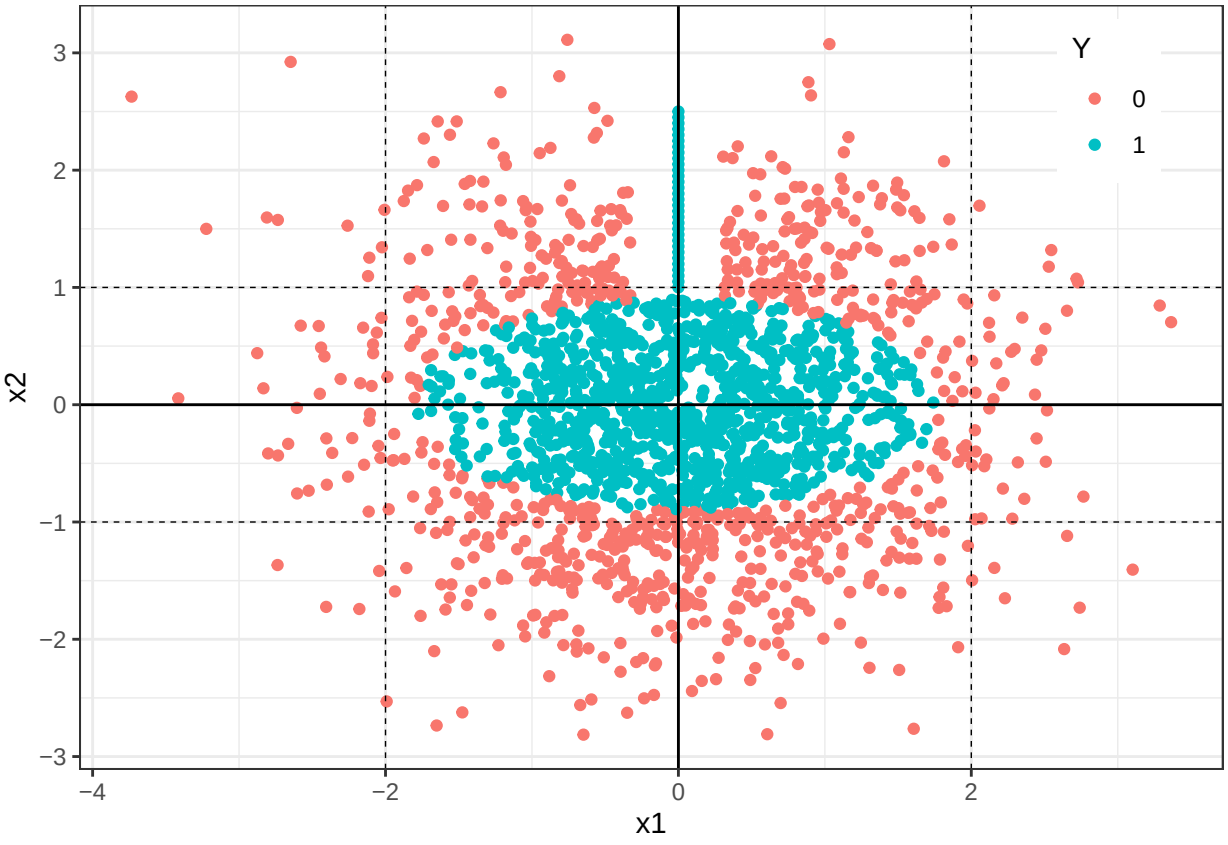
Math 504 HW13

Jeff Gould

4/13/2020

A

```
nn_data <- read_delim("nn.txt", delim = " ")

ggplot(data = nn_data, aes(x = x1, y = x2)) +
  geom_point(aes(color = as.factor(y))) +
  scale_color_discrete(name = "Y") +
  theme_bw() +
  geom_hline(yintercept = 0) +
  geom_vline(xintercept = 0) +
  geom_hline(yintercept = c(-1,1), color = "black", linetype = 2, size = 0.25) +
  geom_vline(xintercept = c(-2,2), color = "black", linetype = 2, size = 0.25) +
  theme(legend.position = c(0.9,0.875))
```



B

Since we are computing with $n = 2$, we will need $(n+1)m = 3m$ parameters to go from $(x_1, x_2) \to \boldsymbol{Z}$. Then to go from $\boldsymbol{Z} \to T_i$ requires $m+1$ parameters, so with $\boldsymbol{T} = (T_1, T_2)$ we need $2(m+1) = 2m+2$ parameters

to go from $\boldsymbol{Z} \to \boldsymbol{T}$, and thus a total of $3m + 2m + 2 = 5m + 2$ will be the dimension of α

C

We write a function `N` that takes a matrix $X = [x^{(1)}x^{(2)}]$, $\alpha = (\beta_0^{(1)}\beta^{(1)} \dots \beta_0^{(m)}\beta^{(1)}\gamma_0^{(1)}\gamma^{(1)}\gamma_0^{(2)}\gamma^{(2)})$ and returns a vector of probabilites that the coordinate pair is equal to 1. While the problem only required us to take in a single observation, by writing it in the manner below it allows us to be more computationally efficient than other methods. But the function can still take in a single (x_1, x_2) observation and return the probability that the point is 1.

First, the function takes the β entries in α and gathers them into a $3 \times m$ matrix, where each column i is $[\beta_0^{(i)}\beta^{(i)}]^T$, and it takes the γ entries and gathers them into a single $(1 + m) \times 2$ matrix, Γ.

Then we calculate the first set of weights: $\boldsymbol{w}_1 = \boldsymbol{X}\boldsymbol{\beta}$.

Next: $\boldsymbol{Z} = \sigma(\boldsymbol{w}_1)$, where σ denotes the sigmoid function

$\boldsymbol{w}_2 = [1\boldsymbol{Z}]\boldsymbol{\Gamma}$

$\boldsymbol{T} = \sigma(\boldsymbol{w}_2)$

$\hat{\boldsymbol{p}} = \frac{\exp(\boldsymbol{T_1})}{\exp(\boldsymbol{T_1})+\exp(\boldsymbol{T_2})}$

```
X <- data.matrix(nn_data[,1:2])
m <- 4
y <- nn_data$y
nn_data <- data.matrix(nn_data)

sigmoid <- function(w){
  return(1 / (1 + exp(-w)))
}

N <- function(x, alpha, m){

  x <- cbind(1,x)
  Beta <- matrix(data = alpha[1:(3 * m)], nrow = 3, byrow = F)
  Gamma <- matrix(data = alpha[(3*m+1):(5*m+2)], ncol = 2, byrow = F)
  w1 <- x %*% Beta
  Z <- sigmoid(w1)
  Z <- cbind(1,Z)
  w2 <- Z %*% Gamma
  T_vec <- sigmoid(w2)
  nn_estimate <- exp(T_vec[,1]) / (exp(T_vec[,1])+exp(T_vec[,2]))

  return(nn_estimate)

}
```

D

From previous work, we know that the log-loss for a logistic regression is:

$$\log L(\alpha) = \sum_{i=1}^{N} y_i \log(P(y_i = 1|x^{(i)}\alpha)) + (1 - y_i)(log(1 - \log(P(y_i = 1|x^{(i)}\alpha))$$

From above, we calculate $P(y_i = 1|x^{(i)}\alpha) = \frac{\exp(T_1)}{\exp(T_1)+\exp(T_2)}$, thus the above equation clearly becomes:

$$\log L(\alpha) = \sum_{i=1}^{N} y_i \log\left(\frac{\exp(T_1)}{\exp(T_1)+\exp(T_2)}\right) + (1-y_i)(log\left(1 - \frac{\exp(T_1)}{\exp(T_1)+\exp(T_2)}\right)$$

In our log loss function below, we first compute $(\hat{\boldsymbol{p}}|\boldsymbol{X}, \alpha)$, which returns a vector of probabilites $= \frac{\exp(T_1)}{\exp(T_1)+\exp(T_2)}$. Then we simply calculate the loss at each observation and sum them together.

```
logL <- function(X, y, alpha, m){

  probs <- N(X, alpha, m)

  logLoss <- sum(
    y * log(probs) + (1 - y) * log(1 - probs)
  )
  return(logLoss)

}
```

E

We write a function that takes a sample of (x_1, x_2, y) observations and returns the gradient vector for the collection of points using the finite difference method. The nested function `fin_diff(i)` takes an input `i` and calculates the corresponding finite difference for the α_i. The `sapply` computes the finite difference for every i and returns the gradient vector.

```
gradLogL <- function(data_samp, alpha, m, h = 1E-8){

  X_samp <- data_samp[,1:2]
  y_samp <- data_samp[,3]

  logLoss <- logL(X_samp, y_samp, alpha, m)

  fin_diff <- function(i){
    alpha_diff <- alpha
    alpha_diff[i] <- alpha_diff[i] + h
    diff <- (logL(X_samp, y_samp, alpha_diff, m) - logLoss) / h
    return(diff)
  }

  i <- 1:length(alpha)
  gradL <- sapply(i, fin_diff)
  return(gradL)
}
```

F

Here we write a stochastic gradient ascent algorithm that takes the observation data (as a matrix input), `nn_data` and a starting α as inputs. From there we select a random sample of rows, size defined by `mini_batch` in the function argument, and calculate the gradient for the observed sample. We then update α with `alpha = alpha + step * gradient`, where `step` is defined in the function arguments. We iterate this `epoch` times,

where `epoch` is another function argument. The function then returns a new α that will closely predict each y_i

```
StochasticGradientAscent <- function(data, alpha,
                                     m = 4,
                                     epoch = 5E6,
                                     mini_batch = 25,
                                     step = 0.01,
                                     with_replacement = FALSE){


  initial_sample_rows <- c(1:nrow(data))
  current_sample_rows <- initial_sample_rows
  L <- logL(X = data[,1:2], y = data[,3], alpha, m)

  for (i in 1:epoch) {

    sample_rows <- sample(current_sample_rows, mini_batch)

    if (!with_replacement) {
      ### Remove rows we just sampled
      current_sample_rows <- current_sample_rows[-sample_rows]
      if (length(current_sample_rows) <= mini_batch) {
        ### If we have gone through all rows, refill our sample space
        current_sample_rows <- initial_sample_rows
      }
    }

    stochSample <- data[sample_rows,]

    stochGrad <- gradLogL(stochSample, alpha, m)

    alpha <- alpha + step * stochGrad

    if(i %% 1000 == 0){
      L <- c(L, logL(X = data[,1:2], y = data[,3], alpha, m))
    }

  }

  output <- list(LossHistory = L, Alphas = alpha)
  return(alpha)

}
```

We generate a starting α matrix, where each column of α is a randomaly generated starting α to train and then compare results. Using `parallel::detectCores()`, we see that 28 cores are available for computation. We will use 20 of them and iterate over `epoch` $= 2.5E6$. Each column of α will be drawn from the standard normal distribution and the multiplied by the column number (or, each column i will be drawn from $N(0, \sigma = i)$)

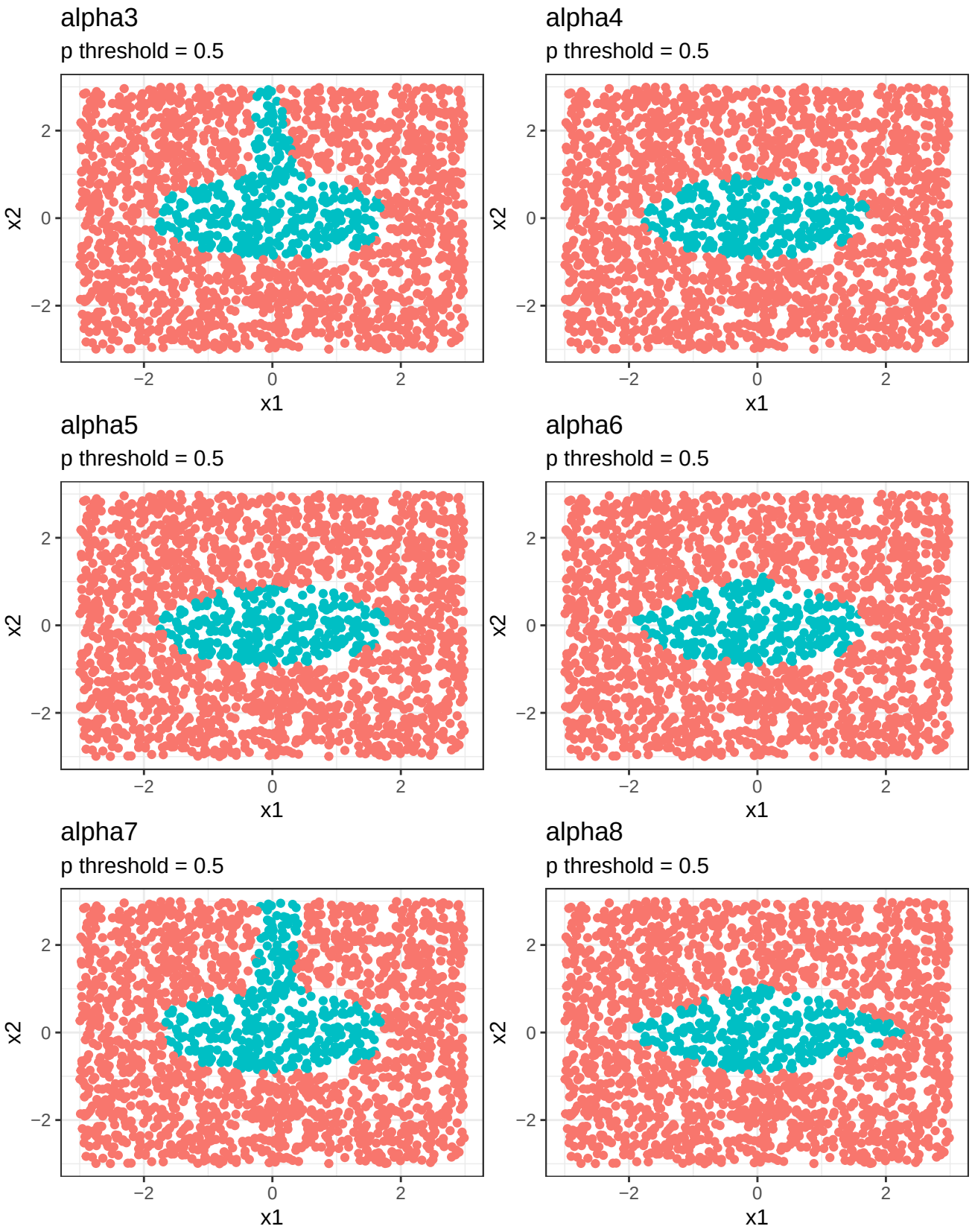
```
i <- 20

alpha_start <- matrix(0, nrow = 5 * m + 2, ncol = i)
for (j in 1:i) {
  alpha_start[,j] <- j * rnorm(5*m+2)
```

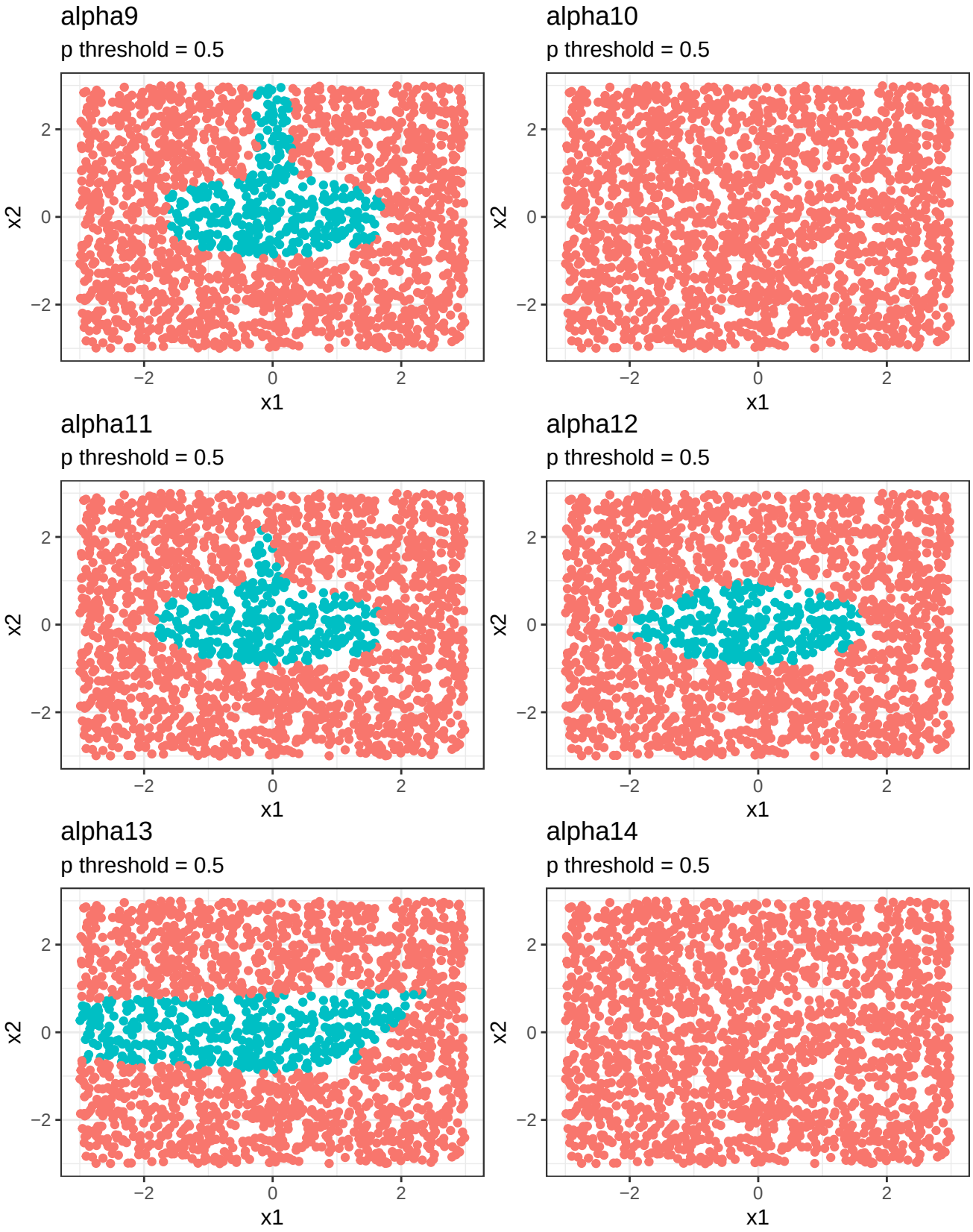
```
}

tictoc::tic()
cl <- makeCluster(i)
parallel::clusterExport(cl, varlist = c("nn_data", "gradLogL", "logL", "N", "sigmoid"))
optimAlphas <- parApply(cl = cl, X = alpha_start, MARGIN = 2,
                        FUN = StochasticGradientAscent, data = nn_data, epoch = 2.5E6)
stopCluster(cl)
tictoc::toc()
```

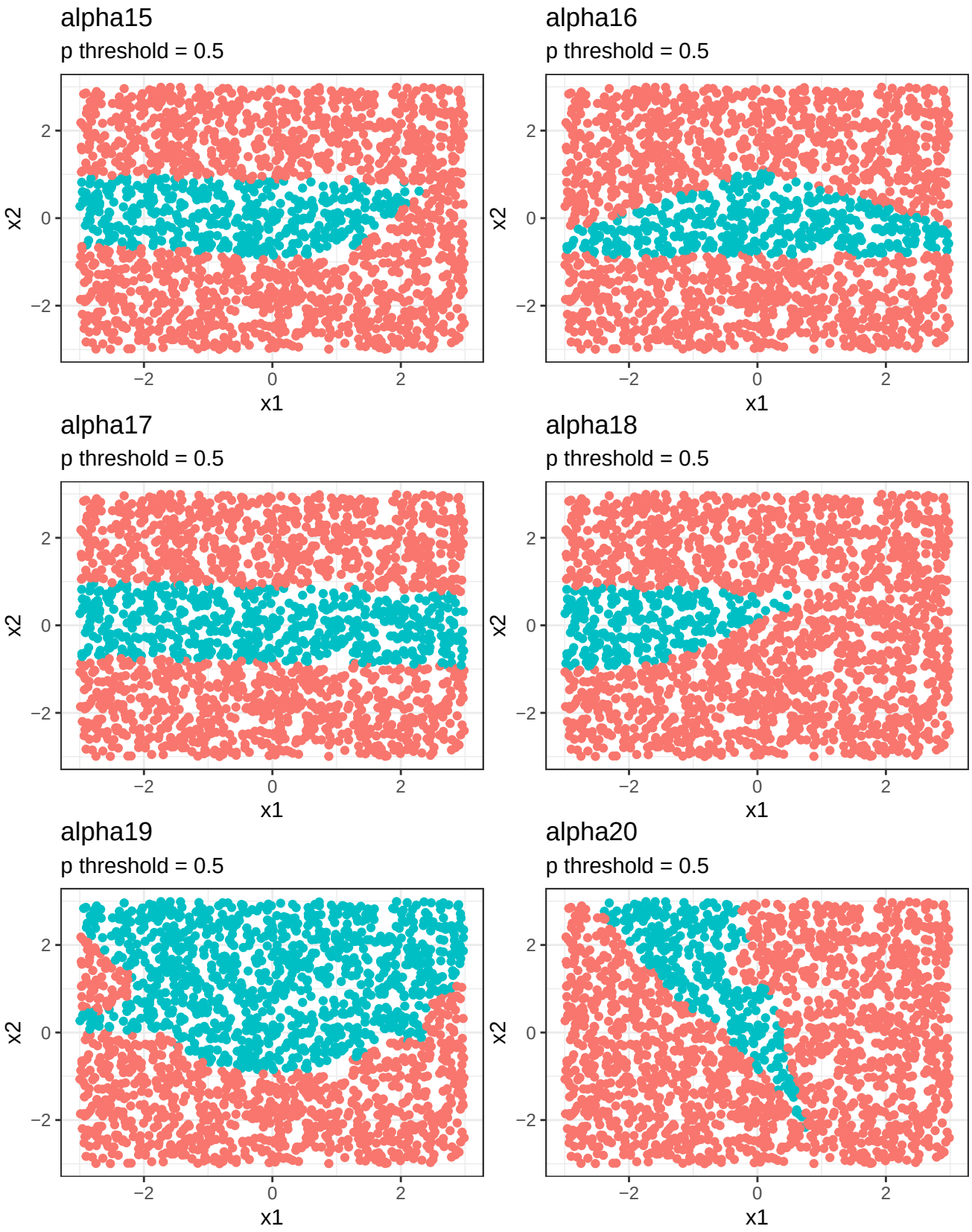
```
## [1] 2503.607 sec elapsed
```

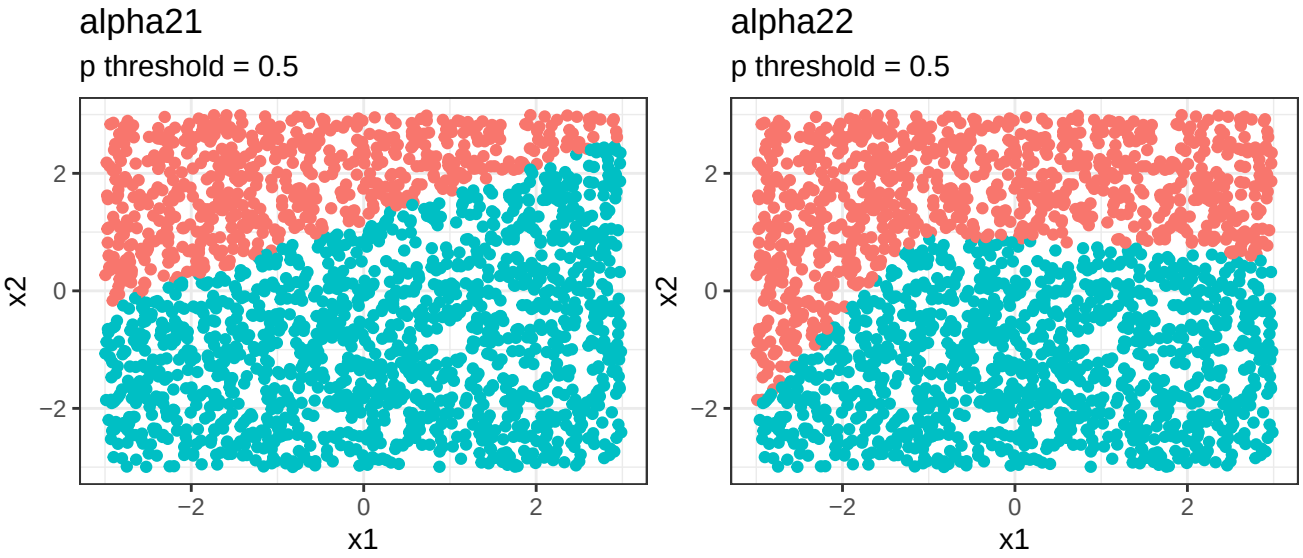
Next we take the output α matrix, and for each column α_i calculate $\hat{p}$ on data randomly drawn from the uniform distribution over $(-3, 3) \times (-3, 3)$. We then use a p-threshold of $p = 0.5$ to define each $\hat{y}$ as 0 or 1, and plot the resulting graph for each α_i

```
test_X <- matrix(runif(4000,-3,3), ncol = 2)
p_hats <- apply(optimAlphas, 2, N, x = test_X, m = 4)
plot_outputs <- as.data.frame(cbind(test_X, p_hats)) %>%
  rename(x1 = V1, x2 = V2) %>%
  mutate_at(3:22, round) %>% ##use p = 0.5 as cutoff to get quick estimates
  mutate_at(3:22, as.factor) %>%
  rename_at(vars(starts_with("V")), funs(sub("V", "alpha",.)))

vec_of_names <- colnames(plot_outputs)[3:22]

make_plot <- function(plot_var){
  col_var <- plot_var

  return(ggplot(data = plot_outputs, aes(x = x1, y = x2, color = !!ensym(col_var))) +
           geom_point(show.legend = F) +
           labs(title = paste0(col_var), subtitle = "p threshold = 0.5")+
           theme_bw())

}

for (i in 1:length(vec_of_names)) {
  print(make_plot(vec_of_names[i]))
}
```


alpha3
p threshold = 0.5
alpha4
p threshold = 0.5
alpha5
p threshold = 0.5
alpha6
p threshold = 0.5
alpha7
p threshold = 0.5
alpha8
p threshold = 0.5
x1
x2
−2
0
2


alpha9
p threshold = 0.5
alpha10
p threshold = 0.5
alpha11
p threshold = 0.5
alpha12
p threshold = 0.5
alpha13
p threshold = 0.5
alpha14
p threshold = 0.5
x1
x2
−2
0
2


alpha15
p threshold = 0.5
alpha16
p threshold = 0.5
alpha17
p threshold = 0.5
alpha18
p threshold = 0.5
alpha19
p threshold = 0.5
alpha20
p threshold = 0.5
x1
x2
−2
0
2



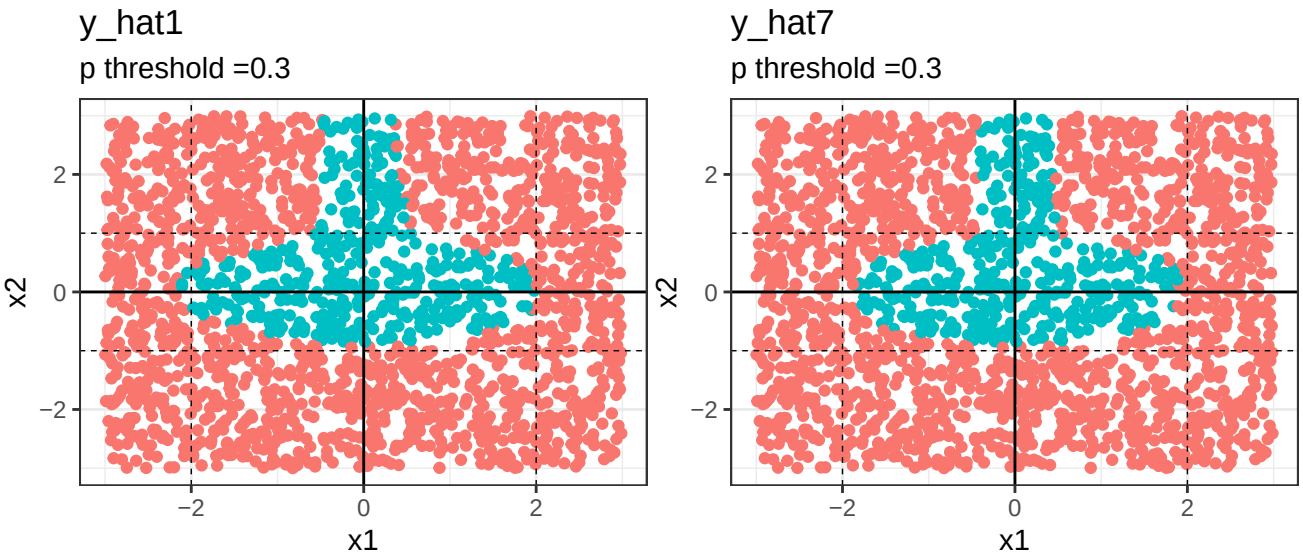
So here we have a few models that were trained very well, able to capture the antenna along with the ellipse, some that were trained well enough to capture the ellipse, but not the antenna, and a few still that were not trained well at all. The random α's increased in variance with each column, and it was the last few α columns which were trained poorly, so this is most likely because they started too far away from a good solution and still need several more epoch's to near a good point.

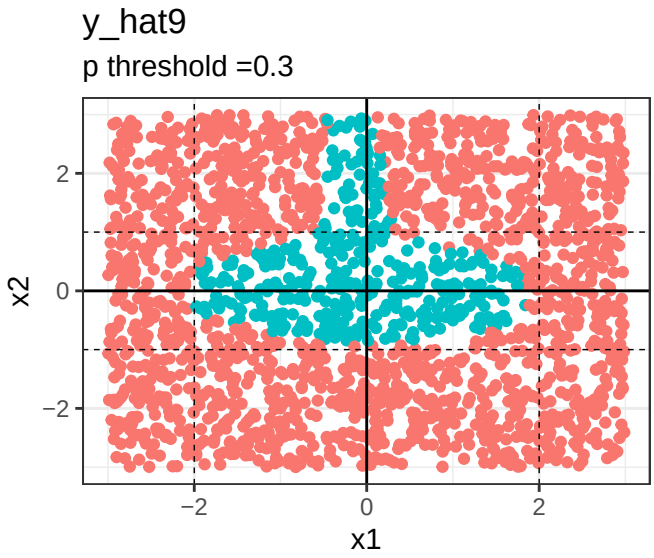
We will select `alpha3`, `alpha9`, and `alpha11` for step **G** (these are actually the first, seventh, and 9th columns in `optimAlpha`, so they should be thought of as $\alpha_1, \alpha_7, \alpha_9$)

G

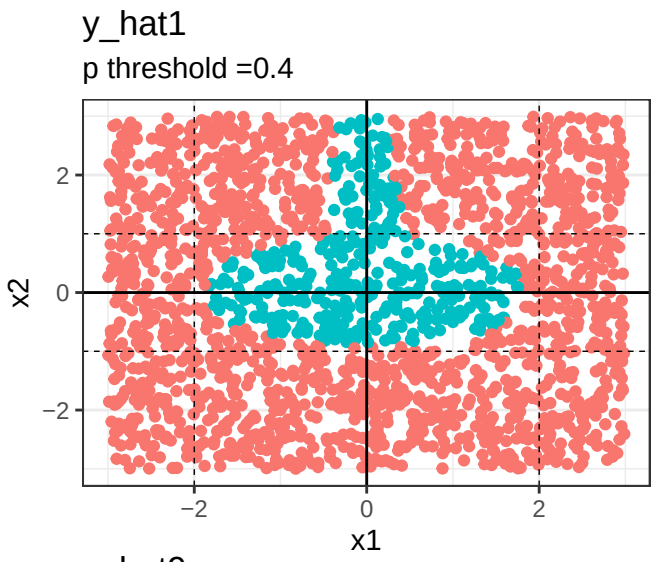
```
init_results <- data.frame(x1 = test_X[,1],
                           x2 = test_X[,2],
                           p_hat1 = p_hats[,1],
                           p_hat7 = p_hats[,7],
                           p_hat9 = p_hats[,9])


make_p_charts <- function(p_thresh) {

  check_results <- init_results %>%
    mutate(
      y_hat1 = ifelse(p_hat1 > p_thresh, 1, 0),
      y_hat7 = ifelse(p_hat7 > p_thresh, 1, 0),
      y_hat9 = ifelse(p_hat9 > p_thresh, 1, 0)
    ) %>%
    mutate_at(6:8, as.factor)

  vec_of_names <- colnames(check_results)[6:8]

  make_plot <- function(plot_var) {
    col_var <- plot_var

    return(
      ggplot(data = check_results, aes(x = x1, y = x2, color = !!ensym(col_var))) +
```

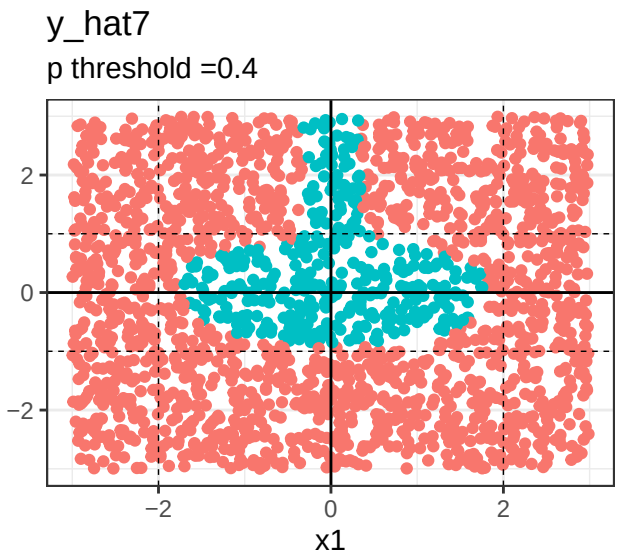
```
        geom_point(show.legend = F) +
        labs(title = paste0(col_var), subtitle = paste0("p threshold =", p_thresh)) +
        theme_bw() +
        geom_hline(yintercept = 0) +
        geom_vline(xintercept = 0) +
        geom_hline(yintercept = c(-1,1), color = "black", linetype = 2, size = 0.25) +
        geom_vline(xintercept = c(-2,2), color = "black", linetype = 2, size = 0.25)
    )
  }



  for (i in 1:length(vec_of_names)) {
    print(make_plot(vec_of_names[i]))
  }


}

make_p_charts(0.3)
```

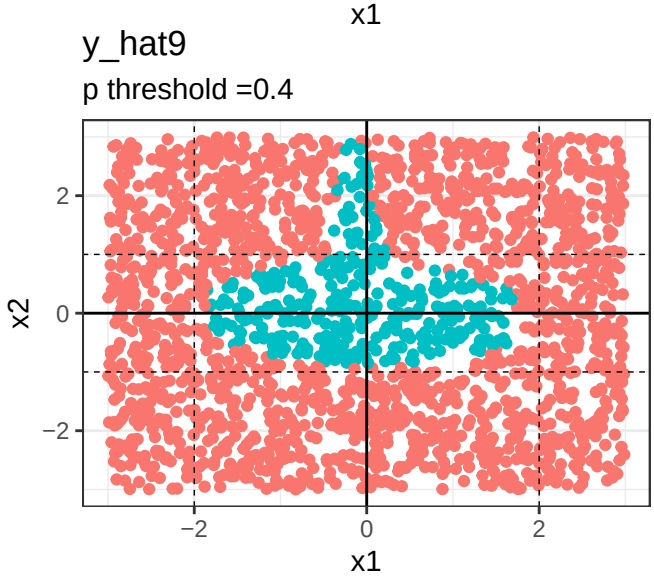




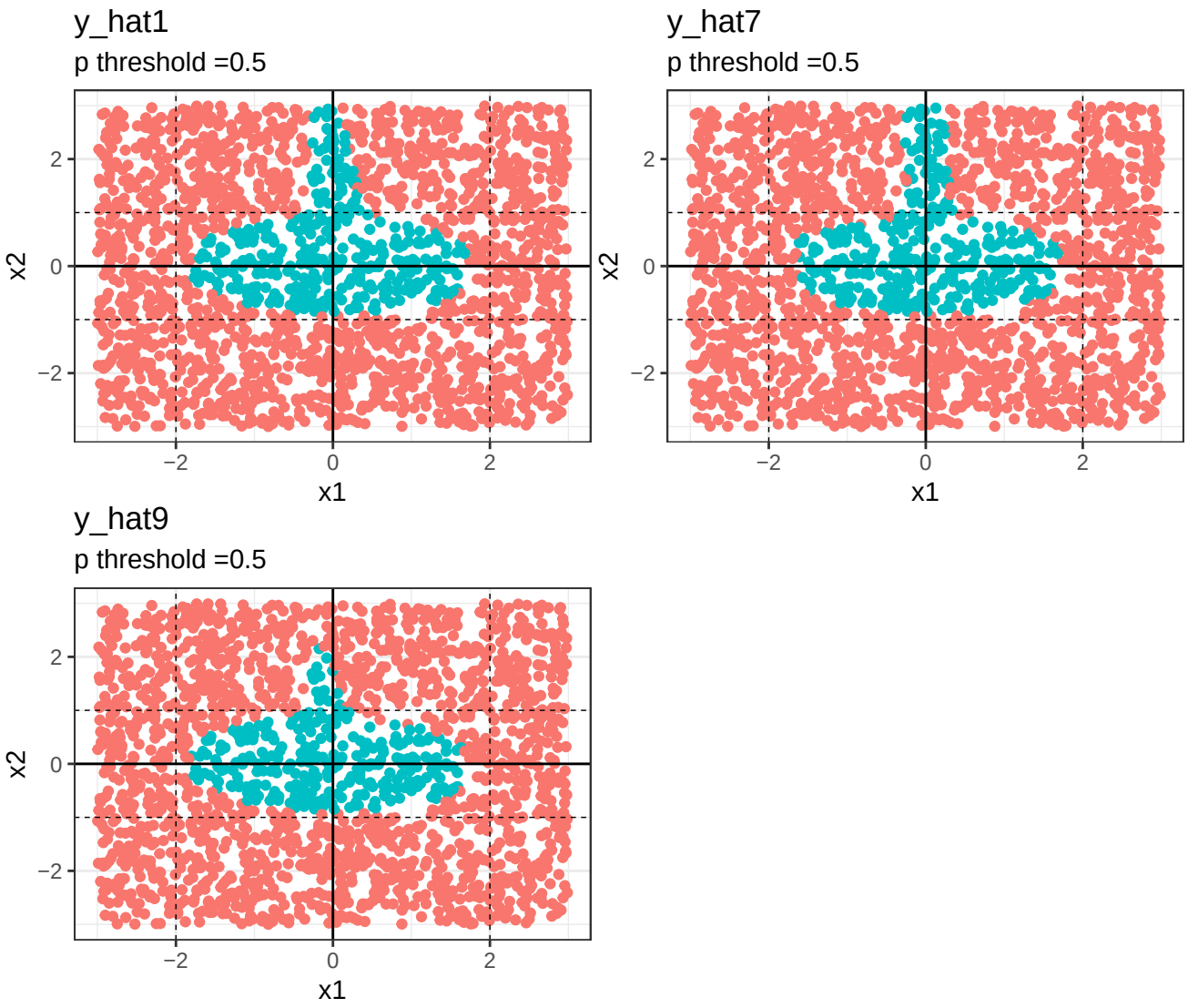

```
make_p_charts(0.4)
```



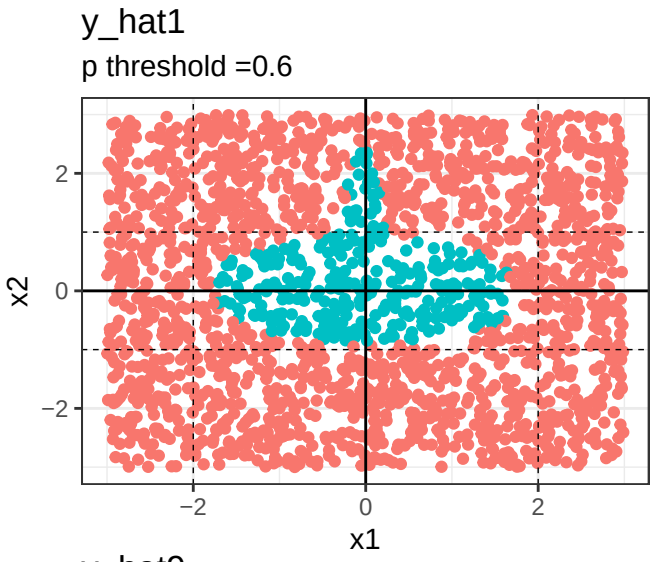


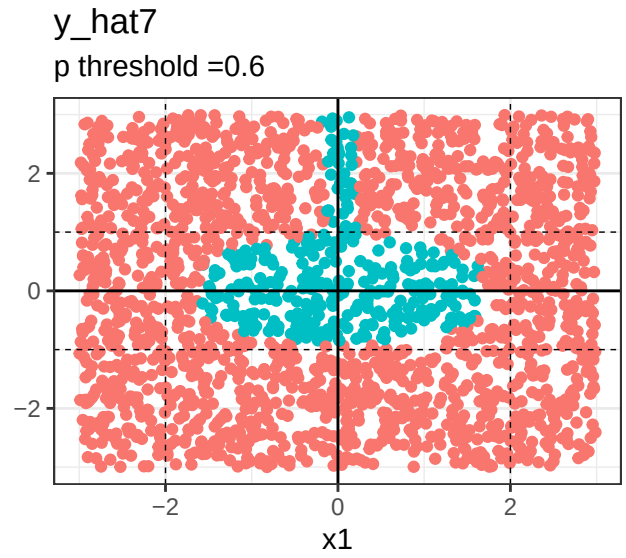


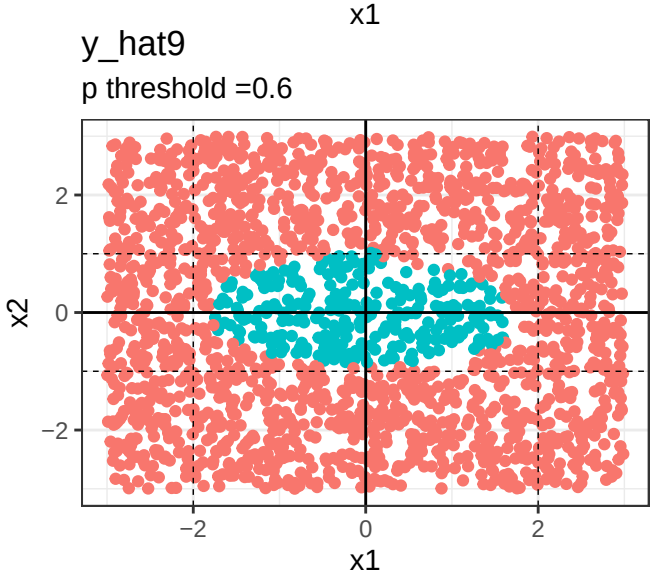
```
make_p_charts(0.5)
```



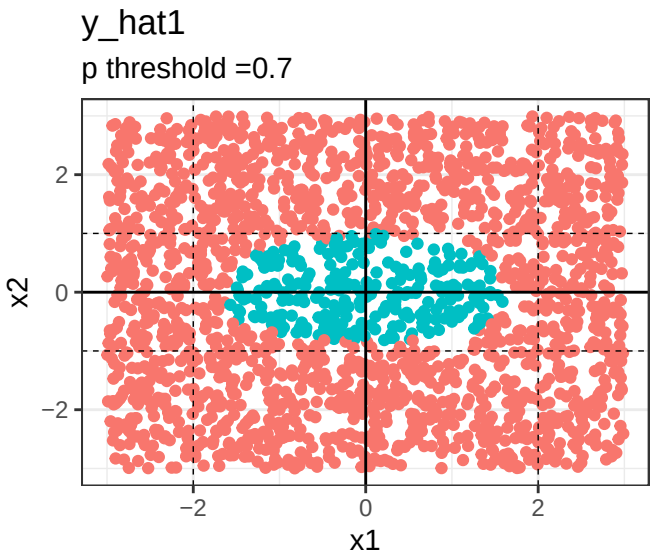

```
make_p_charts(0.6)
```

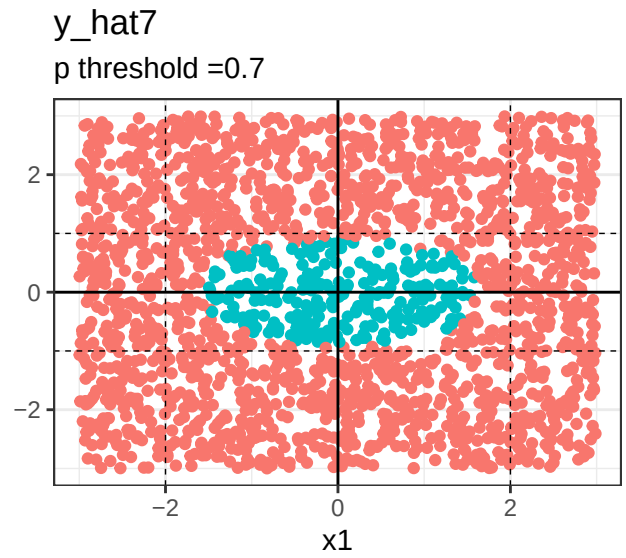


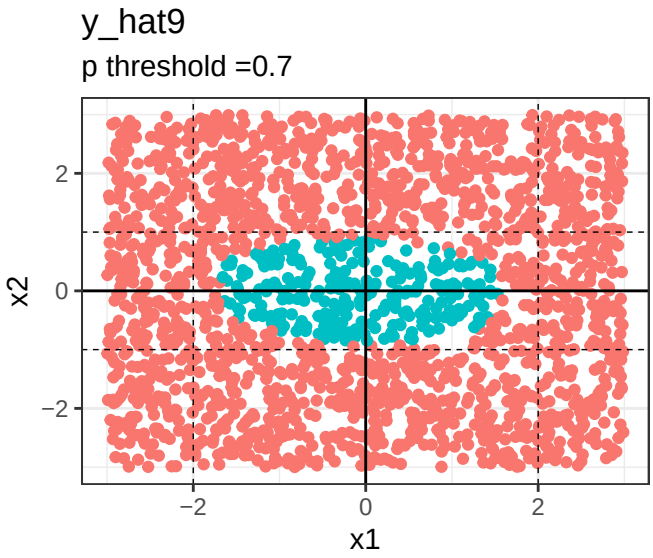





```
make_p_charts(0.7)
```







So at a lower p threshold we capture a pretty good ellipse with a boxy antenna for all three, and the ellipses are a little long. As we increase the p threshold, the ellipse rounds up better and the antenna narrows nicely, but if we increase the p-threshold too high we will lose the antenna. Overall, it looks like the best visualitzation occurs with $\hat{p}^{(7)}$ and $p = 0.6$