for training

# use this line if you want to test with the all other data not used for training
#numTest <- length(files) - numTrain -1
# use this line if you want to especify exactly how many samples to use for test
numTest <- 2

numFiles <- numTrain + numTest # number of files to read
# flag for plot each sample with the thresholds (one pdf for each group of test samples).
# TRUE if you want to plot
plot_flag <- TRUE
```

Run info file output name

```
# run_info_output: .out file name (file with the running information used in the experiment)

if(learn_technique == "svm"){
  run_info_output <- paste("run-info", kernel,sep="_k-")
  run_info_output <- paste(run_info_output, "out",sep=".")
  # output_plot: .pdf file names for the boxplots
  boxplot_output <- paste("boxplot",kernel,sep="_k-")
  boxplot_output <- paste(boxplot_output,"pdf",sep=".")
}else{
  run_info_output <- "run-info_randomForest.out"
  # output_plot: .pdf file names for the boxplots
```

```
  boxplot_output <- "boxplot_randomForest.pdf"
}
```

Main run loop

Main run loop: 3 loops to run using all kernels with all cell populations and run multiple times each cell population

```
# ========== Cell population: cell population to predict thresholds =============
for (cell_population in cell_population_list){ #loop cell-population
  f_measure_list[[counter_fmeasure]] <- c(NA)
  precision_list[[counter_fmeasure]] <- c(NA)
  recall_list[[counter_fmeasure]] <- c(NA)

  # List of files to read
  files <- list.files(path = "trainingFiles", full.names = T, recursive = F, pattern = '*.rds')

  # Matrix for channel densities
  densXchanA <- matrix(NaN, ncol = numFeatures, nrow = length(files))
  densYchanA <- matrix(NaN, ncol = numFeatures, nrow = length(files))
  densXchanB <- matrix(NaN, ncol = numFeatures, nrow = length(files))
  densYchanB <- matrix(NaN, ncol = numFeatures, nrow = length(files))
  # Matrix for threhsolds (high and low)
  threshA <- matrix(NaN, ncol = 2, nrow = length(files))
  threshB <- matrix(NaN, ncol = 2, nrow = length(files))

  # sample matrix in the format to use SVM/RF algorithm
  # 2 columns (one for X value and one for Y value) and each row is a feature (with X and Y values)
  sample_matrixA <- matrix(NaN, ncol= 2, nrow = numFeatures * numTrain)
  sample_matrixB <- matrix(NaN, ncol= 2, nrow = numFeatures * numTrain)

  # List of samples
  samples <- list()
  # List of samples file names (used in the function to compute f-measure)
  samples_names <- list()

  # ========================== GET DATA FROM FILES ================================
  i <- 1
  for (f in files){
    print(i)
    s <- readRDS(f)
    samples[[f]] <- s

    samples_names[[i]] <- sub("trainingFiles","",f[1])
    densXchanA[i, ] <- s[[cell_population]]@densitiesA[[as.character(numFeatures)]]$x
    densYchanA[i, ] <- s[[cell_population]]@densitiesA[[as.character(numFeatures)]]$y
    densXchanB[i, ] <- s[[cell_population]]@densitiesB[[as.character(numFeatures)]]$x
    densYchanB[i, ] <- s[[cell_population]]@densitiesB[[as.character(numFeatures)]]$y

    lowA <- getLowThresholdA(s[[cell_population]], numFeatures)
    highA <- getHighThresholdA(s[[cell_population]], numFeatures)
    lowB <- getLowThresholdB(s[[cell_population]], numFeatures)
```

```
  highB <- getHighThresholdB(s[[cell_population]], numFeatures)
  threshA[i, ] <- c(lowA, highA)
  threshB[i, ] <- c(lowB, highB)

  i <- i + 1
  if (i > numFiles){
    break
  }
}

# run multiple times
run_counter <- 1
# ============  loop: run multiple times each cell population ===============
while (run_counter <= run_times){ # loop run counter: not necessary to read all the samples again
  print(cell_population)
  # ========= random shuffle into samples ==================
  perm <- sample(length(samples))

  samples <- samples[perm]
  samples_names <- samples_names[perm]
  densXchanA <- densXchanA[perm, ]
  densYchanA <- densYchanA[perm, ]
  densXchanB <- densXchanB[perm, ]
  densYchanB <- densYchanB[perm, ]
  threshA <- threshA[perm, ]
  threshB <- threshB[perm, ]

  # ========================== Classification phase ================================
  # ---------------------- Dealing with Training set -------------------------
  i <- 1
  j <- 1
  k <- 1
  class_vectorA <- c()
  class_vectorB <- c()
  while (i <= numTrain){ # (row)
    j <- 1
    flag_leftB <- FALSE
    flag_rightB <- FALSE
    while(j <= numFeatures){ # x,y value for each sample
      sample_matrixA[k, ] <- c(densXchanA[i,j],densYchanA[i,j])
      sample_matrixB[k, ] <- c(densXchanB[i,j],densYchanB[i,j])

      #dealWith sampleA: classify each sample_matrix row as left, right or inside and save into a vec
      # each element in the class_vector is a classifier for each row in the sample_matrix
      if(densXchanA[i,j] < threshA[i,1]){
        # left
        class_vectorA <- c(class_vectorA, "left")
      }
      else if(densXchanA[i,j] > threshA[i,2]){
        # right
        class_vectorA <- c(class_vectorA, "right")
      }
      else{
```

```
      # inside
      class_vectorA <- c(class_vectorA, "inside")
    }

    #dealWith sampleB: same
    if(densXchanB[i,j] < threshB[i,1]){
      # left
      flag_leftB <- TRUE
      class_vectorB <- c(class_vectorB, "left")
    }
    else if(densXchanB[i,j] > threshB[i,2]){
      # right
      flag_rightB <- TRUE
      class_vectorB <- c(class_vectorB, "right")
    }
    else{
      # inside
      if (flag_leftB == FALSE){
        flag_leftB <- TRUE
        class_vectorB <- c(class_vectorB, "left")
      }
      else{
        class_vectorB <- c(class_vectorB, "inside")
      }
    }
    j <- j + 1
    k <- k + 1
  }
  if (flag_rightB == FALSE){
    class_vectorB <- class_vectorB[1:length(class_vectorB)-1]
    class_vectorB <- c(class_vectorB, "right")
  }
  i <- i + 1
}

# ---------------------- Dealing with Test set -------------------------
# just populating the test set into a format to use the SVM function
# class_vector is not necessary once we are going to predict the classification for each row of thi
sample_matrix_testA <- matrix(ncol= 2, nrow = 0)
sample_matrix_testB <- matrix(ncol= 2, nrow = 0)
sample_matrix_aux <- matrix(NaN, ncol=2,nrow = numFeatures)
while (i <= numFiles){
  sample_matrix_aux[,1] <- densXchanA[i,]
  sample_matrix_aux[,2] <- densYchanA[i,]
  sample_matrix_testA <- rbind(sample_matrix_testA,sample_matrix_aux)
  sample_matrix_aux[,1] <- densXchanB[i,]
  sample_matrix_aux[,2] <- densYchanB[i,]
  sample_matrix_testB <- rbind(sample_matrix_testB,sample_matrix_aux)
  i <- i + 1
}

# transform string to factor (in order to use the SVM function)
class_vectorA <- factor(class_vectorA)
```

```
class_vectorB <- factor(class_vectorB)


# ================================ CLASSIFICATION ==================================
# getPredModel: functions.R
model <- getPredModel(learn_technique, sample_matrixA, class_vectorA, kernel)
modelB <- getPredModel(learn_technique, sample_matrixB, class_vectorB, kernel)

# predict using the test data
pred_A <- predict(model, sample_matrix_testA)
pred_B <- predict(modelB, sample_matrix_testB)

# ======================= Find Thresholds ==================================
# find thresholds based on the predicted values (the model classifies each row of the sample_matrix
# based on this classification, we collect the predicted thresholds

# threshold matrix for channel A and channel B: two columns (for low and high thresholds values) an
pred_threshA <- matrix(NaN, nrow = length(pred_A)/numFeatures, ncol = 2)
pred_threshB <- matrix(NaN, nrow = length(pred_B)/numFeatures, ncol = 2)

# compute the thresholds based on the classification prediction
i <- 1
k <- 1
while(k <= numTest){
  j <- 1
  highestA <- -1000
  lowestA <- 1000
  highestB <- -1000
  lowestB <- 1000
  #loop to infer the high and low thresholds: save the lowest and highest Y values ("inside" featur
  # in other words: take the highest and lowest Y values from the features classified as "inside"
  #                 the highest value is going to be the HIGH THRESHOLD
  #                 the lowest value is going to be the LOW THRESHOLD
  while (j <= numFeatures){
    # Dealing with Channel A
    if (as.character(pred_A[i]) == "inside"){
      if (sample_matrix_testA[i,1] < lowestA){
        lowestA <- sample_matrix_testA[i,1]
        lowestA_index <- i
      }
      else if (sample_matrix_testA[i,1] > highestA){
        highestA <- sample_matrix_testA[i,1]
        highestA_index <- i
      }
    }

    # Dealing with Channel B
    if (as.character(pred_B[i]) == "inside"){
      if (sample_matrix_testB[i,1] < lowestB){
        lowestB <- sample_matrix_testB[i,1]
        lowestB_index <- i+(numFeatures*numTest)
      }
      else if (sample_matrix_testB[i,1] > highestB){
```

```
        highestB <- sample_matrix_testB[i,1]
        highestB_index <- i+(numFeatures*numTest)
      }
    }
    j <- j + 1
    i <- i + 1
  }
  # once the highest and lowest ("inside") were found, save the threshold values
  pred_threshA[k, ] <- c(lowestA, highestA)
  pred_threshB[k, ] <- c(lowestB, highestB)

  k <- k + 1
}

# ================================== PLOT ========================================
# if plot_flag is TRUE, plot all the training samples with the actual thresholds and the predicted
if (plot_flag){
  plotPredThresh(learn_technique, numTrain, numTest, samples, cell_population, numFeatures, threshA
}

# ========================== computing f-measure ==============================
# Vectors to store all the information to plot the boxplot
f_measures <- c()
precisions <- c()
recalls <- c()

# loop to compute the f_measure of each sample predicted thresholds (4 thresholds)
i <- 1
while (i <= nrow(pred_threshA)){
  #pred_threshA[i, 1] (low)
  #pred_threshA[i, 2] (high)
  #pred_threshB[i, 1] (low)
  #pred_threshB[i, 2] (high)

  #f_values returns c(precision, recall, f1)
  f_values <- f1.score(samples_names[[length(samples_names)-i+1]], cell_population, c(pred_threshA[

  print("precision, recall, f-measure")
  print(f_values)

  precisions <- c(precisions, f_values[1])
  recalls <- c(recalls, f_values[2])
  f_measures <- c(f_measures, f_values[3])
  i <- i + 1
}

# save the f_measure related values into a list in order to plot all the cell population computed v
# list of vectors where each element from the list represent one cell population
precision_list[[counter_fmeasure]] <- c(precision_list[[counter_fmeasure]],precisions)
recall_list[[counter_fmeasure]] <- c(recall_list[[counter_fmeasure]],recalls)
f_measure_list[[counter_fmeasure]] <- c(f_measure_list[[counter_fmeasure]],f_measures)

run_counter <- run_counter + 1
```

```
  } # ========== end third loop: run counter =============

  # increment the fmeasure counter -> store the next cell population fmeasure info
  counter_fmeasure <- counter_fmeasure + 1
} # ========== end second loop: cell-population ============
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
## [1] 6
## [1] 7
## [1] 8
## [1] 9
## [1] 10
## [1] 11
## [1] 12
## [1] "NOT(Granulocyte Pre)"
##
## Call:
##  randomForest(x = sample_matrix, y = class_vector, importance = TRUE,      proximity = TRUE)
##                Type of random forest: classification
##                      Number of trees: 500
## No. of variables tried at each split: 1
##
##         OOB estimate of  error rate: 1.56%
## Confusion matrix:
##        inside right class.error
## inside    784     8  0.01010101
## right      12   476  0.02459016
##
## Call:
##  randomForest(x = sample_matrix, y = class_vector, importance = TRUE,      proximity = TRUE)
##                Type of random forest: classification
##                      Number of trees: 500
## No. of variables tried at each split: 1
##
##         OOB estimate of  error rate: 1.56%
## Confusion matrix:
##        inside left right class.error
## inside   1257    2     1 0.002380952
## left       10    0     0 1.000000000
## right       7    0     3 0.700000000
## [1] "Plotting:"
## [1] "NOT(Granulocyte Pre)_randomForest.pdf"
## [1] 1

## [1] 2
## [1] "b"

## [1] 3
```

```
## [1] 4
## [1] "b"

## [1] "precision, recall, f-measure"
## [1] 1.0000000 0.9993731 0.9996865
## [1] "precision, recall, f-measure"
## [1] 0.9727256 0.9999950 0.9861718
## [1] "NOT(Granulocyte Pre)"
##
## Call:
##  randomForest(x = sample_matrix, y = class_vector, importance = TRUE,      proximity = TRUE)
##                Type of random forest: classification
##                      Number of trees: 500
## No. of variables tried at each split: 1
##
##         OOB estimate of  error rate: 1.8%
## Confusion matrix:
##        inside right class.error
## inside    779    10  0.01267427
## right      13   478  0.02647658
##
## Call:
##  randomForest(x = sample_matrix, y = class_vector, importance = TRUE,      proximity = TRUE)
##                Type of random forest: classification
##                      Number of trees: 500
## No. of variables tried at each split: 1
##
##         OOB estimate of  error rate: 1.48%
## Confusion matrix:
##        inside left right class.error
## inside   1257    2     1 0.002380952
## left        9    1     0 0.900000000
## right       7    0     3 0.700000000
## [1] "Plotting:"
## [1] "NOT(Granulocyte Pre)_randomForest.pdf"
## [1] 1

## [1] 2
## [1] "b"

## [1] 3

## [1] 4
## [1] "b"

## [1] "precision, recall, f-measure"
## [1] 1.0000000 0.9718792 0.9857391
## [1] "precision, recall, f-measure"
## [1] 1.0000000 0.9684080 0.9839505
## [1] 1
## [1] 2
## [1] 3
## [1] 4
```

```
## [1] 5
## [1] 6
## [1] 7
## [1] 8
## [1] 9
## [1] 10
## [1] 11
## [1] 12
## [1] "CD3 T-cell"
##
## Call:
##  randomForest(x = sample_matrix, y = class_vector, importance = TRUE,      proximity = TRUE)
##                Type of random forest: classification
##                      Number of trees: 500
## No. of variables tried at each split: 1
##
##         OOB estimate of  error rate: 0.86%
## Confusion matrix:
##        inside right class.error
## inside    591     5 0.008389262
## right       6   678 0.008771930
##
## Call:
##  randomForest(x = sample_matrix, y = class_vector, importance = TRUE,      proximity = TRUE)
##                Type of random forest: classification
##                      Number of trees: 500
## No. of variables tried at each split: 1
##
##         OOB estimate of  error rate: 5.55%
## Confusion matrix:
##        inside left right class.error
## inside    738   27     3  0.03906250
## left       33  469     0  0.06573705
## right       8    0     2  0.80000000
## [1] "Plotting:"
## [1] "CD3 T-cell_randomForest.pdf"
## [1] 1

## [1] 2
## [1] "b"

## [1] 3

## [1] 4
## [1] "b"

## [1] "precision, recall, f-measure"
## [1] 0.9106249 0.9943785 0.9506606
## [1] "precision, recall, f-measure"
## [1] 1.0000000 0.8036117 0.8911139
## [1] "CD3 T-cell"
##
## Call:
```

```
##  randomForest(x = sample_matrix, y = class_vector, importance = TRUE,      proximity = TRUE)
##                Type of random forest: classification
##                      Number of trees: 500
## No. of variables tried at each split: 1
##
##         OOB estimate of  error rate: 1.17%
## Confusion matrix:
##        inside right class.error
## inside    593     8  0.01331115
## right       7   672  0.01030928
##
## Call:
##  randomForest(x = sample_matrix, y = class_vector, importance = TRUE,      proximity = TRUE)
##                Type of random forest: classification
##                      Number of trees: 500
## No. of variables tried at each split: 1
##
##         OOB estimate of  error rate: 5.16%
## Confusion matrix:
##        inside left right class.error
## inside    726   28     1  0.03841060
## left       29  486     0  0.05631068
## right       8    0     2  0.80000000
## [1] "Plotting:"
## [1] "CD3 T-cell_randomForest.pdf"
## [1] 1

## [1] 2
## [1] "b"

## [1] 3

## [1] 4
## [1] "b"

## [1] "precision, recall, f-measure"
## [1] 1.0000000 0.7443567 0.8534455
## [1] "precision, recall, f-measure"
## [1] 1.0000000 0.7561703 0.8611583
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
## [1] 6
## [1] 7
## [1] 8
## [1] 9
## [1] 10
## [1] 11
## [1] 12
## [1] "Plasma"
##
```

```
## Call:
##  randomForest(x = sample_matrix, y = class_vector, importance = TRUE,      proximity = TRUE)
##                Type of random forest: classification
##                      Number of trees: 500
## No. of variables tried at each split: 1
##
##         OOB estimate of  error rate: 1.33%
## Confusion matrix:
##        inside left class.error
## inside    467    9 0.018907563
## left        8  796 0.009950249
##
## Call:
##  randomForest(x = sample_matrix, y = class_vector, importance = TRUE,      proximity = TRUE)
##                Type of random forest: classification
##                      Number of trees: 500
## No. of variables tried at each split: 1
##
##         OOB estimate of  error rate: 1.17%
## Confusion matrix:
##        inside left right class.error
## inside    585    2     3 0.008474576
## left        4    6     0 0.400000000
## right       6    0   674 0.008823529
## [1] "Plotting:"
## [1] "Plasma_randomForest.pdf"
## [1] 1

## [1] 2
## [1] "b"

## [1] 3

## [1] 4
## [1] "b"

## [1] "precision, recall, f-measure"
## [1] 1.0000000 0.6666667 0.8000000
## [1] "precision, recall, f-measure"
## [1] 0.8902439 0.9972678 0.9407216
## [1] "Plasma"
##
## Call:
##  randomForest(x = sample_matrix, y = class_vector, importance = TRUE,      proximity = TRUE)
##                Type of random forest: classification
##                      Number of trees: 500
## No. of variables tried at each split: 1
##
##         OOB estimate of  error rate: 1.25%
## Confusion matrix:
##        inside left class.error
## inside    462    9 0.019108280
## left        7  802 0.008652658
```

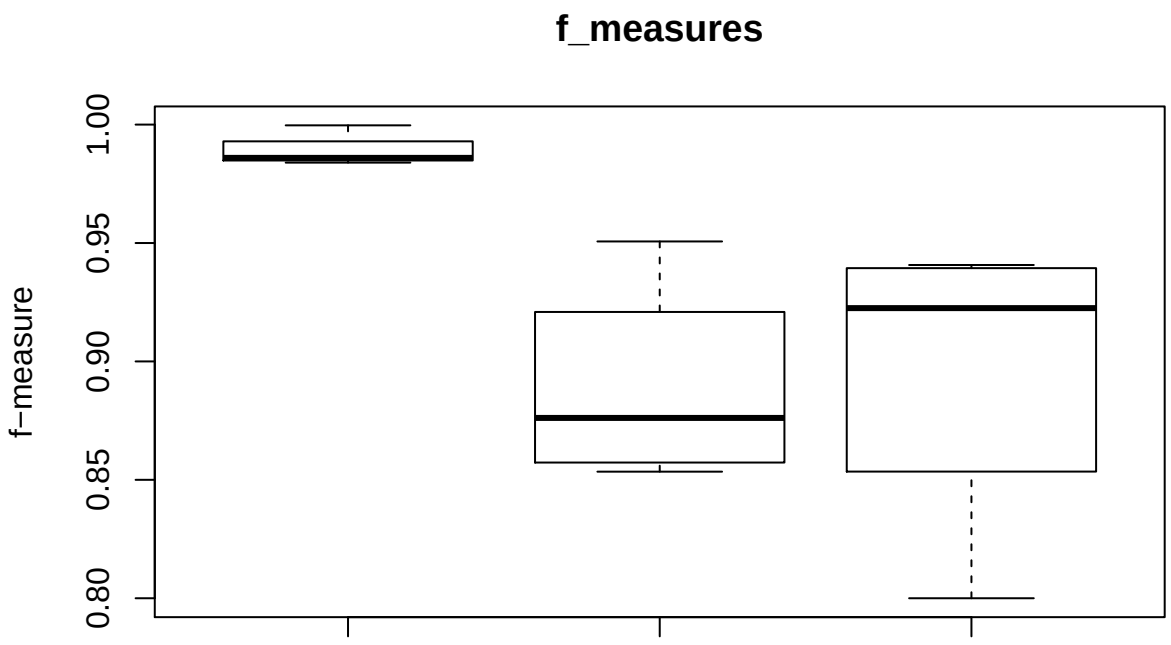
```
##
## Call:
##  randomForest(x = sample_matrix, y = class_vector, importance = TRUE,      proximity = TRUE)
##                Type of random forest: classification
##                      Number of trees: 500
## No. of variables tried at each split: 1
##
##         OOB estimate of  error rate: 1.8%
## Confusion matrix:
##        inside left right class.error
## inside    578    1     6  0.01196581
## left        7    3     0  0.70000000
## right       9    0   676  0.01313869
## [1] "Plotting:"
## [1] "Plasma_randomForest.pdf"
## [1] 1

## [1] 2
## [1] "b"

## [1] 3

## [1] 4
## [1] "b"

## [1] "precision, recall, f-measure"
## [1] 1.0000000 0.8297872 0.9069767
## [1] "precision, recall, f-measure"
## [1] 1.0000000 0.8833333 0.9380531
```

Plot boxplot

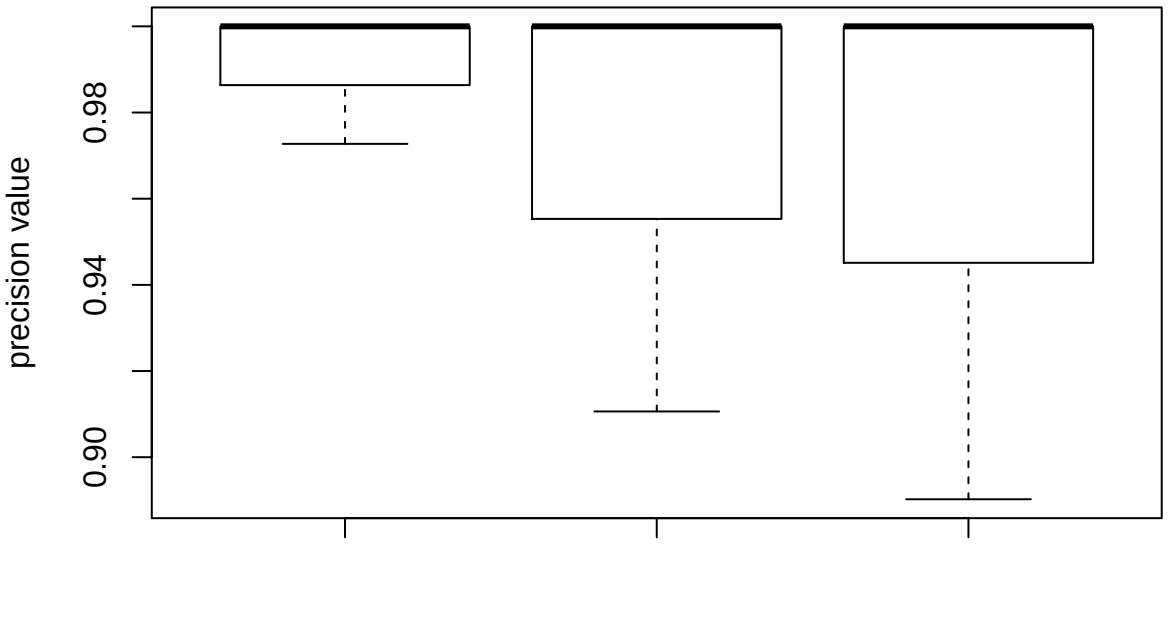
```
counter_fmeasure <- 1

# ====================== plot box plot (all cell populations into one plot) =========================
print("printing boxplot")
```

```
## [1] "printing boxplot"
```

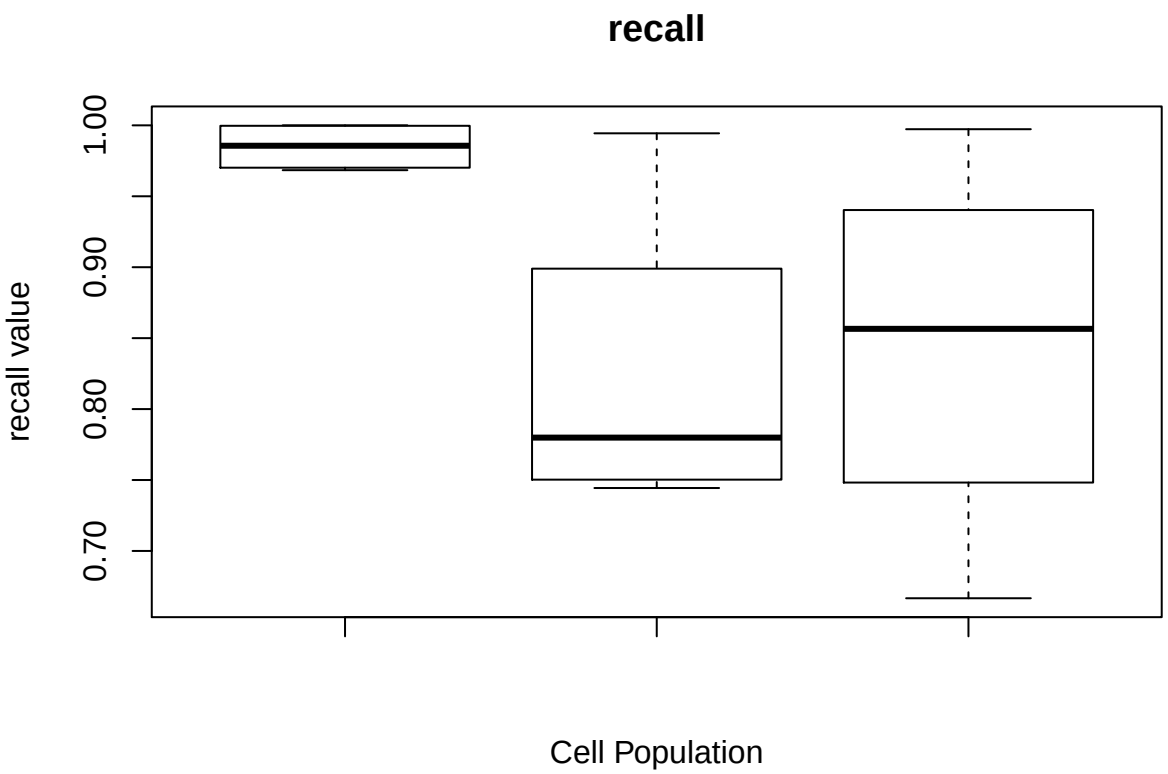
```
#pdf(boxplot_output)

boxplot(xlab="Cell Population", ylab="f-measure",
        f_measure_list[[1]], # cell population
        f_measure_list[[2]],
        f_measure_list[[3]]
        )
title(main="f_measures", sub="", xlab="", ylab="")
```




```
boxplot(xlab="Cell Population", ylab="precision value",
        precision_list[[1]],
        precision_list[[2]],
        precision_list[[3]]
)
title(main="precision", sub="", xlab="", ylab="")
```




```
boxplot(xlab="Cell Population", ylab="recall value",
        recall_list[[1]],
        recall_list[[2]],
        recall_list[[3]]
        )
title(main="recall", sub="", xlab="", ylab="")
```




```
# clean all the lists for the next iteration (using another kernel for SVM)
f_measures_list <- list()
recall_list <- list()
precision_list <- list()

#dev.off()
```

Save the experiment info

```
# ==================== save training/test data info and running info (kernel, cell population) =======
# save all the running parameters used in the experiment in order to be possilble to reproduce if nec

run_data <- matrix(NaN, ncol = 2, nrow = 3 + length(cell_population_list) + length(samples_names))

run_data[1,] = c("kernel", kernel)
run_data[2,] = c("training", toString(numTrain))
run_data[3,] = c("test", toString(numTest))

i <- 4
j <- 1
for (cell_population in cell_population_list){
  run_data[i,] = c(paste("cellPopulation",toString(j),sep="-"),cell_population)
  i <- i + 1
  j <- j + 1
}
```

```
j <- 1
while (j < length(samples_names)-numTest){
  run_data[i,] = c(paste("trainingFile",toString(j),sep="-"), samples_names[[j]])
  j <- j + 1
  i <- i + 1
}
test_file_counter<-1
while (j <= length(samples_names)){
  run_data[i,] = c(paste("testFile",toString(test_file_counter),sep="-"), samples_names[[j]])
  j <- j + 1
  i <- i + 1
  test_file_counter <- test_file_counter + 1
}

write.table(run_data, file=run_info_output, sep = ":", row.names=FALSE, col.names=FALSE)
```