Assignment 1

Ian Tongs

01/09/2020

```
# Load All Libraries:
library(dplyr)
library(readr)
library(tidyverse)
library(ggplot2)
library(gridExtra)
library(grid)
library(cowplot)
```

Question 1:

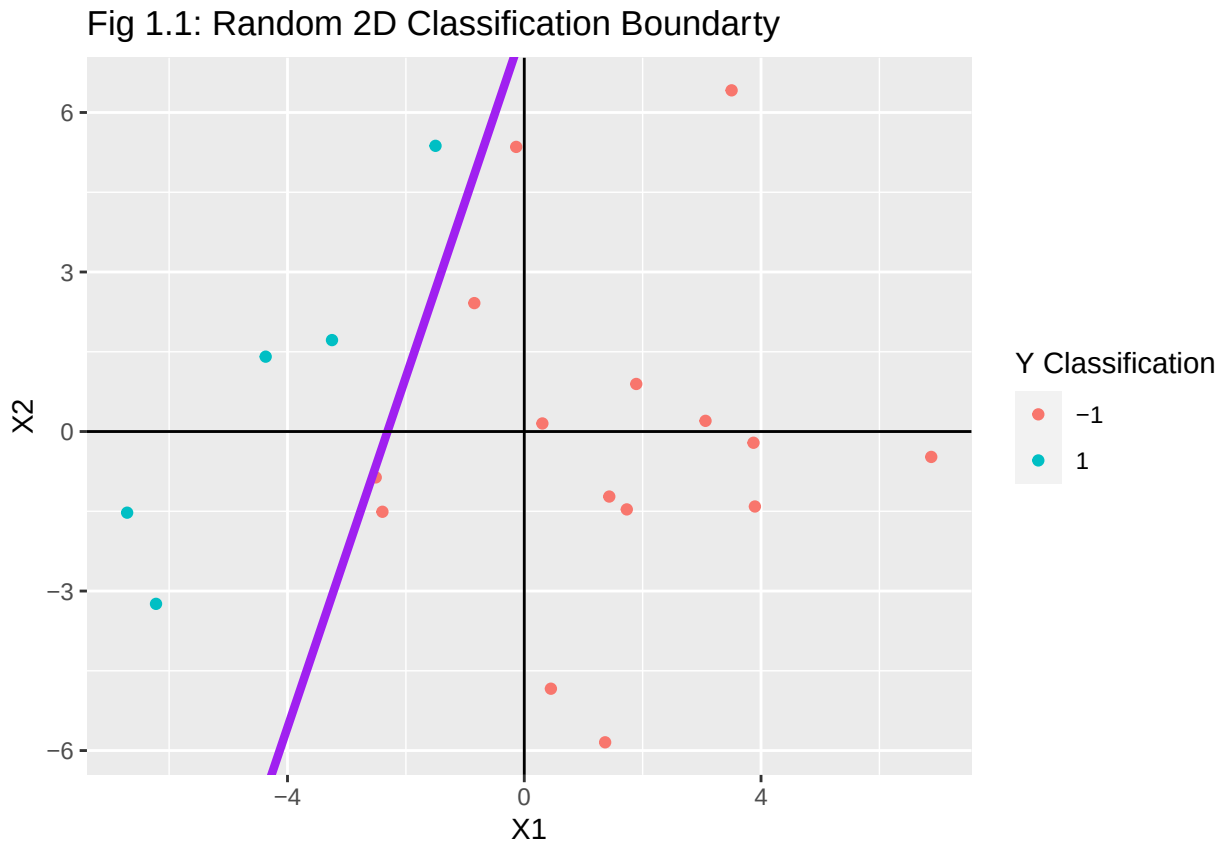
Perceptron Learning Algorithm Example in 2 Dimensions

Generating the sample dataset, function, and learning algorithm:

Construct our linearly separate dataset:

```
# What we need to do:
## Choose some function f in the 2D space
## Generate 20 random examples and assign by the function f
### This is now our training set
## Build a perceptron algorithm
## Train the perceptron algorithm on this dataset
## See how long it takes to converge as g -> f.


# Set Seed:
set.seed(27765369)

# Choose f, being a random line in two dimensions. As:
# Of the form: c1*X1 + c2*X2 + c3 = 0
# or rather: X2 = (-c1/c2)*X2 + (-c3/c2)

c1 <- runif(1, -3, 3)
c2 <- runif(1, -3, 3)
c3 <- runif(1, -3, 3)


# Create the random dataset
X1 <- rnorm(20, mean=0, sd=3)
```

```r
X2 <- rnorm(20, mean=0, sd=3)

# Check where to divide for X1:
dividing_line <- -(c1/c2)*X1 -(c3/c2)

# Use sign for Y if above this:
Y <- sign(X2 - dividing_line)


# Set the dataset:
Data_set <- tibble(X1, X2, 1, Y)


# Plot it:
Initial_plot <- ggplot(Data_set, aes(x=X1, y=X2)) +
  geom_point(aes(colour = factor(Y))) + labs(color = "Y Classification") +
  geom_abline(intercept = -(c3/c2), slope = -(c1/c2), color="purple", linetype="solid", size=1.5) +
  ggtitle("Fig 1.1: Random 2D Classification Boundarty") +
  geom_vline(xintercept = 0, color="black") +
  geom_hline(yintercept = 0, color="black")

Initial_plot
```

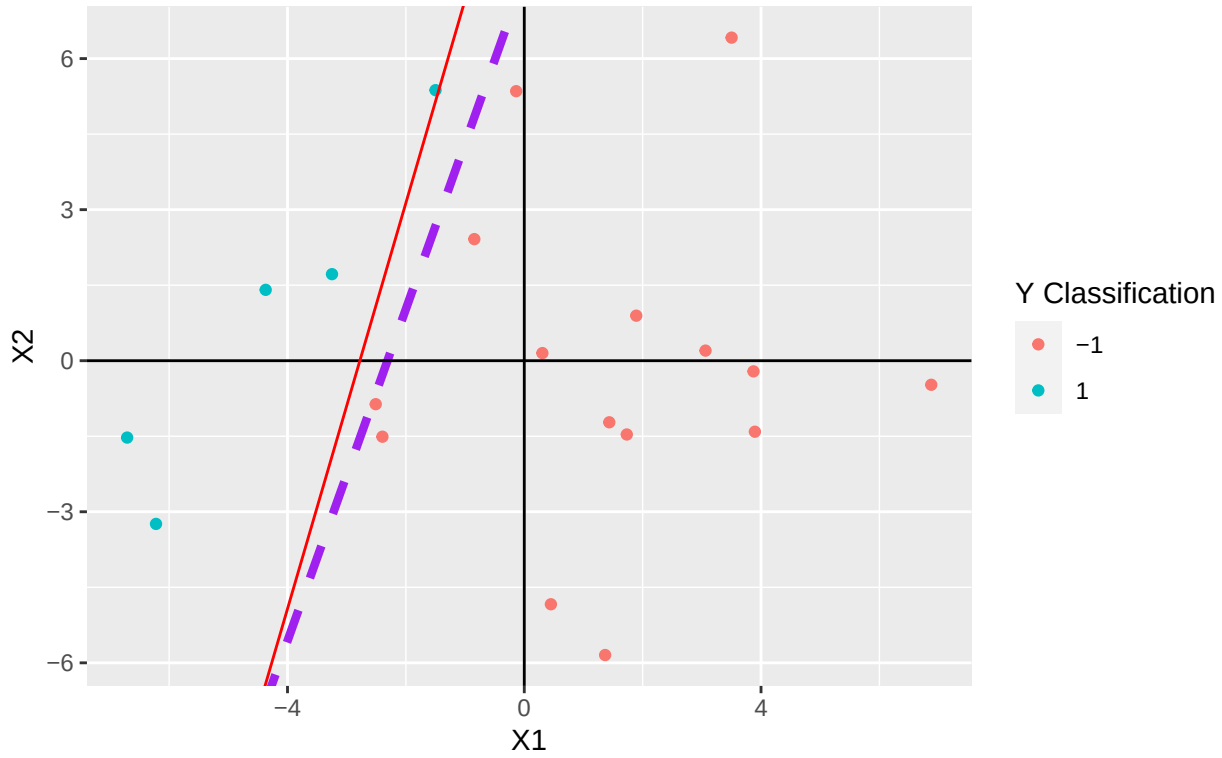



```r
# add legend for the line colours!
```

Run the Perceptron Learning Algorithm on the dataset and plot it:

```r
#### Now we want to implement perceptron:


# Set initial weights
weights = c(1, 2, 3)

# Start the Timing:
start_time <- Sys.time()

# Use the sign to assign initial outcomes:
h_val <- sign( as.matrix(Data_set %>% select(1:3)) %*% weights )


iterations <- 0
# Create a loop to repeatedly update the estimates:
while (sum(h_val == (Data_set %>% pull(Y))) < 20){
  for (i in (1:20)){
    # Primary if clause
    if (h_val[i] != Y[i]){
      weights <- weights + Y[i] *  as.matrix(Data_set %>% select(1:3))[i, ]
      # iterations += 1
      iterations <- iterations + 1
    }
  }
  h_val <- sign( as.matrix(Data_set %>% select(1:3)) %*% weights )
}

# End the Timing:
end_time <- Sys.time()

time <- end_time - start_time


Later_Plot <- ggplot(Data_set, aes(x=X1, y=X2)) +
  geom_point(aes(colour = factor(Y))) + labs(color = "Y Classification") +
  geom_abline(intercept = -(c3/c2), slope = -(c1/c2), color="purple", linetype="dashed", size=1.5) +
  ggtitle("Fig 1.2: Random 2D Classification Boundarty with Perceptron Boundary \nPurple is the True li
  geom_vline(xintercept = 0, color="black") +
  geom_hline(yintercept = 0, color="black") +
  geom_abline(intercept = (-weights[3]/weights[2]), slope = (-weights[1]/weights[2]), color="red")

Later_Plot
```

Fig 1.2: Random 2D Classification Boundarty with Perceptron Boundary
Purple is the True line, Red the PLA dividing line




```
# add legend for the line colours!
```

And report the iterations and time taken for the PLA to converge on this dataset:

```
print(paste("For this dataset, for convergence to occue with with PLA, we required", iterations, "updat
```

```
## [1] "For this dataset, for convergence to occue with with PLA, we required 75 updates."
```

```
print(paste("For this dataset, it took", time, "seconds for the PLA to converge to a solution."))
```

```
## [1] "For this dataset, it took 0.3801109790802 seconds for the PLA to converge to a solution."
```

From the graph, we saw that the PLA correctly separated all the sample points. However, there was a noticeable discrepance between the true dividing line and the PLA dividing line, as we would expect with a small sample size such as $n = 20$.

Question 2:

Single Bin and Multiple Bin sampling

Part A:

We may start by creating code to simulate the coin tosses:

```
# Build a coin flipper function - flipping 1000 coins 10 times each, and recording the frequency
# Select the following coins:
  # First coin flipped
  # A random coin
  # The minimum heads frequency

sample_coins <- c("Tails", "Heads")
flip_matrix <- matrix(ncol = 10, nrow = 1000)

# Loop for flipping 1000 coins 10 times each
for (i in 1:1000) {
  flip_matrix[i, ] <- sample(sample_coins, 10, replace = TRUE)
}

# Loop to find the minimum value:
min_val <- 10
min_id <- 0
for (i in 1:1000) {
  cur_count <- sum(flip_matrix[i, ] == "Heads")
  # make min if min
  if (min_val > cur_count) {
    min_val <- cur_count
    min_id <- i
  }
}

# Choose the three bits:
c_1 <- flip_matrix[1, ]
c_min <- flip_matrix[min_id, ]
c_rand <- flip_matrix[sample(1000, 1), ]

# Find v vals:
v_1 <- mean(c_1 == "Heads")
v_min <- mean(c_min == "Heads")
v_rand <- mean(c_rand == "Heads")
```

Now we can report our sample μ (average) for our three selected coins:

```
# Find Mu:
mu <- mean(v_1, v_min, v_rand)
mu
```

```
## [1] 0.8
```

We should note from theory that each of the three coins are drawn from iid bins (assuming they are fair coins) with $\mu_{theoretical} = 0.5$, as each coin flip set is a repeated bernoulli experiment.

Part B:

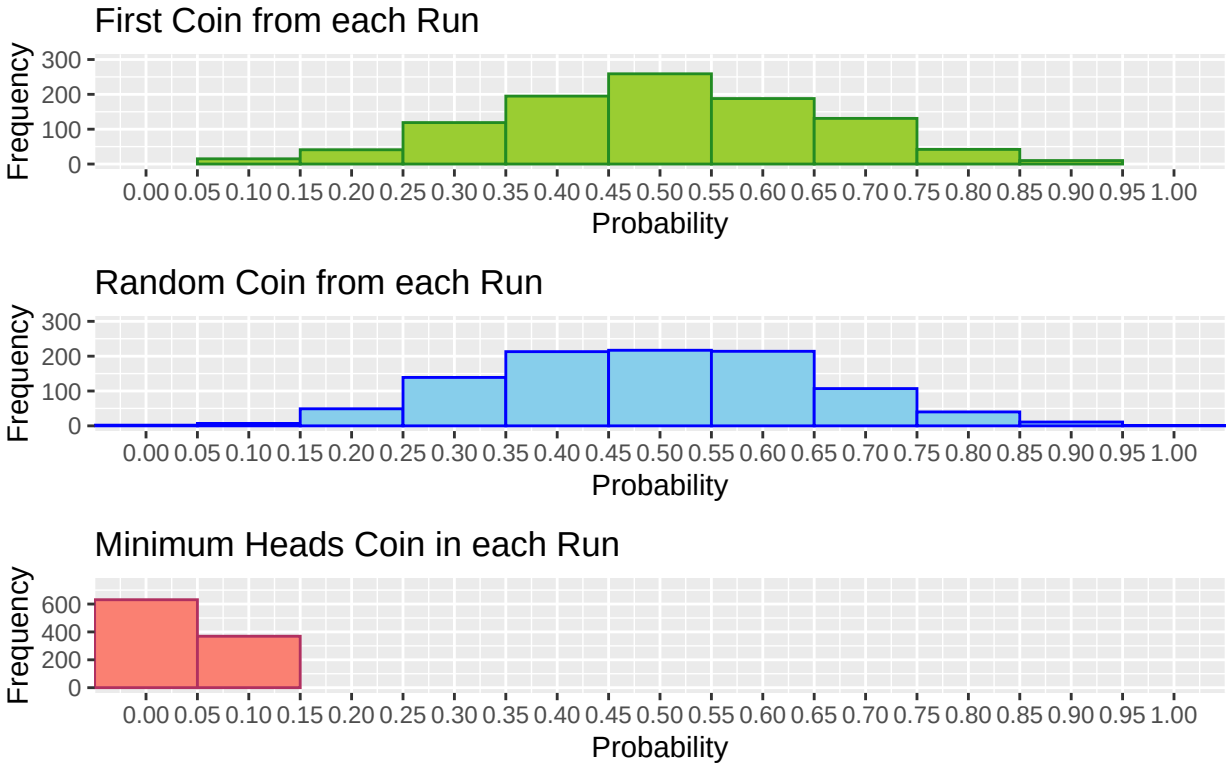
We can now repeat this process 1000 times to get our sample dataset:

```
# repeat process 100000 times and plot the histograms

# Assign the number of runs:
runs <- 1000


# Define the output matrices:
mat_v_1 <- matrix(nrow=runs)
mat_v_min <- matrix(nrow=runs)
mat_v_rand <- matrix(nrow=runs)


# Run a loop of this fot each of the runs:
for (j in 1:runs) {

  # Loop to find the minimum value:
  min_val <- 10
  min_id <- 0

  # resamply flip matrix
  for (i in 1:1000) {
    flip_matrix[i, ] <- sample(sample_coins, 10, replace = TRUE)
  }

  # find min
  for (i in 1:1000) {
    cur_count <- sum(flip_matrix[i, ] == "Heads")
    # make min if min
    if (min_val > cur_count) {
      min_id <- i
      min_val <- cur_count
    }
  }

  # Choose each c:
  c_1 <- flip_matrix[1, ]
  c_min <- flip_matrix[min_id, ]
  c_rand <- flip_matrix[sample(1000, 1), ]


  # Put into result matrices:
  mat_v_1[j] <- mean(c_1 == "Heads")
  mat_v_min[j] <- mean(c_min == "Heads")
  mat_v_rand[j] <- mean(c_rand == "Heads")
}

# Now we should combine the results into a single dataframe:
Q2b_results_df <- data.frame(mat_v_1, mat_v_min, mat_v_rand)
```

Now we can plot the results of this:

```r
# Plot each result:
plot_c1 <- ggplot(Q2b_results_df) +
  coord_cartesian(xlim = c(0,1), ylim = c(0,0.3*runs)) +
  scale_x_continuous(breaks=seq(0,1,0.05)) +
  labs(x="Probability", y="Frequency") +
  geom_histogram(binwidth=0.1, aes(mat_v_1), fill="yellow green",
                          colour="forest green") +
  ggtitle(paste("First Coin from each Run"))

plot_crand <- ggplot(Q2b_results_df) +
  coord_cartesian(xlim = c(0,1), ylim = c(0,0.3*runs)) +
  scale_x_continuous(breaks=seq(0,1,0.05)) +
  labs(x="Probability", y="Frequency") +
  geom_histogram(binwidth=0.1, aes(mat_v_rand), fill="sky blue",
                 colour="blue") +
  ggtitle(paste("Random Coin from each Run"))

plot_cmin <- ggplot(Q2b_results_df) +
  coord_cartesian(xlim = c(0,1), ylim = c(0,0.75*runs)) +
  scale_x_continuous(breaks=seq(0,1,0.05)) +
  labs(x="Probability", y="Frequency") +
  geom_histogram(binwidth=0.1, aes(mat_v_min), fill="salmon",
                 colour="maroon") +
  ggtitle(paste("Minimum Heads Coin in each Run"))



# Combine all the plots into a single plot and display
plot_Q2B<- plot_grid(plot_c1, plot_crand, plot_cmin, nrow=3)

# Add the title:
q2b_title <- ggdraw() + draw_label(paste("Figure 2.1: Simulated Probability Distributions of \nHead Fli

#Combine the two and display
Full_plot_Q2B <- plot_grid(q2b_title, plot_Q2B, ncol=1, rel_heights=c(0.1, 1))
Full_plot_Q2B
```

Figure 2.1: Simulated Probability Distributions of Head Flips for Coins in 1000 Coins



Part C:

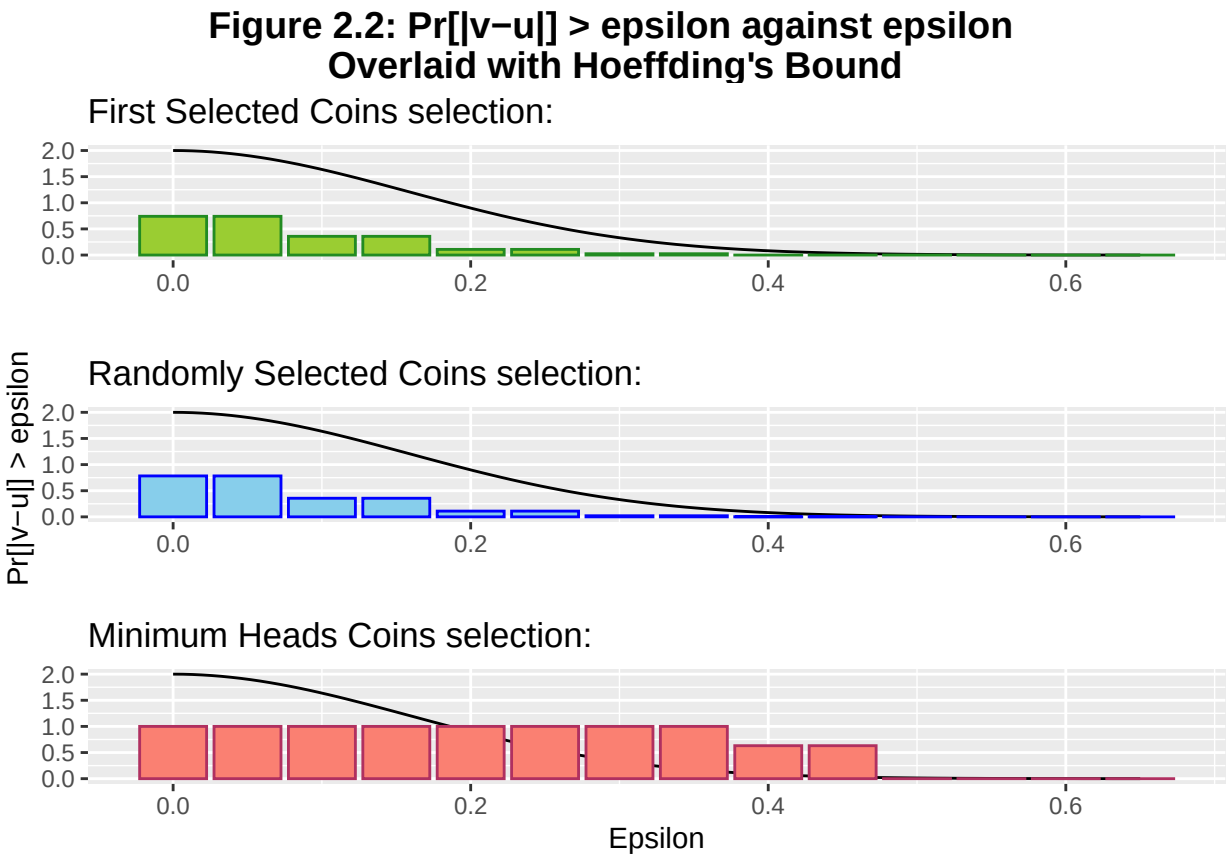
Find the sample proportions for each coin case at each value of epsilon, we can plot them against the Hoeffding Bound. It is worth noting that the inequality given in the assignment sheet is not given as an absolute value, whereas in the notes it is. The notes equation was used in its stead. It should also be noted we should reference the theoretical value of μ as well, as we are trying to determine if these estimators of our iid bins (or, more accurately, iid data generating functions) are valid.

```
# Define the domain of epsilon:
epsilon <- seq(0, 0.65, 0.05)
n <- length(epsilon)
N <- 10

# Add dataset for the Hoffding bound:
hoeffding_bound <- function (epsilon, N = 10){
  bound_val <- 2*exp((-2*epsilon^2)*N)
  return (bound_val)
}

# Set up the values we want to analyse
Pr_c1 <- matrix(nrow = n)
Pr_crand <- matrix(nrow = n)
Pr_cmin <- matrix(nrow = n)

# Find the probability frequencies:
```

```
for (i in 1:length(epsilon)) {
  Pr_c1[i] <- sum(abs(mat_v_1 - mu) > epsilon[i])/runs
  Pr_crand[i] <- sum(abs(mat_v_rand - mu) > epsilon[i])/runs
  Pr_cmin[i] <- sum(abs(mat_v_min - mu) > epsilon[i])/runs
}

# Establish this as a dataframe
Pr_dataframe <- data.frame(epsilon, Pr_c1, Pr_crand, Pr_cmin)
```

And we can now plot it as required:

```
# Plot the results:
plot_2ci <- ggplot(data = Pr_dataframe, aes(x=epsilon)) +
  stat_function(fun = hoeffding_bound) +
  geom_histogram(binwidth=0.1,
                         aes(y=Pr_c1),
                         stat='identity',
                         fill="yellow green",
                         colour="forest green") +
  ggtitle(paste("First Selected Coins selection:")) +
  labs(y = "", x = "")

plot_2cii <- ggplot(data = Pr_dataframe, aes(x=epsilon)) +
  stat_function(fun = hoeffding_bound) +
  geom_histogram(binwidth=0.1,
                 aes(y=Pr_crand),
                 stat='identity',
                 fill="sky blue",
                 colour="blue") +
  ggtitle(paste("Randomly Selected Coins selection:")) +
  labs(y = "Pr[|v-u|] > epsilon ", x = "")

plot_2ciii <- ggplot(data = Pr_dataframe, aes(x=epsilon)) +
  stat_function(fun = hoeffding_bound) +
  geom_histogram(binwidth=0.1,
                 aes(y=Pr_cmin),
                 stat='identity',
                 fill="salmon",
                 colour="maroon") +
  ggtitle(paste("Minimum Heads Coins selection:")) +
  labs(y = "", x = "Epsilon")


# Combine all the plots into a single plot and display
plot_Q2C<- plot_grid(plot_2ci, plot_2cii, plot_2ciii, nrow=3)

# Add the title:
q2c_title <- ggdraw() + draw_label(paste("Figure 2.2: Pr[|v-u|] > epsilon against epsilon \nOverlaid wi

#Combine the two and display
Full_plot_Q2C <- plot_grid(q2c_title, plot_Q2C, ncol=1, rel_heights=c(0.1, 1))
Full_plot_Q2C
```



Part D:

The first coin and the randomly selected coin obey the Hoeffding bound. This is evident as their probabilities at each level of epsilon are less than the bound. Both of these coin selections were made randomly - the random coins selected randomly at each step and the first coins set selected once but as a random sample at each step. This indicates that these coins should be unbiased, as is indicated by there adgerence to the bound.

The minimum proportion of heads coin, however, does not obey the bound. It may be observed that by $\epsilon \geq 0.2$ the probability exceeds the Hoeffding bound. This leads us to the conclusion that v_{min} does not 'say anything about' $\mu_{theoretical}$ and we must conclude that it is a biased estimator of $\mu_{theoretical}$ if it is used to estimate $\mu_{theoretical}$.

Part E:

We can relate Part D to the bins in the Figure 1 of the assignment sheet through our understanding of the sets of coins we selected throughout this question. For c_1, we looked at multiple samples from a single bin and as such we could conclude that the sample value v formed a good approximation for $\mu_{theoretical}$, the out of sample value associated with the bin from which it was drawn. The same can be said for c_{rand}, as it is drawn from iid random bins each with the same $\mu_{theoretical}$.

By contrast, we observe that c_{min} does not perform well as an estimator for $\mu_{theoretical}$. This is not surprising as c_{min} is drawn from the minimum samples at each trial of the experiment, which essentially acts to increase the E_{out} at each trial of this experiment. It can be said therefore that c_{min} is not drawn randomly from the bins depicted in Figure 1 of the assignment sheet. One could, however, consider the entire experiment at each trial to be a single bin of minimum head proportions from which to draw a sample, and using the rules of statistics we could find a $\mu_{augmented}$ that represents this bin. In this case, c_{min} would produce a valid estimator v_{min} for this bin.

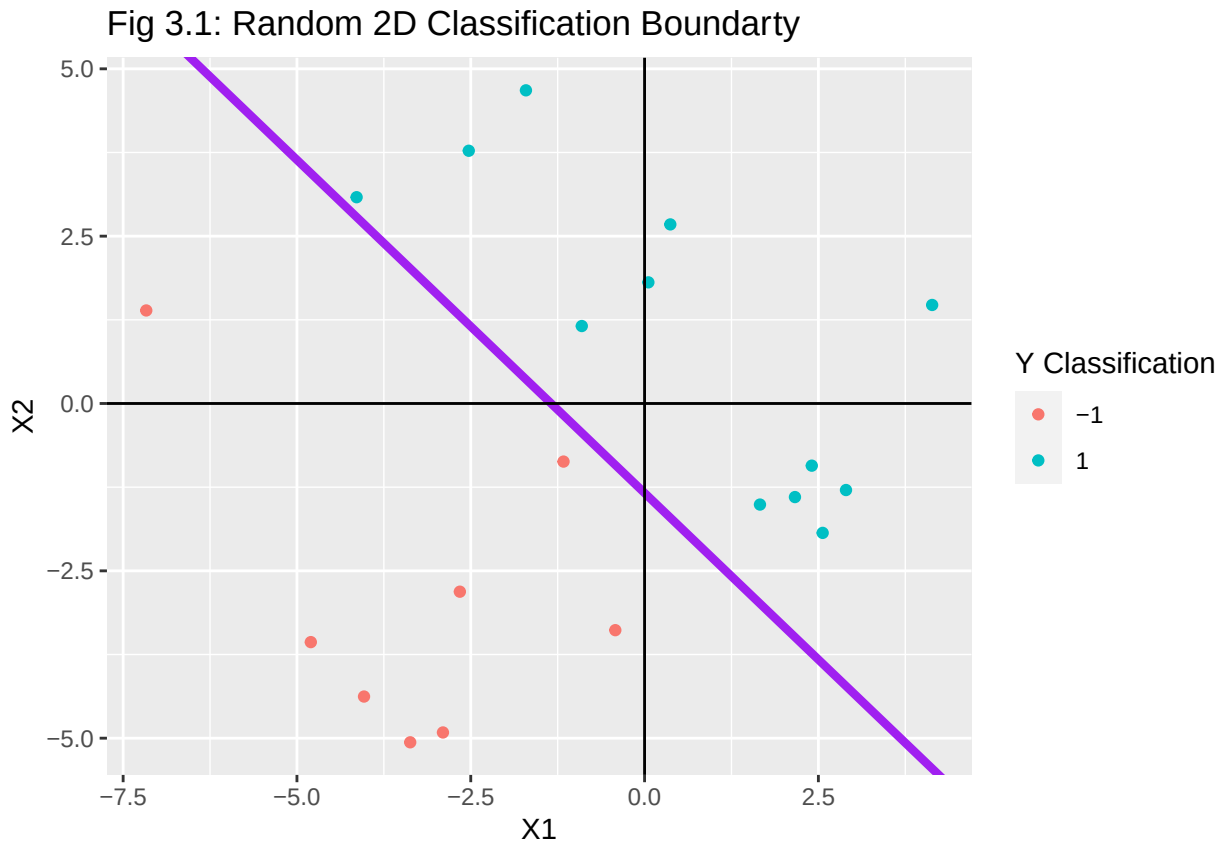
Question 3:

Perceptron Learning Algorithm Analysis in 2 and 10 Dimensions

Part A:

Generating a linearly separable dataset, just as in Question 1:

```
# Choose f, being a random line in two dimensions. As:
# Of the form: c1*X1 + c2*X2 + c3 = 0
# or rather: X2 = (-c1/c2)*X2 + (-c3/c2)

c1 <- runif(1, -3, 3)
c2 <- runif(1, -3, 3)
c3 <- runif(1, -3, 3)


# Create the random dataset
X1 <- rnorm(20, mean=0, sd=3)
X2 <- rnorm(20, mean=0, sd=3)

# Check where to divide for X1:
dividing_line <- -(c1/c2)*X1 -(c3/c2)

# Use sign for Y if above this:
Y <- sign(X2 - dividing_line)


# Set the dataset:
Data_set <- tibble(X1, X2, 1, Y)


# Plot it:
Initial_plot <- ggplot(Data_set, aes(x=X1, y=X2)) +
  geom_point(aes(colour = factor(Y))) + labs(color = "Y Classification") +
  geom_abline(intercept = -(c3/c2), slope = -(c1/c2), color="purple", linetype="solid", size=1.5) +
  ggtitle("Fig 3.1: Random 2D Classification Boundarty") +
  geom_vline(xintercept = 0, color="black") +
  geom_hline(yintercept = 0, color="black")

Initial_plot
```



Part B:

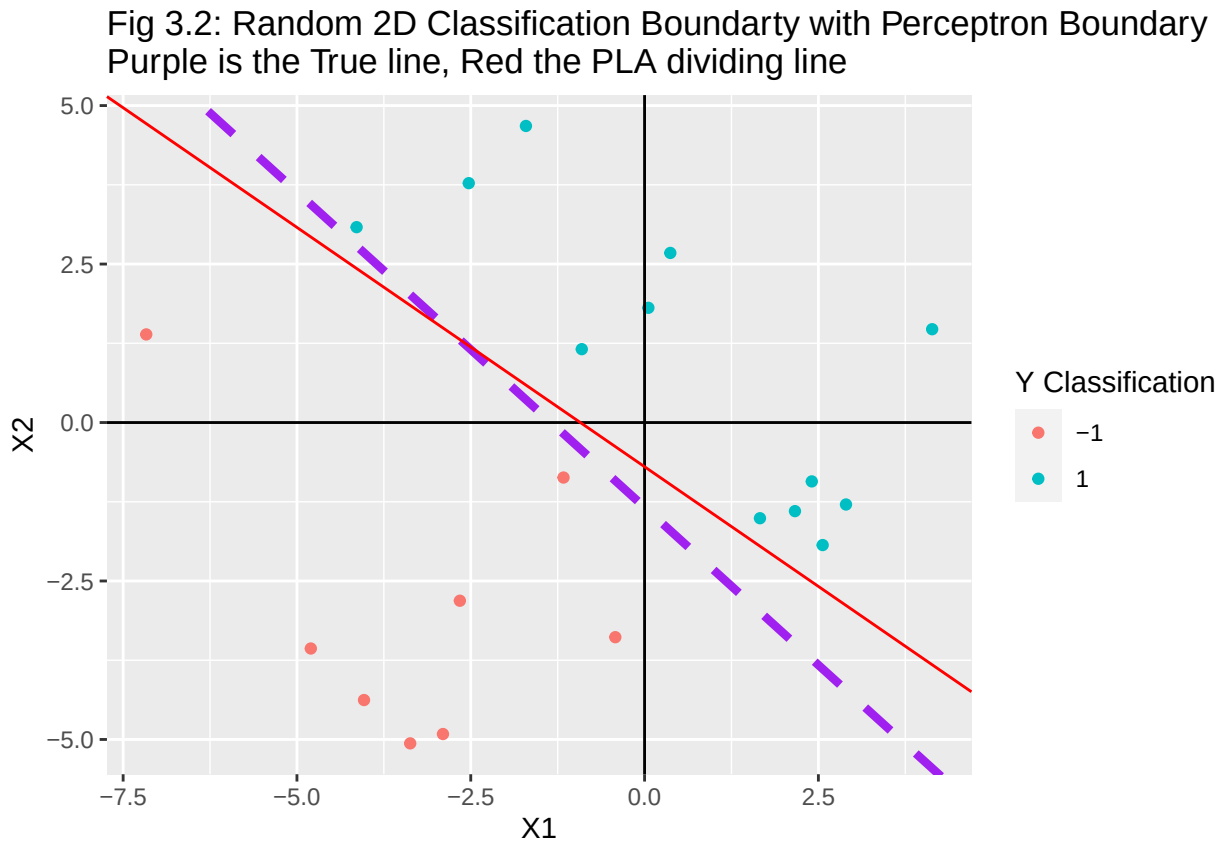
Run the perceptron algorithm algorithm:

```
#### Now we want to implement perceptron:


# Set initial weights
weights = c(1, 2, 3)


# Use the sign to assign initial outcomes:
h_val <- sign( as.matrix(Data_set %>% select(1:3)) %*% weights )

iterations <- 0
# Create a loop to repeatedly update the estimates:
while (sum(h_val == (Data_set %>% pull(Y))) < 20){
  for (i in (1:20)){
    # Primary if clause
    if (h_val[i] != Y[i]){
      weights <- weights + Y[i] *  as.matrix(Data_set %>% select(1:3))[i, ]
      # iterations += 1
      iterations <- iterations + 1
    }
  }
```

```
  h_val <- sign( as.matrix(Data_set %>% select(1:3)) %*% weights )
}
```

Now plot the results:

```
Later_Plot <- ggplot(Data_set, aes(x=X1, y=X2)) +
  geom_point(aes(colour = factor(Y))) + labs(color = "Y Classification") +
  geom_abline(intercept = -(c3/c2), slope = -(c1/c2), color="purple", linetype="dashed", size=1.5) +
  ggtitle("Fig 3.2: Random 2D Classification Boundarty with Perceptron Boundary \nPurple is the True li
  geom_vline(xintercept = 0, color="black") +
  geom_hline(yintercept = 0, color="black") +
  geom_abline(intercept = (-weights[3]/weights[2]), slope = (-weights[1]/weights[2]), color="red")

Later_Plot
```




```
# add legend for the line colours!
```

Now report the number of updates:

```
# Print the number of iterations:
print(paste("For this dataset, for convergence to occue with with PLA, we required", iterations, "updat
```

```
## [1] "For this dataset, for convergence to occue with with PLA, we required 1 updates."
```

Our hypothesis appears to be mostly correct. It identifies each point correctly as is required for algorithm termination. However, the line for our hypothesis g clearly is not the same as out target function f. This is likely as a result of the small sample size, as we do not have the data to make the hypothesis more accurate.

Part C:

It seems plagmatic to make this into a function to save time later on:

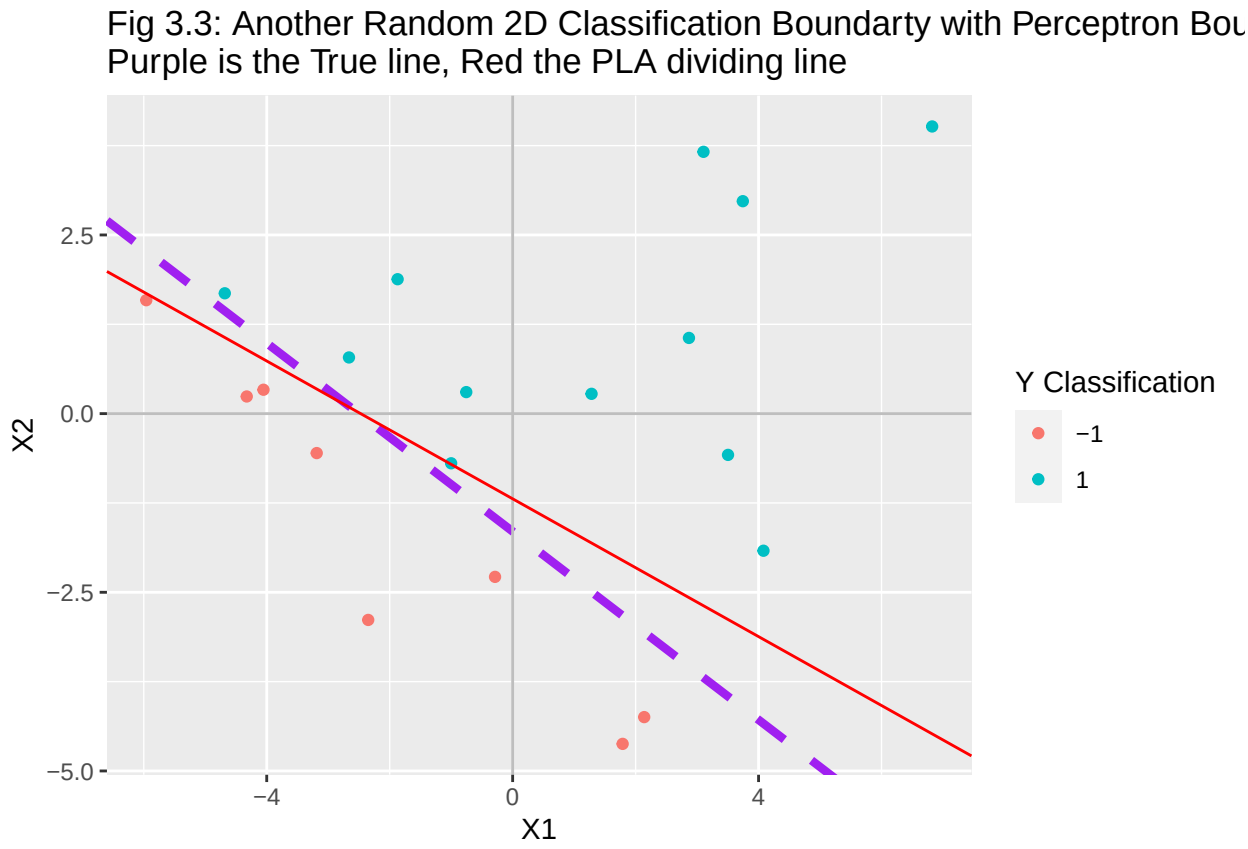
```
# Because of this repetition I'm going to make my sampling a function:
two_d_PLA_function <- function(sample_size){
  c1 <- runif(1, -3, 3)
  c2 <- runif(1, -3, 3)
  c3 <- runif(1, -3, 3)

  # record the real line values:
  real_weights <- c(c1, c2, c3)

  # Create the random dataset
  X1 <- rnorm(sample_size, mean=0, sd=3)
  X2 <- rnorm(sample_size, mean=0, sd=3)

  # Check where to divide for X1:
  dividing_line <- -(c1/c2)*X1 -(c3/c2)

  # Use sign for Y if above this:
  Y <- sign(X2 - dividing_line)


  # Set the dataset:
  Data_set <- tibble(X1, X2, 1, Y)

  # Set initial weights
  weights = c(1, 2, 3)


  # Use the sign to assign initial outcomes:
  h_val <- sign( as.matrix(Data_set %>% select(1:3)) %*% weights )

  # record updates done:
  iterations <- 0

  # Create a loop to repeatedly update the estimates:
  while (sum(h_val == (Data_set %>% pull(Y))) < sample_size){
    for (i in (1:sample_size)){
      # Primary if clause
      if (h_val[i] != Y[i]){
        weights <- weights + Y[i] *  as.matrix(Data_set %>% select(1:3))[i, ]
        # iterations += 1
        iterations <- iterations + 1
      }
    }
    h_val <- sign( as.matrix(Data_set %>% select(1:3)) %*% weights )
  }

  return(list(iterations, weights, Data_set, real_weights, Y))
}
```

Now to repeat the experiment:

```
sample_size <- 20

sample_3c <- two_d_PLA_function(sample_size)
```

Plot the result:

```
q3c_plot <- ggplot(sample_3c[[3]], aes(x=X1, y=X2)) +
  geom_point(aes(colour = factor(sample_3c[[5]]))) + labs(color = "Y Classification") +
  geom_abline(intercept = -(sample_3c[[4]][[3]]/sample_3c[[4]][[2]]), slope = -(sample_3c[[4]][[1]]/samp
  ggtitle("Fig 3.3: Another Random 2D Classification Boundarty with Perceptron Boundary, n=20 \nPurple
  geom_vline(xintercept = 0, color="grey") +
  geom_hline(yintercept = 0, color="grey") +
  geom_abline(intercept = (-sample_3c[[2]][3]/sample_3c[[2]][2]), slope = (-sample_3c[[2]][1]/sample_3c

q3c_plot
```



And confirm the number of iterations:

```
print(paste("For this dataset, for convergence to occue with with PLA, we required", sample_3c[1], "upda
```

```
## [1] "For this dataset, for convergence to occue with with PLA, we required 52 updates."
```

We should note that our results are roughly as accurate as we obsered in Part B. Indeed, the number of updates was identical. It might be said the hypothesis is less close than in Part B, but not by much.

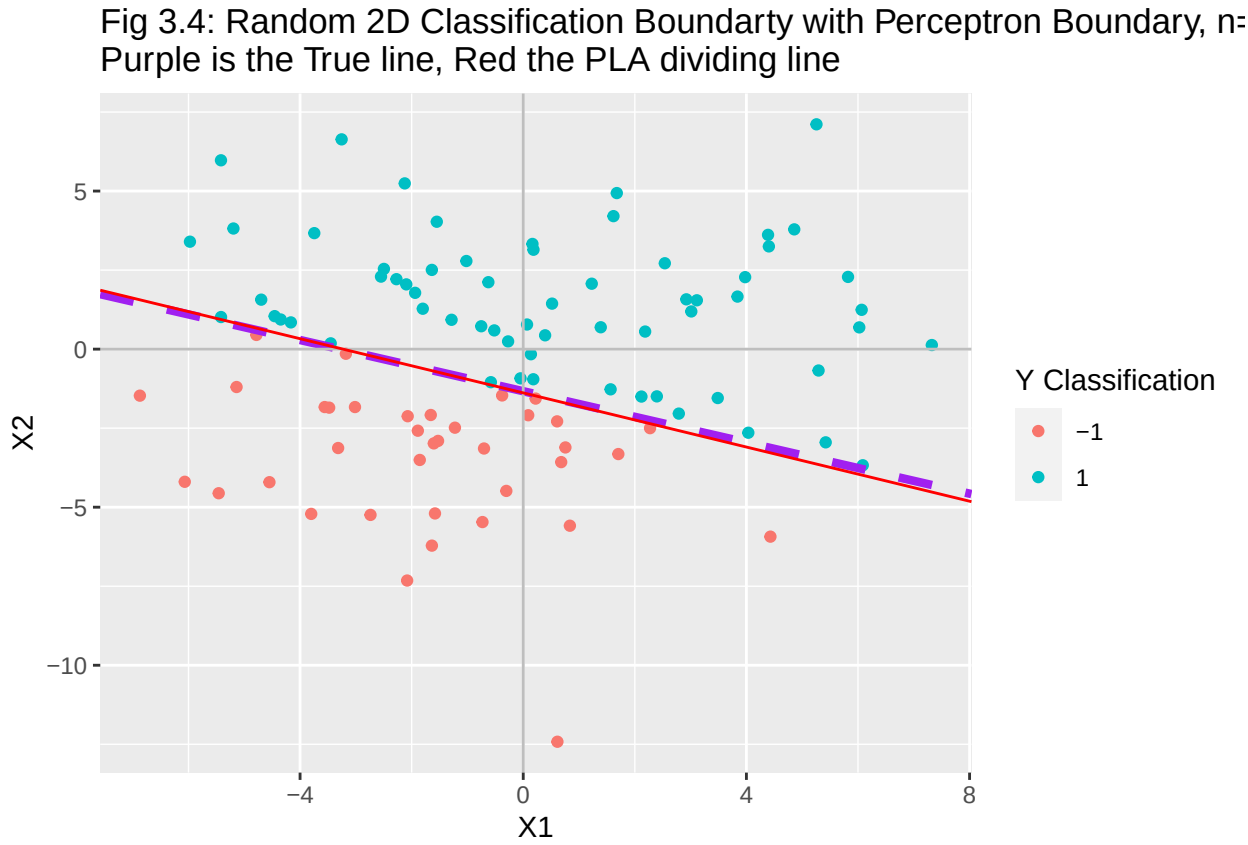
Part D:

Now with a larger sample size, we can generate the dataset and run the PLA:

```
sample_size <- 100

sample_3d <- two_d_PLA_function(sample_size)
```

And plot the result:

```
q3d_plot <- ggplot(sample_3d[[3]], aes(x=X1, y=X2)) +
  geom_point(aes(colour = factor(sample_3d[[5]]))) + labs(color = "Y Classification") +
  geom_abline(intercept = -(sample_3d[[4]][[3]]/sample_3d[[4]][[2]]), slope = -(sample_3d[[4]][[1]]/samp
  ggtitle("Fig 3.4: Random 2D Classification Boundarty with Perceptron Boundary, n=100 \nPurple is the
  geom_vline(xintercept = 0, color="grey") +
  geom_hline(yintercept = 0, color="grey") +
  geom_abline(intercept = (-sample_3d[[2]][3]/sample_3d[[2]][2]), slope = (-sample_3d[[2]][1]/sample_3d

q3d_plot
```



And print the number of updates:

```
print(paste("For this dataset, for convergence to occue with with PLA, we required", sample_3d[1], "upda
```

```
## [1] "For this dataset, for convergence to occue with with PLA, we required 215 updates."
```

Like before, all points are identified correctly, though the number of updates for convergence has gone up by a significant amount. There has been an improvement in accuracy of g relative to f as well.

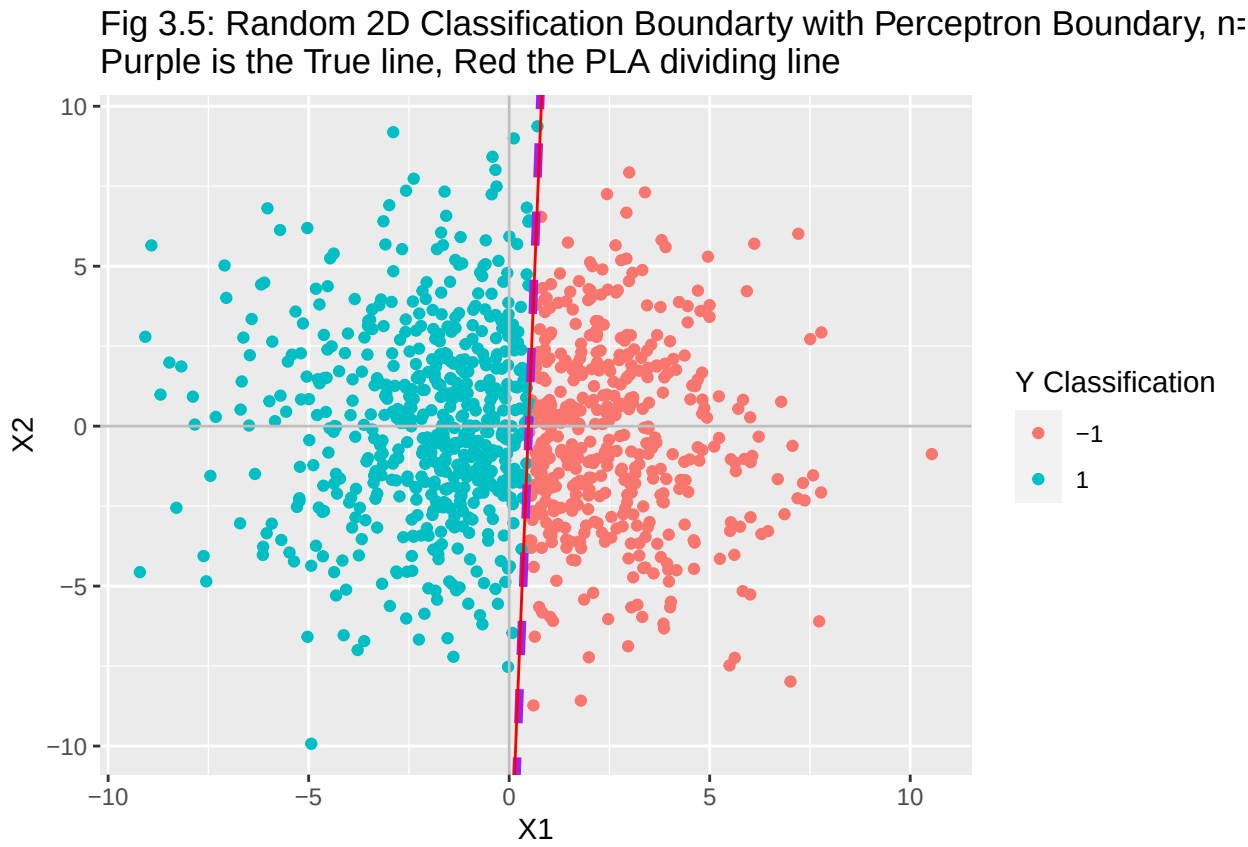
Part E:

Now with an even larger sample size, we can generate the dataset and run the PLA:

```
sample_size <- 1000

sample_3e <- two_d_PLA_function(sample_size)
```

Then we can plot the result:

```
q3e_plot <- ggplot(sample_3e[[3]], aes(x=X1, y=X2)) +
  geom_point(aes(colour = factor(sample_3e[[5]]))) + labs(color = "Y Classification") +
  geom_abline(intercept = -(sample_3e[[4]][[3]]/sample_3e[[4]][[2]]), slope = -(sample_3e[[4]][[1]]/samp
  ggtitle("Fig 3.5: Random 2D Classification Boundarty with Perceptron Boundary, n=1000 \nPurple is the
  geom_vline(xintercept = 0, color="grey") +
  geom_hline(yintercept = 0, color="grey") +
  geom_abline(intercept = (-sample_3e[[2]][3]/sample_3e[[2]][2]), slope = (-sample_3e[[2]][1]/sample_3e

q3e_plot
```



And print the number of updates:

```
print(paste("For this dataset, for convergence to occur with with PLA, we required", sample_3e[1], "upda
```

```
## [1] "For this dataset, for convergence to occur with with PLA, we required 1512 updates."
```

In this final dataset PLA analysis, we note that the number of updates has increased by a lot. This is to be expected as the time to convergence is at least partially correlated with the sample size. At the same time, our hypothesis is now a lot more accurate than in Part B, as is seen by how much f and g overlap.

Part F:

Given what Part G requires, I think I will make a function for this again:

```
# Seems best to make a function for this:
ten_d_PLA_function <- function(sample_size){

  # Choose f, being a random line in two dimensions. As:
  # Of the form: c1*X1 + c2*X2 + c3*X3 + ... + c10*X10 + c11 = 0
  # or rather: X1 = (-c2/c1)*X2 + (-c3/c1)*X3 + (-c4/c1)*X4 + ... + (-c3/c2)

  c1 <- runif(1, -3, 3)
  c2 <- runif(1, -3, 3)
  c3 <- runif(1, -3, 3)
  c4 <- runif(1, -3, 3)
  c5 <- runif(1, -3, 3)
  c6 <- runif(1, -3, 3)
  c7 <- runif(1, -3, 3)
  c8 <- runif(1, -3, 3)
  c9 <- runif(1, -3, 3)
  c10 <- runif(1, -3, 3)
  c11 <- runif(1, -3, 3)

  # record the real line values:
  real_weights <- c(c1, c2, c3, c4, c5, c6, c7, c8, c9, c10, c11)

  # Create the random dataset
  X1 <- rnorm(sample_size, mean=0, sd=3)
  X2 <- rnorm(sample_size, mean=0, sd=3)
  X3 <- rnorm(sample_size, mean=0, sd=3)
  X4 <- rnorm(sample_size, mean=0, sd=3)
  X5 <- rnorm(sample_size, mean=0, sd=3)
  X6 <- rnorm(sample_size, mean=0, sd=3)
  X7 <- rnorm(sample_size, mean=0, sd=3)
  X8 <- rnorm(sample_size, mean=0, sd=3)
  X9 <- rnorm(sample_size, mean=0, sd=3)
  X10 <- rnorm(sample_size, mean=0, sd=3)

  # Check where to divide for X1:
  dividing_line <- -(c2/c1)*X2 -(c3/c1)*X3 -(c4/c1)*X4 -(c5/c1)*X5 -(c6/c1)*X6 -(c7/c1)*X7 -(c8/c1)*X8 -

  # Use sign for Y if above this:
  Y <- sign(X1 - dividing_line)


  # Set the dataset:
  Data_set <- tibble(X1, X2, X3, X4, X5, X6, X7, X8, X9, X10, 1, Y)

  # Set initial weights
  weights = c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11)


  # Use the sign to assign initial outcomes:
  h_val <- sign( as.matrix(Data_set %>% select(1:11)) %*% weights )
```

```
  # record updates done:
  iterations <- 0

  # Create a loop to repeatedly update the estimates:
  while (sum(h_val == (Data_set %>% pull(Y))) < sample_size){
    for (i in (1:sample_size)){
      # Primary if clause
      if (h_val[i] != Y[i]){
        weights <- weights + Y[i] *  as.matrix(Data_set %>% select(1:11))[i, ]
        # iterations += 1
        iterations <- iterations + 1
      }
    }
    h_val <- sign( as.matrix(Data_set %>% select(1:11)) %*% weights )
  }

  return(list(iterations, weights, Data_set, real_weights, Y, h_val))
}
```

Now to test this out:

```
q3f_sample <- ten_d_PLA_function(1000)

print(paste("For this dataset, for convergence to occur with with PLA, we required", q3f_sample[1], "up
```

```
## [1] "For this dataset, for convergence to occur with with PLA, we required 3720 updates."
```

```
#print(q3f_sample[[1]])
#print(q3f_sample[[2]])
# out_y <- q3f_sample[[6]]
# test_y <- q3f_sample[[5]]
#full_y <- list(out_y, test_y)
# identical(out_y, test_y)
# Sample and Hypothesis Ys Match
```

Part G:

Best to update the function from Part F slightly:

```
# Simplify the function output to save on space complexity:
# Seems best to make a function for this:
ten_d_PLA_function_simple <- function(sample_size){

  # Choose f, being a random line in two dimensions. As:
  # Of the form: c1*X1 + c2*X2 + c3*X3 + ... + c10*X10 + c11 = 0
  # or rather: X1 = (-c2/c1)*X2 + (-c3/c1)*X3 + (-c4/c1)*X4 + ... + (-c3/c2)

  c1 <- runif(1, -3, 3)
  c2 <- runif(1, -3, 3)
  c3 <- runif(1, -3, 3)
  c4 <- runif(1, -3, 3)
  c5 <- runif(1, -3, 3)
  c6 <- runif(1, -3, 3)
  c7 <- runif(1, -3, 3)
```

```r
c8 <- runif(1, -3, 3)
c9 <- runif(1, -3, 3)
c10 <- runif(1, -3, 3)
c11 <- runif(1, -3, 3)

# record the real line values:
real_weights <- c(c1, c2, c3, c4, c5, c6, c7, c8, c9, c10, c11)

# Create the random dataset
X1 <- rnorm(sample_size, mean=0, sd=3)
X2 <- rnorm(sample_size, mean=0, sd=3)
X3 <- rnorm(sample_size, mean=0, sd=3)
X4 <- rnorm(sample_size, mean=0, sd=3)
X5 <- rnorm(sample_size, mean=0, sd=3)
X6 <- rnorm(sample_size, mean=0, sd=3)
X7 <- rnorm(sample_size, mean=0, sd=3)
X8 <- rnorm(sample_size, mean=0, sd=3)
X9 <- rnorm(sample_size, mean=0, sd=3)
X10 <- rnorm(sample_size, mean=0, sd=3)

# Check where to divide for X1:
dividing_line <- -(c2/c1)*X2 -(c3/c1)*X3 -(c4/c1)*X4 -(c5/c1)*X5 -(c6/c1)*X6 -(c7/c1)*X7 -(c8/c1)*X8 -

# Use sign for Y if above this:
Y <- sign(X1 - dividing_line)


# Set the dataset:
Data_set <- tibble(X1, X2, X3, X4, X5, X6, X7, X8, X9, X10, 1, Y)

# Set initial weights
weights = c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11)


# Use the sign to assign initial outcomes:
h_val <- sign( as.matrix(Data_set %>% select(1:11)) %*% weights )

# record updates done:
iterations <- 0

# Create a loop to repeatedly update the estimates:
while (sum(h_val == (Data_set %>% pull(Y))) < sample_size){
  for (i in (1:sample_size)){
    # Primary if clause
    if (h_val[i] != Y[i]){
      weights <- weights + Y[i] *  as.matrix(Data_set %>% select(1:11))[i, ]
      # iterations += 1
      iterations <- iterations + 1
    }
  }
  h_val <- sign( as.matrix(Data_set %>% select(1:11)) %*% weights )
}
```

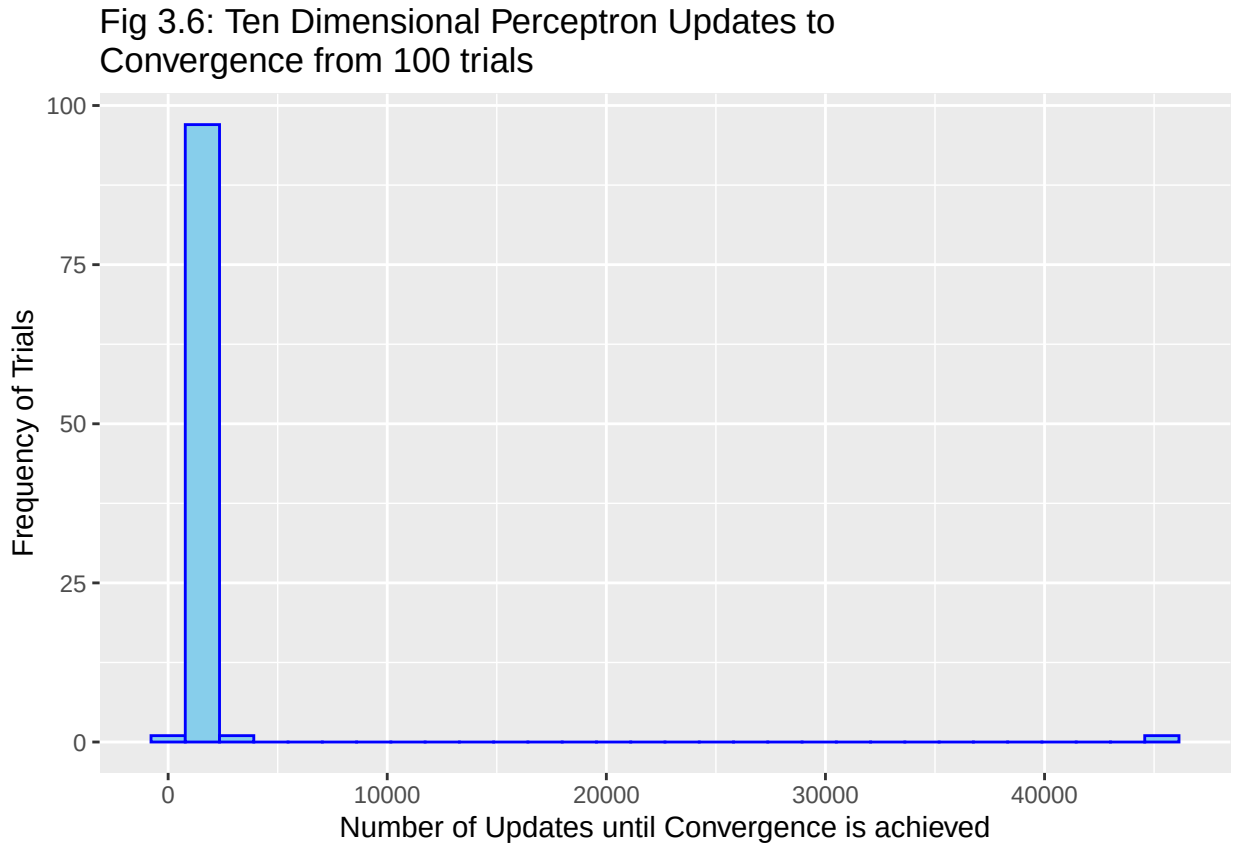
```
  return(iterations)
}
```

Now to run it many times as required:

```
# Main loop:

trials <- 100
sample_size <- 1000

updates <- matrix(nrow=trials)

for (i in 1:trials) {
  sample_output <- ten_d_PLA_function_simple(sample_size)

  updates[i] <- sample_output
}
```

And plot our output:

```
# Plot Histogram:
Histogram_plot <- ggplot() +
  aes(updates) +
  geom_histogram(fill="sky blue",
                 colour="blue") +
  labs(title=paste("Fig 3.6: Ten Dimensional Perceptron Updates to \nConvergence from", trials, "trials
       x = "Number of Updates until Convergence is achieved",
       y = "Frequency of Trials"
       )
Histogram_plot
```



Part H:

Overall, from the results of this exercise, we find our experimental results reflect what we would expect from theory.

As we increase the sample size N, we find that while the number of iterations increases exponentially and we also find that the accuracy of our hypothesis g becomes significantly closer to f as we gain information in the form of additional datapoints. Hence, we can say $accuracy \propto N$ and that - at a guess - we can say $running\_time \propto \exp\{N\}$

From parts F and G we found that as we increase the number of dimensions, we do not necessarily increase the running times by a massive amount - moving from ~ 1500 updates in Part E to ~ 1000 updates in Part F (in my RMD sample). We did find in Part G that some samples could vary wildly from most other samples, but this could conceivably happen in two dimensions. Also we did not gather a large sample of repeated size 1000 trials in two dimensions to appropriately compare. In higher dimensions, we are still only trying to get the same number of sample points to correctly identify so aside from any overhead to time complexity that may come from performing calculations in higher dimensional matrices, the algorithm is not observed to be consistently slower with higher dimensional data. We did not observe a major change in accuracy given the algorithm converges successfully and we did not measure how closely g reflects f analytically or graphically for higher dimensional data.