Machine Learning

Rob Hayward

June 28, 2014

Introduction

These are the notes from the Peng Machine Learning course.

Introduction to the methods and model representations. This is an example of *supervised learning*. There is a training set that is used to develop a model. m is the number of training examples, x are the input variables or features, y are the output variables (x, y) is a training example, $(x^{(i)}), y^{(i)}$ is a training set.

The aim is to get the parameters of a hypothesis. The hypothesis is

$$h_\theta(x) = \theta_0 + \theta_1 x \tag{1}$$

This is a linear regression model. The aim is to chose the values of the parameters θ_0 and θ_1 to minimise the difference between actual and forecast values.

Objective function

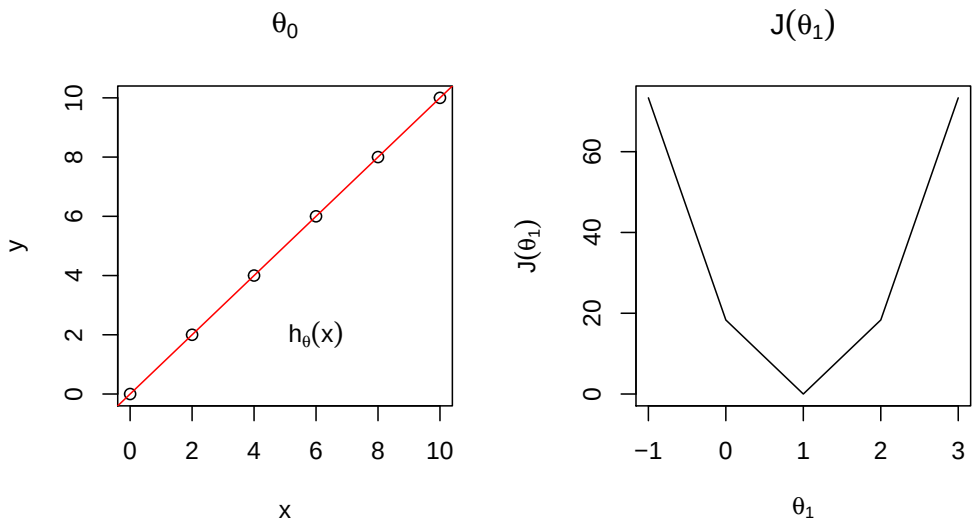
$$\text{minimise}_{\theta_0,\theta_1)} \text{ for} \frac{1}{2m} \sum (h_0(x) - y)^2 \tag{2}$$

The cost function

$$J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^{m} (h_0 x^{(i)} - y^{(i)})^2 \tag{3}$$

This is also called the *squared error function*

A simplified version can just look at one of the parameters (θ_1). This would be a line through the origin. There are two functions: $h_\theta(x)$ is a function of x; $J(\theta_1)$ is a function of θ_1.



Clearly the value of θ_1 that will minimise the cost function is one.

Now we have the same case with both papameters changing. This can be shown using *contour plots* or *contour figures*.

Econometric Sense

```
x0 <- c(1, 1, 1, 1, 1)  # column of 1's
x1 <- c(1, 2, 3, 4, 5)  # original x-values
# create the x- matrix of explanatory variables
x <- as.matrix(cbind(x0, x1))
# create the y-matrix of dependent variables
y <- as.matrix(c(3, 7, 5, 11, 14))
m <- nrow(y)
# implement feature scaling
x.scaled <- x
x.scaled[, 2] <- (x[, 2] - mean(x[, 2]))/sd(x[, 2])
# analytical results with matrix algebra
solve(t(x) %*% x) %*% t(x) %*% y  # w/o feature scaling
```

```
##    [,1]
## x0  0.2
## x1  2.6
```

```
solve(t(x.scaled) %*% x.scaled) %*% t(x.scaled) %*% y  # w/ feature scaling
```

```
##     [,1]
## x0 8.000
## x1 4.111
```

```r
# results using canned lm function match results above
summary(lm(y ~ x[, 2]))  # w/o feature scaling
```

```
##
## Call:
## lm(formula = y ~ x[, 2])
##
## Residuals:
##    1    2    3    4    5
##  0.2  1.6 -3.0  0.4  0.8
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)    0.200      2.132    0.09    0.931
## x[, 2]         2.600      0.643    4.04    0.027 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 2.03 on 3 degrees of freedom
## Multiple R-squared:  0.845,Adjusted R-squared:  0.793
## F-statistic: 16.4 on 1 and 3 DF,  p-value: 0.0272
```

```r
summary(lm(y ~ x.scaled[, 2]))  # w/feature scaling
```

```
##
## Call:
## lm(formula = y ~ x.scaled[, 2])
##
## Residuals:
##    1    2    3    4    5
##  0.2  1.6 -3.0  0.4  0.8
##
## Coefficients:
##               Estimate Std. Error t value Pr(>|t|)
## (Intercept)      8.000      0.909    8.80   0.0031 **
## x.scaled[, 2]    4.111      1.017    4.04   0.0272 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 2.03 on 3 degrees of freedom
## Multiple R-squared:  0.845,Adjusted R-squared:  0.793
## F-statistic: 16.4 on 1 and 3 DF,  p-value: 0.0272
```

```r
# define the gradient function dJ/dtheata: 1/m * (h(x)-y))*x where h(x) =
# x*theta in matrix form this is as follows:
grad <- function(x, y, theta) {
    gradient <- (1/m) * (t(x) %*% ((x %*% t(theta)) - y))
    return(t(gradient))
}
# define gradient descent update algorithm
grad.descent <- function(x, maxit) {
    theta <- matrix(c(0, 0), nrow = 1)  # Initialize the parameters
    alpha = 0.05  # set learning rate
    for (i in 1:maxit) {
        theta <- theta - alpha * grad(x, y, theta)
    }
    return(theta)
}
# results without feature scaling
print(grad.descent(x, 1000))
```

```
##          x0  x1
## [1,] 0.2001 2.6
```

```r
# results with feature scaling
print(grad.descent(x.scaled, 1000))
```

```
##      x0    x1
## [1,]  8 4.111
```

```r
# ------------------------------------------------------------------------
# cost and convergence intuition
# ------------------------------------------------------------------------
# typically we would iterate the algorithm above until the change in the
# cost function (as a result of the updated b0 and b1 values) was extremely
# small value 'c'. C would be referred to as the set 'convergence' criteria.
# If C is not met after a given # of iterations, you can increase the
# iterations or change the learning rate 'alpha' to speed up convergence
# get results from gradient descent
beta <- grad.descent(x, 1000)
# define the 'hypothesis function'
h <- function(xi, b0, b1) {
    b0 + b1 * xi
}
```

```r
# define the cost function
cost <- t(mat.or.vec(1, m))
for (i in 1:m) {
    cost[i, 1] <- (1/(2 * m)) * (h(x[i, 2], beta[1, 1], beta[1, 2]) - y[i, ])^2
}
totalCost <- colSums(cost)
print(totalCost)

## [1] 1.24

# save this as Cost1000
cost1000 <- totalCost
# change iterations to 1001 and compute cost1001
beta <- (grad.descent(x, 1001))
cost <- t(mat.or.vec(1, m))
for (i in 1:m) {
    cost[i, 1] <- (1/(2 * m)) * (h(x[i, 2], beta[1, 1], beta[1, 2]) - y[i, ])^2
}
cost1001 <- colSums(cost)
# does this difference meet your convergence criteria?
print(cost1000 - cost1001)

## [1] 1.515e-11
```