Gradient Descent

1. Data set :-

For implementing, the gradient descent algorithm we used a small dataset. It is recorded as various state's population and drinking data in order to predict the death rate caused by cirrhosis. It consists of **5 columns and 46 rows**. These five rows represent various factors like **urban population, wine and liquor consumption etc.** which are independent variables. Moreover, the last column consists of the **death rate**, which is a dependent variable, dependent upon the first four attributes. For more info, you can refer to **"ReadMe.txt"** file attached to this folder which includes sources of this data as well. And **"data.txt"** contains the actual data.

2. Theoretical aspect of gradient descent:

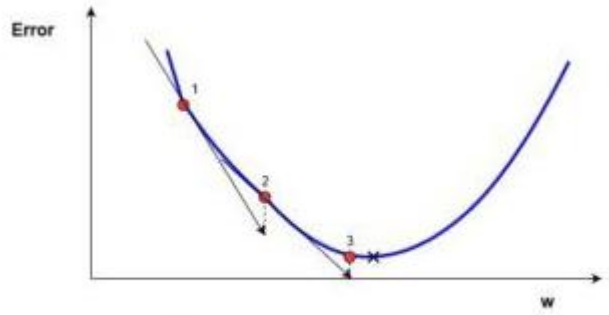
- We all know that Gradient Descent is an optimization technique used to minimize some function by iteratively shifting in the direction of steepest path as defined by the negative of the slope.
  Que:- But when do we use it?

  Ans:- Whenever we encounter a problem in which we do have to find out a minimum or maximum of a function Gradient Descent can be used. Mostly we use it to minimize cost or error functions.
  Que:- Why do we use gradient descent for linear regression?
  Ans:- The main reason is the computational complexity. In most of the cases, it is computationally cheaper(faster) to find the solution using gradient descent.
  Que:- Does it have any drawbacks?
  Ans:- Yes, quite a few, but they can be dealt with efficiently. Some of them are like if there are more than one minima in the graph, reaching global minima is not always true, careful selection of learning rate and the threshold is also very imp, and in some cases, it can be slow like in case of large datasets.

  The whole idea behind Gradient descent:-

  It requires a cost or an error function to be worked upon(to minimize it). There are many available corresponding to each machine learning algorithm like linear regression, SVM, NN etc. Minimizing any function means finding deepest valley in that function. This is exactly what our gradient descent works. It can also be used to find maximum of a function just by finding minimum of that function's inverse. Minimizing an error function increase the accuracy of our model. We do this by iterating over training dataset while tweaking the parameters(/weights and biases) of our model.

  Analogically this can be seen as, walking down into a valley. Actually, gradient descent is **a greedy approach** in which one always choose the best path to walk upon, which is the steepest downward direction. This may prevent us from reaching the global minima, which are somewhere no in our path. This depends upon our starting position.



Cost Function "J" plotted against one weight.

Source:- https://hackernoon.com/gradient-descent-aynk-7cbe95a778da

In linear regression we have associated weights for each independent variables. And dependent variable is assigned as the summation of those weights(**W**) * attribute values(**X**). And let the cost function we get is **J**. Here **J** is a function of these weights.

In principle, the error function is just for observing the error with the training examples while the derivative of the error function with respect to one weight is where we need to move that one weight in order to reduce the error for that training example. But we need to do this for each weight to update value of each weight.

Now, comes the **learning rate**. It is also called the **hyper-parameter**. This can be thought of as a "**step size in the right direction**", where right direction comes from **dJ/dW.**

Error function used in linear regression is **MSE(mean-squared error)**. MSE is given as:-

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)^2$$

Where **m** is total no of training examples.

The derivative of this with respect to any weight is :

$$\frac{\partial}{\partial\theta_j}\mathrm{MSE}\,(\theta) = \frac{2}{m}\sum_{i=1}^{m}(\theta^T \cdot \mathbf{x}^{(i)} - y^{(i)})\,x_j^{(i)}$$

Here j is the j_{th} weight, use for derivation.

So, in each iteration, we go over all the training examples and compute its MSE until it converges. We update weight like given below :

$$\text{Repeat until convergence } \{$$

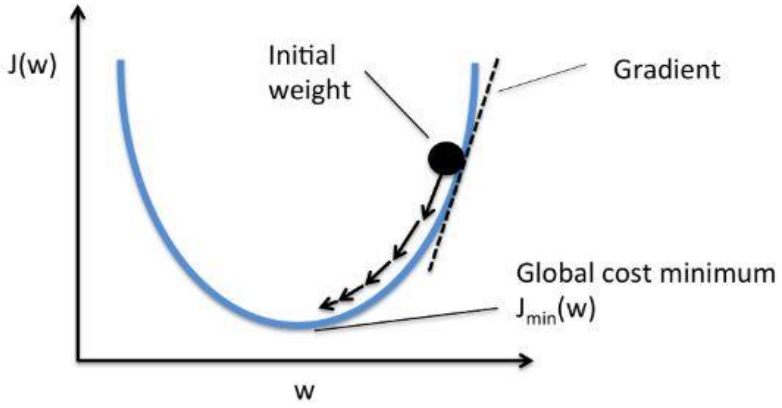
$$\theta_j \leftarrow \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$

$$\}$$

This converges if

1) difference between two consecutive calculated MSE is less than a threshold.

2) It surpasses suggested number of iterations



Source:- https://hackernoon.com/gradient-descent-aynk-7cbe95a778da

Depiction of how gradient descent works. In the beginning, the step size was largely due to the high value of slope, and then it keeps on decreasing as the slope decreases.

3. Implementation of Gradient descent on selected dataset(linear regression)

Since the dataset we are using has five columns, the **5th column is dependent** upon the first four columns. Since we are using linear regression, we can write predicted variable = sum(weight of a variable * value of a variable).

Y_pred = w0 + w1*x1+w2*x2*w3*x3+w4*x4

The pseudocode used for this implementation is:-
1. **Initialize** the weights *W* randomly.
2. Calculate the error value.
3. Calculate the gradients *G* of error function w.r.t parameters. This is done using partial differentiation: ***G = ∂J(W)/∂W.*** The value of the gradient *G* depends on the data, the current values of the weights, and the error function.
4.Update the weights as ***W = W – ηGX,*** Where *η –> learning rate.*
5. Repeat until any pre-defined **termination criteria** are met.

We, need to be very careful about this learning rate value**, High values of *learning rate* may overshoot the minimum**, and **shallow values will reach the minimum very slowly**. Moreover, one of the essential things is to **update the weights simultaneously.**

This algorithm is implemented in **"GD.R"** file attached to this folder.
For the checking, the correctness of accuracy we also used built-in library in R. On our dataset the **least MSE** we are getting using a linear model library is **100.2415**. Furthermore, through our gradient descent implementation model also, we are getting the **same MSE of 100.2415.**

4. Problems Identified:-

There are quite a few problems encountered in this simple implementation of Gradient descent.

Some of these are:-
1. The value of learning rate that is being accepted in the function was of the order 0.001. which is quite small and we required 1,000,000 number of iteration to get the best accuracy.

2. We have observed that the 3rd step in the above applied algorithm of gradient descent, to calculate G we need to traverse through the whole data(i.e. each and every row of input) in each iteration. So, if there are n number of iteration and there are m rows in our data then step 3 alone will take n*m amount of time which is huge if m is very high, which is generally the case.

5. Solutions Proposed:-

Hereby are the solutions proposed to the above problems are Identified:

1. For this problem we can have **normalized data** as input. This will significantly reduce the data values to the range of **-1 to +1**. Since, our independent variable's size decreases, we can increase our step size which is the learning rate. And hence, less number of iterations will be required.

2. To deal with this type of problem we have a special type of GD variation -> **Stochastic Gradient Descent.**
What **SGD** does is that, instead of traversing the whole data points, it traverse only one random data point, and each traverse is counted as one iteration. This forces us to increase the number of iterations by many folds. And in most of the cases it reduces accuracy a bit. But it's a **tradeoff between accuracy and time** and **SGD favors time**. Its not very good generalization but still it performs well.

6. Solution for 1:-

In order to process input data well, we normalized the independent variables.
The formula used to normalize data for any attribute:-

New_value = (Old_value - Average)/(Maximum - Minimum)

We manipulate the inputs here and the rest of the process of as same as the simple gradient descent.
The reason for doing this to allow bigger values of learning rate which will help in reaching the minima in minimum possible iterations.
This solution is implemented in the R language in file **"NGD.R"** attached with this file.

7. Solution for 2:-

We do have to apply **stochastic gradient descent** here.
The whole process here also is same as simple gradient descent except the part of updating the weights.
In GD at the time of updating of weights we find G, which uses all rows of data, but here we will only use only one row of dataset to calculate the same G.

We do this by randomly selecting any row and using it for calculating G. Therefore in this scenario: $\mathbf{W=W—\eta G_i X_i}$, where **i** is any randomly selected row.

The rest of the part is same as simple gradient descent.
This will make each iteration significantly faster but cause increase in number of iterations. This variation is introduced to compromise the accuracy for time.

This solution is implemented in R language in file **"SGD.R"** attached with this file.

8. Comparison of different variation of gradient descent.

Here comes the comparison part,

1. As stated earlier in our dataset, "GD.R" was not accepting any value of learning rate above the order of **0.001**, which was significantly reducing the speed of our program by increasing the number of iterations ~ **1,000,000**. However, after using normalized input data in the algorithm("**implemented in NGD.R"**) is accepting

learning rate value up to the order of **1**. Moreover, significantly reduce the number of iteration to the order of just **1,000**.

Note:- We made this comparison as both were producing the same **MSE** values of **100.2415**. This is the lowest MSE we can get, this is confirmed by using the library's linear model's MSE.

Hence we can conclude this solution helped significantly(**1000 times**) in term of time producing the same least possible MSE. The time got reduced as the number of iterations got reduced from **1,000,000** to **1,000**. And each iteration in both case take nearly equal amount of time.

2. As we know that each iteration of **"GD.R"** traverses through each row of the dataset, it will be quite a slow process. To tackle this problem, we introduced stochastic gradient descent(**implemented in "SGD.R"**). SGD traverse through only one row in each iteration, improving a notable amount of time. Since it is not a very good generalization, it will lack in accuracy always.

In **"GD.R"** learning rate was **0.001**, no. of iterations were **1,000,000** and accuracy obtained was **100.2415** which is as least as possible, but when we apply same constraints to **"SGD.R"**, the accuracy we obtained was **100.522** which is higher than **100.2415**. Now let us take a look on time taken by each.

Time taken by GD.R is on average 41.33 seconds.

Time taken by SGD.R is on average 18.48 seconds.

Here we can observe SGD.R significantly decreases the amount of time required by the algorithm.

We can observe the tradeoff between accuracy and time consumed.

Summary : GD -> time: 41 sec and MSE: 100.2415

SGD-> time: 18 sec and MSE: 100.522

It seems viable to go for **SGD** in this case as the accuracy we are sacrificing is very negligible, but the time we are saving is more than **60%**. So, it is a no brainer, and this solution is also perfect.