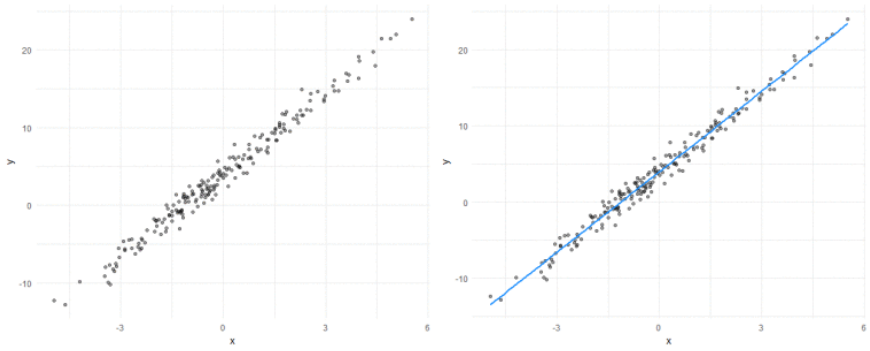
Conor McDonald Follow
Data Scientist @ Amazon, PhD, Economics (University of Leeds). R, Python and applied social/data science. @conormacd blogging at https://conorsdatablog.wordpres
Nov 27, 2017 · 8 min read

Machine learning fundamentals (I): Cost functions and gradient descent

This is part one of a series on machine learning fundamentals. ML fundamentals (II): Neural Networks can be found at https://towardsdatascience.com/machine-learning-fundamentals-ii-neural-networks-f1e7b2cb3eef

In this post I'll use a simple linear regression model to explain two **machine learning** (ML) fundamentals; (1) cost functions and; (2) gradient descent. The linear regression isn't the most powerful model in the ML tool kit, but due to its familiarity and interpretability, it is still in widespread use in research and industry. Simply, linear regression is used to estimate linear relationships between continuous *or/and* categorical data and a continuous output variable—you can see an example of this in a previous post of mine https://conorsdatablog.wordpress.com/2017/09/02/a-quick-and-tidy-data-analysis/.

As I go through this post, I'll use X and y to refer to variables. If you prefer something more concrete (as I often do), you can imagine that y is sales, X is advertising spend and we want to estimate how advertising spend impacts sales. Visually, I'll show how a linear regression learns the best line to fit through this data:



What does the machine learn?

One question that people often have when getting started in ML is:

"What does the machine (i.e. the statistical model) actually learn?"

This will vary from model to model, but in simple terms the model learns a function *f* such that *f*(*X*) maps to *y*. Put differently, the model learns how to take *X* (i.e. features, or, more traditionally, independent variable(s)) in order to predict *y* (the target, response or more traditionally the dependent variable).

In the case of the simple linear regression (***y* ~ b0 + b1 * *X*** where *X* is one column/variable) the model "learns" (read: estimates) two parameters;

- b0: the bias (or more traditionally the intercept); and,
- b1: the slope

The bias is the level of *y* when *X* is 0 (i.e. the value of sales when advertising spend is 0) and the slope is the rate of predicted increase or decrease in *y* for each unit increase in *X* (i.e. how much do sales increase per pound spent on advertising). Both parameters are scalars (single values).

Once the model learns these parameters they can be used to compute estimated values of *y* given new values of *X*. In other words, you can use these learned parameters to predict values of *y* when you don't know what *y* is—hey presto, a predictive model!

Learning parameters: Cost functions

There are several ways to learn the parameters of a LR model, I will focus on the approach that best illustrates statistical learning; minimising a **cost function**.

Remember that in ML, the focus is on **learning from data**. This is perhaps better illustrated using a simple analogy. As children we typically learn what is "right" or "good" behaviour by being told NOT to do things or being punished for having done something we shouldn't. For example, you can imagine a four year-old sitting by a fire to keep warm, but not knowing the danger of fire, she puts her finger into it and gets burned. The next time she sits by the fire, she doesn't get burned, but she sits too close, gets too hot and has to move away. The third time she sits by the fire she finds the distance that keeps her warm without exposing her to any danger. In other words, through experience and

feedback (getting burned, then getting too hot) the kid learns the optimal distance to sit from the fire. The heat from the fire in this example acts as a **cost function—it helps the learner to correct / change behaviour to minimize mistakes.**

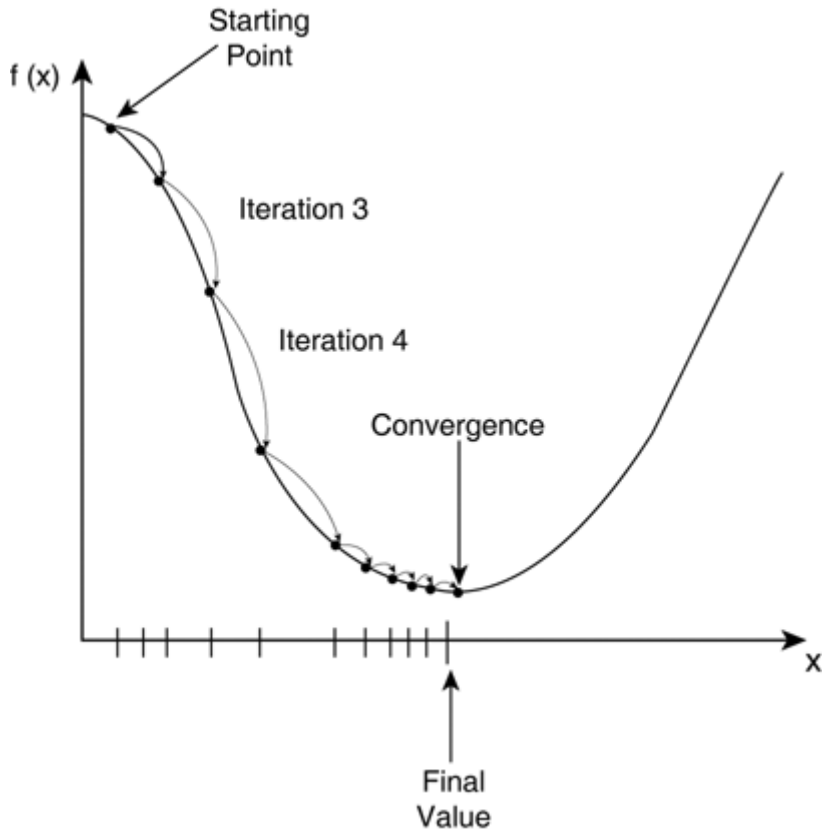
In ML, cost functions are used to estimate how badly models are performing. Put simply, ***a cost function is a measure of how wrong the model is in terms of its ability to estimate the relationship between X and y.*** This is typically expressed as a difference or distance between the predicted value and the actual value. The cost function (you may also see this referred to as *loss* or *error*.) can be estimated by iteratively running the model to compare estimated predictions against "ground truth"—the known values of *y*.

The objective of a ML model, therefore, is to find parameters, weights or a structure that **minimises** the cost function.

Minimizing the cost function: Gradient descent

Now that we know that models learn by minimizing a cost function, you may naturally wonder how the cost function is minimized—enter **gradient descent**. Gradient descent is an efficient optimization algorithm that attempts to find a local or global minima of a function.

Gradient descent enables a model to learn the gradient or *direction* that the model should take in order to reduce errors (differences between actual y and predicted y). Direction in the simple linear regression example refers to how the model parameters b0 and b1 should be tweaked or corrected to further reduce the cost function. As the model iterates, it gradually converges towards a minimum where further tweaks to the parameters produce little or zero changes in the loss—also referred to as convergence.



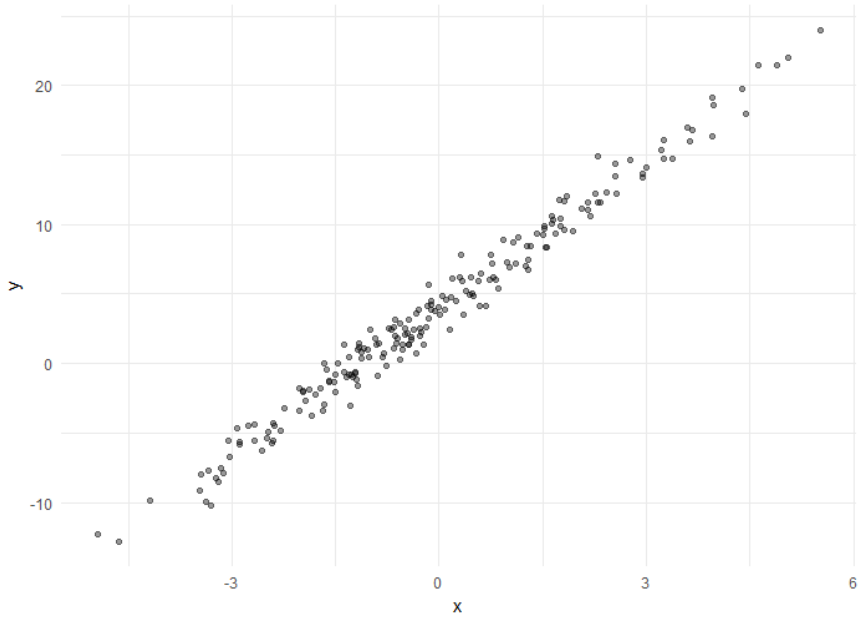
At this point the model has **optimized the weights** such that they minimize the cost function. This process is integral (no calculus pun intended!) to the ML process, because it greatly expedites the learning process—you can think of it as a means of receiving corrective feedback on how to improve upon your previous performance. The alternative to the gradient descent process would be brute forcing a potentially infinite combination of parameters until the set that minimizes the cost are identified. For obvious reasons this isn't really feasible. Gradient descent, therefore, enables the learning process to make corrective updates to the learned estimates that move the model toward an optimal combination of parameters.

Observing learning in a linear regression model

To observe learning in a linear regression, I will set the parameters b0 and b1 and will use a model to learn these parameters from the data. In other words, we know the ground truth of the relationship between X and y and can observe the model learning this relationship through iterative correction of the parameters in response to a cost (note: the code below is written in R).

```r
library(dplyr)
library(ggplot2)

n     <- 200 # number of observations
bias  <- 4
slope <- 3.5
dot   <- `%*%` # defined for personal preference

x   <- rnorm(n) * 2
x_b <- cbind(x, rep(1, n))
y   <- bias + slope * x + rnorm(n)
df  <- data_frame(x = x,  y = y)

learning_rate <- 0.05
n_iterations  <- 100
theta         <- matrix(c(20, 20))

b0    <- vector("numeric", length = n_iterations)
```

Here I define the bias and slope (equal to 4 and 3.5 respectively). I also add a column of ones to X (for the purposes of enabling matrix multiplication). I also add some Gaussian noise to y to mask the true parameters—i.e. create errors that are purely random. Now we have a dataframe with two variables, X and y, that appear to have a positive linear trend (as X increases values of y increase).



Next I define the learning rate—this controls the size of the steps taken by each gradient. If this is too big, the model might miss the local minimum of the function. If it too small, the model will take a long time to converge (copy the code and try this out for yourself!). Theta stores the parameters b0 and b1, which are initialized with random values (I have set these these both to 20, which is suitably far away from the true parameters). The n_iterations value controls how many times the model will iterate and update values. That is, how many times the model will make predictions, calculate the cost and gradients and update the weights. Finally, I create some placeholders to catch the values of b0, b1 and the sum of squared errors (SSE) upon each iteration of the model (creating these placeholders avoids iteratively growing a vector, which is very inefficient in R).

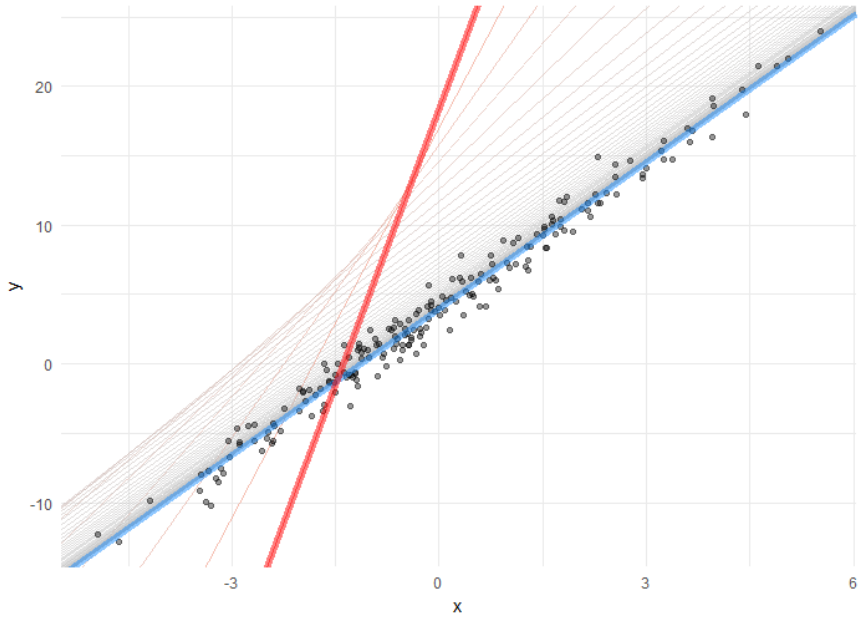
The ***SSE*** in this case is the cost function. It is simply the ***sum of the squared differences between predicted y and actual y (i.e. the residuals)***

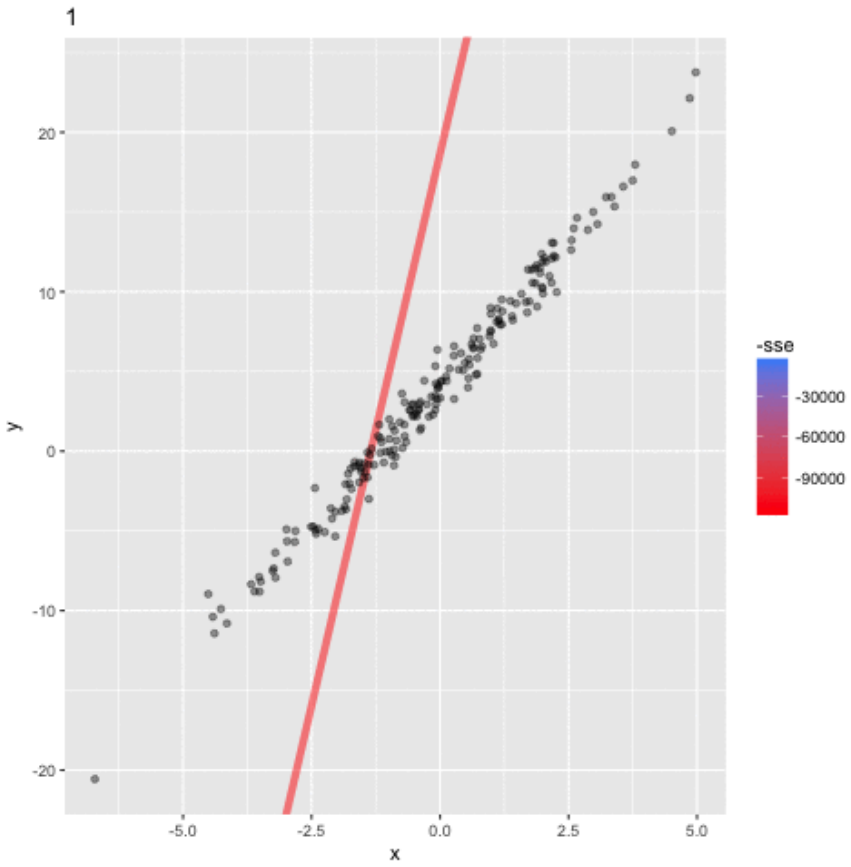
Now, we run the loop. On each iteration the model will predict *y* given the values in theta, calculate the residuals, and then apply gradient descent to estimate corrective gradients, then will update the values of theta using these gradients—this process is repeated 100 times. When the loop is finished, I create a dataframe to store the learned parameters and loss per iteration.

```
for (iteration in seq_len(n_iterations)) {

  residuals_b        <- dot(x_b, theta) - y
  gradients          <- 2/n * dot(t(x_b), residuals_b)
  theta              <- theta - learning_rate * gradients

  sse_i[[iteration]] <- sum((y - dot(x_b, theta))**2)
  b0[[iteration]]    <- theta[2]
  b1[[iteration]]    <- theta[1]

}
```

When the iterations have completed we can plot the lines than the model estimated.

```
p1 <- df %>%
  ggplot(aes(x=x, y=y)) +
  geom_abline(aes(intercept = b0,
                  slope = b1,
                  colour = -sse,
                  frame = model_iter),
              data = model_i,
              alpha = .50
              ) +
  geom_point(alpha = 0.4) +
  geom_abline(aes(intercept = b0,
                  slope = b1),
              data = model_i[100, ],
              alpha = 0.5,
              size = 2,
              colour = "dodger blue") +
  geom_abline(aes(intercept = b0,
                  slope = b1),
              data = model_i[1, ],
              colour = "red",
              alpha = 0.5,
              size = 2) +
```

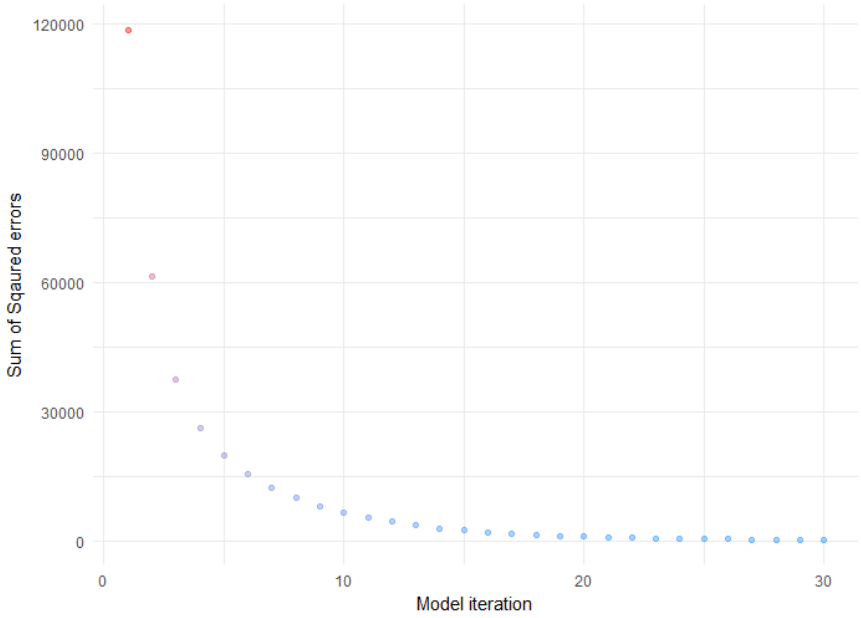




The first thing to notice is the thick red line. This is the line estimated from the initial values of b0 and b1. You can see that this doesn't fit the data points well at all and because of this it is has the highest error (SSE). However, you can see the lines gradually moving toward the data points until a line of best fit (the thick blue line) is identified. In other words, upon each iteration the model has learned better values for b0 and b1 until it finds the values that minimize the cost function. The final values that the model learns for b0 and b1 are 3.96 and 3.51 respectively—so very close the parameters 4 and 3.5 that we set!

Voilla! Our machine! it has learned!!

We can also visualize the decrease in the SSE across iterations of the model. This takes a steep decline in the early iterations before converging and stabilizing.



We can now use the learned values of b0 and b1 stored in theta to predict values *y* for new values of *X*.

```
predict_from_theta <- function(x) {

  x <- cbind(x, rep(1, length(x)))
  dot(x, theta)

}

predict_from_theta(rnorm(10))
           [,1]
 [1,] -1.530065
 [2,]  8.036859
 [3,]  6.895680
 [4,]  3.170026
```

Summary

This post presents a very simple way of understanding machine learning. It goes without saying that there is a lot more to ML, but gaining an initial intuition for the fundamentals of what is going on "underneath the hood" can go a long way toward improving your understanding of more complex models.

. . .