Linear regression with gradient descent

30/03/2015

For the implementation of gradient descent for linear regression I draw from Digithead's blog post.

Andrew Ng defines gradient descent for linear regression as:

repeat until convergence {

$\theta_0 := \theta_0 - \alpha\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})$

$\theta_1 := \theta_1 - \alpha\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)}) \cdot x^i$

}

Where α is the training rate, m is the number of training examples, and the term on the right is the familiar squared error term after multiplication with the partial derivative $\frac{\delta}{\delta\theta_0}$ or $\frac{\delta}{\delta\theta_1}$ as appropriate.

Digithead's implementation of this is quite slick, and it took me a while to get my head around the vectorised implementation, so I will break it down here, for my own memory if nothing else:

```r
# Start by loading the data and splitting into vectors

"ex1data1.txt" %>%
  read.csv(header =  FALSE) %>%
  set_colnames(c("x","y")) -> ex1data1

X <- cbind(1,matrix(ex1data1$x))

y <- ex1data1$y
```

To run linear regression as a matrix multiplication it is necessary to add a column of ones, so that $x_0 = 1$. This means that when matrix X is multiplied by the parameter matrx θ, the intercept $\theta_0 = \theta_0 \times 1$. i.e.:

$$\begin{bmatrix} x_0^0 & x_0^1 \\ x_1^0 & x_1^1 \\ x_2^0 & x_2^1 \\ \vdots & \vdots \\ x_3^0 & x_m \end{bmatrix} \times \begin{bmatrix} \theta_0 \\ \theta_1 \end{bmatrix} = \begin{bmatrix} \theta_0 + (x_0^1 \times \theta_1) \\ \theta_0 + (x_1^1 \times \theta_1) \\ \theta_0 + (x_2^1 \times \theta_1) \\ \vdots \\ \theta_0 + (x_m^1 \times \theta_1) \end{bmatrix} \approx a + bx$$

We define the usual squared error cost function: $J(\theta_0, \theta_1) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x) - y)^2$ except that in Digithead's implementation below h_θ is defined by the matrix multiplication of X and θ as described above, and rather than multiplying by $\frac{1}{2m}$, he divides by $2m$.

```r
cost <- function(X, y, theta) {
  sum( (X %*% theta - y)^2 ) / (2*length(y))
}
```

Alpha is set at a low number initially, and the number of iterations set to 1000.

```r
alpha <- 0.01
num_iters <- 1000
```

A vector and a list are initialised to handle the history of the cost function $J(\theta_0, \theta_1)$ and the parameters θ at each iteration.

```r
cost_history <- double(num_iters)
theta_history <- list(num_iters)
```

The coefficients for regression are initialised to zero

```r
theta <- matrix(c(0,0), nrow=2)
```

Finally, the gradient descent is implemented as a for loop:

```r
for (i in 1:num_iters) {
  error <- (X %*% theta - y)
  delta <- t(X) %*% error / length(y)
  theta <- theta - alpha * delta
  cost_history[i] <- cost(X, y, theta)
  theta_history[[i]] <- theta
}
```

This makes a reasonably large jump, so I'll break down each line of this loop, for my own understanding.

Recall that Andrew Ng defines the final algorithm for gradient descent for linear regression to be:

repeat until convergence {

$\theta_0 := \theta_0 - \alpha\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})$

$\theta_1 := \theta_1 - \alpha\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)}) \cdot x^{(i)}$

}

The first and second lines of the loop handle the term $(h_\theta(x^{(i)}) - y^{(i)}) \cdot x^{(i)}$. The first line:

```r
error <- (X %*% theta - y)
```

does our linear regression by matrix multiplication, as mentioned above. The second line:

```r
delta <- t(X) %*% error / length(y)
```

does both the sum ($\sum_{i=1}^{m}$) and the element-wise multiplication denoted by the $\cdot x^{(i)}$. In the latter case this takes every single error (predicted - observed) score from the `error` function and multiplies it by the transpose X (`t(X)`), which includes $x_0 = 1$.

To give a snippet of this calculation from the first iteration:

$$\begin{bmatrix} 1.00 & 1.00 & \cdots & 1.00 \\ 6.11 & 5.53 & \cdots & 5.44 \end{bmatrix} \begin{bmatrix} -17.59 \\ -9.13 \\ \vdots \\ -0.617 \end{bmatrix} = \begin{bmatrix} (1.00 \times -17.59) + (1.00 \times -9.13) + \cdots + (1.00 \times -0.617) \\ (6.11 \times -17.59) + (5.53 \times -9.13) + \cdots + (5.44 \times -0.617) \end{bmatrix}$$

So this will end with a two dimensional vector (or a 2×1 dimensional matrix) `delta`$\in \mathbb{R}^2$. The end of this line divides by the length of the vector `y`, or in the notation that I have been using so far: m, and this is in place of multiplying by $\frac{1}{m}$.

The third line of the loop:

```
theta <- theta - alpha * delta
```

updates θ using the learning rate α multiplied by `delta`, whilst the next line:

```
cost_history[i] <- cost(X, y, theta)
```

applies the sum of squares cost function to the parameters θ following the update, and saves this out to the double-precision vector `cost_history` initialised earlier.

Finally, the last line of the code saves out the parameter vector θ to the list `theta_history`. The loop then repeats.

So let's run it and see what happens…and just out of interest, I have included a call to `system.time` so we can measure how long it takes.

```
system.time(
for (i in 1:num_iters) {
  error <- (X %*% theta - y)
  delta <- t(X) %*% error / length(y)
  theta <- theta - alpha * delta
  cost_history[i] <- cost(X, y, theta)
  theta_history[[i]] <- theta
}
)
```

```
##    user  system elapsed
##   0.025   0.000   0.025
```

```
print(theta)
```

```
##             [,1]
## [1,] -3.241402
## [2,]  1.127294
```

We can check these values using the built in regression function in R which uses the normal equation, also with a call to `system.time`.

```
system.time(
model <- lm(ex1data1$y~ex1data1$x)
)
```

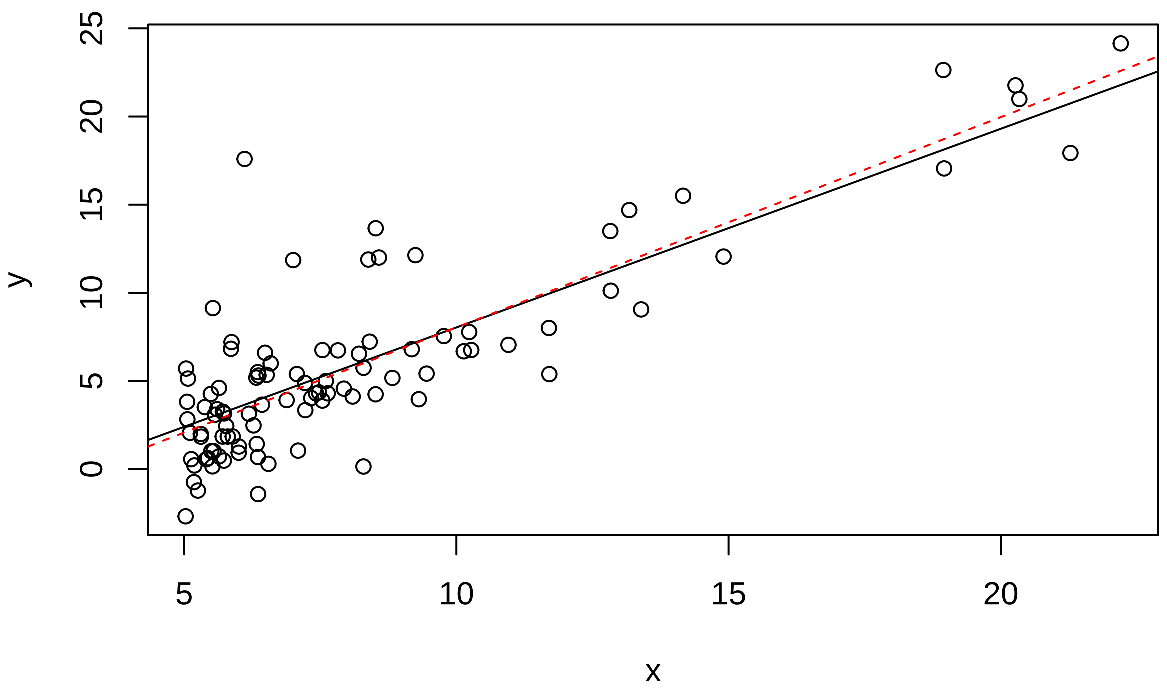
```
##    user  system elapsed
##   0.003   0.000   0.004
```

```
print(model)
```

```
##
## Call:
## lm(formula = ex1data1$y ~ ex1data1$x)
##
## Coefficients:
## (Intercept)   ex1data1$x
##      -3.896        1.193
```

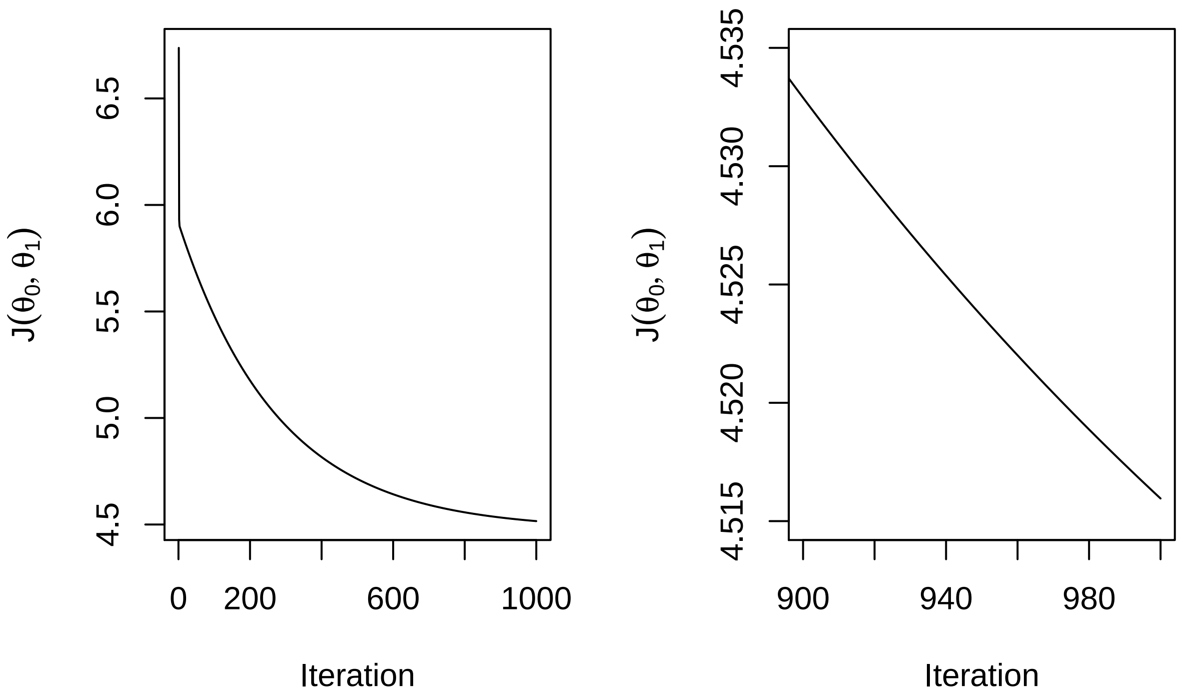
So interestingly this shows us that the gradient decsent run for 1000 iterations has stopped short of finding the correct answer, and also took 7 times longer. This may mean that α is too small, or that there were not enough iterations. The answer is close, but still not quite the same as the answer derived from the noraml equation:

```r
plot(ex1data1)
abline(a=theta[1],b=theta[2])
abline(model,col="red",lty=2)
```



Plotting $J(\theta_0, \theta_1)$ for each iteration would indicate that we had not yet minimised $J(\theta_0, \theta_1)$, and that it is continuing to fall with each further iteration:

```r
par(mfrow=c(1,2))

plot(
  cost_history,
  type = "l",
  ylab = expression(J(theta[0],theta[1])),
  xlab = "Iteration"
  )

plot(
  cost_history,
  type = "l",
  ylab = expression(J(theta[0],theta[1])),
  xlab = "Iteration",
  xlim = c(900,1000),
  ylim = c(4.515,4.535)
  )
```



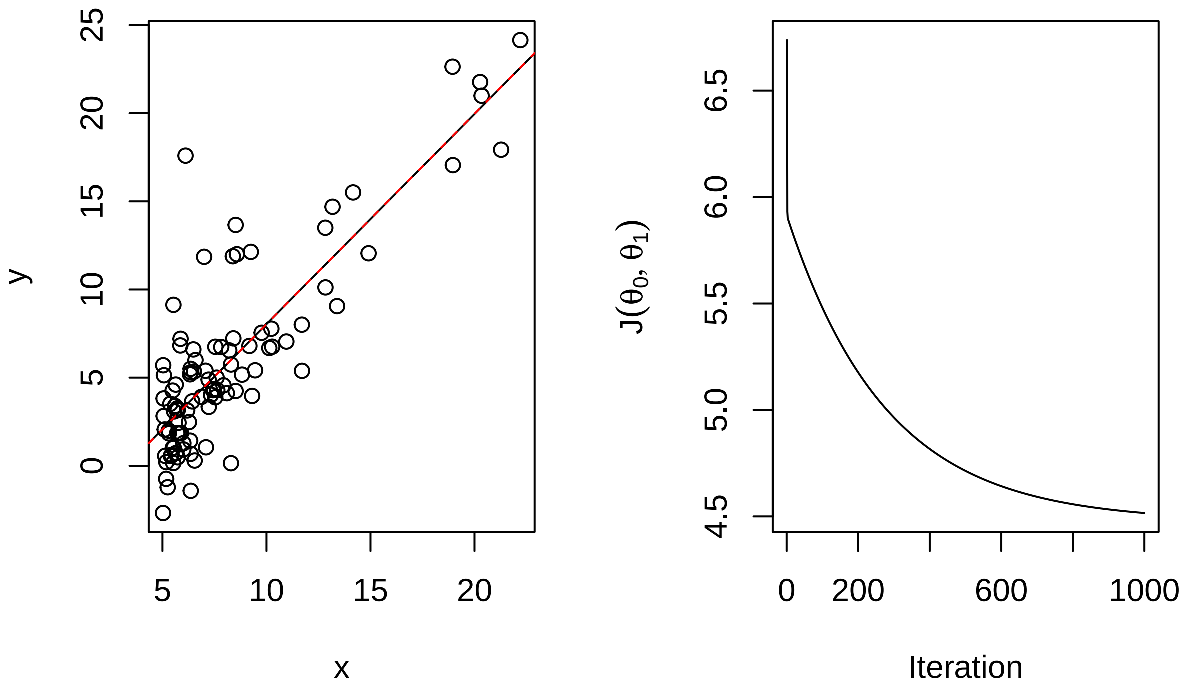
This time I try gradient descent again with having rolled the code into a self-contained function to take arguments and follow the notation that Andrew Ng has stuck to in the machine learning course. In addition, the cost function has been changed to the vectorised form:

$$J(\theta) = \frac{1}{2m}(X\theta - \vec{y})^T(X\theta - \vec{y})$$

```
grad <- function(alpha,j,X,y,theta) {

#   J <- function(X, y, theta) {
#     sum( (X %*% theta - y)^2 ) / (2*length(y))
#     }

  # The cost function vectorises to:

  J <- function(X, y, theta) {
    (1/2*length(y)) * t((X %*% theta - y)) %*% (X %*% theta - y)
    }

  theta_history <<- matrix(nrow=j,ncol=ncol(X)+1)

  for (i in 1:j) {
    error <- (X %*% theta - y)
    delta <- t(X) %*% error / length(y)
    theta <- theta - alpha * delta
    theta_history[i,] <<- c(theta,J(X, y, theta))

    if (i > 1) {

      if (
        isTRUE(
          all.equal(
            theta_history[i,3],
            theta_history[i-1,3]
            #tolerance = # can set a tolerance here if required.
              )
          )
        ) {

        theta_history <<- theta_history[1:i,]
        break

        }
      }

    }

  list(
    theta = theta,
    cost = theta_history[i,3],
    iterations = i
    )

  }

theta <- matrix(c(0,0), nrow=2)
out <- grad(0.02,3000,X,y,theta) %>% print
```

```
## $theta
##           [,1]
## [1,] -3.885737
## [2,]  1.192025
##
## $cost
## [1] 42123.91
##
## $iterations
## [1] 1656
```

```
par(mfrow=c(1,2))

plot(ex1data1)
abline(a=out[[1]][1],b=out[[1]][2])
abline(model,col="red",lty=2)

plot(
  cost_history,
  xlab="Iteration",
  ylab=expression(J(theta[0],theta[1])),
  type="l"
  )
```



---

3 Linear regression with multiple variables

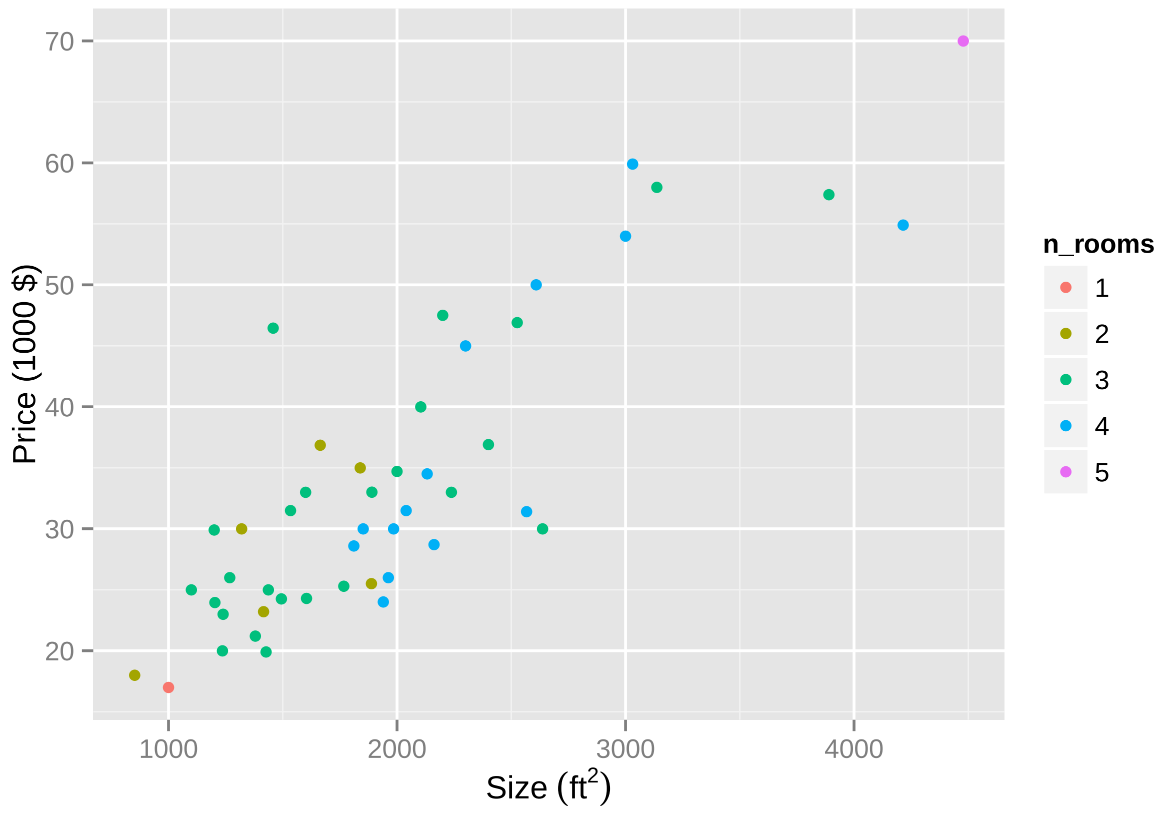
Load the data dn produce some summaries:

```
"ex1data2.txt" %>%
  read.csv(
    header = FALSE,
    col.names = c("size","n_rooms","price")
    ) %>%
  dplyr::mutate(
    n_rooms = factor(n_rooms)
    ) -> house_prices


house_prices %>% head
```

```
##   size n_rooms  price
## 1 2104       3 399900
## 2 1600       3 329900
## 3 2400       3 369000
## 4 1416       2 232000
## 5 3000       4 539900
## 6 1985       4 299900
```

Let's also plot it out of interest:

```r
library(ggplot2)

house_prices %>%
  ggplot(
    aes(
      x = size,
      y = price,
      colour = n_rooms
      )
    ) +
  geom_point()+
  scale_x_continuous(expression(Size~(ft^2)))+
  scale_y_continuous(
    "Price (1000 $)",
    breaks = seq(2e+05,7e+05,1e+05),
    labels = seq(20,70,10)
    )
```



3.1 Feature normalisation/scaling

To copy the exercise document:

> Your task here is to complete the code in featureNormalize.m to
>
> - Subtract the mean value of each feature from the dataset.
> - After subtracting the mean, additionally scale (divide) the feature values by their respective "standard deviations."

and in the file deatureNormalize.m, we get:

> First, for each feature dimension, compute the mean of the feature and subtract it from the dataset, storing the mean value in mu. Next, compute the standard deviation of each feature and divide each feature by it's standard deviation, storing the standard deviation in sigma.
>
> Note that X is a matrix where each column is a feature and each row is an example. You need to perform the normalization separately for each feature.

```r
feature_scale <- function(x) {

  # Convert all factors to numeric
  # Note that this will also allow the conversion of string features

  for (i in 1:ncol(x)) {
    x[,i] %>% as.numeric -> x[,i]
    }

  # Set up matrices to take outputs

  mu <- matrix(nrow=1,ncol=ncol(x))
  sigma <- matrix(nrow=1,ncol=ncol(x))
  scaled <- matrix(nrow=nrow(x),ncol=ncol(x))

  # Define feature scaling function

  scale <- function(feature) {
    (feature - mean(feature))/sd(feature)
    }

  # Run this for each of the features

  for (i in 1:ncol(x)) {

    mu[,i] <- mean(x[,i])
    sigma[,i] <- sd(x[,i])
    scaled[,i] <- scale(x[,i])

    }

  # And output them together as a list

  list(
    mu = mu,
    sigma = sigma,
    scaled = scaled
    )
  }

scaled_features <- feature_scale(house_prices[,-3])

range(scaled_features[[3]][,1])
```

```
## [1] -1.445423  3.117292
```

```r
range(scaled_features[[3]][,2])
```

```
## [1] -2.851859  2.404508
```

3.2 Gradient descent

Implementation note:

In the multivariate case, the cost function can also be written in the vectorised form:

$$J(\theta) = \frac{1}{2m}(X\theta - \vec{y})^T(X\theta - \vec{y})$$

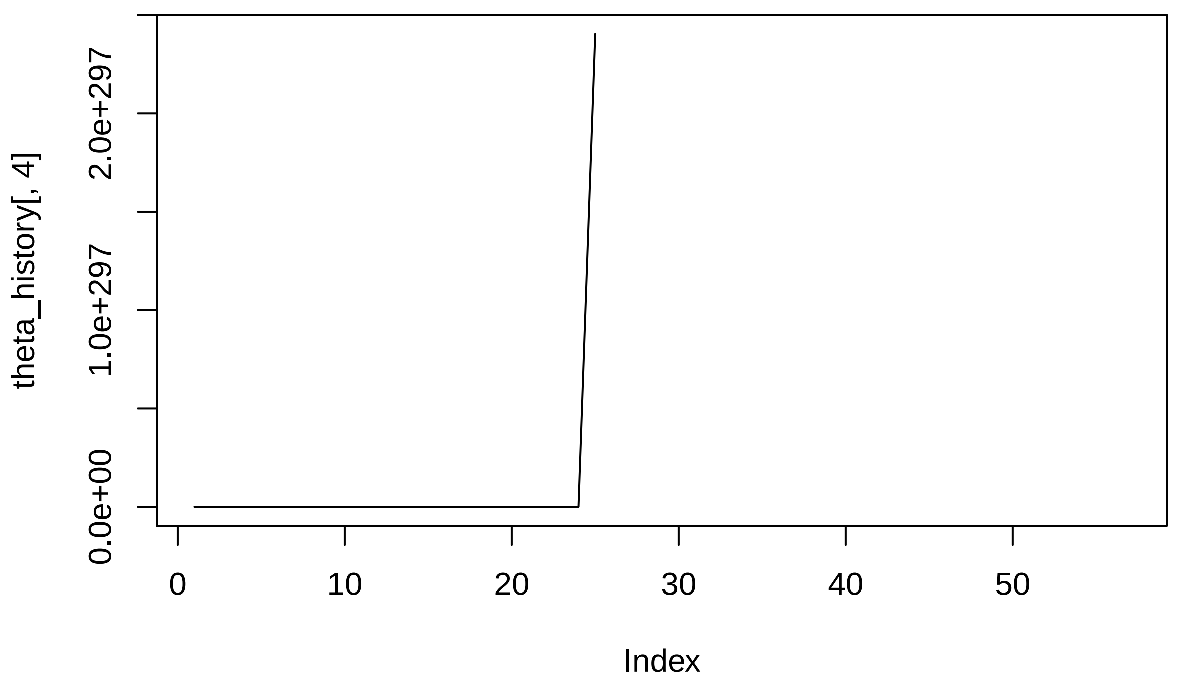
Where:

$$X = \begin{bmatrix} (x^{(1)})^T \\ (x^{(2)})^T \\ (x^{(3)})^T \\ \vdots \\ (x^{(m)})^T \end{bmatrix} \vec{y} = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ y^{(3)} \\ \vdots \\ y^{(m)} \end{bmatrix}$$

```r
X <- matrix(ncol=ncol(house_prices)-1,nrow=nrow(house_prices))
X[,1:2] <- cbind(house_prices$size,house_prices$n_rooms)
X <- cbind(1,X)
y <- matrix(house_prices$price,ncol=1)
theta <- matrix(rep(0,3),ncol=1)



multi_lin_reg <- grad(
  alpha = 0.1,
  j = 1000,
  X = X,
  y = y,
  theta = theta
  ) %>% print
```

```
## $theta
##      [,1]
## [1,]  NaN
## [2,]  NaN
## [3,]  NaN
##
## $cost
## [1] NaN
##
## $iterations
## [1] 57
```

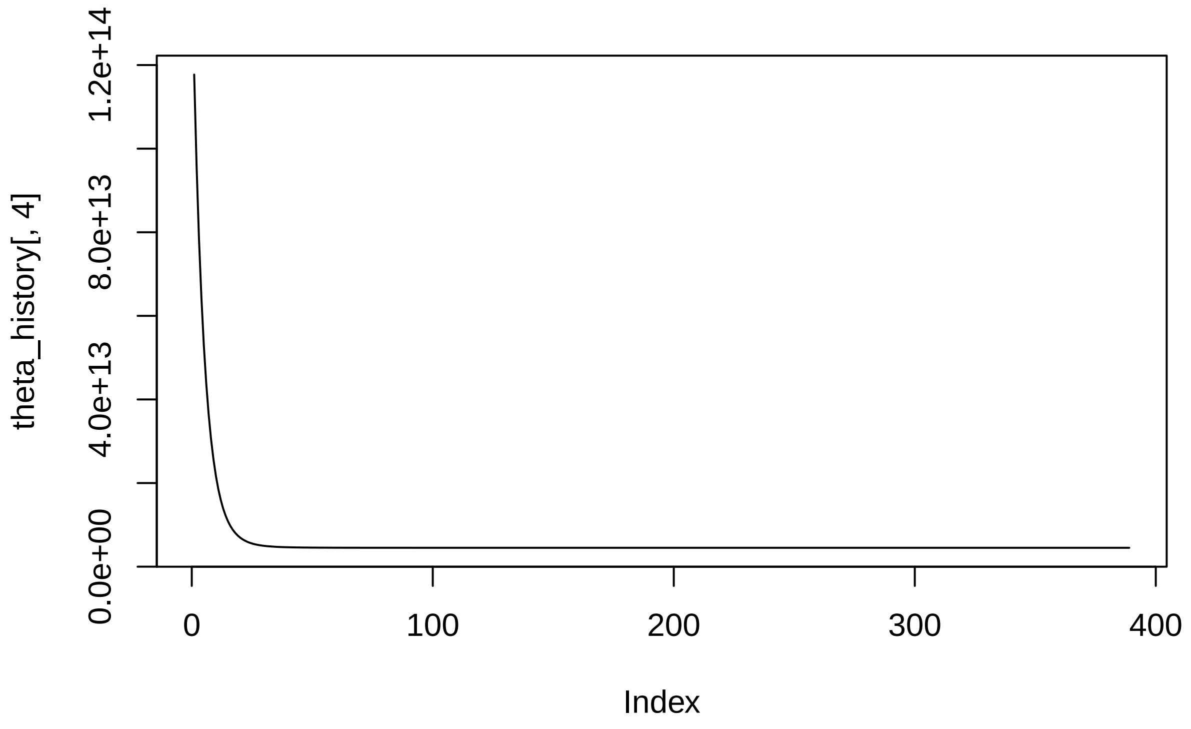
```r
plot(theta_history[,4],type="l")
```



```
X[,2:3] <- feature_scale(X[,2:3])[[3]]

multi_lin_reg <- grad(
  alpha = 0.1,
  j = 1000,
  X = X,
  y = y,
  theta = theta
  ) %>% print
```

```
## $theta
##             [,1]
## [1,] 340412.660
## [2,] 110631.048
## [3,]  -6649.472
##
## $cost
## [1] -6649.472
##
## $iterations
## [1] 389
```

```
plot(theta_history[,4],type="l")
```



Great, convergence after 389 iterations. Now a multiple linear regression the traditional way:

```r
model <- lm(
  price ~ size + n_rooms,
  data = house_prices %>% mutate(n_rooms = as.integer(n_rooms))
  )
coef(model)
```

```
## (Intercept)        size     n_rooms
##  89597.9095    139.2107  -8738.0191
```

Ok So the parameters don't match, but this is because we have scaled the features. The output from the two models will be exactly the same:

```r
house_prices %<>%
  dplyr::mutate(
    vector_pred = (X %*% multi_lin_reg$theta),
    pred = coef(model)[1] + (coef(model)[2] * size) + (coef(model)[3]*as.integer(n_rooms))
    )

identical(
  c(house_prices$vector_pred),
  c(house_prices$pred)
  )
```

```
## [1] FALSE
```

Ok not identical, how come?

```r
mean(house_prices$pred - house_prices$vector_pred)
```

```
## [1] 3.244767e-10
```

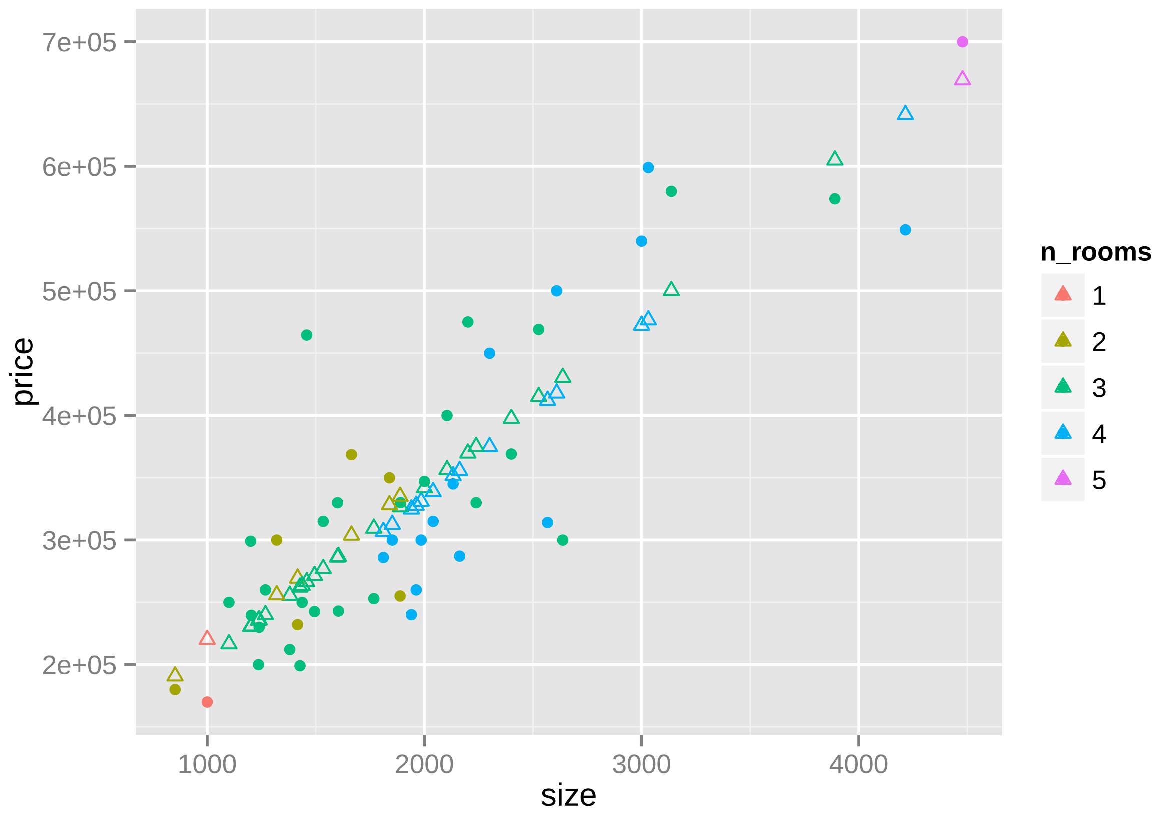
Ok so they differ by a pretty small amount, try again:

```r
all.equal(
  c(house_prices$vector_pred),
  c(house_prices$pred)
  )
```

```
## [1] TRUE
```

And now let's plot the actual data with predictions from the multiple regression.

```r
house_prices %>%
  ggplot(
    aes(
      x = size,
      y = price,
      colour = n_rooms
      )
    )+
  geom_point()+
  geom_point(
    aes(
      x = size,
      y = pred
      ),
    shape = 2,
    )
```



```r
theta <- matrix(c(1,2,3),ncol=1)
```