HW13

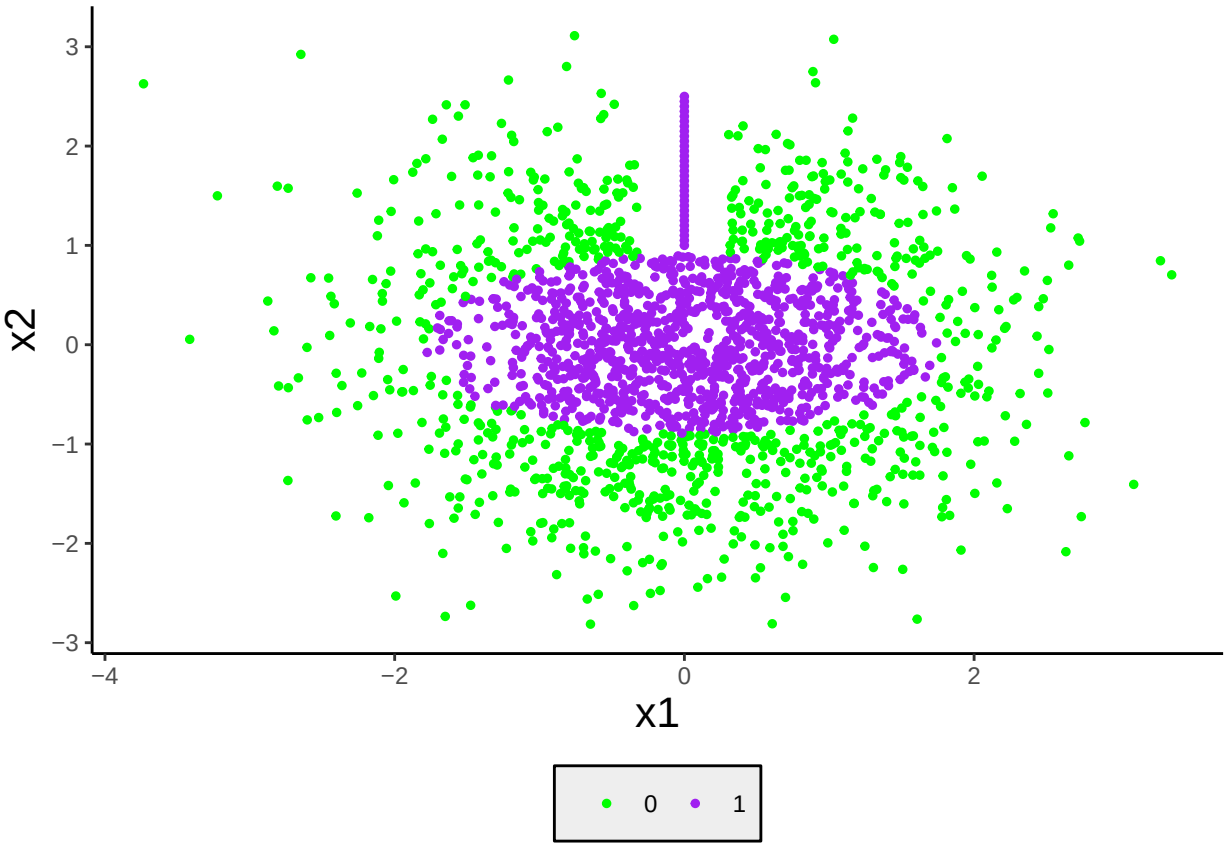
Ines Pancorbo

4/9/2020

```
# reading and prepping data
data <- read.csv("nn.txt", header = TRUE, sep = ' ', stringsAsFactors = TRUE)
x1 <- data$x1
x2 <- data$x2
y <- data$y
X <- cbind(1, x1, x2)
```

a)

```
library(ggplot2)

# Plotting dataset, two different colors for the two different classes of y
ggplot(data, aes(x = x1, y = x2, color = as.factor(y))) +
  geom_point(size=1) +
  scale_color_manual(values = c("green", "purple")) +
  theme_classic() +
  theme(legend.position = "bottom",
        legend.background = element_rect(fill = "#EEEEEE", color = "black"),
        legend.title = element_blank(),
        axis.title = element_text(size = 16))
```



From the plot, it looks like for the most part points with coordinates $(x1, x2)$ in the set $[-2, 2]$ X $[-1, 1]$ are labeled as 1, and points with coordinates $(x1, x2)$ not in the set $[-2, 2]$ X $[-1, 1]$ are labeled as 0. We would want to see a similar pattern when we visualize the classifier in 1 g).

b)

What is the dimension of α in terms of m? We have 1947 datapoints. Each datapoint $x^{(i)} = (x_1^{(i)}, x_2^{(i)}) \in R^2$ and we have a hidden layer of size m. The hidden layer is of dimension $m(2+1) = 3m$ and the output layer is of dimension $2(m+1) = 2m+2$. And so the dimension of α in terms of m is $3m + 2m + 2 = 5m + 2$.

c)

```
################################ helper functions ######################################
#sigmoid function
sigmoid <- function(x) 1 / (1 + exp(-x))

# get_W1 (weight matrix for hidden layer)
get_W1 <- function(alpha, m) matrix(alpha[1:(3*m)], nrow = 3, ncol = m)

# get_W2 (weight matrix for output layer)
get_W2 <- function(alpha, m) matrix(alpha[(3*m+1):length(alpha)], nrow = (m+1), ncol = 2)
########################################################################################
```

```
NN <- function(X, alpha, m){
  W1 <- get_W1(alpha, m)
```

```
  W2 <- get_W2(alpha, m)

  linear_layer <- X %*% W1

  Z <- sigmoid(linear_layer)
  Z <- cbind(1,Z)
  output <- sigmoid(Z %*% W2)

  exp_output <- exp(output)
  prob <- exp_output[,1]/rowSums(exp_output)
  return(prob)
}
```

d)

From previous HWs we have shown that the log likelihood function is $logL(\alpha) = \sum_{i=1}^{N}[y_i log(P(y = 1|x^{(i)}, \alpha)) + (1 - y_i)log(1 - P(y = 1|x^{(i)}, \alpha))]$ for a sample of size N. This follows because you can think of the collection of datapoints (N datapoints, Y_i) as a random sample of bernoulli random variables (each datapoint is 1 with probability $P(y = 1|x^{(i)}, \alpha)$ and 0 with probability $1 - P(y = 1|x^{(i)}, \alpha)$). So the likelihood function would be $L(\alpha) = \prod_{i=1}^{N} P(y = 1|x^{(i)}, \alpha)^{y_i}(1 - P(y = 1|x^{(i)}, \alpha))^{(1-y_i)}$. Taking logs, the log likelihood would then be $logL(\alpha) = \sum_{i=1}^{N}[y_i log(P(y = 1|x^{(i)}, \alpha)) + (1 - y_i)log(1 - P(y = 1|x^{(i)}, \alpha))]$.

In a previous HW we showed that for logistic regression with datapoints in R^n, given that $P(y = 1|x^{(i)}, \alpha) = \frac{1}{1+e^{-(\alpha_0+\alpha_1 x_1^{(i)}+\ldots+\alpha_n x_n^{(i)})}}$, the log likelihood function is $logL(\alpha) = \sum_{i=1}^{N}[y_i log(P(y = 1|x^{(i)}, \alpha)) + (1-y_i)log(1-P(y = 1|x^{(i)}, \alpha))] = \sum_{i=1}^{N}[y_i log(\frac{1}{1+e^{-(\alpha_0+\alpha_1 x_1^{(i)}+\ldots+\alpha_n x_n^{(i)})}}) + (1 - y_i)log(1 - \frac{1}{1+e^{-(\alpha_0+\alpha_1 x_1^{(i)}+\ldots+\alpha_n x_n^{(i)})}})]$. We have just plugged in for the corresponding $P(y = 1|x^{(i)}, \alpha)$.

Similarly, plugging in for the correct $P(y = 1|x^{(i)}, \alpha)$ in the neural net case: In the neural net case, we have that $P(y = 1|x^{(i)}, \alpha) = \frac{e^{T_1}}{e^{T_1}+e^{T_2}}$, where T_1, T_2 are the outputs of the neural net given an input $x^{(i)}$. Therefore, $logL(\alpha) = \sum_{i=1}^{N}[y_i log(P(y = 1|x^{(i)}, \alpha)) + (1 - y_i)log(1 - P(y = 1|x^{(i)}, \alpha))] = \sum_{i=1}^{N}[y_i log(\frac{e^{T_1}}{e^{T_1}+e^{T_2}}) + (1 - y_i)log(1 - \frac{e^{T_1}}{e^{T_1}+e^{T_2}})]$.

```
# log likelihood function, the datapoints X, and the labels y
# are declared as global variables so no need to pass to function
# The function will have access to them

logL <- function(alpha, r = NULL){
  W1 <- get_W1(alpha, m)
  W2 <- get_W2(alpha, m)

  # for general evaluation when needed
  # especially plotting
  if (is.null(r)){
    prob <- NN(X, alpha, m)
    return(sum(y*log(prob)+(1-y)*log(1-prob)))
  }

  # for stochastic gradient use
  # r is a random sample
  else {
    linear_layer <- X[r,] %*% W1
    Z <- sigmoid(linear_layer)
```

```
    Z <- cbind(1,Z)
    output <- sigmoid(Z %*% W2)

    exp_output <- exp(output)
    prob <- exp_output[,1]/rowSums(exp_output)
    return(sum(y[r]*log(prob)+(1-y[r])*log(1-prob)))
  }
}
```

e)

```
# stochastic gradient of logL using finite differences

sg_logL <- function(alpha){
  # picking optimal step size, shown in class to be 10^-8
  h <- 1e-8
  # random batch for stochastic gradient
  # we have 1947 datapoints, let's choose 5
  # (just for the sake of not using 1)
  # sample with replacement
  r <- sample(1:nrow(X), 5, replace=T)
  l <- logL(alpha, r)

  # alphas needed for each approx of first partial
  alpha_calc <- matrix(rep(alpha, length(alpha)), ncol = length(alpha), byrow = TRUE) +
    diag(h, nrow = length(alpha), ncol = length(alpha))

  # grad vector
  grad <- (apply(alpha_calc, MARGIN = 1, FUN = logL, r = r) - l)/h
  return(grad)
}
```

f)

Initialize the parameters (the α vector) to random numbers between -1 and 1, and set $m = 4$.

```
set.seed(7)
m <- 4
num_of_params <- 5*m+2
alpha <- runif(num_of_params, -1, 1)
```

Train function (the stochastic steepest gradient ascent function) used to find optimal α.

```
# stochastic gradient ascent
SGA <- function(alpha, eps = 1e-10){

  g <- sg_logL(alpha)
  l <- logL(alpha)
  s <- 0.1
  iter <- 0
  x_values <- c(iter)
  y_values <- c(l)
```

```
  repeat{
    alpha <- alpha + s*g
    next_l <- logL(alpha)

    if (abs(l-next_l) < eps){
      break
    }

    l <- next_l
    g <- sg_logL(alpha)
    iter <- iter + 1

    if(iter %% 100 == 0){
      x_values <- append(x_values, iter)
      y_values <- append(y_values, l)
    }
  }
  return(list(max=alpha, iter=iter, x_values=x_values, y_values=y_values))
}
```

Lets find the optimal alpha and lets plot Iterations vs. logL Value (to double-check convergence of stochastic steepest gradient ascent).

```
start_time <- Sys.time()
SGA <- SGA(alpha)
end_time <- Sys.time()
end_time - start_time
```

```
## Time difference of 2.353022 mins
```

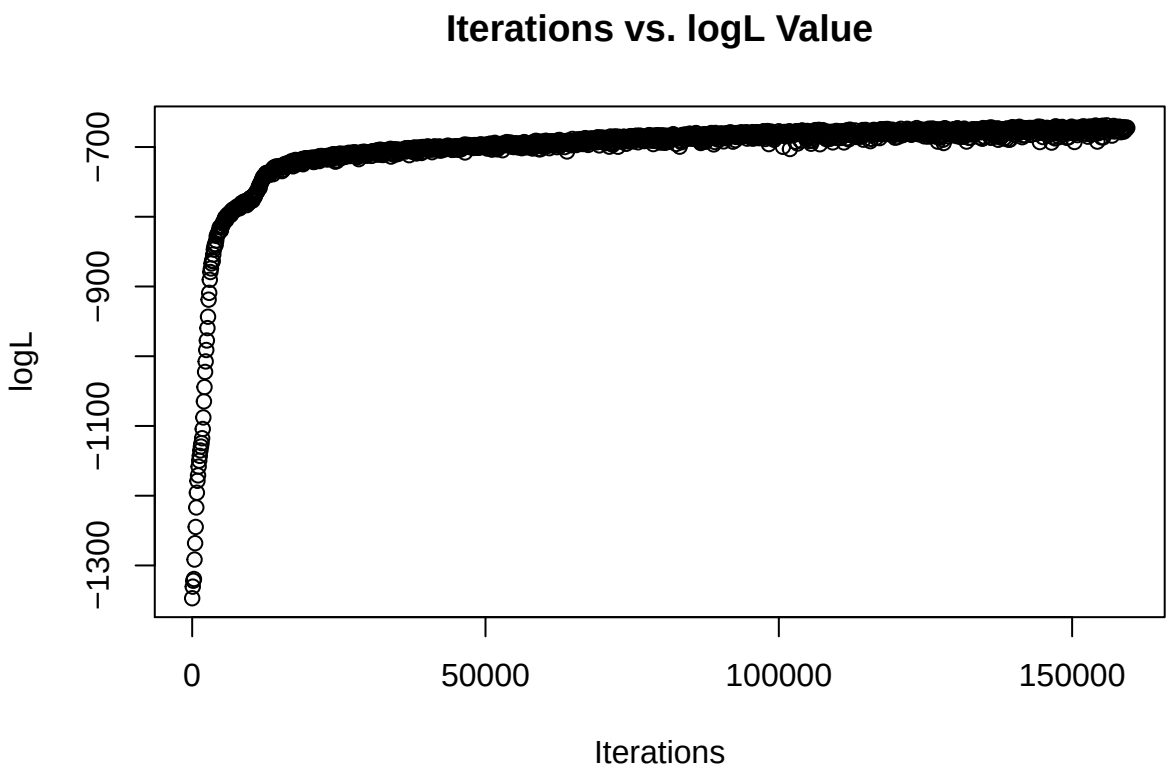
```
# print number of iterations
SGA$iter
```

```
## [1] 159400
```

```
alpha_star <- SGA$max
# print alpha_star
alpha_star
```

```
##  [1]  -3.66665784  -3.39701031   0.21513735 -10.75714326   0.08876202
##  [6]  14.96877898  -6.96108720  -0.07625682  -9.44180540   4.02101790
## [11]  -3.41424536  -0.41170976  -3.06942505 -22.26881684 -18.29989164
## [16] -25.27276153  22.82632406   3.06829809  22.26049965  18.29336082
## [21]  25.26352577 -22.81768635
```

```
plot(SGA$x_values, SGA$y_values,
     xlab = "Iterations", ylab = "logL",
     title(main = "Iterations vs. logL Value"))
```



g)

```
# defining F function
F_func <- function(x, p){
  y_pred <- ifelse(x >= p, 1, 0)
  return(y_pred)
}
```

Now lets visualize the classifier

```
# generating random coordinate points to test the classifier
x1_test <- 4*runif(10000) - 2
x2_test <- 4*runif(10000) - 2
X_test <- cbind(1, x1_test, x2_test)
```

```
# lets feed the generated points from above to the neural net
probs_pred <- NN(X_test, alpha_star, m)

# the neural net will give us probabilities, so numbers between 0 and 1
# lets print the first couple outputs from the neural net, and
# see what the probabilities are. This will help in determining an
# appropriate probability cut-off p
# I will print 100
probs_pred[1:100]
```

```
##   [1] 0.2699432 0.2689414 0.3071112 0.2689414 0.6829198 0.7303729 0.2689414
##   [8] 0.7310584 0.7310582 0.7310586 0.2689466 0.3265254 0.7310585 0.2689729
##  [15] 0.2689414 0.7310586 0.4279526 0.2689420 0.7310507 0.2689739 0.2697083
##  [22] 0.7310584 0.7308722 0.7310586 0.7310586 0.2689461 0.7303667 0.4468113
##  [29] 0.2689415 0.7310586 0.2696335 0.2689711 0.2689894 0.2697286 0.2689414
```

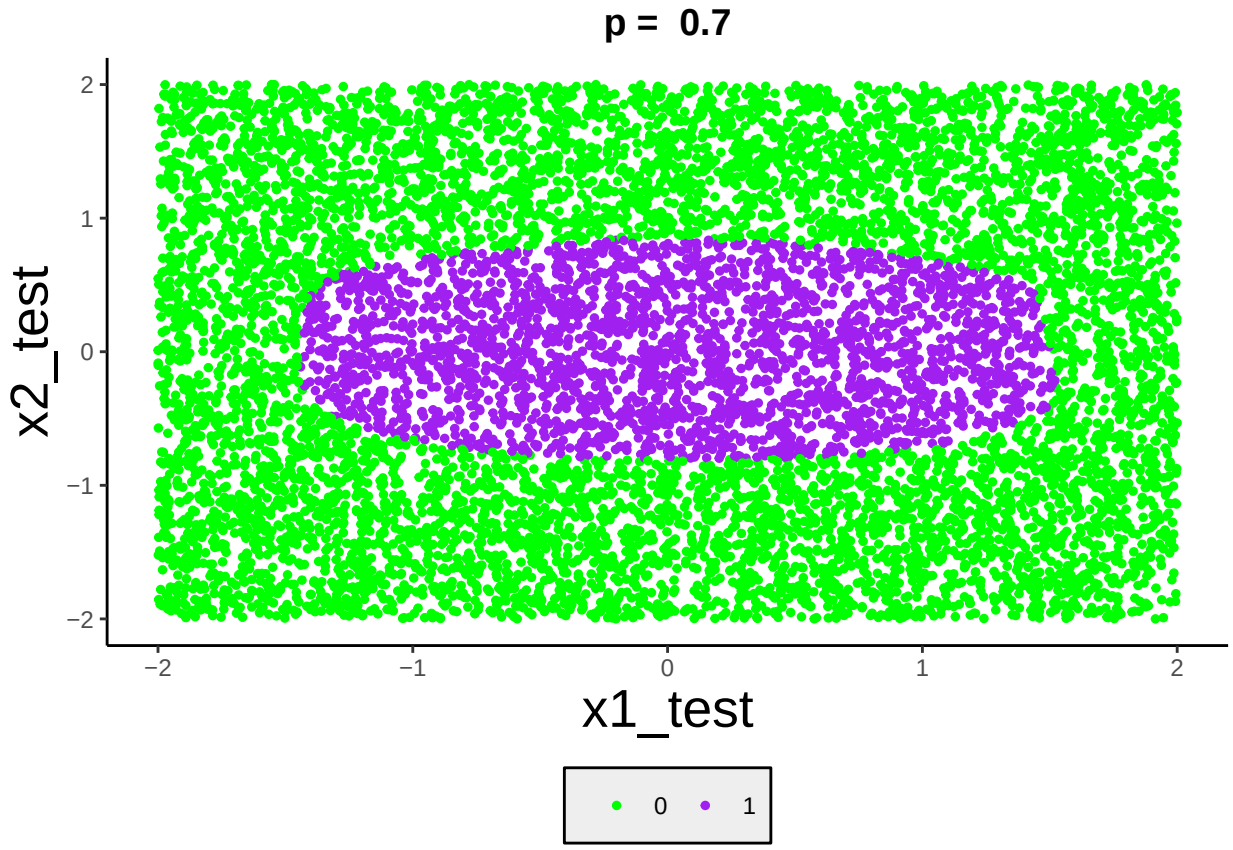
```
##  [36] 0.5933258 0.7310470 0.7309195 0.2694403 0.2689646 0.2689414 0.3615145
##  [43] 0.2689414 0.3225418 0.5786524 0.2689423 0.2695842 0.7310585 0.2691247
##  [50] 0.4134947 0.2689532 0.7310533 0.4569352 0.2689414 0.4445158 0.2689414
##  [57] 0.3519746 0.2689414 0.7310580 0.7310586 0.3925795 0.7310585 0.7293873
##  [64] 0.2689414 0.7310586 0.2689414 0.2973443 0.2689414 0.2689414 0.3928229
##  [71] 0.2689414 0.2689417 0.7310584 0.2693893 0.2689414 0.2689414 0.4327266
##  [78] 0.7269578 0.6258900 0.2693815 0.7308486 0.2689427 0.2689414 0.2713291
##  [85] 0.2689414 0.2701822 0.2689414 0.7310586 0.6957336 0.7310580 0.2689414
##  [92] 0.7310171 0.2697093 0.2696545 0.7310513 0.2689414 0.2689414 0.2691431
##  [99] 0.7310501 0.7308815
```

From the print statement, it looks like any p between $0.60 - 0.70$ might be a good choice since the neural net will be able to classify (for the most part the neural net outputs probabilites in the range $0.20 - 0.30$ and $0.60 - 0.70$).

If I choose a p greater than $0.70 - 0.75$ my classifier will not be able to classify. We can visualize this with plots.

Lets first visualize with $p = 0.7$:

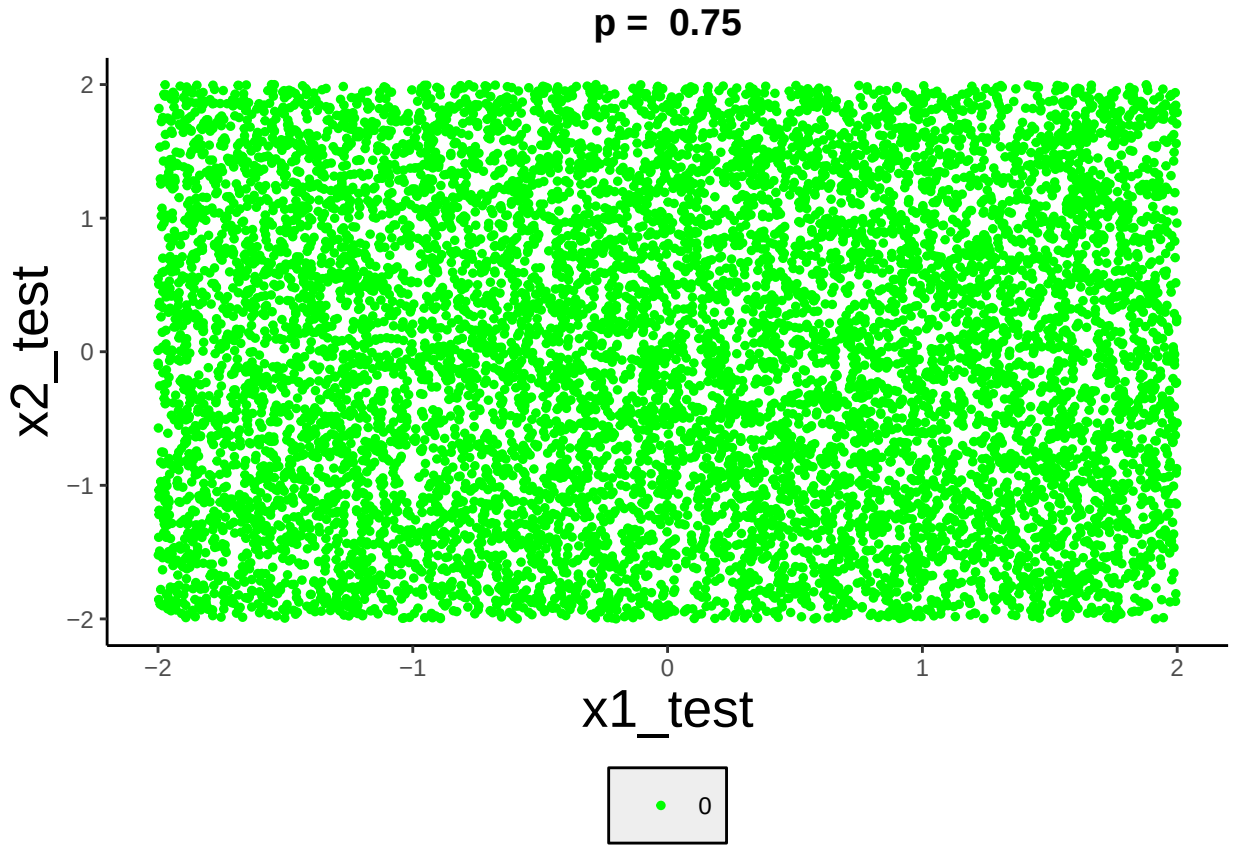
```
plotting <- function(p){
  y_pred <- F_func(probs_pred, p)

  ggplot(data.frame(x1_test=x1_test,x2_test=x2_test,y_pred=y_pred))+
  geom_point(aes(x = x1_test, y = x2_test, color = as.factor(y_pred)), size=1) +
  scale_color_manual(values = c("green", "purple")) +
  theme_classic() +
  ggtitle(paste("p = ", p)) +
  theme(legend.position = "bottom",
        legend.background = element_rect(fill = "#EEEEEE", color = "black"),
        legend.title = element_blank(),
        axis.title = element_text(size = 20),
        plot.title = element_text(size=14, face='bold', hjust = 0.5))
}

plotting(0.7)
```



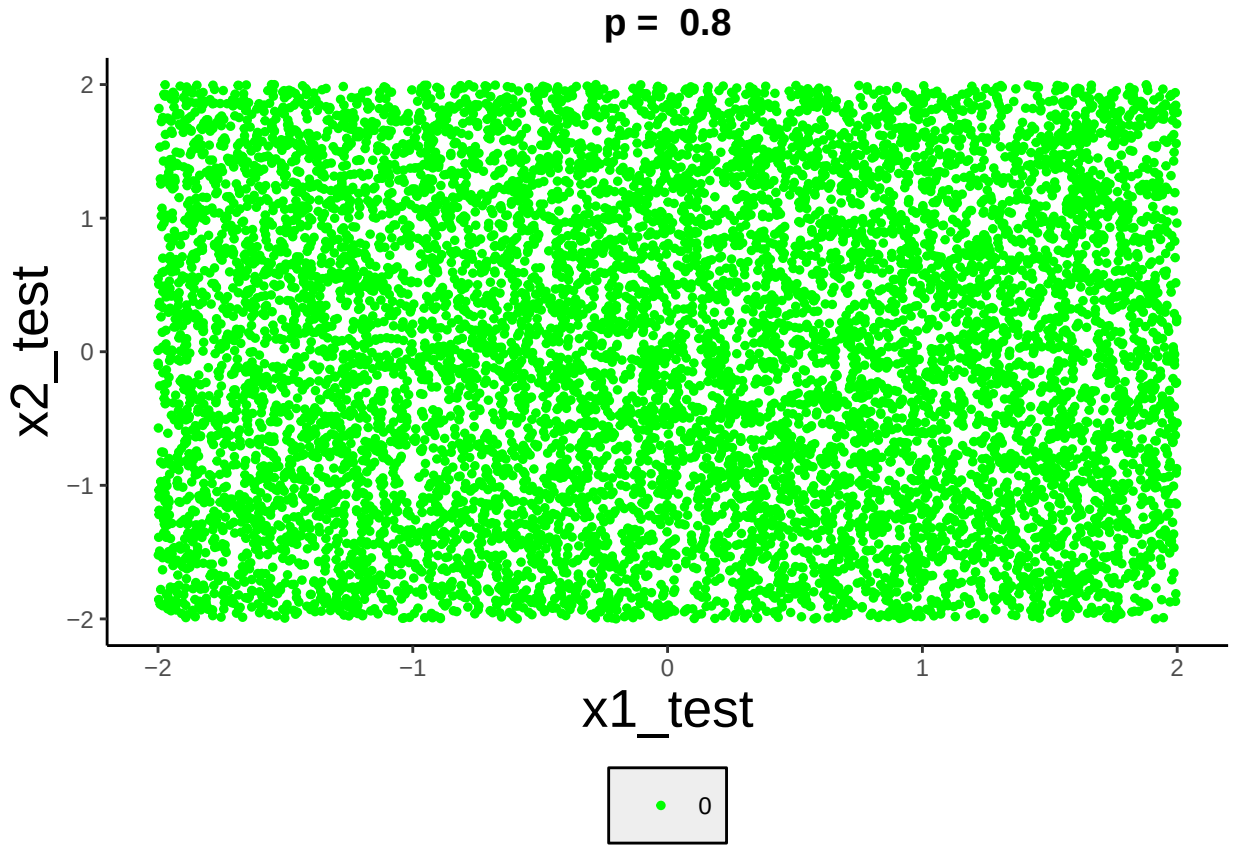
The plot is similar to the plot generated using the nn.txt data (since there is no antenna, this is a medium fit).

Now lets visualize with a p greater than $0.70 - 0.75$:

```
plotting(0.75)
```



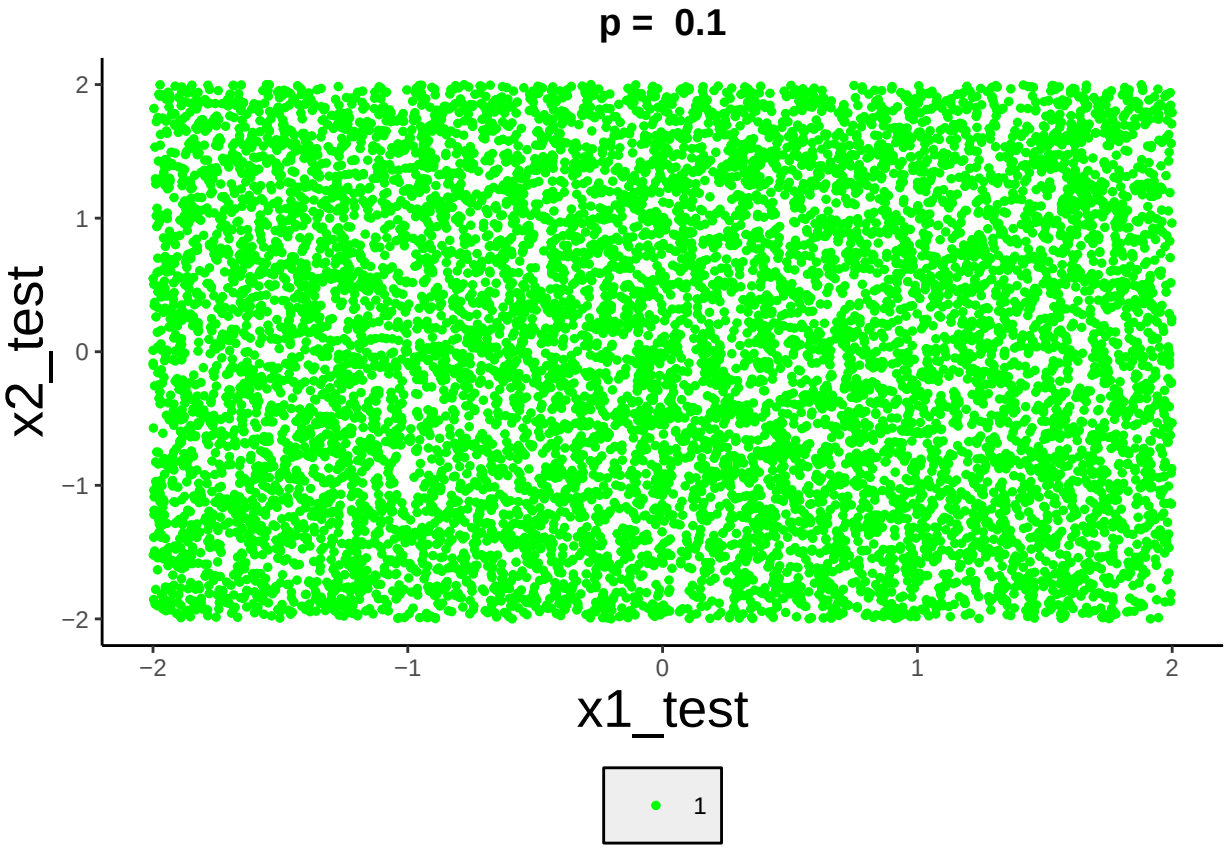

```
plotting(0.8)
```

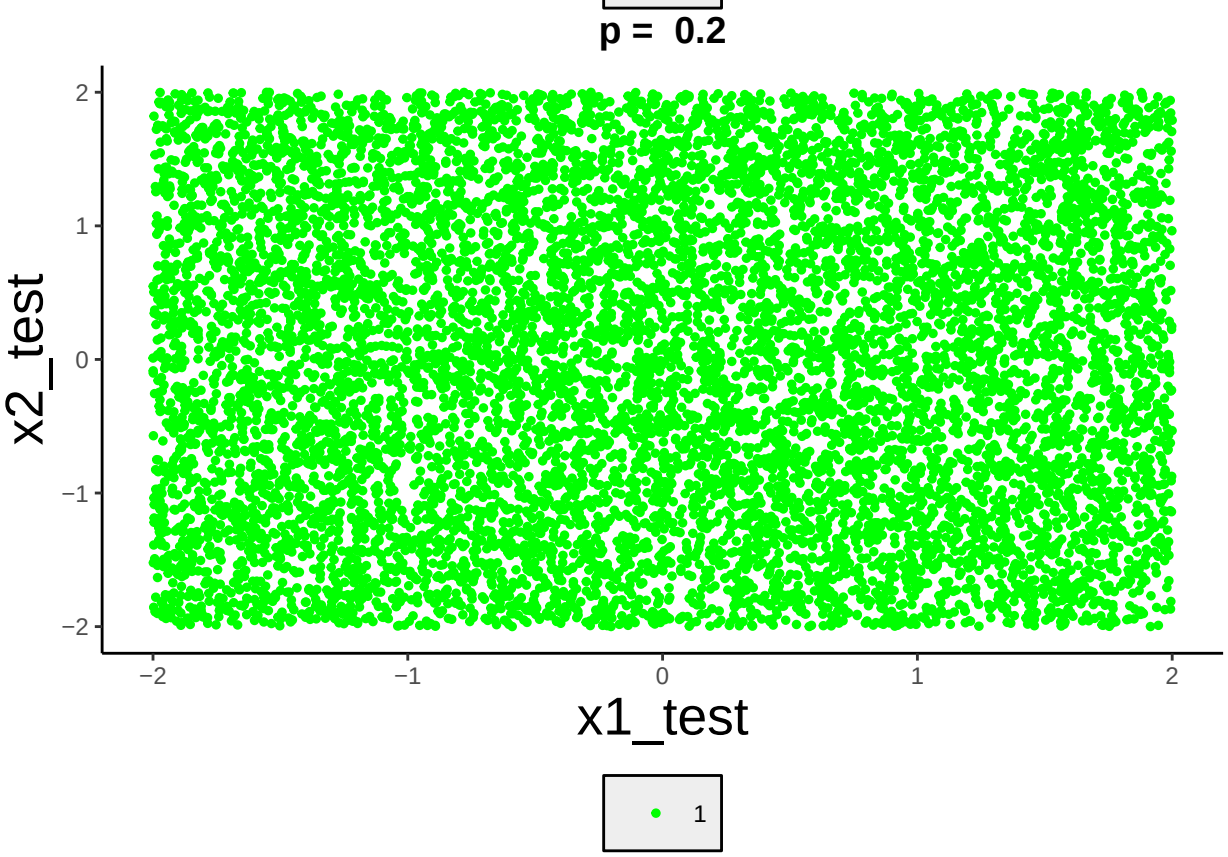


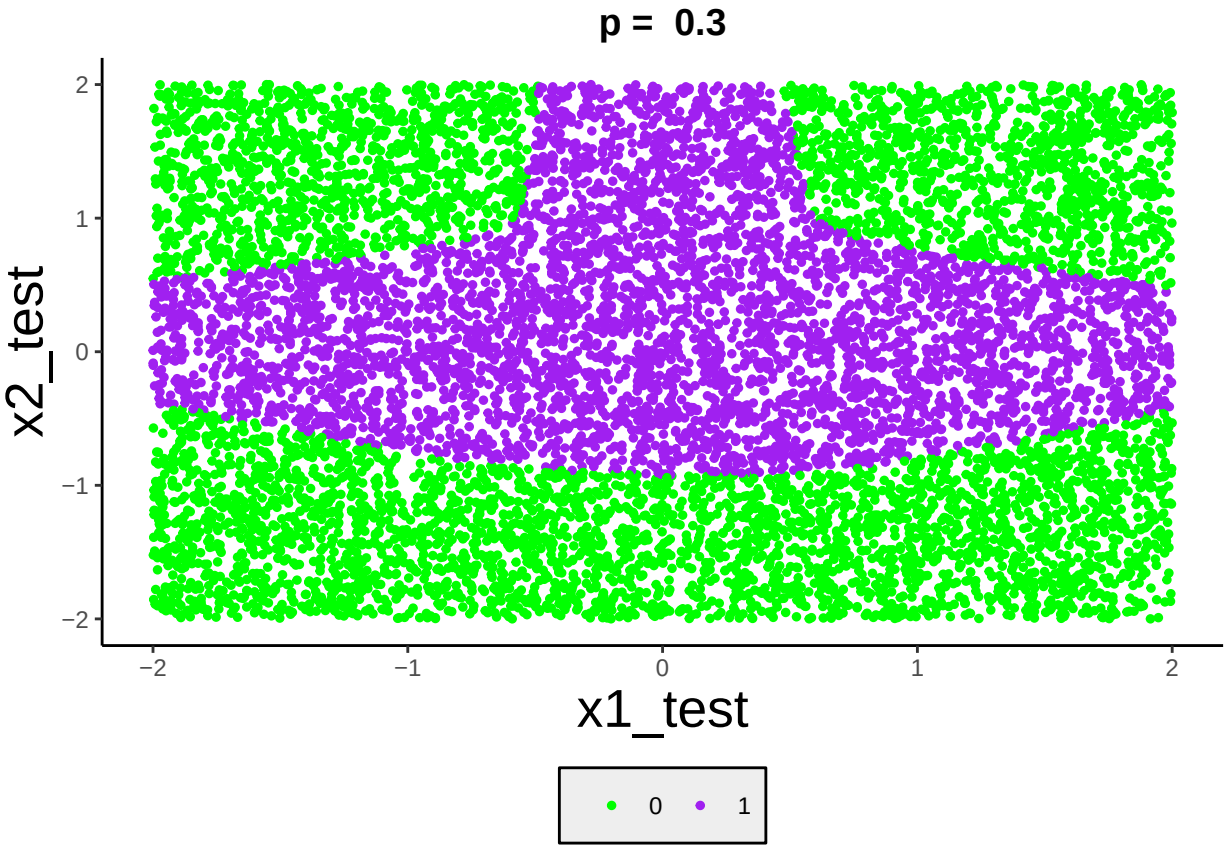
The above two plots make sense. A cutoff higher than $0.70 - 0.75$ will do a poor job of classifying the test data. As seen before the neural net doesn't output probabilities higher than $0.70 - 0.75$ (for the most part the neural net gives probabilites in the range $0.20 - 0.30$ or in the range $0.70 - 0.80$).

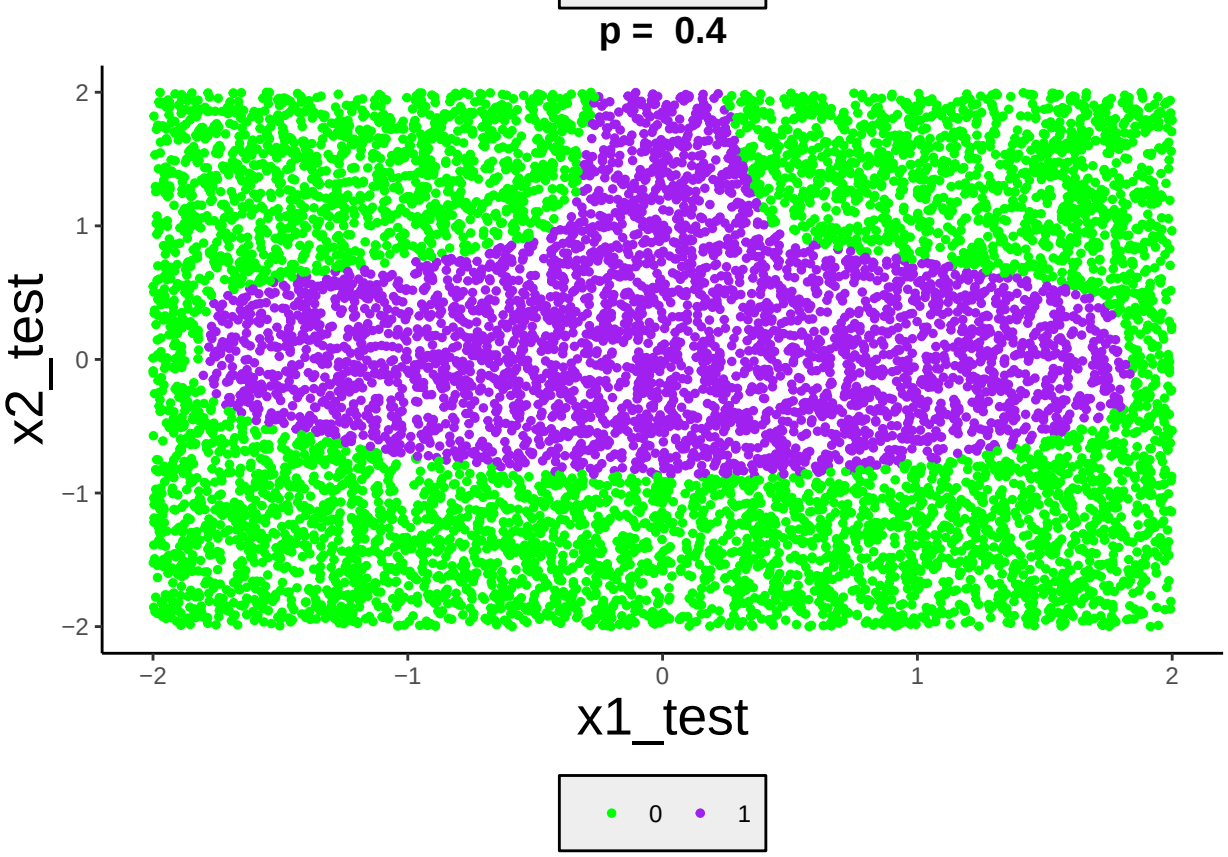
Lastly, lets use a p smaller than 0.7. Lets choose $0.1, 0.2, 0.3, 0.4, 0.5, 0.6$ and see how the neural net + classifier F perform.

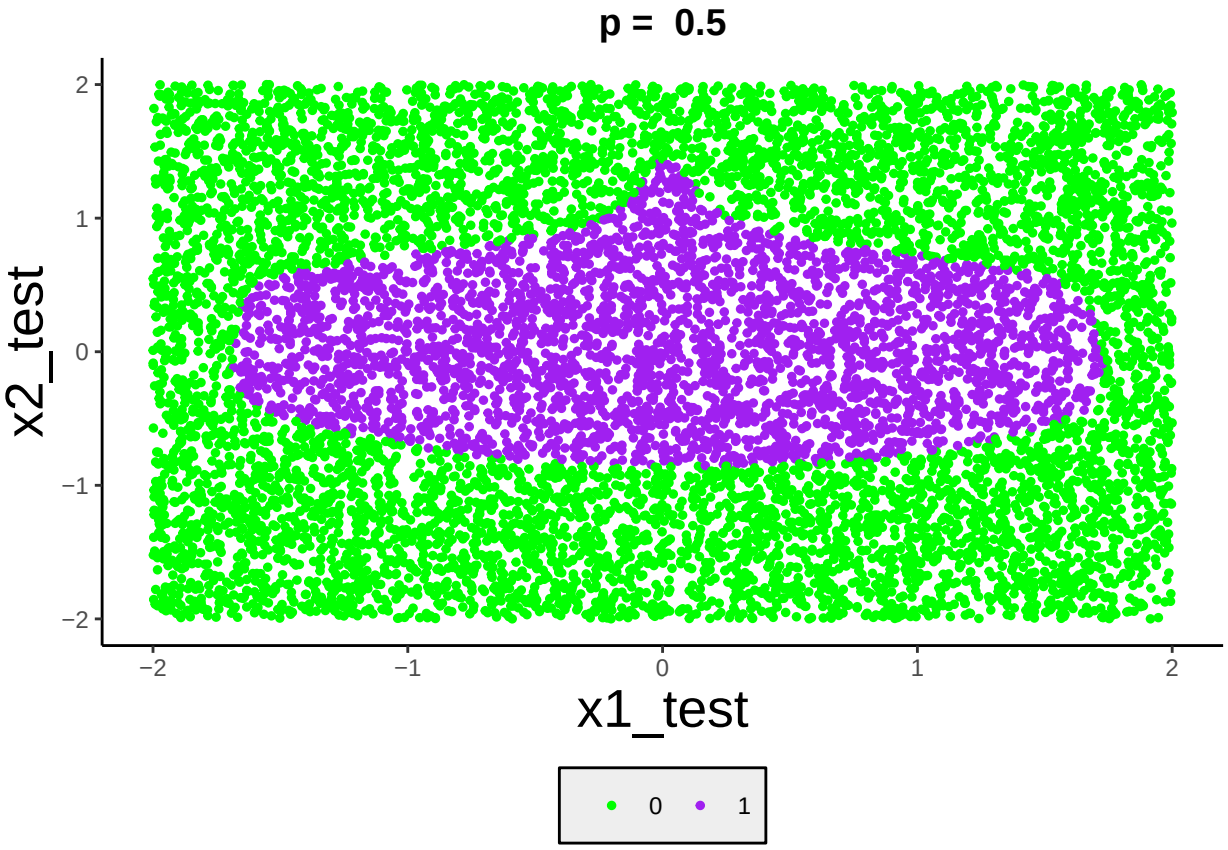
```
for (p in seq(0.1,0.6,0.1)){
  print(plotting(p))
}
```

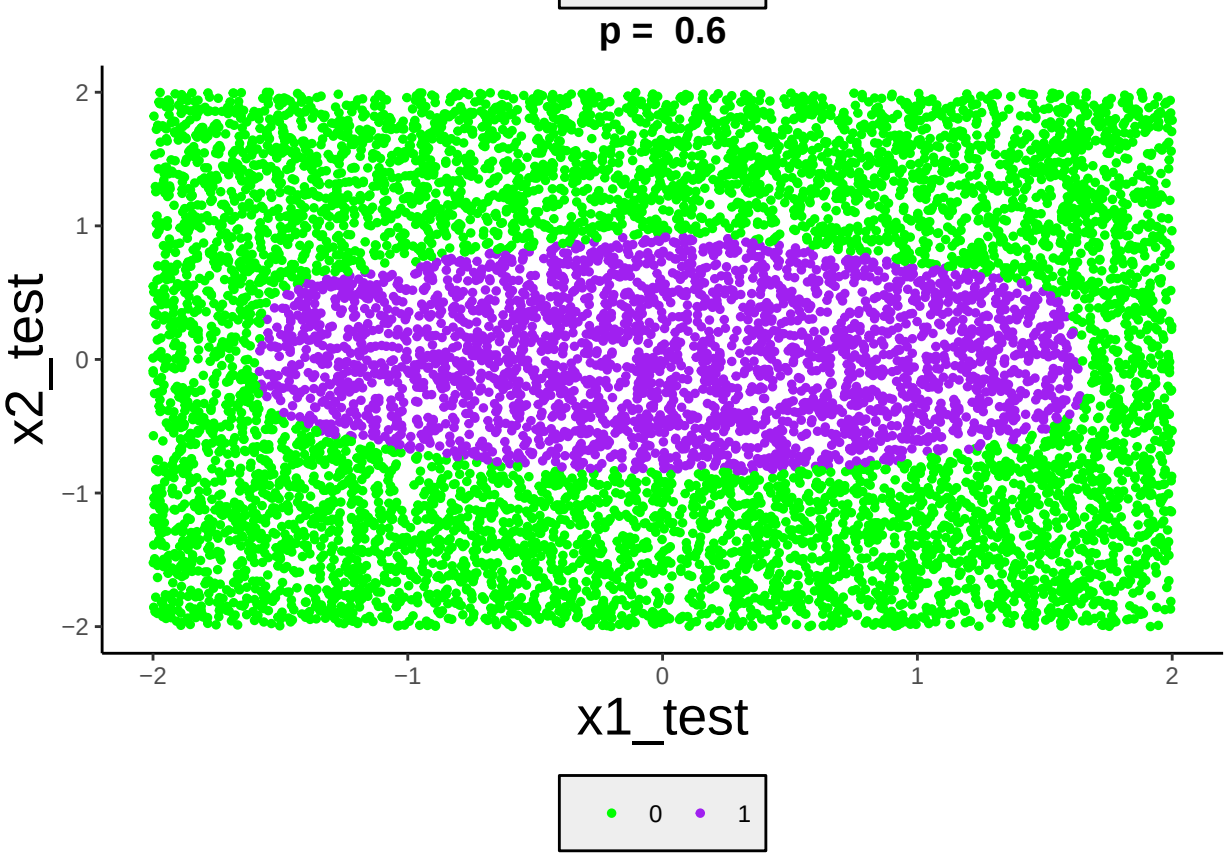

p = 0.1
x2_test
x1_test
2
1
0
−1
−2
−2
−1
0
1
2
1


p = 0.2
x2_test
x1_test
2
1
0
−1
−2
−2
−1
0
1
2
1


p = 0.3
x2_test
x1_test
2
1
0
−1
−2
−2
−1
0
1
2
0
1


p = 0.4
x2_test
x1_test
2
1
0
−1
−2
−2
−1
0
1
2
0
1





From the above plots, you get a better fit with a $p = 0.4$ as you can see the antenna (even with a

$p = 0.3, p = 0.5$). With a $p = 0.6$ you get again, similar to $p = 0.7$, a medium fit. And using a p smaller than 0.3 you get a very poor fit (you cannot classify).