Linear Regression

Mads Møller

LinkedIn: https://www.linkedin.com/in/madsmoeller1/

Mail: mads.moeller@outlook.com

This paper is the first in a series, which are going to go into the depth behind behind the mathematics, statistics and code implementation of different machine learning algorithms. We will use Python as our framework for implementing these algorithms. At the top of each paper a link to the GitHub Repository including the code will be available. I hope you will enjoy this series of papers and hopefully get a good understanding of how the various algorithms works.

1 Univariate Linear Regression

Before we go into linear regression let's define what we mean about regression. Regression is all about modelling a target value, which we denote y based on independent variables, which are going to be denoted by X. A univariate linear regression model is characterized by one independent variable (x) and a dependent variable (y). To understand the mathematics behind simple linear regression we will inspect figure 1 below:

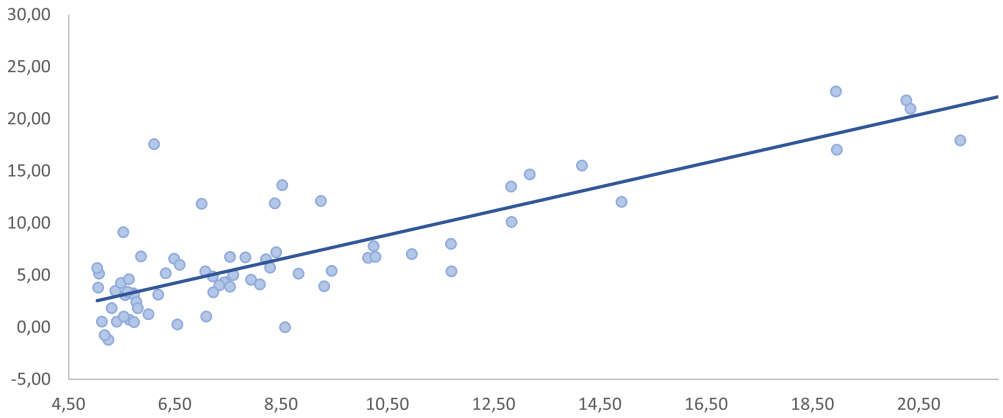


Figure 1: Linear Regression

We refer to the blue line from figure 1 as *the best fit straight line*. What linear regression is really all about is finding the line that fits our observations best. A line can be written mathematically as:

$$\hat{y} = w_0 + w_1 x \tag{1}$$

Here x is called our independent variable (input variable), w_1 is called the *weight* and w_0 is called the *bias term*. Therefore, we want to find those values of w_0 and w_1 which gives us the best model fit. In order to find the best line we have to set up a formula which can figure out what line minimizes the distance from all points to the line. First of all we need to get familiar with two essential machine learning principles called *cost functions* and *gradient descent*.

1.1 Cost Function

If we inspect figure 1 again, the blue line represent predictions of our dependent variable y. So the problem that linear regression solves is fitting the line which minimizes the distance between the actual values (observations on the graph) and our predicted values (the line). This is also referred to as the *cost*. Therefore, the function we are going to minimize is called the *cost function*. In linear regression we define our cost function as:

$$J(w_0, w_1) = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2 = \frac{1}{n}\sum_{i=1}^{n}((w_0 + w_1 x_i) - y_i)^2 \tag{2}$$

We denote our cost function with "J", the predicted values as $\hat{y}$, the observations from our dataset as y and the number of observations n. This cost function is also referred to as the *mean squared error* or just *MSE*. By using MSE we are going to minimize the error between the predicted value and the actual value. As seen from equation 2, we square the error difference and sum over all the n datapoints and divide that value by the total number of data points. The reason of why we square the error difference is because we need to be able to handle negative differences and add that value to our cost as well.

However, in order to minimize our cost function we need to figure out how to change the weights w_0 and w_1, such that our cost is minimized. To understand how to change the weights properly we have to define an *optimization algorithm* for our cost function.

1.2 Gradient Descent

Gradient Descent is a simple optimization algorithm. An optimization algorithm which tries to minimize our cost function. The math behind Gradient Descent is a simple process:

$$w_i = w_i - \alpha\frac{\partial J}{\partial w_i} \tag{3}$$

Gradient Descent repeats this process until convergence. All of the weights has to be updated simultaneously, so for our simple univariate linear regression the two weights $i = \{0, 1\}$ are going to be updated simultaneous. The variable α is called the *learning rate*, and the value of α decides how much the weights are being changed in each iteration. The partial derivatives w.r.t. the weights can easily be calculated from equation 2:

$$\frac{\partial J}{\partial w_0} = \frac{2}{m}\sum_{i=1}^{m}(\hat{y}_i - y_i) \tag{4}$$

$$\frac{\partial J}{\partial w_1} = \frac{2}{m}\sum_{i=1}^{m}(\hat{y}_i - y_i)x_i \tag{5}$$

The intuition of gradient descent is easy to illustrate. Let's inspect figure 3.1:

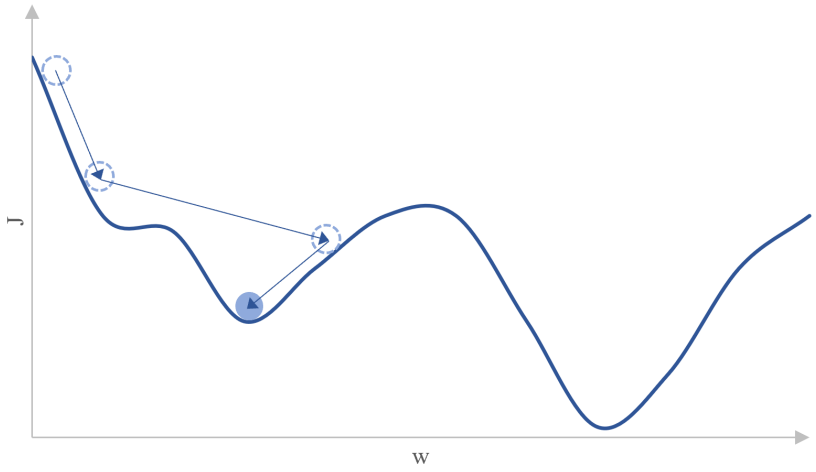


Figure 2: Gradient Descent

The most intuitive way of understanding Gradient Descent is trying to imagine a ball on a graph, as in figure 3.1. The graph is our cost function w.r.t. the weight w. The ball represents each iteration in Gradient Descent for this example. The ball calculates the slope of our function for that specific place on our cost function (the weight). If the slope is positive our weight will decrease with the slope times our learning rate and our ball will roll down, which is seen in equation 6. On the other hand, if our slope is negative our weight will increase with the slope times the learning rate.

The cost function can be a very complex function, which is the case in figure 3.1. Here Gradient Descent converge to a local minimum, which means that our cost could have been reduced even more. To avoid this one could change the learning which indicates how big a step we want to change our weight in each iteration. However, a too high learning rate can can also lead to a situation where our cost function cannot converge and find a minimum. However for a linear regression model as ours gradient descent will work just fine. For more complex machine learning models which we will cover later, there are better *optimization algorithms* to avoid ending up in a local minimum.

2 Multivariate Linear Regression

We have seen the fundamentals of a simple univariate linear regression. This can easily be generalized to a multivariate situation. We now have the following situation with n features/columns:

$$\hat{y} = w_0 + w_1x_1 + w_2x_2 + \cdots + w_nx_n = \sum_{i=0}^{n} w_ix_i = \underline{w}^T\underline{x} \tag{6}$$

Where $\underline{w} = \begin{pmatrix} w_0 \\ w_1 \\ \vdots \\ w_n \end{pmatrix} \in \mathbb{R}^{n+1}$ and $\underline{x} = \begin{pmatrix} 1 \\ x_1 \\ \vdots \\ x_n \end{pmatrix} \in \mathbb{R}^{n+1}$

As in the case of simple univariate linear regression we aim to find the optimal weights such that the regression line fits our data best (has the minimum cost). For linear regression the optimal weights can be found analytically with what is called the **normal equation** or we can use gradient descent.

2.1 Normal Equation

One could imagine our feature vector from earlier containing multiple (or all) rows from our data. Let us imagine we have a dataset with n features and m rows. From equation 6 we now have:

$$\underbrace{\underline{\hat{y}}}_{m\times 1} = \underbrace{\underline{\underline{X}}}_{m\times(n+1)} \cdot \underbrace{\underline{w}}_{(n+1)\times 1} \tag{7}$$

We can now rewrite the least squares cost function from equation 2:

$$\begin{aligned} J(\underline{w}) &= \frac{1}{n} \cdot (\underline{\underline{X}} \cdot \underline{w} - \underline{y})^T (\underline{\underline{X}} \cdot \underline{w} - \underline{y}) \\ &= \frac{1}{n} ((\underline{\underline{X}} \cdot \underline{w})^T - \underline{y}^T)(\underline{\underline{X}} \cdot \underline{w} - \underline{y}) \\ &= \frac{1}{n} ((\underline{\underline{X}} \cdot \underline{w})^T (\underline{\underline{X}} \cdot \underline{w}) - (\underline{\underline{X}} \cdot \underline{w})^T \underline{y} - \underline{y}^T (\underline{\underline{X}} \cdot \underline{w}) + \underline{y}^T \underline{y}) \\ &= \frac{1}{n} (\underline{w}^T \underline{\underline{X}}^T \underline{\underline{X}} \cdot \underline{w} - 2(\underline{\underline{X}} \cdot \underline{w})^T y + \underline{y}^T \underline{y}) \end{aligned} \tag{8}$$

Our least sqares cost function is a convex function. Therefore in order to find the values of $\underline{w}$, which minimizes the cost we can simply use calculus:

$$\frac{\partial J}{\partial \underline{w}} = 2\underline{\underline{X}}^T \underline{\underline{X}} \cdot \underline{w} - 2\underline{\underline{X}}^T \underline{y} = 0 \iff \underline{\underline{X}}^T \underline{\underline{X}} \cdot \underline{w} = \underline{\underline{X}}^T \underline{y} \tag{9}$$

We now assume that the matrix $\underline{\underline{X}}^T \underline{\underline{X}}$ is invertible, such that we can get an expression for our vector of weights:

$$\underline{w} = (\underline{\underline{X}}^T \underline{\underline{X}})^{-1} \underline{\underline{X}} y \tag{10}$$

This equation is called the **normal equation**. In cases of a small number of feautures the normal equation has been proven to work more effective than applying gradient descent.

2.2 Multivariate Gradient Descent

We will now generalize gradient descent from section 1.2. The updates from gradient descent are exactly the same as in the case of univariate linear regression. However, we can generalize the derivatives from equation 5 such that gradient descent has to simultaneously update the weights using gradient descent:

$$w_j = w_j - \alpha \frac{2}{m} \sum_{i=1}^{m} (\hat{y}^{(i)} - y^{(i)}) x_j^{(i)} \tag{11}$$

Gradient descent for multivariate linear regression can now easily be implemented. One has to be aware that $x_0 = 1$ in order to correctly implement the algorithm. In practice we will add a column of ones to our feature matrix X to handle this.

3 Implementation of linear regression

For our implementation of linear regression we will firstly implement the algorithm from scratch. I like to use **Object Oriented Programming** when coding these algorithms from scratch. If you are not familiar with the concepts of object oriented programming i will recommend you to read this article before trying to understand the code implementation. Afterwards we will see how linear regression can be implemented using the Python machine learning library *Scikitlearn*.

3.1 Implementation - From scratch

For implementing linear regression without fancy machine learning packages we are going to use the following libraries:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

The data we are using only contain an independent- and a dependent variable:

```
data = pd.read_csv('LR_data.txt', header = None)
X = data.iloc[:, 0]
y = data.iloc[:, 1]
```

We are now ready to create a Linear Regression class:

```
class LinearRegression:
    def __init__(self, X, y):
        self.m = len(y)
        self.X = np.vstack((np.ones(self.m), X)) #adding column of ones to X
        self.y = y
        self.w = np.zeros(self.X.shape[0]) #weighs initialized as zero
```

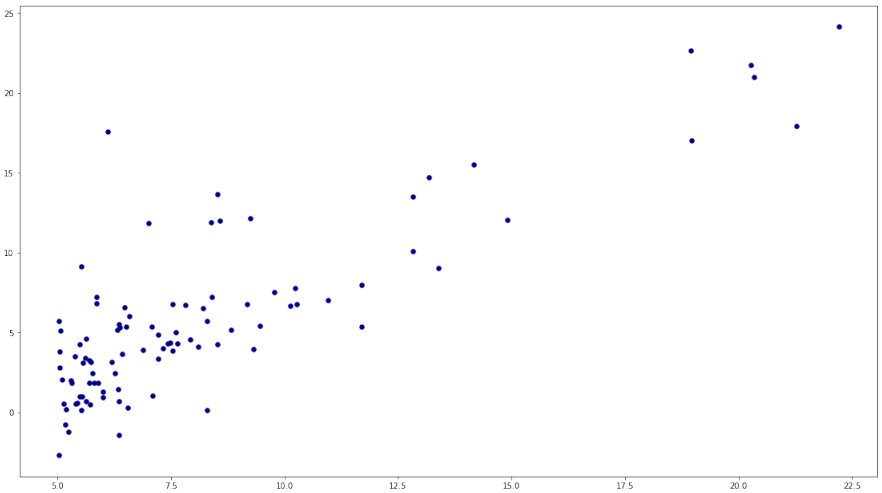
We will now create some functions within our **LinearRegression** class. We are going to create two functions for visualization, a cost function and gradient descent implemented with equation 11:

```
def createScatter(self):
    if self.X.shape[0] == 2:
        plt.figure(figsize=(19.20,10.80)) #HD
        plt.scatter(self.X[1], self.y, s = 30,color='darkblue', marker = 'o')
        plt.show()
```

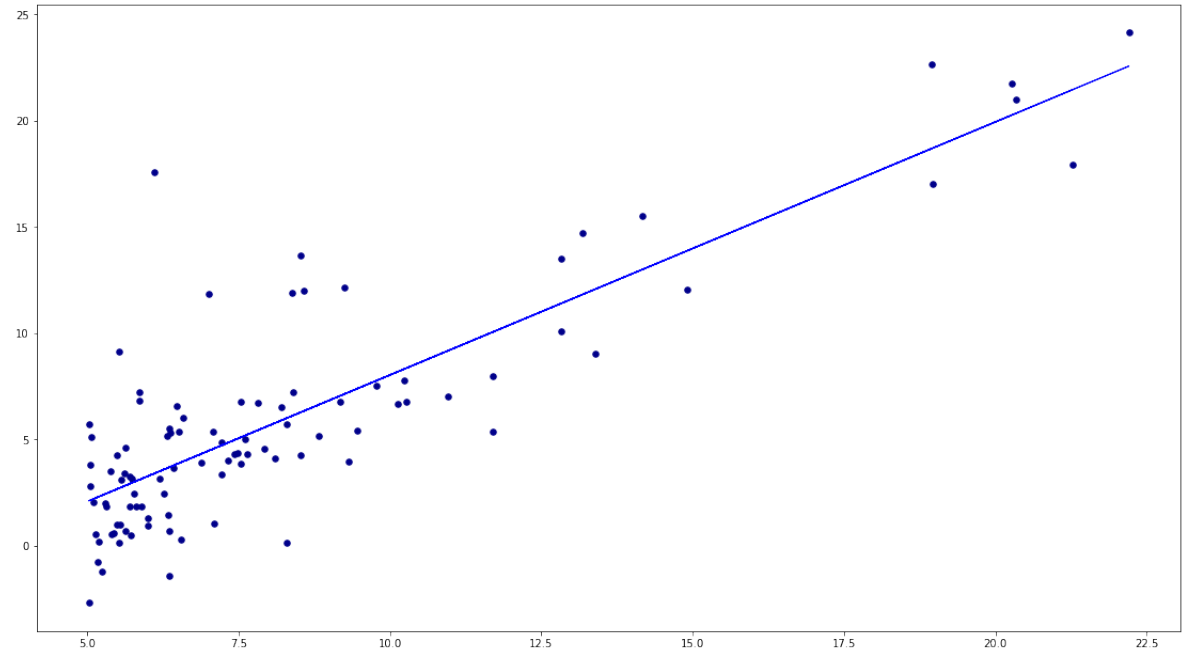
```
        else:
            print(f'Cannot create scatter plot for multivariate linear regression.')

    def cost(self):
        return 1/self.m*sum((self.X.T.dot(self.w)-self.y)**2)

    def gradientDescent(self, alpha = 0.01, iterations = 1500):
        cost_iteration = []
        for i in range(iterations):
            cost_iteration.append(self.cost())
            self.w = self.w - alpha*2/self.m * self.X.dot((self.X.T.dot(self.w)-self.y))
        return self.w, cost_iteration

    def plotLinearRegression(self, pltLabel = 'Linear Regression', xL = "", yL = ""):
        y_pred = self.X.T.dot(self.w)
        plt.figure(figsize=(19.20,10.80))
        plt.scatter(self.X[1], self.y, s = 30,color='darkblue', marker = 'o')
        plt.plot(self.X[1], y_pred, color='blue')
        plt.xlabel(xL)
        plt.ylabel(yL)
        plt.title(pltLabel)
        plt.show()
```

This is all of the code for implementing our first machine learning algorithm. Let us try to see how our algorithm works for our dataset:

```
LR = LinearRegression(X, y)
LR.createScatter()
```



```
w, costs = LR.gradientDescent(iterations = 1500)
LR.plotLinearRegression(pltLabel ="")
print('weights: ', w)
```




```
weights:  [-3.87813769  1.19126119]
```

3.2 Implementation - Scikit-Learn

Scikit-Learn is a machine learning library for Python, propably the most used one. Implementing machine learning models are easy using Scikit-Learn. You just have to learn the syntax. We will import the following libraries:

```
import pandas as pd
import numpy as np
from sklearn.linear_model import LinearRegression
```

In order to use Sklearn's algorithm you have to know how to prepare the data for the specific algorithm. Therefore, for linear regression we need to reshape our inputs such that they fit the algorithm:

```
data = pd.read_csv('LR_data.txt', header = None)
X = np.array(data.iloc[:, 0]).reshape(-1, 1)
y = np.array(data.iloc[:, 1]).reshape(-1, 1)
```

Now we can implement the algorithm for our dataset:

```
from sklearn.linear_model import LinearRegression
regressor = LinearRegression()
regressor.fit(X, y)
```

Let us take a look on the weight from Sklearn's fit:

```
print(regressor.intercept_)
print(regressor.coef_)
weights:  [-3.89578088  1.19303364]
```

The weight are pretty similar for Scikit-Learn implementation compared with our algorithm we made from scratch.