Logistic Regression

A review of week 2, lecture 6 in R

Further Reading

This is a good time to point out that ISLR does not teach a lot of these topics the same way as the Andrew Ng course. They really take some time to explain how to use logistic regression in the real world. Andrew Ng, however, just goes over the mechanics because he's just preparing us for neural networks.

They also do not touch gradient descent at all. Ng shows us gradient descent because, again, it's all to better understand neural networks.

- ISLR videos: Introduction to Classification
- ISLR videos: Logistic Regression and Maximum Likelihood
- ISLR videos: Multivariate Logistic Regression and Confounding
- ISLR videos: Case-Control Sampling and Multiclass Logistic Regression
- ISLR book section 4.3: Logistic Regression (4.1-4.2 introduce classification)

Working with Classification Data

Before going into examples, the best thing is to make sure we have a good handle on our dataset. We'll be using the `iris` dataset again, but as you can see below, the species are in some kind of "factor" format.

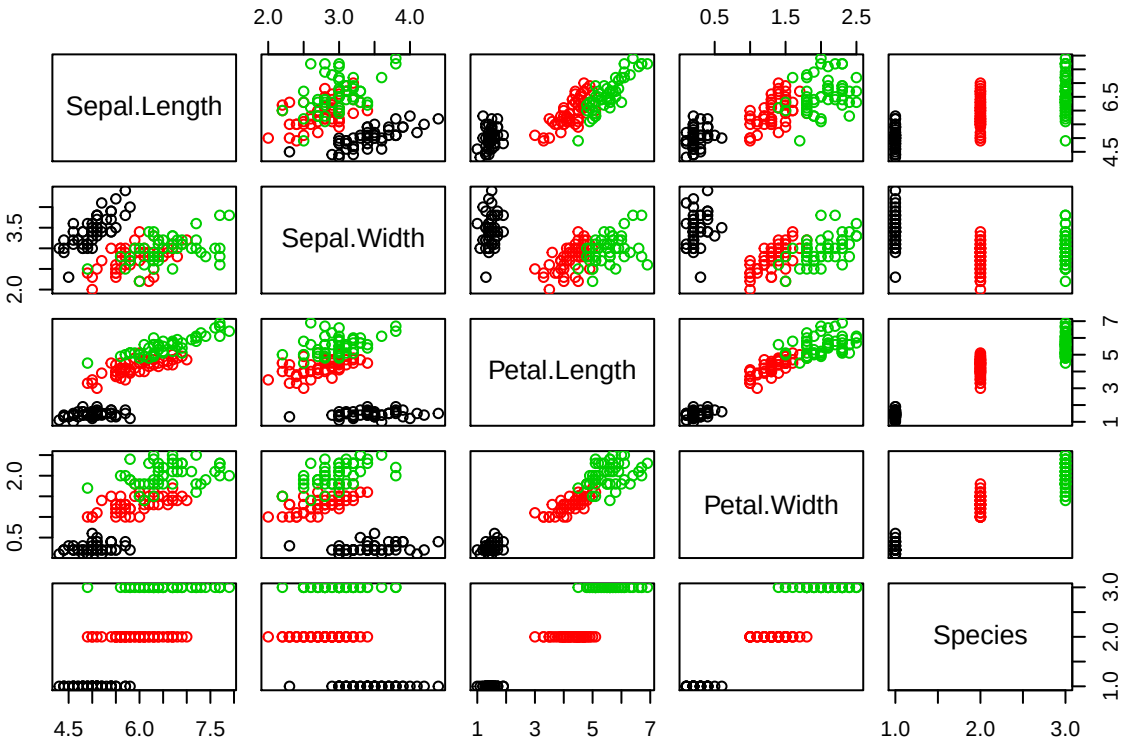
```
# Some data exploration
str(iris) # Properties of the columns
```

```
## 'data.frame':    150 obs. of  5 variables:
##  $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
##  $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
##  $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
##  $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
##  $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...
```

```
head(iris) # Display the first 5 rows of the data
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

```
plot(iris, col=iris$Species) # A plot matrix of the data, with color
```



The classes, the species, are factors rather than 1s and 0s. R factors are a way for R to store character strings as integers but convert them back into text for display, using a look-up table.

There's a very clever solution available to fix this in R, often shown on Stack Overflow. Many models in R are able to convert factors into dummy variables for their own immediate use. The `model.matrix()` function performs this step but then just returns the processed data as-is without running a model.

The ~ equation is an R "formula". It's just a syntax for econometricians and statisticians to write models in a familiar format. (If we had wanted a dependent y variable, we'd have placed it to the left of the tilde ~ character.)

`model.matrix()` is great for getting dummy variables, but `glm()` only accepts data frames. The data must be converted back into a data frame.

```
# Use a formula to get a model-ready matrix, but then convert back to a data frame
dummy_iris <- data.frame(
    model.matrix(~ Sepal.Length + Sepal.Width + Petal.Length + Petal.Width + Species - 1, iris)
)
str(dummy_iris) # So you can see what things look like now. A lot of extra info has been added
```

```
## 'data.frame':    150 obs. of  7 variables:
##  $ Sepal.Length     : num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
##  $ Sepal.Width      : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
##  $ Petal.Length     : num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
##  $ Petal.Width      : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
##  $ Speciessetosa    : num  1 1 1 1 1 1 1 1 1 1 ...
##  $ Speciesversicolor: num  0 0 0 0 0 0 0 0 0 0 ...
##  $ Speciesvirginica : num  0 0 0 0 0 0 0 0 0 0 ...
```

Logistic Regression the R Way

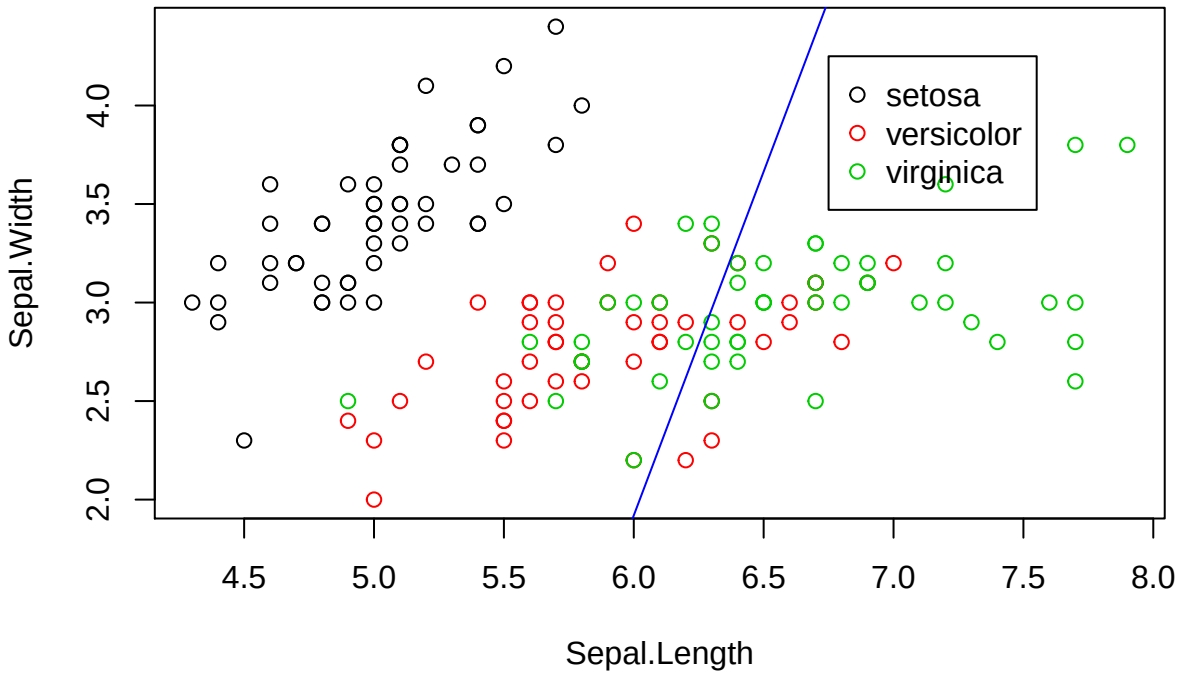
Before getting into gradient descent, this is how you'd perform a logistic regression the "normal" way.

```r
logit <- glm(Speciesvirginica ~ Sepal.Length + Sepal.Width, binomial, dummy_iris)
summary(logit)
```

```
##
## Call:
## glm(formula = Speciesvirginica ~ Sepal.Length + Sepal.Width,
##     family = binomial, data = dummy_iris)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -1.9098  -0.5901  -0.2182   0.6181   2.5815
##
## Coefficients:
##              Estimate Std. Error z value Pr(>|z|)
## (Intercept)  -14.1836     3.1172  -4.550 5.36e-06 ***
## Sepal.Length   2.6025     0.4406   5.907 3.48e-09 ***
## Sepal.Width   -0.7458     0.6413  -1.163    0.245
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 190.95  on 149  degrees of freedom
## Residual deviance: 115.90  on 147  degrees of freedom
## AIC: 121.9
##
## Number of Fisher Scoring iterations: 6
```

As you can see, I manually calculate the decision boundary. I could not find an easy way to draw a standard equation in base R.

```r
plot(Sepal.Width ~Sepal.Length, iris, col=iris$Species) # Plot the original table
legend(6.75, 4.25, legend=unique(iris$Species), col=1:length(iris$Species), pch=1) # Legend
# Plot the decision boundary
abline(
    -logit$coefficients[1]/logit$coefficients[3],
    -logit$coefficients[2]/logit$coefficients[3],
    col="blue"
    )
```



Feature Rescaling Redux

We have to rescale/normalize our data since Ng has again indicated that this is necessary for convergence. Here is another way of normalizing data, forcing it between -1 and 1.

Subtract the minimum from the values then divide these new values by their maximum. This gets the data between 0 and 1, and you may want to stop here. To get the data between -1 and 1, simply multiply by 2 then subtract 1.

This is good time to show that you can start a new line in R code as long as some operator is left at the end of the line before. Arithmetic operators and commas all work.

```
# Make a copy of our dummied data table as a matrix for gradient descent
iris_nom <- data.matrix(dummy_iris)
for(i in 1:4){
    iris_nom[,i] <- 2*(iris_nom[,i]-min(iris_nom[,i])) /
        max(iris_nom[,i]-min(iris_nom[,i]))-1
}
# The 1L is not necessary. 1 works fine too. R programmers have arbitrary conventions. :-)
iris_nom <- cbind(1L, iris_nom) # Add intercept
```

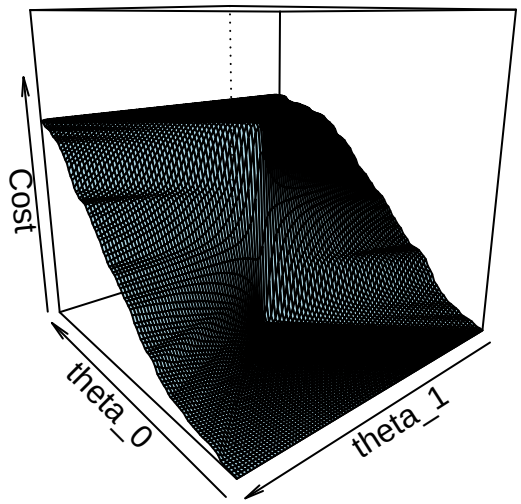
Short Detour: Logistic Regression with Quadratic Cost

Professor Ng mentions that the quadratic function below is not suitable for logistic regresion because it is not convex.

$$J(\theta_0, \theta_1) = \frac{1}{2}(\sigma(X^T\theta) - y^{(i)})^2$$

Here is the best I could do to demonstrate this with graphs. To better see the 3D graph, you have to run the code in R and zoom in.

```
x <- seq(-100, 100, length=100)
y <- seq(-100, 100, length=100)
# Functions are objects in R, so you can just enter a function definition as an argument.
# Nothing stops you from defining another function within this function either.
z <- outer(x, y, function(x,y){
    sigmoid <- function(input){
        return(1/(1+exp(-input)))
    }
    weights <- cbind(x, y)
    return(colMeans(0.5*(sigmoid(tcrossprod(as.matrix(iris_nom[,1:2]), weights) - iris_nom[,8])^2)))
}
)
persp(x, y, z, theta = 230, phi = 15, zlim=c(0, 0.75), shade=0.05,
    col="lightblue", xlab = "theta_0", ylab = "theta_1", zlab = "Cost")
```

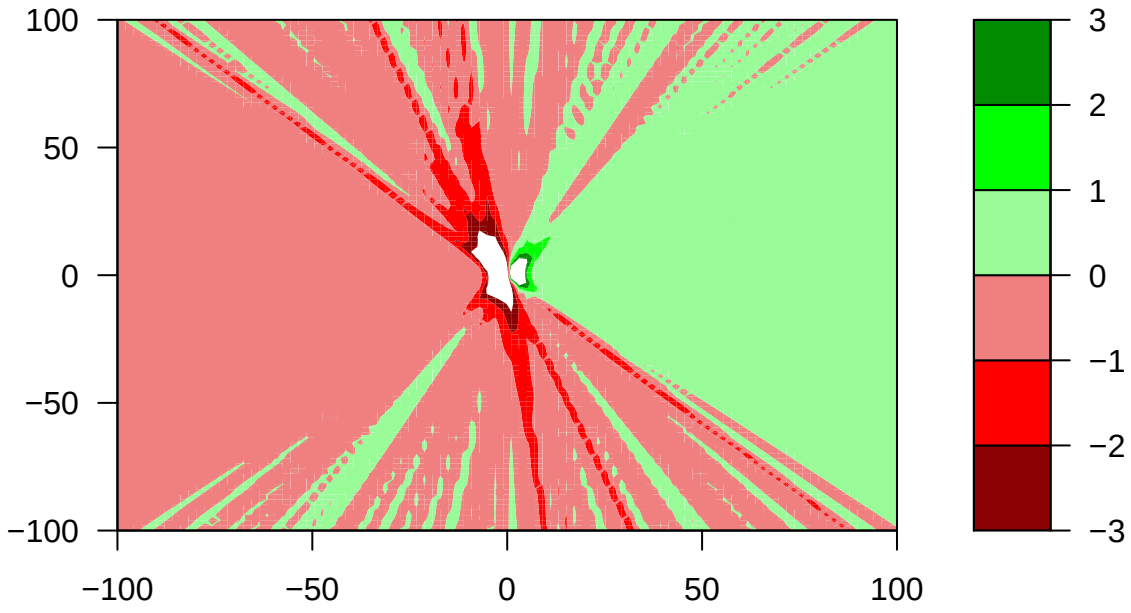


A less obvious demonstration is done by graphing the cost function's gradient. Where green and red meet there will be zero slopes, which are local minimums our gradient descent algorithm can get stuck in.

I have tried running gradient descent with the MSE/quadratic cost function, and it did not converge.

```
x <- seq(-100, 100, length=100)
y <- seq(-100, 100, length=100)
# Again, I define functions in situ
z <- outer(x, y, function(x,y){
    sigmoid <- function(input){
        return(1/(1+exp(-input)))
    }
    s_prime <- function(input){
        return(sigmoid(input) * (1 - sigmoid(input)))
    }
    weights <- cbind(x, y) # Bind the two function inputs into a matrix, for matrix multiplication
    activation <- tcrossprod(iris_nom[,1:2], weights)
    return(colMeans(crossprod(iris_nom[,1:2], (activation - iris_nom[,8]) * s_prime(activation))))
}
)
# All the colors you could possibly want in R:
# http://www.stat.columbia.edu/~tzheng/files/Rcolor.pdf
filled.contour(x, y, z,
```

```
    levels=c(-3, -2, -1, 0, 1, 2, 3),
    col=c("red4", "red1", "lightcoral", "palegreen", "green1", "green4"))
```



Gradient Descent with Negative Log-likelihood Cost Function

Here is the negative log-likelihood cost function.

$$J(\theta) = -\frac{1}{m}\left[\sum_{i=1}^{m} y^{(i)} log(h_\theta(x^{(i)})) + (1 - y^{(i)}) log(1 - h_\theta(x^{(i)}))\right]$$

Although the equation looks complicated, it has the bonus of eliminating a few terms in the update function. So that it is once again only

$$\frac{\partial J(\theta)}{\partial \theta} = X^T \Delta$$

```
initial_theta <- matrix(rnorm(3), nrow=3) # This creates a vector of three rows
training_matrix <- iris_nom[,c(1, 2, 3, 8)] # Only select the columns we want
# col 1: intercept
# col 2: Sepal.Length
# col 3: Sepal.Width
# col 4: Speciesvirginica (output)

# The sigmoid function
sigmoid <- function(input){
    return(1/(1+exp(-input)))
}

# Negative log-likelihood cost (now fully vectorized)
log_cost <- function(theta){
    activation <- training_matrix[,1:3] %*% theta
    return(-colMeans(training_matrix[,4]*log(sigmoid(activation)) + (1-training_matrix[,4])*log(1 - sig
}
```

```
# A gradient descent function
gradient_descent_log <- function(learning_rate, number_of_epochs){
    # Set the initial parameters
    theta <- initial_theta
    # Start a history
    theta_history <- theta
    for(epoch in 1:number_of_epochs){
        delta <- sigmoid(training_matrix[,1:3] %*% theta) - training_matrix[,4]
        theta <- theta - learning_rate/nrow(training_matrix)*crossprod(training_matrix[,1:3], delta)
        theta_history <- c(theta_history, theta)
    }
    # The theta history is a long vector so it needs to be made into a matrix
    theta_history <- matrix(theta_history, ncol=3, byrow=TRUE)
    # Since the cost function is vectorized, it is much faster
    # Transpose the_history (thetas as rows, epochs as cols) to calculate cost all at once!
    return(cbind(theta_history, log_cost(t(theta_history))))
}
# Run the algorithm and save the history and this time time ourselves
start_timing <- proc.time()[3]
history_log <- gradient_descent_log(1.0, 250)
print(sprintf("Training took %f seconds.", proc.time()[3] - start_timing))
```

```
## [1] "Training took 0.026000 seconds."
```

Gradient descent gives results that are very close to the GLM method. You can get the values much closer if you let the code run for more than a thousand epochs.

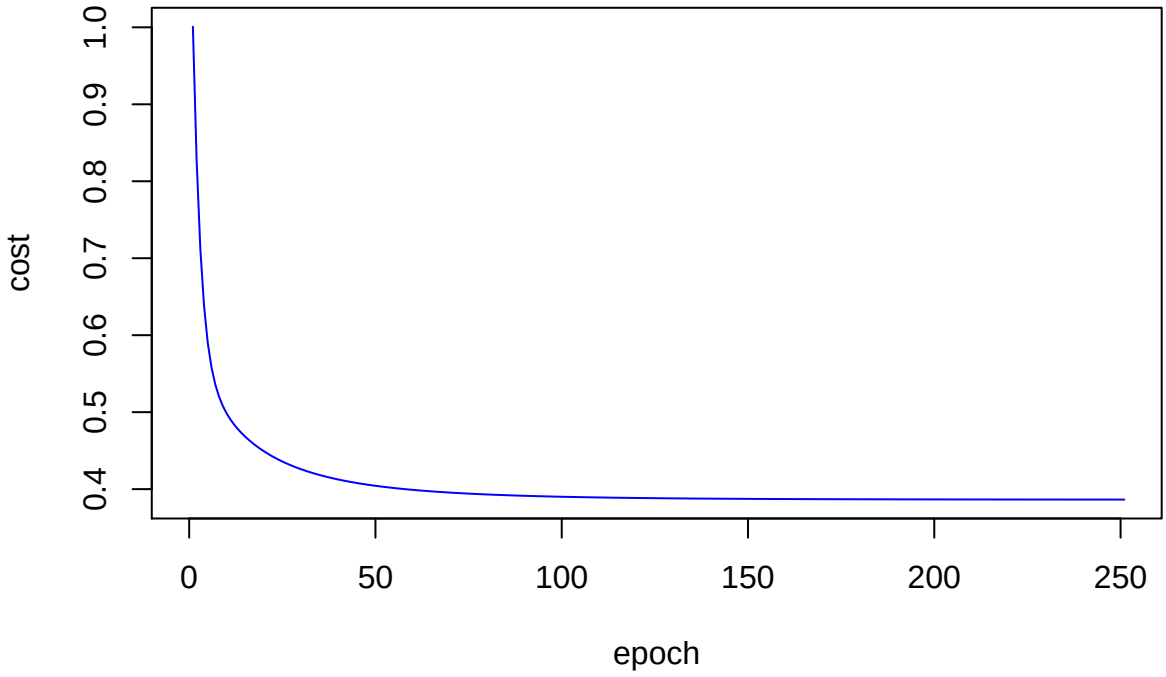
```
logit <- glm(Speciesvirginica ~ Sepal.Length + Sepal.Width, binomial, data.frame(iris_nom))
print(logit$coefficients)
```

```
##  (Intercept) Sepal.Length  Sepal.Width
##   -0.6946827    4.6845551   -0.8949422
```

```
tail(history_log, 1)[,1:3]
```

```
## [1] -0.6888236  4.5539989 -0.8883386
```

```
plot(1, type="n", xlab="epoch", ylab="cost",
    ylim=c(
        min(history_log[,4]),
        max(history_log[,4])
    ),
    xlim=c(0, nrow(history_log)))
lines(x=1:nrow(history_log), y=history_log[,4], col="blue")
```



Advanced Optimization

R also has access to the advanced optimization algorithms mentioned at the end of the lecture. As you can see below, `optim()` runs faster than our own gradient descent.

I named the function arguments to show that `optim` takes initial parameter values, a cost function, and a gradient function.

```
start_timing <- proc.time()[3]
optim(
    par=matrix(rnorm(3), nrow=3),
    fn=log_cost,
    gr=function(theta) crossprod(training_matrix[,1:3],
        sigmoid(training_matrix[,1:3] %*% theta) - training_matrix[,4]),
    method="BFGS"
)
```

```
## $par
##            [,1]
## [1,] -0.6946653
## [2,]  4.6847299
## [3,] -0.8948248
##
## $value
## [1] 0.3863304
##
## $counts
## function gradient
##       14        8
##
## $convergence
## [1] 0
##
```

```
## $message
## NULL
```

```
print(sprintf("Training took %f seconds.", proc.time()[3] - start_timing))
```

```
## [1] "Training took 0.005000 seconds."
```