My notes on Regression

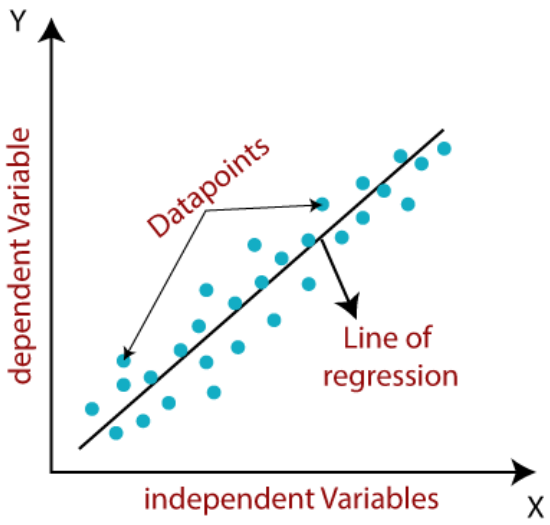
Supervised vs Unsupervised Machine Learning

Supervised Learning is when an input data is given to the model to train on and its task is to predict something when given new data . Eg. Regression , Classification

Unsupervised Learning is when the task of the model is to cluster the data so as to make it more informative . Eg Clustering , Summarisation

Linear Regression Model

Given an input data , a linear regression model tries to fit a straight line to it



As the name suggests , equation is given by

$$y = w * x + b$$

where ‘y’ is the dependent variable and ‘x’ is the independent variable.

The line of best fit is determined by minimizing the sum of squared errors between the predicted values and the actual values.

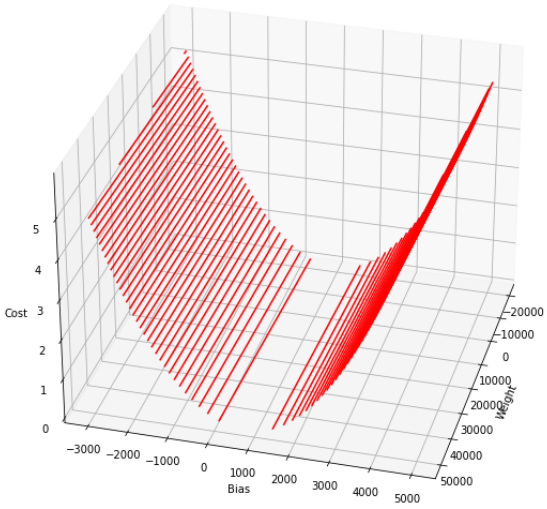
The loss function in this type of model is the “sum of squared error” given by

$$L(w,b) = \frac{1}{2m}\sum_{i=1}^{m}(f(x_i) - y_i)^2$$

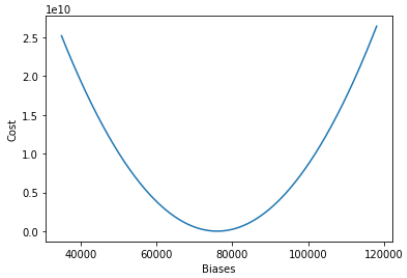
If we can select parameters , ‘w’ and ‘b’ that minimise the loss function , then that gives us the line of best fit . Using this line , we can make predictions for new dataset

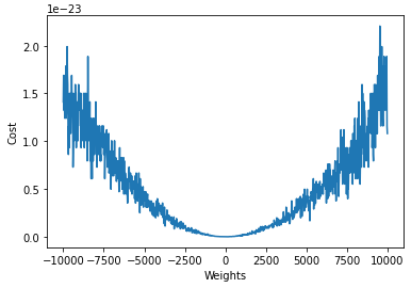
If we plot this loss function for different values of ‘w’ and ‘b’ , it turns out to be a 3D parabola or a “soup-bowl kind of shape” , and thus is convex

> A convex function is one which has only 1 global minima

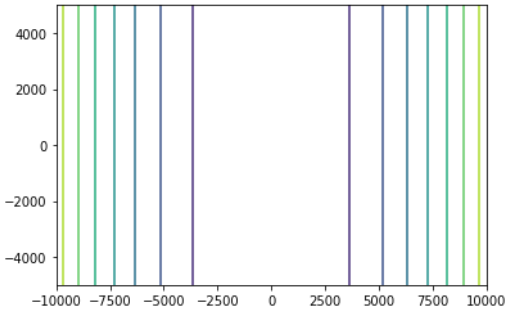


On Plotting individual (keeping only 1 non-zero) plots of the cost function with parameters ‘w’ and ‘b’ , We see parabolic shapes in graph





On plotting contour - plots of this loss function , we can see that this loss function can have same value for many values of ‘w’ and ‘b’ but the minimum value of the function lies at the centre of these curves



Now , how do we minimise the cost function by selecting appropriate parameters ‘w’ and ‘b’ ?

Here comes the **Gradient Descent** algorithm which helps us reduce the cost function by finding appropriate values of weights and biases

INTUITION - The algorithm finds the minimum of a function by iteratively adjusting its parameters in the direction of **steepest descent**. Imagine you are standing on a hill and want to get to its bottommost point (it has 1 bottommost point) , you will look around and move in the direction of steepest slope to do so

Steps

1. The algorithm starts with an initial guess for the weights and biases . Lets say (w,b)=(0,0)
2. Then , it calculates the gradient of the loss function with respect to each parameter

$$\frac{dL}{dw_j} = \frac{1}{m}\sum_{i=1}^{m}(f(x_i) - y_i)x_{i,j}$$

$$\frac{dL}{db} = \frac{1}{m}\sum_{i=1}^{m}(f(x_i) - y_i)$$

3. The algorithm then updates the parameters in the opposite direction of the gradient, taking into account a **learning rate (α)** that determines the step size of each update.

$$w_{n+1} = w_n - \alpha\frac{dL}{dw}$$

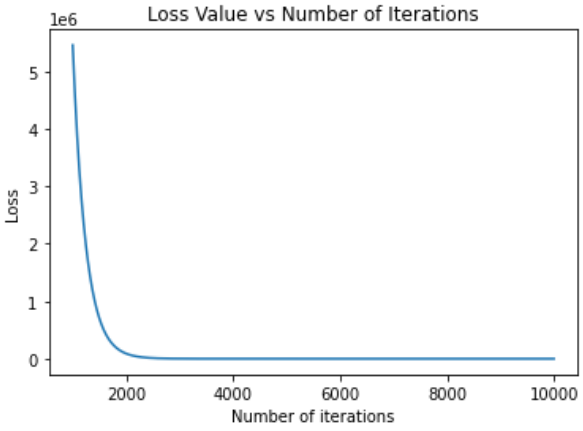
$$b_{n+1} = b_n - \alpha\frac{dL}{db}$$

This process is repeated until the algorithm converges to a minimum of the loss function.

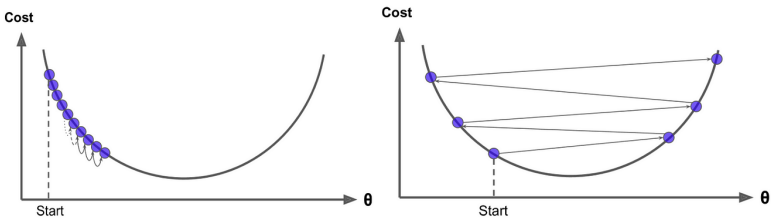
Now , there are 2 ways to check whether the algorithm should end or not

- Fix the number of iterations - The process stops after the desired number of iterations are completed
- Set a threshold - Compare the loss function values with its previous values and if the values decrease less than that value or start increasing for a set number of iterations , then stop the algorithm

If the code works well and the learning rate is accurately set , then the value of the loss function should decrease after each iteration and it should flatten out at the end depicting we have reached the end of gradient descent



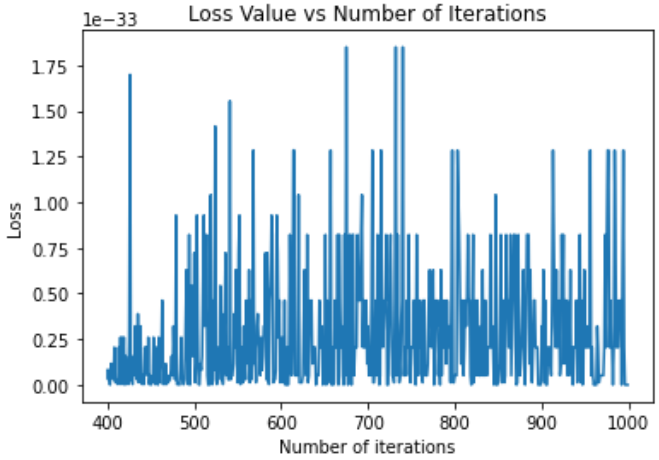
A note on Learning Rate



Learning rate too small Learning rate too high

A low learning rate will make the gradient descent work ok but it may take up a lot of time and also sometimes may not reach the bottommost point of the soup bowl ,

While if the Learning rate is too high , loss may actually start increasing because the values of parameters may keep oscillating and bouncing



Therefore , if the loss function curve doesn't flatten out or starts increasing , then you may wanna take a look at the learning rate of your gradient descent

> Here , i took example of a dataset with only 1 independent variable (called dimension) but for datasets with multiple independent variables , gradient descent algorithm works pretty fine with the only key point to be kept in mind that : **ALWAYS DO SIMULTANEOUS UPDATE FOR DIFFERENT WEIGHTS**

> Sometimes , gradient descent may take a lot of time to work due to various reasons like number of iterations , learning rate etc. but there is a way to make the code run faster - **VECTORISATION .** You can check out the code to see that all of it vectorised which makes the cpu to do parallel computation making gradient descent a lot faster

Handling Multiple Features in Regression

1. Feature Scaling

Real-world data is diverse and can vary greatly in magnitude. Using the data as is may result in parameter scales that also vary greatly, leading to a skinny contour plot.

> For eg. if we have 2 features , say , f1 and f2 with $1 < f_1 < 10$ and $1000 < f_2 < 10000$, then the parameters corresponding to them , w1 and w2 , will have great disparity in scale

A skinny contour plot means that the parameters will end up bouncing back and forth until they reach a minimum loss value making gradient descent as a whole , a lot slower.

Here is where feature scaling comes into play . It means scaling down the values of features so that they lie in a range very close to $[-1, 1]$. Here are some methods to do that

- **Mean Normalisation**

  We take the mean value of the features , lets say μ and scale the features such that

$$x_1 = \frac{x_1 - \mu}{max(X) - min(X)}$$

- **Z score Normalisation**

  This method makes use of the central limit theorem of converting a distribution into a normal random variable $N(0, 1)$. Here , we calculate the “standard deviation” of the distribution , lets say σ and convert features such that

$$x_1 = \frac{x_1 - \mu}{\sigma}$$

> The goal of feature scaling is to make the dataset scattered and centered around the origin within a range of [-1,1]

2. Feature Engineering

It is the process of selecting the right features for your model . It can be done by combining 2 or more features, creating additional features or omitting some features.

It can be done with the use of domain knowledge or something called ‘Mutual Inclusion Score’ of features

a) Mutual Information Score (Information Gain)

Given 2 random variables , X and Y , MI score measures the information that X and Y share . It is designed such that when X and Y are independent , then this score is 0 , else it is a non-negative number , mainly the entropy contained in X (or Y)

> the **entropy** of a random variable is the average level of "information", "surprise", or "uncertainty" inherent to the variable's possible outcomes. It is given by

$$H(X) = -\sum p(x) log(p(x))$$

Coming back to the MI score or Information gain , it is given by

$$I(X;Y) = \sum_x \sum_y P_{X,Y}(x,y) log\left(\frac{P_{X,Y}(x,y)}{P_X(x)P_Y(y)}\right)$$

But how does it relate with entropies ?

$$I(X:Y) = \sum_x \sum_y p(x,y) log(p(x,y)) - \sum_x log(p(x)) \sum_y p(x,y) - \sum_y log(p(y)) \sum_x p(x,y)$$

$$= -H(X,Y) - \sum_x log(p(x))p(x) - \sum_y log(p(y))p(y)$$

$$I(X,Y) = H(X) + H(Y) - H(X,Y)$$

Also , if we use the property that

$$p(x,y) = p(x)p(y/x)$$

We can easily prove that

$$I(X:Y) = H(Y) - H(Y/X) = H(X) - H(X/Y)$$

But this type of method works for discrete random variable or dataset having discrete values . For continuous random variables, summation is replaced by integration . For such datasets , one way is to discretise the dataset into a set number of intervals and then calculate mutual information score. Another way is to use the k-nearest neighbour method .

b) **Polynomial Regression**

Feature Engineering helps us fit not only straight lines but also curves to the data . One such method which comes into play here is called Polynomial Regression

Polynomial Regression is a type of regression analysis in which the relationship between the independent variable x and the dependent variable y is modeled as an nth degree polynomial.

$$y = b_0 + b_1 x + b_2 x^2 + ... + b_n x^n$$

where y is the dependent variable, x is the independent variable, b0, b1, b2, ..., bn are the coefficients, and n is the degree of the polynomial.

Higher degree polynomials can fit more complex curves to the data, but can also be more prone to overfitting.

Logistic Regression or Classification

Classification is a type of supervised learning where the goal is to predict a categorical output variable based on input variables. The output variable can have only a discrete set of values.

Here , we have to predict whether a set of input features belong to a certain category or not . Let , if they belong , be denoted as y=1 , else , y=0 .

Therefore , we want the output of our function to be between 0 and 1 . That is why , we use the sigmoid function , which outputs value between 0 and 1

Just like regression , we first form a linear equation

$$z = w * x + b$$

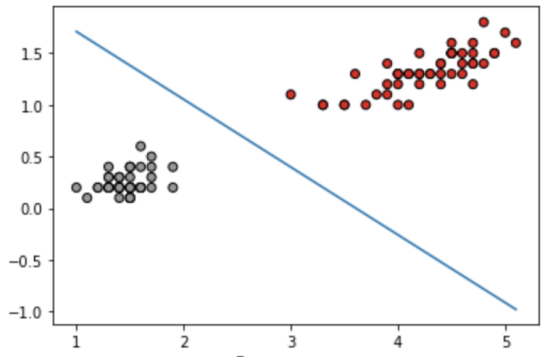
Then , we apply the sigmoid function such that

$$f_{w,b}(x) = \frac{1}{1 + e^{-z}}$$

Where this probability (f(x)) reaches 0.5

(Equal probability of y being 0 or 1) , i.e where $w * x + b = 0$, it forms something called decision boundary , to the right of which , we predict 1 and to the left of which we predict 0 .

Here , this decision boundary is a straight line

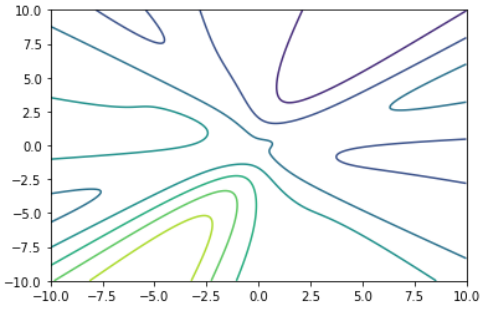


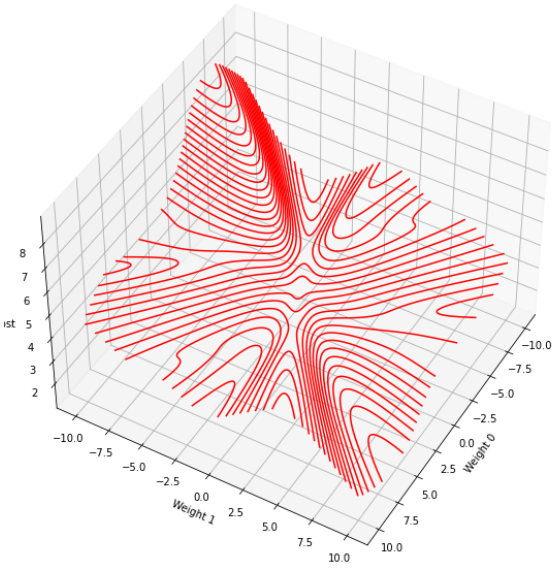
The decision boundary can be of various shaped depending on the kind of function used . For eg if a degree-1 polynomial is used , we have $w_0 * x_0 + w_1 * x_1 + b = 0$ which is a straight line

Loss function

Loss function used in Linear Regression (Mean squared Error) will not work here as the prediction equation is different here and if the MSE loss function is used , it will give us a non-convex loss function with respect to w and b

This is a non-convex contour plot of cost vs weights when the loss is taken to be mean squared



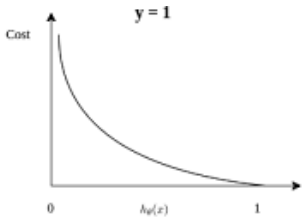


The loss function for logistic regression is determined for 2 separate cases

Case-1

When y=1 , we want to punish the function when it tends to predict the value 0 ,which we do by the term

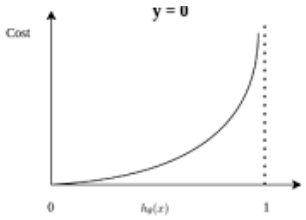
$$-\log(f_{w,b}(x_i))$$



Case - 2

When y=0 , we want to punish the function when it tends to predict the value 1 ,which we do by the term

$$-\log(1-f_{w,b}(x_i))$$

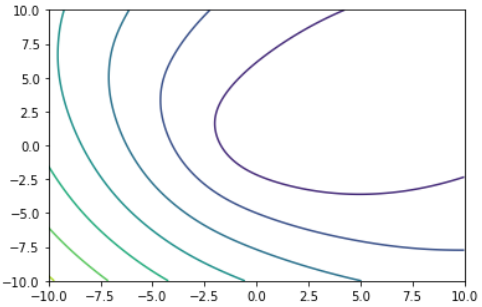


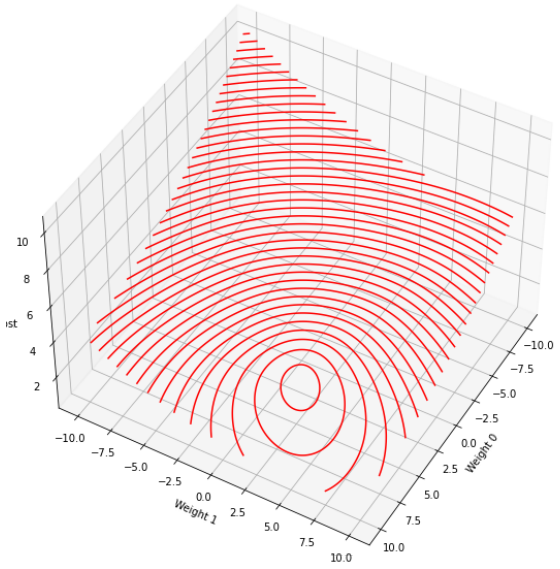
Thus , our complete loss function can be written as (In simplified form)

$$L(w,b) = -\frac{1}{m}\sum_{i=1}^{m}[y_i \log(f_{w,b}(x_i)) + (1-y_i)\log(1-f_{w,b}(x_i))]$$

where ‘m’ is the number of training examples, ‘y’ is the true label (0 or 1) and ‘f’ is the predicted probability of the positive class

When the loss function is taken to be as described above , contour plot of loss vs weights is convex





> We are so keen on making our loss function convex so that we can apply the gradient descent algorithm to this loss function to reach the global minima

Gradient Descent Algorithm

Gradient descent is a general algorithm and can be applied to all kinds of convex functions .

The only difference we have here is that the predicting function and loss function is different here . Therefore ,

$$\frac{dL}{dw_j} = -\frac{1}{m}\sum_{i=1}^{m} \frac{y_i.e^{-z_i}}{f_{w,b}(x_j).(1+e^{-z_i})^2}x_j - \frac{(1-y_i).e^{-z_i}}{(1-f_{w,b}(x_i).(1+e^{-z_i})^2)}(x_j)$$

$$= \frac{1}{m}\sum_{i=1}^{m} x_j \frac{e^{-z_i}}{(1+e^{-z_i})^2}\left(\frac{f_{w,b}(x_i)-y_i}{f_{w,b}(x_i)*(1-f_{w,b}(x_i))}\right)$$

$$f_{w,b}(x_i)*(1-f_{w,b}(x_i)) = \frac{e^{-z_i}}{(1+e^{-z_i})^2}$$

Therefore ,

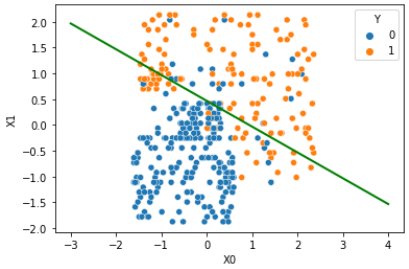
$$\frac{dL}{dw_j} = \frac{1}{m}\sum_{i=1}^{m} x_{i,j}\left(f_{w,b}(x_i) - y_i\right)$$

This is very similar to the descent in linear regression but keep in mind that here the prediction function is totally different

Using this , we can get the values of weights and biases and can therefore make predictions and also draw the optimum decision boundary

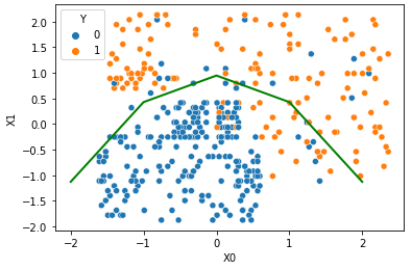
When used a degree-1 polynomial i.e

$$z = w_0 * x_0 + w_1 * x_1 + b$$



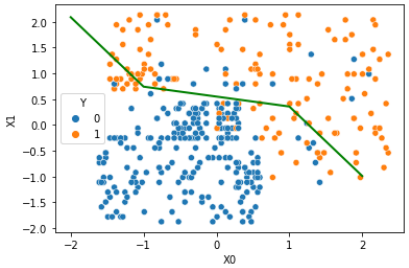
When used a different polynomial such that

$$z = w_0 * x_0^2 + w_1 * x_1 + b$$



When used another polynomial such that

$$z = w_0 * x_0^3 + w_1 * x_1 + b$$



Regularization

Sometimes , our model is overfit due to high coefficient values of certain terms which increases the complexity of the model and gives rise to high variation , though low bias . To handle this problem we add a regularization term to our model which prevents the coefficients from going unnecessarily high

Ridge - Regularization (L2-norm)

Here , we add a term which is proportional to the square of the weights to the loss term which penalises the coefficients at each iteration of gradient descent . Therefore , our loss function becomes

$$L(w,b) = \frac{1}{2m}\sum_{i=1}^{m}(f(x_i) - y_i)^2 + \frac{\lambda}{2m}\sum_{j=1}^{n} w_j^2$$

Where λ is the regularizing parameter

Therefore , during gradient descent , we have

$$\frac{dL}{dw_j} = \frac{1}{m}\sum_{i=1}^{m}(f(x_i) - y_i)x_{i,j} + \frac{\lambda}{m}w_j$$

The effect this type of change has is that , before every iteration of gradient descent , our coefficient is shrunk to a certain amount

$$w_j = w_j - \alpha\left(\frac{1}{m}\sum_{i=1}^{m}(f(x_i) - y_i)x_{i,j} + \frac{\lambda}{m}w_j\right)$$

$$= w_j\left(1 - \alpha\frac{\lambda}{m}\right) - regular\ update$$

Lasso - Regularization (L1 - norm)

Here , we add a term which is proportional to the absolute value of the weights to the loss term which penalises the coefficients at each iteration of gradient descent . Therefore , our loss function becomes

$$L(w,b) = \frac{1}{2m}\sum_{i=1}^{m}(f(x_i) - y_i)^2 + \frac{\lambda}{m}\sum_{j=1}^{n}|w_j|$$

Therefore , during gradient descent we have

$$\frac{dL}{dw_j} = \frac{1}{m}\sum_{i=1}^{m}(f(x_i) - y_i)x_{i,j} + \frac{\lambda}{m}\frac{|w_j|}{w_j}$$

> The difference between Lasso - Regularization and Ridge - Regularization is that , Ridge tries to make the coefficients close to 0 while Lasso attempts to make it exactly 0 . Therefore , Lasso works well for the data where a small number of parameters influence the result while the rest are very close to 0 . On

> the other hand , RIdge works well when many parameters have almost the same influence