Linear Regression

Dave Miller

1/31/2019

Modeling

- In general, a predictive model with one indicator variable will have the form
$$Y = f(X) + \epsilon$$
for some function f, which we will define, and error ϵ.

- In this section we are ultimately going to build a **linear model** called **ordinary linear regression**.

Simple Linear Regression

We use the standard equation of any line with the slope and intercept defined using β_0 and β_1 respectively. In general, a linear model with response variable Y and indicator variable X is given by:

$$Y = \beta_0 + \beta_1 X + \epsilon,$$

where:

- Y is the response variable,
- X is the indicator variable,
- β_0 is the intercept,
- β_1 is the coefficient (slope term) representing the linear relationship,
- ϵ is a mean-zero random error term (more on that later).

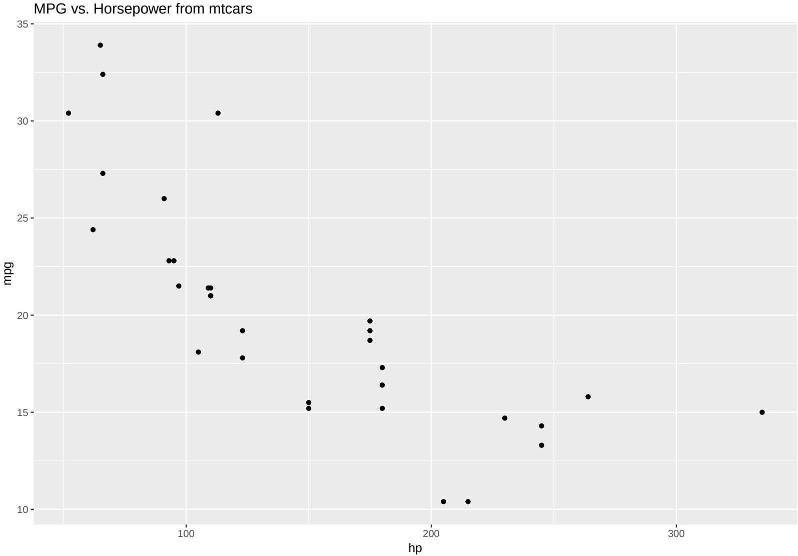
Modeling note

- X and Y are written as capital letters because they are actually **vectors** that contain all of the x and y points respectively:

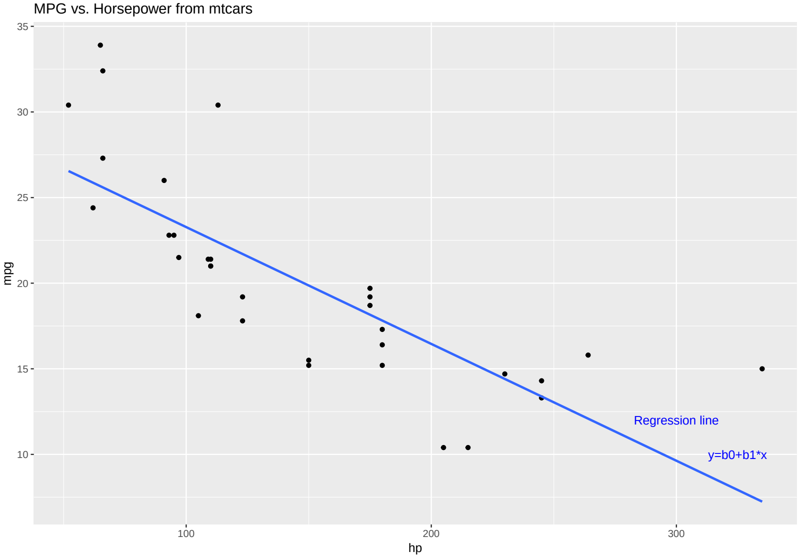
$$X = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \quad Y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix},$$

where the (x_k, y_k) pairs are given in the dataset, and n is the number of points in the set.

Looking at data: mtcars



Simple Linear Regression



Finding the regression line in R

R has many built in functions to assist in model building, including the `lm()` function (linear model) for linear regression.To build this model in R we use the formula notation $Y \sim X$:

```
model1 <- lm(mpg ~ hp, data = mtcars)
```

Finding the regression line in R

We can find the coefficients the of the fitted regression line using the `coefficients()` command:

```
coefficients(model1)
```

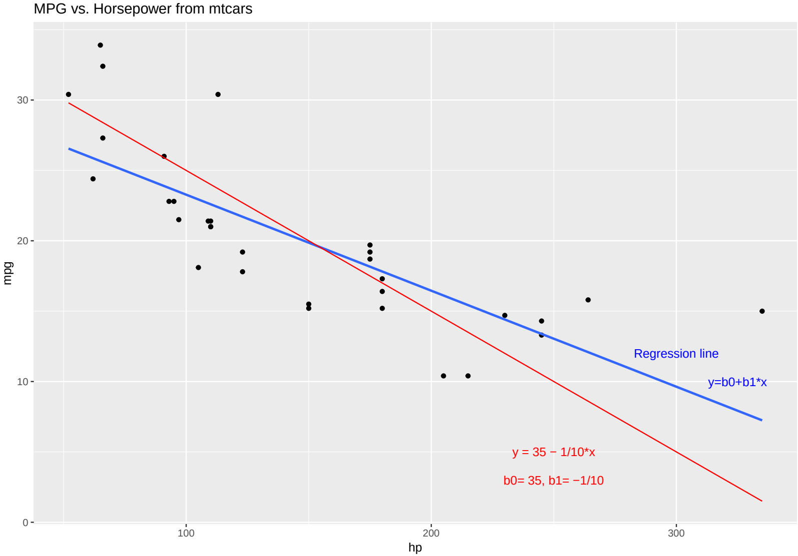
```
## (Intercept)          hp
## 30.09886054 -0.06822828
```

Least Squares Regression Lines

- The goal in creating the model is to find the values of β_0, β_1 that 'fit' our data.

- What does it mean to 'fit' our data?

Least Squares Regression Lines



Least Squares Regression Lines

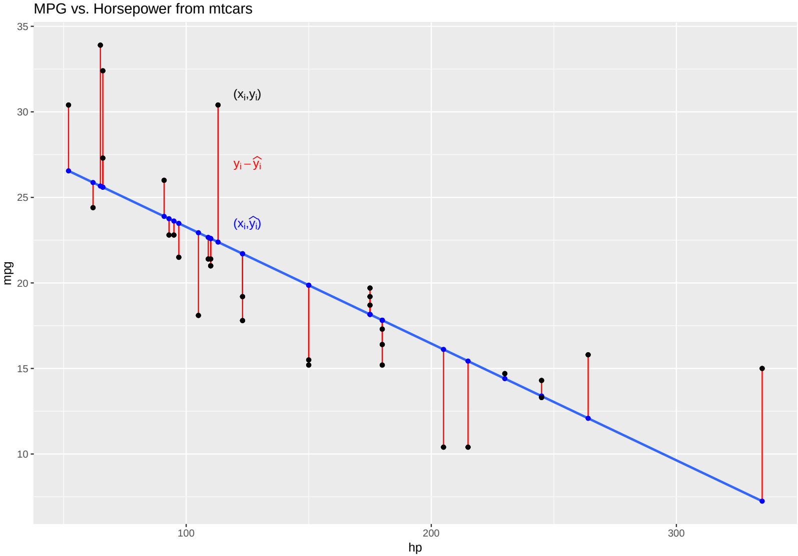
- Our goal is to **minmize the error**, ϵ, in the given model.
- As shown in the picture on the next slide, the error is defined as the difference between the actual value and the predicted value (also called the *residual*), formally:

$$\epsilon_i = y_i - \widehat{y_i},$$

where $\widehat{y_i}$ is the **predicted** value from our model, defined by

$$\widehat{y_i} = \beta_0 + \beta_1 x_i.$$

Least Squares Regression Lines



Least Squares Regression Lines

- The blue line above is called the **Least Squares Regression Line** because it minimizes the sum of the **mean squared error**, or

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \widehat{y_i})^2.$$

- Plugging in the equation of $\widehat{y_i}$ above gives

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - (\beta_0 + \beta_1 x_i))^2.$$

Least Squares Regression Lines

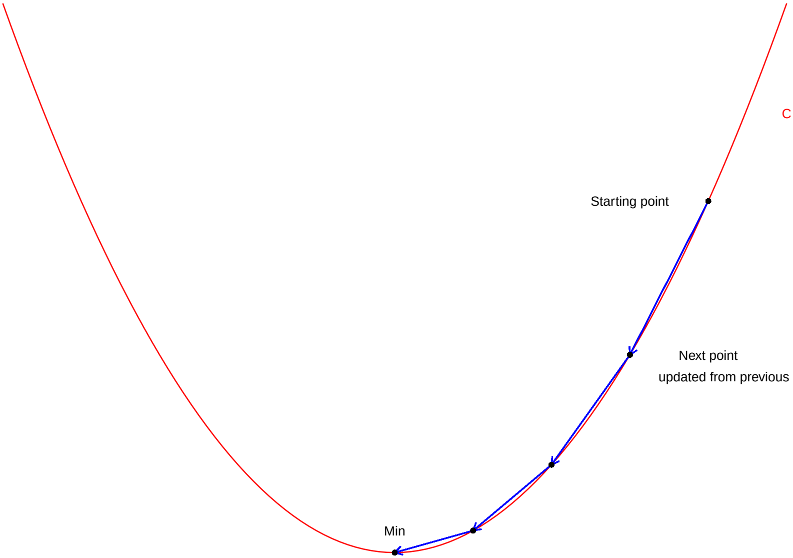
- The goal of the least squares regression line is to *find* β_0 *and* β_1 *such that the mean squared error (MSE) equation above is a minimum*.
- In machine learning, *MSE* is called the **cost function**, C.
- In the equation above, x_i and y_i are known values from our dataset, leaving β_0 and β_1 as our only unknowns. For this reason, we will re-write the equation above as a function of two variables:

$$C(\beta_0, \beta_1) = \frac{1}{n}\sum_{i=1}^{n}(y_i - (\beta_0 + \beta_1 x_i))^2.$$

Gradient Descent

- There are various ways of finding the minimum value of the mean squared error from Calculus or Linear Algebra, many of which are slow or not accurate.
- **Gradient descent** is an optimzation algorithm that looks for the minimum cost by computing the cost function at a given point and updating the weights β_0 and β_1 with slightly better cost.
- The process is repeated until the minimum is reached (or close enough). Gradient descent is represnted visually on the next slide.

Gradient Descent



The Gradient from Calculus III

Reminder that we are looking to minimize the cost function: $C(\beta_0, \beta_1)$.
From Calclus III, the gradient of C, or ∇C, is given by

$$\nabla C = \begin{bmatrix} \frac{\partial C}{\partial \beta_0} \\ \frac{\partial C}{\partial \beta_1} \end{bmatrix}.$$

Gradient Descent Algorithm

- Overall, the gradient descent algorithm is as follows:

1. Pick a starting value for β_0, β_1.
2. Update the β terms by:

$$\beta_0 := \beta_0 - \alpha \frac{\partial C}{\partial \beta_0}$$

$$\beta_1 := \beta_1 - \alpha \frac{\partial C}{\partial \beta_1}$$

   where α is the **learning rate**, which is chosen by the user.
3. Repeat step 2. until at (or near) the minimum.

Note: The substraction in the equations is due to the fact that we want to go *down* the curve. Also, the notation $:=$ implies that we are *updating* the β terms, and that they are defined by the previous point.

Computing the gradient

- Using MSE as our cost function, from Calculus we get

$$\frac{\partial C}{\partial \beta_0} = -\frac{2}{n}\sum_{i=1}^{n}(y_i - (\beta_0 + \beta_1 x_i))$$

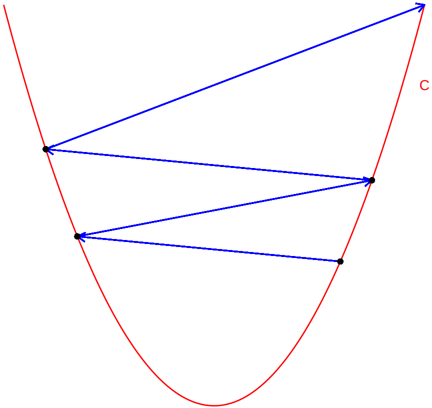
$$\frac{\partial C}{\partial \beta_1} = -\frac{2}{n}\sum_{i=1}^{n}x_i(y_i - (\beta_0 + \beta_1 x_i)).$$

Note, these will be different using different cost functions (MSE or RMSE).

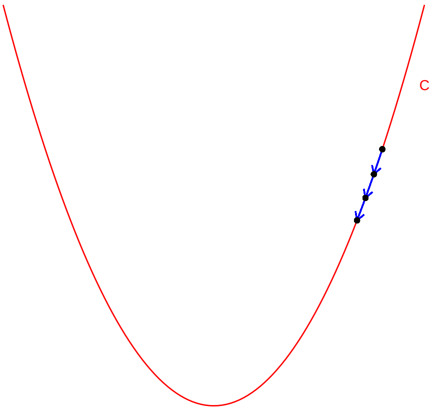
Learning rate

- Different α values will have different consequences on the algorithm.
  - If α is too large, the algorithm may overshoot the minimum, or even diverge.
  - If α is too small, gradient descent can be too slow.

Large leraning rate



Small learning rate



Finding the minimum

- Finally, we need to satisfy step 3, where we stop the algorithm when we are at or near the minimum.

- Again, from Calculus III, it is known that the minimum satisfies the property that

$$\nabla C(\text{min}) = \begin{bmatrix} \frac{\partial C}{\partial \beta_0}(\text{min}) \\ \frac{\partial C}{\partial \beta_1}(\text{min}) \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}.$$

Finding the minimum

- It will take gradient descent far too long if we require the algorithm to reach the exact minimum point.
- Instead, we ask that it gets ''close enough", which we will define in the algorithm, ensuring efficiency. This implies that we want ∇C to be close to 0, or more techincally:

$$||\nabla C|| < \epsilon,$$

where $||\nabla C||$ is called the **norm** of C (more specifically, the L1 norm), defined as

$$||\nabla C|| = \left|\frac{\partial C}{\partial \beta_0}\right| + \left|\frac{\partial C}{\partial \beta_1}\right|,$$

and $\epsilon > 0$ is a very small number.

Algorithm in R

Overall the gradient descent algorithm can be written in a function in R as found on GitHub.

We can run the function by using the following code, with $\epsilon = 0.001$, $\alpha = 0.00001$, $x =$ hp and $y =$ mpg:

```
grad_descent(x = mtcars$hp, y = mtcars$mpg, alpha =
0.00001, epsilon = 0.001)
```

```
## [1] "beta0: 30.096, beta1: -0.0682"
```

Trying other things on your own

- Try running the code with different alpha values and see what happens.
- Try splitting the data into training and testing sets and finding the line.
  - Once you find the line, evaluate the model by finding the training and test error.
  - Do we have overfitting? Underfitting? What can we do to improve the model?