Machine Learning HW1 (Perceptron & Winnow)

Daniel Truver

2018/01/24

Perceptron Algorithm and Convergence Analysis

(1)

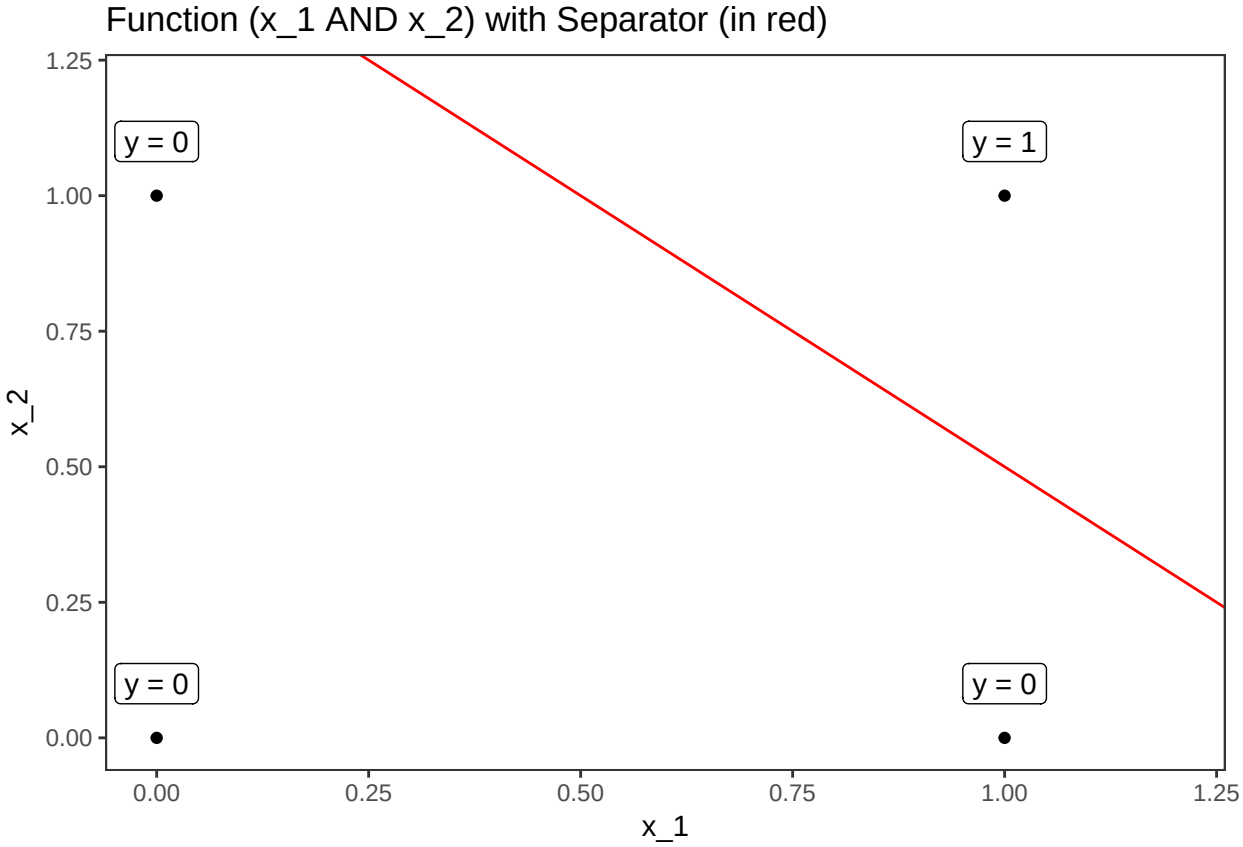
We consider Boolean functions

$$f : \{0,1\}^n \to \{0,1\}$$

(a) Consider

$$y = f(x_1, x_2) = x_1 \text{ AND } x_2 = x_1 \cdot x_2$$

Then, $y = 1$ iff $x_1 = x_2 = 1$. We can separate these points with a line in the plane.



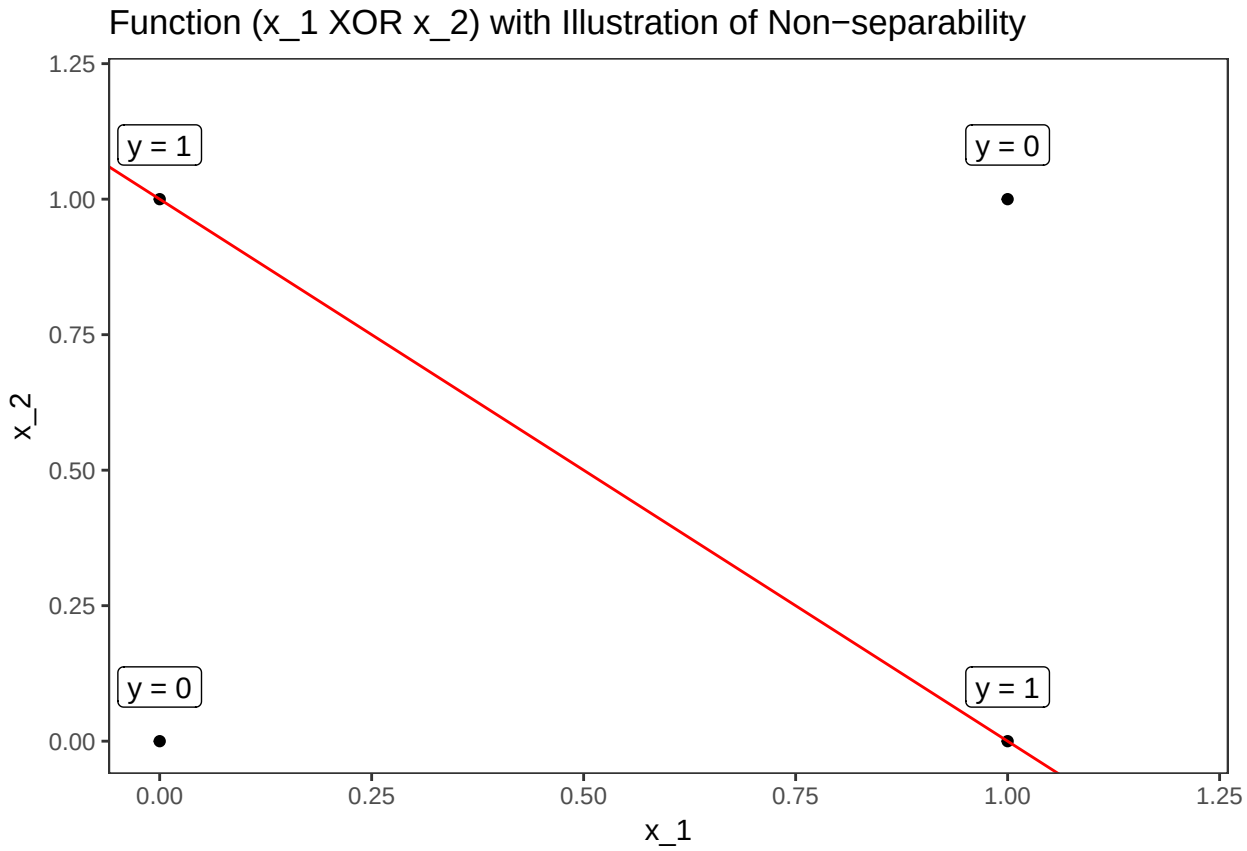
(b) Consider

$$y = f(x_1, x_2) = x_1 \text{ XOR } x_2 = x_1 + x_2 (\text{mod } 2)$$

Then, for

$$(x_1, x_2) \in \{0,1\} \times \{0,1\}, \quad y = 1 \iff x_1 \neq x_2.$$

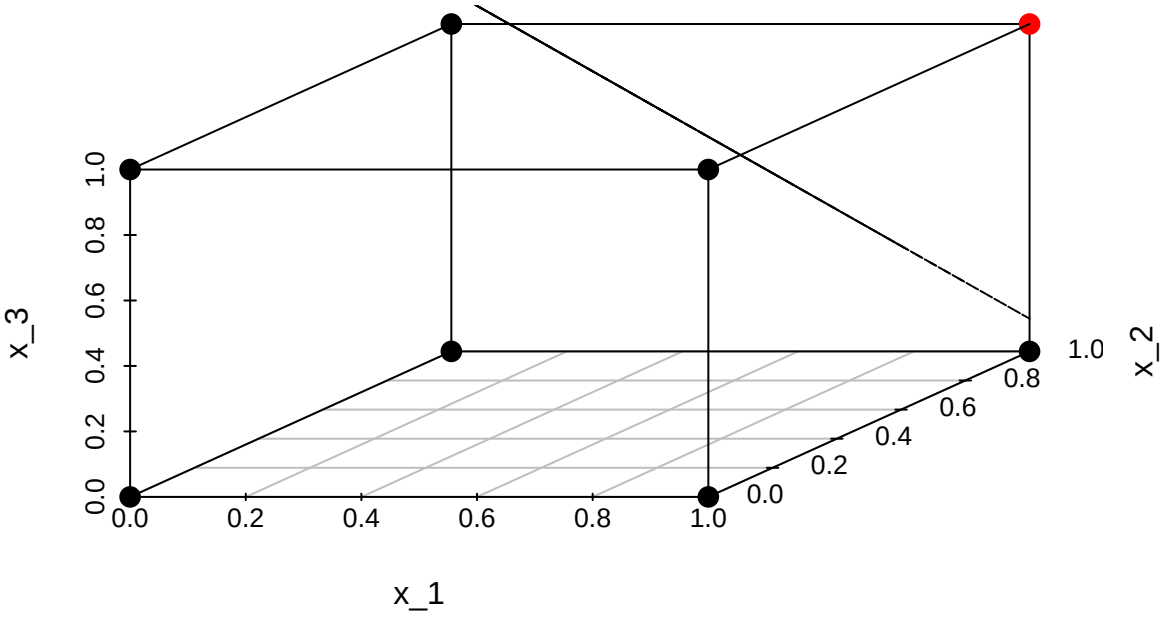
Graphically, we can see there is no separator for points where $y = 0$ and $y = 1$ since $(0, 1), (1, 0)$ are colinear with $(0, 0), (1, 1)$ on opposite sides of the shared line.



(c) Consider

$$y = f(x_1, x_2, x_3) = x_1 \text{ AND } x_2 \text{ AND } x_3 = x_1 \cdot x_2 \cdot x_3$$

Then, we have a case similar to (a) where $y = 1 \iff x_1 = x_2 = x_3 = 1$.



(2)

Since our classifier is $sign(f(x))$, the decision boundary is the hyperplane $0 = \beta_0 + \beta^T x = \beta_0 + \beta_1 x_1 + \cdots + \beta_p x_p$. From calculus, at least for me, we know the distance between a point x_0 and a hyperplane is the projection of $x_0 - x$, for x in the plane, onto the normal vector of the plane, here given by β. Let B denote the decision boundary, x denote a point on B, and x_0 denote an fixed point. The signed Euclidean distance is then:

$$\begin{aligned} D(x_0, B) &= proj_\beta(x_0 - x) \\ &= \beta \cdot (x_0 - x)/||\beta||_2 \\ &= (\beta^T x_0 - \beta^T x)/||\beta||_2 \\ &= (\beta_0 + \beta^T x_0)/||\beta||_2 \quad \text{since } x \text{ is a point on } B \\ &= f(x_0)/||\beta||_2 \end{aligned}$$

Multiplying both sides by y, since y is given for x in this problem, we get

$$yD(x, B) = yf(x)/||\beta||_2.$$

If x is misclassified, then the sign of y will differ from the sign of $f(x)$ and this value will be negative. Therefore, this quantity is the margin.

(3)

I feel like I'm missing something here.

The perceptron initializes at $w^{(0)} = 0$, so $||w^{(0)} - w^{sep}||_2^2 = ||w^{sep}||_2$. If we normalize w^{sep} by dividing by its magnitude and recaculate the margin, then we get

$$y_i(x_i \cdot w^{sep})/||w^{sep}||_2 \geq 1/||w^{sep}||$$

We have met the conditions of the perceptron convergence bound theorem proved in class. Note, even if $\exists i : ||x_i|| > 1$, rescaling would change the bound to

$$1/(||w^{sep}|| \cdot ||x_i||) < 1/||w^{sep}||,$$

so $1/||w^{sep}||$ is still an upper bound, and the theorem gives us

$$T < ||w^{sep}||^2,$$

what was to be shown.

Programming Assignment (using R)

Oh dear, data loading and data cleaning. Let's just do a quick Google search to see if anyone knows how to handle these file types in R.

```
# special thanks to: https://gist.github.com/brendano/39760
# Load the MNIST digit recognition dataset into R
# http://yann.lecun.com/exdb/mnist/
# assume you have all 4 files and gunzip'd them
# creates train$n, train$x, train$y  and test$n, test$x, test$y
# e.g. train$x is a 60000 x 784 matrix, each row is one digit (28x28)
# call:  show_digit(train$x[5,])   to see a digit.
# brendan o'connor - gist.github.com/39760 - anyall.org

load_mnist <- function() {
  load_image_file <- function(filename) {
    ret = list()
    f = file(filename,'rb')
    readBin(f,'integer',n=1,size=4,endian='big')
    ret$n = readBin(f,'integer',n=1,size=4,endian='big')
    nrow = readBin(f,'integer',n=1,size=4,endian='big')
    ncol = readBin(f,'integer',n=1,size=4,endian='big')
    x = readBin(f,'integer',n=ret$n*nrow*ncol,size=1,signed=F)
    ret$x = matrix(x, ncol=nrow*ncol, byrow=T)
    close(f)
    ret
  }
  load_label_file <- function(filename) {
    f = file(filename,'rb')
    readBin(f,'integer',n=1,size=4,endian='big')
    n = readBin(f,'integer',n=1,size=4,endian='big')
    y = readBin(f,'integer',n=n,size=1,signed=F)
    close(f)
    y
  }
  train <<- load_image_file('mnist/train-images-idx3-ubyte')
  test <<- load_image_file('mnist/t10k-images-idx3-ubyte')

  train$y <<- load_label_file('mnist/train-labels-idx1-ubyte')
  test$y <<- load_label_file('mnist/t10k-labels-idx1-ubyte')
}
load_mnist()
```

Now that we're done with the loading–dear god, 200MB of memory–we proceed with data cleaning.

```
# get observations of 4 or 9
keep_train = (train$y == 4 | train$y == 9)
keep_test = (test$y == 4 | test$y == 9)
# subset the data
x_train = train$x[keep_train,]
```

```
y_train = train$y[keep_train]

x_test = test$x[keep_test,]
y_test = test$y[keep_test]
# norm function
vnorm = function(x){sqrt(sum(x^2))}
train_norms = apply(x_train, 1, vnorm)
x_train = x_train/max(train_norms)
# recode the response so 9 is 1, 4 is -1
recode_1 = which(y_train == 9)
y_train[recode_1] = 1
y_train[-recode_1] = -1
n_train = length(y_train)
#testing set
test_norms = apply(x_test, 1, vnorm)
x_test = x_test/max(test_norms)
#recode test response
recode_1 = which(y_test == 9)
y_test[recode_1] = 1
y_test[-recode_1] = -1
n_test = length(y_test)
```

Now that the data is in the format we like, $||x_i||_2 \leq 1, y \in \{-1, 1\}$, we can get started on the perceptron.

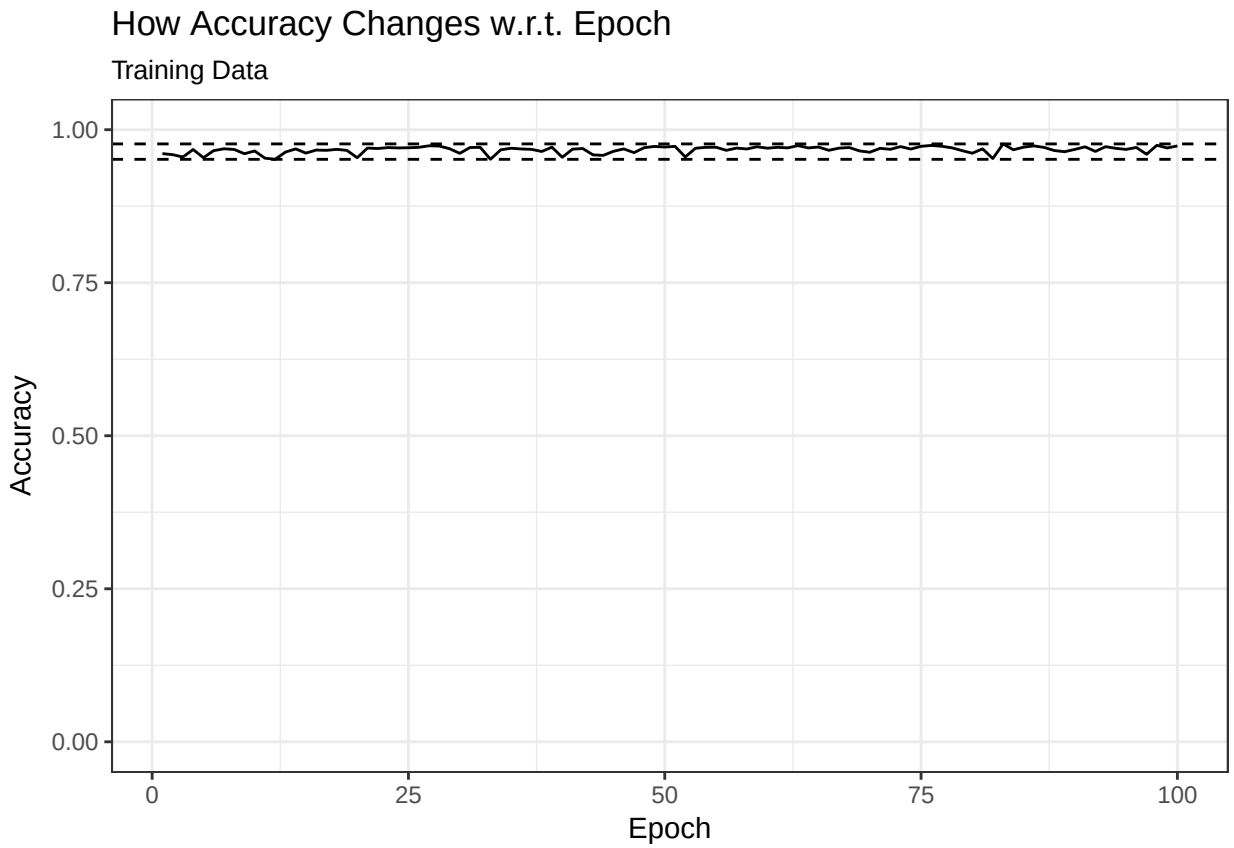
(1) Perceptron

```
perceptron = function(X, y, I){
  accuracy_epoch = data.frame(matrix(NA, nrow = I, ncol = 2)) # accuracy-epoch matrix
  colnames(accuracy_epoch) = c("accuracy", "epoch")
  W = matrix(NA, nrow = I, ncol = ncol(X))
  w = rep(0, dim(X)[2])
  for (j in 1:I){
    for (i in seq_along(y)){
      if (y[i]*(w %*% X[i,]) <= 0){
        w = w + y[i]*X[i,] # update step
      }
    }
    accuracy = sum((X %*% w) * y > 0)/length(y) # calculate propotion of correctly classified points
    accuracy_epoch[j, "accuracy"] = accuracy
    accuracy_epoch[j, "epoch"] = j
    W[j,] = w
    if (accuracy == 1){
      break # no need to continue if we have perfect separation
    }
  }
  return(list("a.e" = accuracy_epoch, "w" = w, "W" = W))
}
```

(a) Running the perceptron on the training set.

```
res_train = perceptron(X = x_train, y = y_train, I = 100)
```

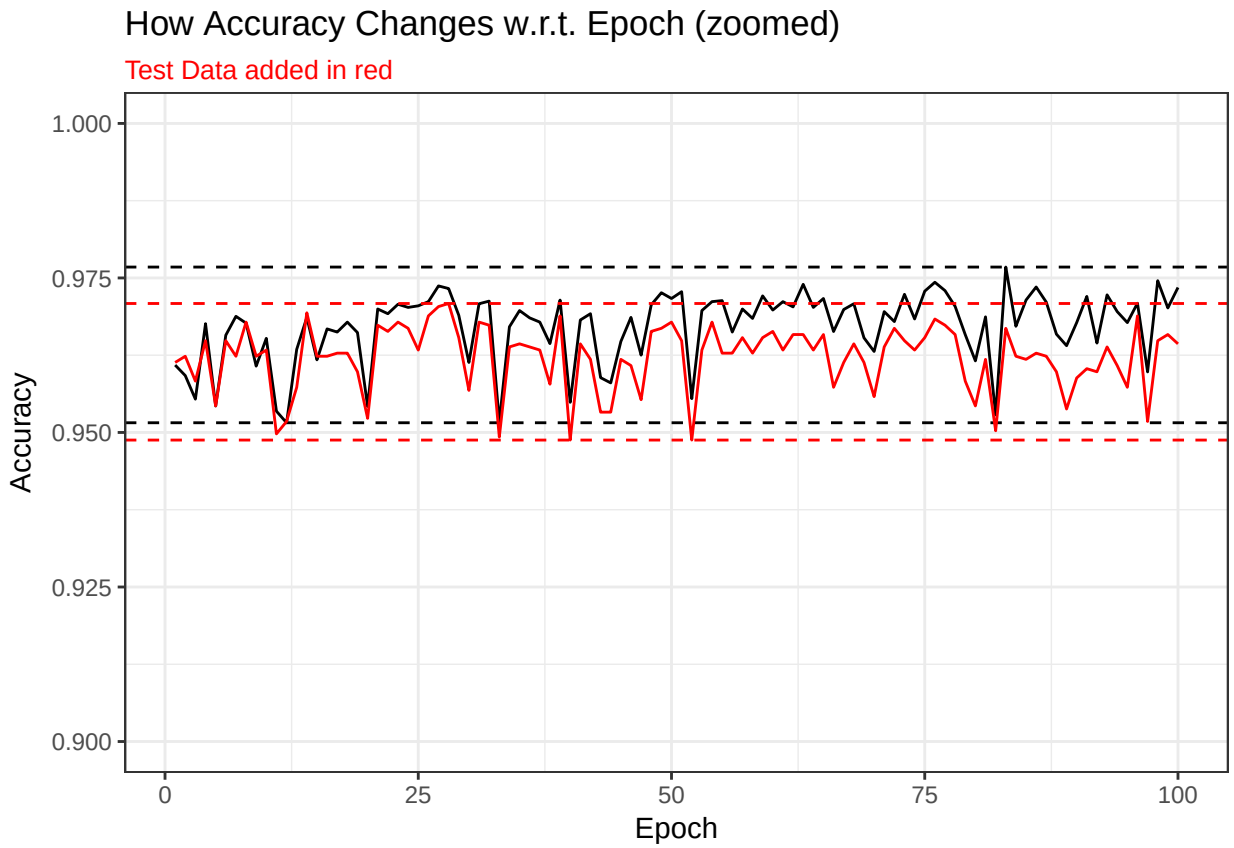
That was faster than expected, let's look at these delicious accuracy-epoch plots.



Suprisingly, at least to me, we see here that accuracy is quite high after 1 epoch, not monotonic, and stays close to 0.96. The actual range is [0.9516,0.9768]. It reaches its maximum value at epoch 83 and its minimum at epoch 12. From this plot, it does not seem that increasing or decreasing the maximum number of epochs will have much of an effect on the accuracy. Hmm. . . suspicious.

(b) Since this is the test set, and I'm usually a test set purist, I'm going to interpret the question to mean we take the values of w obtained from training the perceptron and see how well they classify the test set, without actually training a new perceptron on the test set. Well, what do you know? We conveniently kept a copy of w at each epoch.

```
W = res_train$W
n_epoch = 1:nrow(W)
accuracy_test = lapply(n_epoch,
                       function(j){
                         sum(y_test * (x_test %*% W[j,]) > 0)/n_test #check accuracy via dot product
                         }
                       )
a.eTest = data.frame("accuracy.test" = unlist(accuracy_test),
                     "epoch.test" = n_epoch)
```



The accuracy for the test data is generally lower than the accuracy for the training data, but the curves are very close; they tend to follow the same pattern of increasing and decreasing as well.

(c) The confusion matrix is how I refer to my brain in private.

```
w = res_train$w # last w obtained from training the perceptron
our.results = rep(NA, length(y_test))
# get results on test data
our.results[(x_test %*% w > 0)] = 1
our.results[(x_test %*% w <= 0)] = -1
# initialize confusion matix
confusion = data.frame(matrix(NA, nrow = 2, ncol = 2))
colnames(confusion) = c("actual.yes", "actual.no")
rownames(confusion) = c("predicted.yes", "predicted.no")
# fill in confusion matrix
confusion["predicted.yes", "actual.yes"] = sum(our.results == 1 & our.results == y_test)
confusion["predicted.yes", "actual.no"] = sum(our.results == 1 & our.results != y_test)
confusion["predicted.no", "actual.no"] = sum(our.results == -1 & our.results == y_test)
confusion["predicted.no", "actual.yes"] = sum(our.results == -1 & our.results != y_test)
nameTheColumns = c("Actual Yes, y = 1", "Actual No, y = -1")
knitr::kable(confusion,
             col.names = nameTheColumns,
             caption = "Confusion Matrix for Test Data")
```

Table 1: Confusion Matrix for Test Data

	Actual Yes, y = 1	Actual No, y = -1
predicted.yes	972	34

	Actual Yes, y = 1	Actual No, y = -1
predicted.no	37	948

The accuracy after the last epoch is 0.9643395.

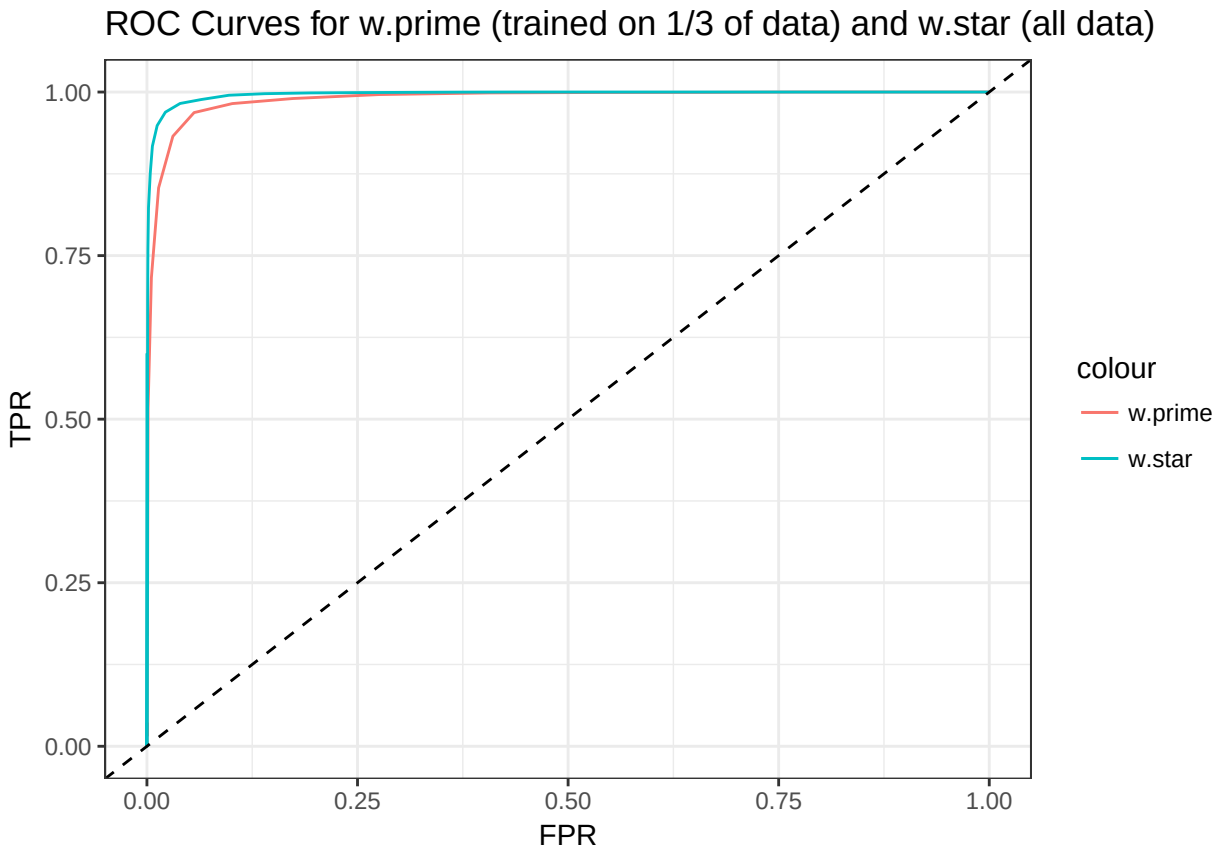
(d)

```
# instead of modifying the function and risk breaking it...
# we'll just extract the first third of the training data and run 1 epoch
third.train = floor(n_train/3)
x_train.third = x_train[1:third.train,]
y_train.third = y_train[1:third.train]
res_third = perceptron(x_train.third, y_train.third, I = 1)
w.prime = res_third$w
res_train = perceptron(x_train, y_train, I = 100)
w.star = res_train$w
```

```
#check n_b values for the threshold b
n_b = 100
ROC.third.matrix = data.frame(matrix(NA, nrow = n_b, ncol = 2))
colnames(ROC.third.matrix) = c("TPR", "FPR")
ROC.full.matrix = data.frame(matrix(NA, nrow = n_b, ncol = 2))
colnames(ROC.full.matrix) = c("TPR", "FPR")
i = 1
for (b in seq(-5, 5, length.out = n_b)){
  results.third = sign(x_train %*% w.prime - b)
  results.full = sign(x_train %*% w.star - b)
  # get the confusion matrix entries
  TP.full = sum(results.full == 1 & results.full == y_train)
  FP.full = sum(results.full == 1 & results.full != y_train)
  FN.full = sum(results.full == -1 & results.full != y_train)
  TN.full = sum(results.full == -1 & results.full == y_train)

  TP.third = sum(results.third == 1 & results.third == y_train)
  FP.third = sum(results.third == 1 & results.third != y_train)
  FN.third = sum(results.third == -1 & results.third != y_train)
  TN.third = sum(results.third == -1 & results.third == y_train)
  #store in matrix for later plotting
  ROC.full.matrix[i,"TPR"] = TPR.full = TP.full/(TP.full + FN.full)
  ROC.full.matrix[i,"FPR"] = FPR.full = FP.full/(FP.full + TN.full)

  ROC.third.matrix[i,"TPR"] = TPR.third = TP.third/(TP.third + FN.third)
  ROC.third.matrix[i,"FPR"] = FPR.third = FP.third/(FP.third + TN.third)
  i = i + 1
}
```



We see from the graph, that the ROC curve for w^* is always greater than or equal to the curve for w'. Therefore, if we use use w^*, we have to accept fewer false positives to reach a desired amount of true positives.

(e) Trapezoids with uneven intervals it is.

```
N = nrow(ROC.full.matrix) - 1
trapezoids.full = lapply(2:N, # get areas of all our trapezoids
                         function(i){
                           (ROC.full.matrix[i-1,"FPR"] - ROC.full.matrix[i,"FPR"])*
                             (ROC.full.matrix[i-1,"TPR"] + ROC.full.matrix[i, "TPR"])/2
                         })
AUC.star = sum(unlist(trapezoids.full))
trapezoids.third = lapply(2:N, # get areas of all our trapezoids
                         function(i){
                           (ROC.third.matrix[i-1,"FPR"] - ROC.third.matrix[i,"FPR"])*
                             (ROC.third.matrix[i-1,"TPR"] + ROC.third.matrix[i, "TPR"])/2
                         })
AUC.prime = sum(unlist(trapezoids.third))
```

AUC for w^* is 0.9967.
AUC for w' is 0.9892.

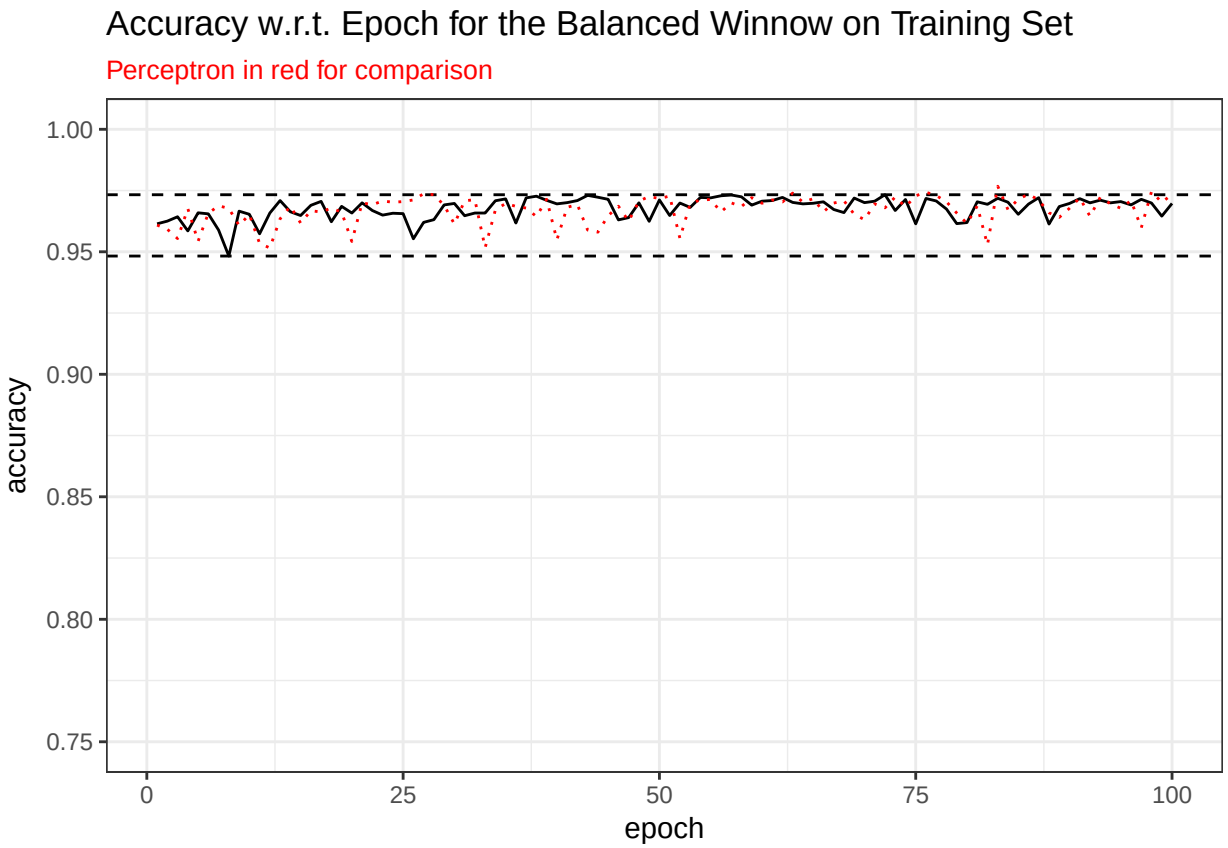
These results are in line with the ROC curves we obtained in the previous section: w^* has better performance than w'. They are very close, however, suggesting that the perceptron is converging quickly to its final state.

(2) Balanced Winnow

```
balanced_winnow = function(X, y, I, eta = 1){
  p = ncol(X)
  accuracy_epoch = data.frame(matrix(NA, nrow = I, ncol = 2))
  colnames(accuracy_epoch) = c("accuracy", "epoch")
  W_n = matrix(NA, nrow = I, ncol = ncol(X))
  W_p = matrix(NA, nrow = I, ncol = ncol(X))
  w_p = w_n = rep(.5/p, p)
  for (j in 1:I){
    for (i in 1:nrow(X)) {
      if (( y[i] * (w_p %*% X[i,] - w_n %*% X[i,]) ) <= 0 ){
        w_p = w_p * exp(eta*y[i]*X[i,])
        w_n = w_n * exp(-eta*y[i]*X[i,])
        s = sum(w_n + w_p)
        w_p = w_p/s
        w_n = w_n/s
      }
    }
    accuracy = sum((X %*% (w_p - w_n)) * y > 0)/length(y) # calculate propotion of correctly classified
    accuracy_epoch[j, "accuracy"] = accuracy
    accuracy_epoch[j, "epoch"] = j
    W_p[j,] = w_p
    W_n[j,] = w_n
  }
  return(list("a.e" = accuracy_epoch, "w_p" = w_p,
              "w_n" = w_n, "W_n" = W_n, "W_p" = W_p))
}
```

(a) Oh boy, another function, I can't wait to run it.

```
confusion.matrix = function(predicted, actual){ # I finally get smart and write a function for this
  conf = data.frame(matrix(NA, 2, 2))
  colnames(conf) = c("actual.yes", "actual.no")
  rownames(conf) = c("predicted.yes", "predicted.no")
  conf["predicted.yes", "actual.yes"] = sum(predicted == 1 & predicted == actual)
  conf["predicted.yes", "actual.no"] = sum(predicted == 1 & predicted != actual)
  conf["predicted.no", "actual.yes"] = sum(predicted == -1 & predicted != actual)
  conf["predicted.no", "actual.no"] = sum(predicted == -1 & predicted == actual)
  return(conf)
}
res_train.winnow = balanced_winnow(x_train, y_train, I = 100)
w_p = res_train.winnow$w_p
w_n = res_train.winnow$w_n
res_test = sign(x_test %*% (w_p - w_n))
accuracy_test = sum(y_test * res_test > 0)/n_test
conf.test = confusion.matrix(predicted = res_test, actual = y_test)
```



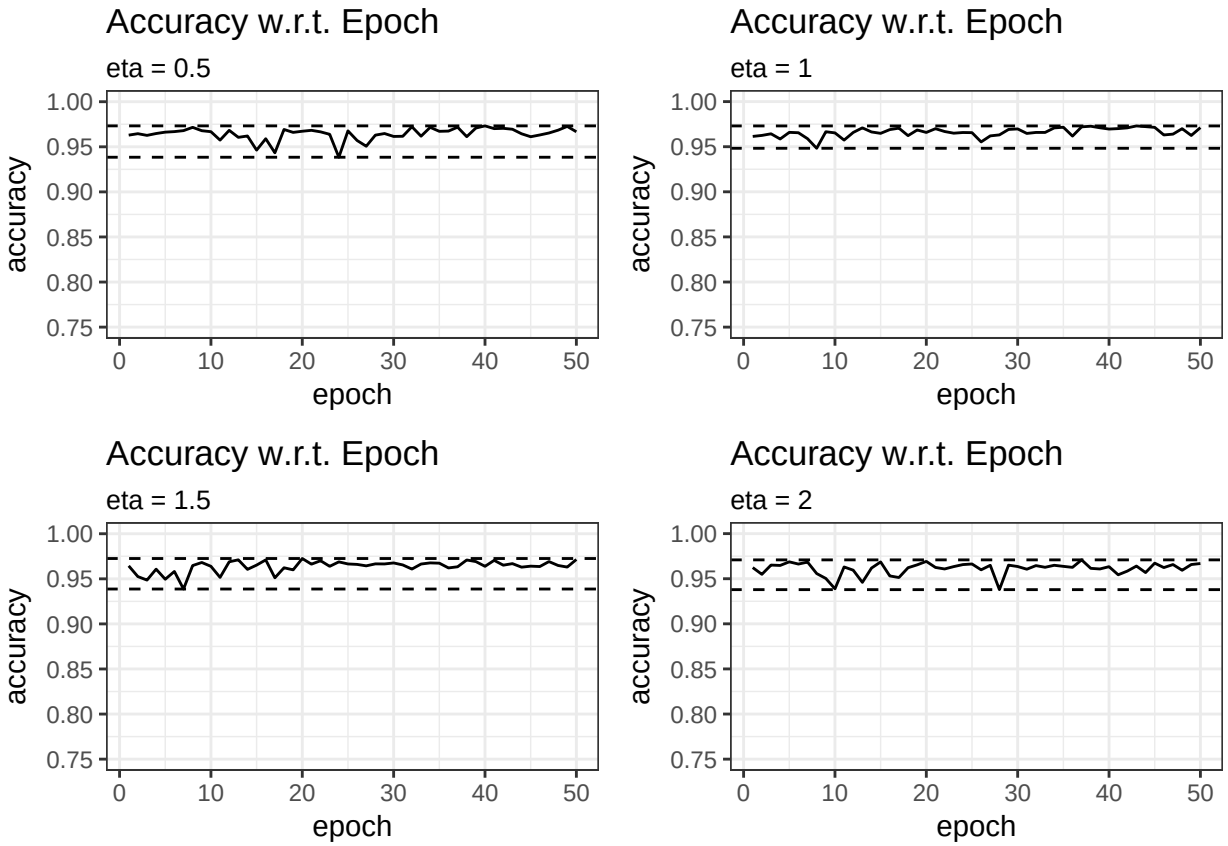

```
## Accuracy for test data: 0.9638
```

Table 2: Confusion Matrix for Balanced Winnow (test data)

	Actual Yes	Actual No
predicted.yes	957	20
predicted.no	52	962

(b) Well, the only way I know of to test different values of the parameter η is to make a list of possible values and throw them all into the function to see which one gives the best accuracy at the end. Known colloquially as the "head meet wall" method.

```
ETA = c(0.5, 1, 1.5, 2)
plot.list = list()
for (eta in ETA){
  res = balanced_winnow(x_train, y_train, 50, eta) # 50 epochs for the sake of time
  g = ggplot(data = res$a.e, aes(x = epoch, y = accuracy)) +
    geom_line() +
    geom_hline(yintercept = min(res$a.e$accuracy), lty = "dashed") +
    geom_hline(yintercept = max(res$a.e$accuracy), lty = "dashed") +
    ggtitle("Accuracy w.r.t. Epoch",
            subtitle = paste0("eta = ", eta)) +
    theme_bw() +
    ylim(.75,1)
  plot.list[[which(ETA == eta)]] = g
}
```



There do not seem to be many differences in the accuracy between these values of η. The optimal η is probably $1/e$ because when in doubt, that always seems to be a reasonable guess. Or something equally as obscure and unexpected.