Conceptual Understanding of Loss Functions and Optimization Strategies

Part 1

Error function plays an important role in training a model. Be it classification or regression, clustering or reinforced learning, algorithms learn the patterns from data by working on a error function. During training, the algorithms use the error function (a.k.a. Loss Function) to improvise their learning in every iteration. The term "learning" refers to the process of extracting the relation between predicted variable (Y) and predictor variables (X) or identification of natural groups in the given data (Clusters), or the logical steps need to be taken given the situation. In this blog, "learning" refers to extracting the relation between dependent variable Y and predictor variables X (supervised learning method). The end result of the training stage is a model that has learnt the true patterns reflecting the relation between X and y from the data. The model learning is represented in form of coefficients.

The purpose of error function is to help assess how well the model has learnt at every iteration. If value of error function at any iteration is greater than zero, this may indicate scope for further learning and becoming a better model. For this, the optimizer works on the error function to modify the coefficients in every iteration and in the process, drive the model towards perfection. On completion of training stage, the values of the coefficients represent the relation between y and X. Let us call them the optimal coefficients values. Thus, the error function and optimizer play an important role in training a model.

The error function can be represented by different mathematical functions and there are different types of optimizers that can be used to train a model. For example, in regression one can use SSE (Sum of Squared Error) function or MAE (Mean Absolute Error) function along with Gradient Descent optimization algorithm to find the best fit plane.

In this blog we will discuss the error functions which are mathematically called convex functions and the gradient descent optimizer along with its variants which work on the convex error function during the training process.

.

Error Function

In ordinary terms, EF is a mathematical formula that calculates the difference between what a model is expected to predict (ground truth) and what it predicts given the input data. SSE (Sum of Squared Error) is one of the most popular ways of representing the error because mathematics guarantees that such functions will be convex (discussed below in "Squared Error Functions") in form. This is a very important characteristic that helps in training models efficiently.

The relation between target variable (y) and input variables (X) can be expressed in multiple ways such as a mathematical formula (Linear Regression), in form of rules (Decision Trees) and likelihood ratios (Naïve Bayes). Whichever way it is represented, the following discussion is relevant in all cases. We will take the Linear Regression way to discuss the concepts.

In Linear Regression, the dependency of the target variable (y) on the independent variables (X) is represented as

$$Y^{\wedge} = w_1X_1 + w_2X_2 + w_3X_3 + ... + b$$

The list of coefficients (w_1, w_2, w_3, ... b) for a trained model represent pattern of relation between the y and X variables learnt from the data. They are called the coefficients of the model a.k.a. model parameters (MP).

During the training iterations, the first iteration begins with random values assigned to the coefficients and the random values almost never represent the patterns in the data. When they are used to estimate y^ through the formula, the output (y^) for the given X is never the same as expected value (y). The difference between y^ and expected y is expressed as squared error function.

1. Sum of Squared Error Function

The Sum of squared error function (SSE) is a quadratic function is expressed as -

$$\mathbf{SSE = \sum^{n}_{i} (y_i - y_i^{\wedge})^2}$$

i - index number of the data points in the given dataset
n - number of data points/ records/instances in the given dataset
y_i – Ground truth for X_i
y^- predicted value of X_i
The position of y_i and y_i^ can be interchanged as the difference is squared

Since y_i^ (predicted value) is the result of WX_i + b, The SSE can also be expressed as

$$\mathbf{SSE = \sum^{n}_{i} (y_i - (WX_i + b)^2} \text{ given } y_i^{\wedge} = WX_i + b$$

"W" and "b" represent all the coefficients and X represents all the predictor variables in the data. There will be as many elements in W as in X.

In this expression of SSE (y_i, X_i) represent an instance /record /data point with index i and both are given to us as training data. Which means we can influence SSE only by modifying values of W and b. Thus, we can express SSE as a function of W and b. SSE = f (W, b). Before we get there, mathematically, a quadratic function (output = input raised to power 2), is guaranteed to have a parabolic shape when plotted with input varying from -∞ to + ∞. The output will always be positive.

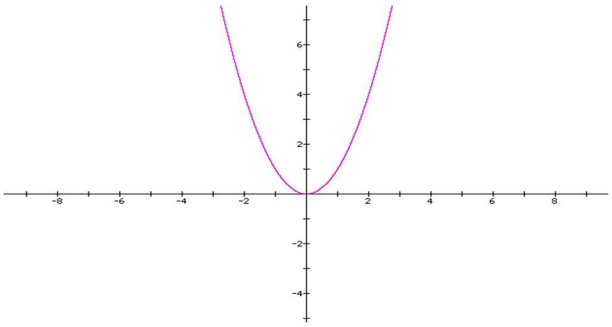

Fig 1

The function may be shifted horizontally or lifted up depending on the way the function is expressed but in all the cases, it will look parabolic. Such functions are strictly convex functions (f'' (W) >= 0 for all values of input W). There is only one point in the input W where the SSE is the least, represented by the lowest point in the curve (a.k.a. global minima). For training models efficiently, this is a desirable property of the error function. Given any random W, the optimizer will find the right value of W that minimizes SSE during the training iterations. For this it uses a mathematical concept called the gradient of a function at a point and decide what the value of W should be.

2. SSE as a Function of W,b and its Gradient

SSE, when plotted against W, b, should theoretically look like a bowl as shown below. The vertical axis E represents error expressed as SSE.

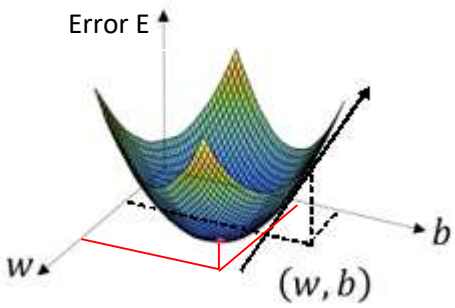

Fig 2

Gradient of a function refers to the rate of increase of the output of the function at a given input value. For the SSE function, gradient for a particular value of W and b is the increase in SSE for a unit change in W and b.

The gradient at any point in this function is represented by a vector (arrow) which geometrically is tangent to the function i.e. touches the function and does not go thru it at that point. In the figure above, the black arrow represents the gradient (direction of increase in error) at a specific point

(W,b). The length of the arrow represents the magnitude of increase in SSE for a unit change in W and b.

Thus, gradient vector reflects both the magnitude of increase and the direction of increase of the output i.e. the direction of change of the input W and b that results in increase in SSE.

The lowest point in the SSE function (represented by red dot) gives us the combination of optimal W and b (shown in red lines) which together give us the best fit model with lowest SSE. Lowest SSE usually is higher than zero.

Note: In the figure above, SSE is shown as a function of two variables W and b. This is only to explain the concepts of convexity. In most of the data SSE will be a function of multiple coefficient variables. Though we cannot visualize the function, the mathematical properties remain intact in the higher dimensions just as they are in the three dimensions discussed here.

The optimizer applies a mathematical technique called partial derivatives to estimate the gradient at a point (combination of W and b). Gradient at a point is the direction of increase in SSE with a change in W and b at that point. The optimizer uses the gradient to drive the model towards the optimal combination of the coefficients W and b.

In this two-dimensional figure it will be obvious that W and b have to move in the direction of the origin to find the least error combination but in higher dimensions where W represents more than two coefficients, it will not be so obvious. This optimization method is called the gradient descent as it will take the model against the gradient at every point except at the minima where gradient will be 0.

3. Gradient Descent

All algorithms start building the models with the coefficients initialized with some random values which are always very different from the optimal value. As a result, the resulting model will do erroneous predictions. Let the training error at the start of the training be E. This could be any point on the error function, for example as shown below (W,b).

The objective of training is to move from the current position on the EF to the optimal position ($W_{optimal}$, $b_{optimal}$) , shown as red dot, which will result in the lowest error E.

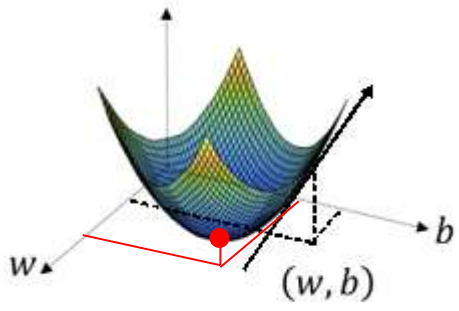

Fig 3

The challenge is, we do not have a prior knowledge of how the error function looks. We can get a holistic view of the error function only when we have evaluated all combinations of the W and b. There are infinite combinations! This means when we start the training, we are in the following situation.

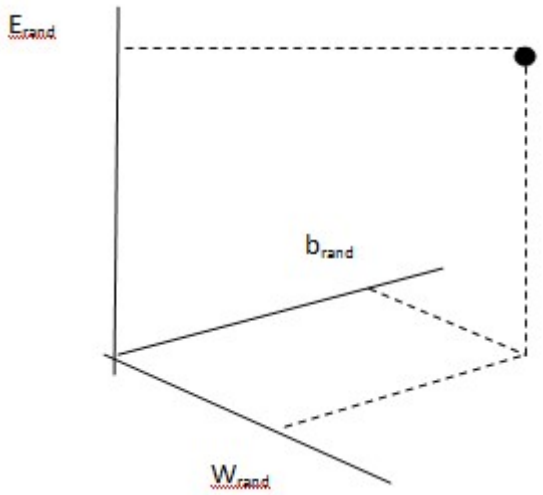

Fig 4

We don't have a prior information on where the minima of the error function is. This is often compared to an imaginary situation where you are on top of a hill and want to reach the valley. You are surrounded by thick fog and as a result cannot see your own feet, forget the valley. One way to find the path down to the valley is to continuously feel the ground with your feet one direction at a time and move in the direction where there is a drop (hope to reach the valley with all bones intact!)

This process of feeling the ground around you in one direction at a time is similar to concept of partial derivatives in mathematics. Using this mathematical technique, the optimizer evaluates how the invisible error function behaves when we change the W and b a small bit one at a time. For e.g. what happens when we replace W with (W + ΔW) or with (W - ΔW). **We note down the change where the error increases and by how much**. This is the gradient of the error function at a point with respect to given W. Similarly, we find the gradient of the error function at the given point with respect to b. The cumulative effect of the two gradients is the overall gradient of the function at that point.

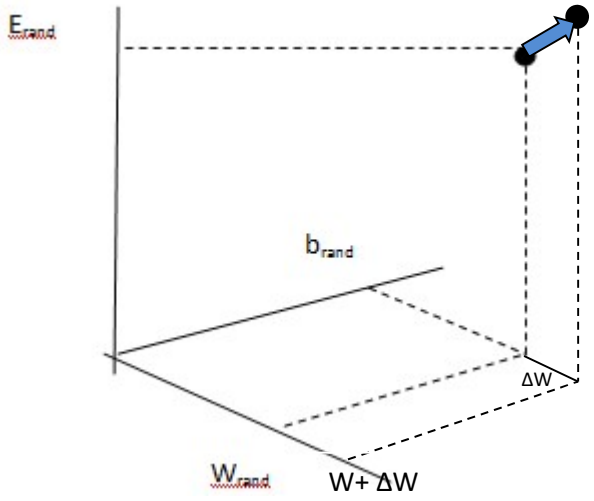

Fig 5

In the figure above, a small (Δ) increase in W, keeping b constant, pushes the error up (blue arrow). This is the gradient of the error function with respect to W at the location (W_{rand}, b_{rand})

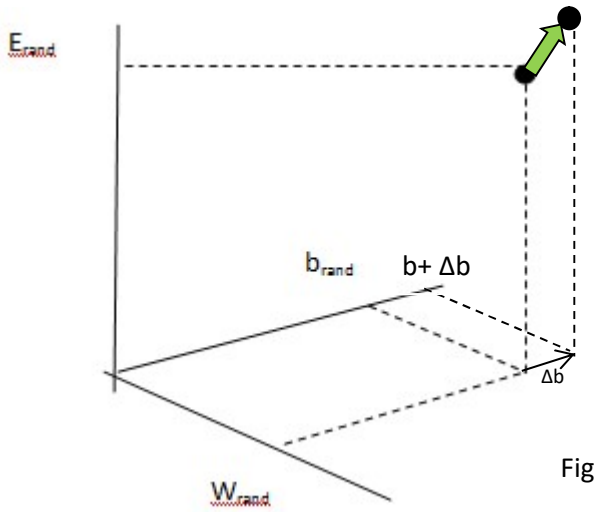

Fig 6

In figure 'B', for a small increase in b, error increases. The green arrow is the gradient of the error function with respect to b keeping W constant at (W_{rand}, brand).

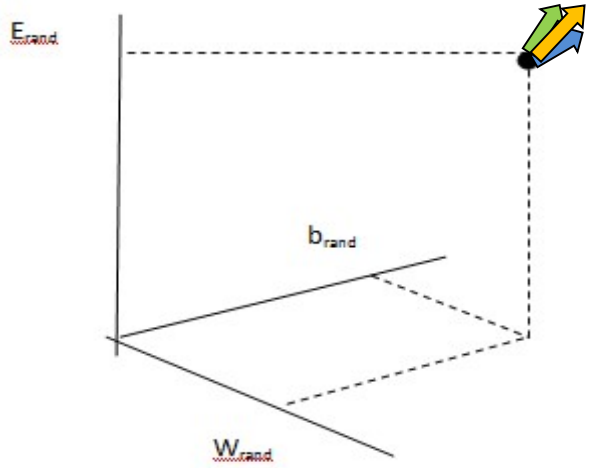

Fig 7

Thus, the cumulative effect of modifying both W and b by Δ will be represented by the orange arrow that results from vector addition of the green and the blue arrow. But we want to reduce error...we want to go towards the lowest point in the invisible error function... If the Orange arrow (gradient at current (W,b) points in the direction of increase of error, we have to go bang opposite i.e. flip the orange arrow. That is the direction in which the minimum error point is likely to be (cannot be sure whether it is or not).

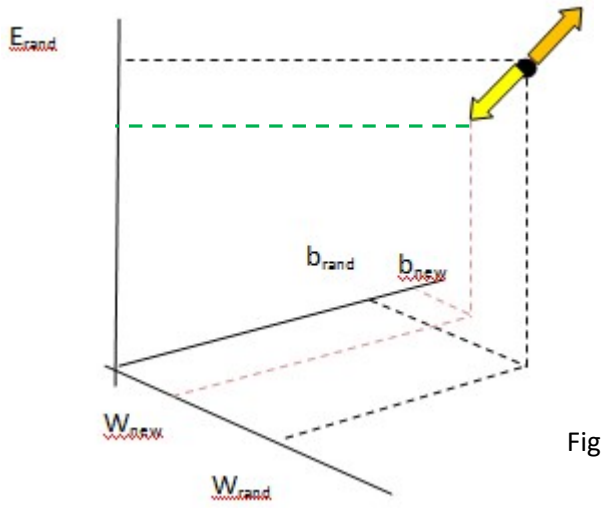

Fig 8

The anti-gradient vector (shown in yellow) points to new W and b combination which has lesser error (shown in green dash line) than the previous combination of W and b resulted. Continuing the iterations, the same set of steps (partial derivative calculations) is repeated in every iteration, driving the model towards the $W_{optimal}$, $b_{optimal.}$

Since the change is in opposite direction of the gradient at a point, this approach is called **gradient descent** (we try to descend down to the minima). Do we have gradient ascent also? Yes, we have and that is not useful in the current context of this discussion and will leave it out of scope.

Given that mathematics guarantees convex functions to have only one global minimum, does this mean the optimizer using gradient descent will reach the global minima i.e. will it be able to find the $W_{optimal}$, $b_{optimal}$ every time it starts from some random W and b? The answer is in negative.

There are two major reasons for this…

1. The optimizer may oscillate around the minima. In some cases, even end up increasing the error!
2. Given the data distribution, the convex error function may not be strictly convex i.e. may not be the smooth continuous function with one absolute minimum as shown in theory

As a result, the optimizer may either fail to find the optimal coefficient values irrespective of the number of iterations or end up in a sub-optimal combination of coefficients and stop the training. Let us explore these two reasons in detail.

4. Oscillations and Learning Rate

When does optimizer oscillate around the minima? This happens when the descent step takes algorithm to wrong side of the minima

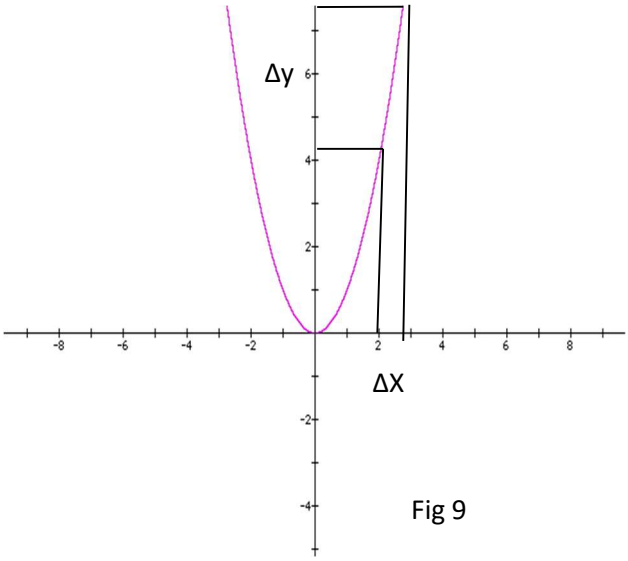

Fig 9

In the diagram above, small change in X (ΔX) results in a relatively large change in y (Δy). Thus the gradient at the point X,y = (2,4.5) is a vector of large size and let us assume it is the black arrow which is always tangent to the function at the point.

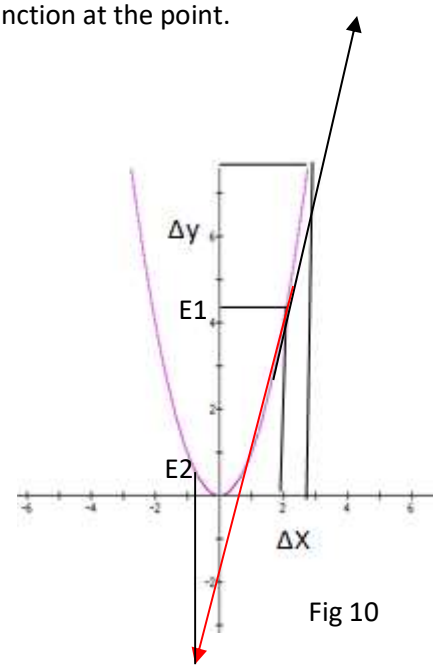

Fig 10

Since objective is to descend, the gradient vector is flipped resulting in the red vector. When we move (modify the coefficients) to the location of the head of the red arrow, we end up on the wrong side of the minima! Though the error is reduced, compared to the previous error, it has jumped over to the other side. This kind of jumping over the minima can happen multiple times given the gradient characteristics of the error function.

To prevent such oscillations around the minima (thus wasting precious training resources), the gradient is scaled down using learning rate (LR) to dampen the oscillations. LR is a hyper parameter that one needs to fix and when fixed with the right value, will have the effect of speeding up the training due to suppression of the oscillations. LR controls the magnitude of the descent on the error function and tries to keep the movement close to the error function. If we connect all the combinations of the parameters that the optimizer goes thru (especially in the vicinity of the global

minima), the path will approximate the shape of the error function. LR helps the optimizer to approximate EF as much as possible.

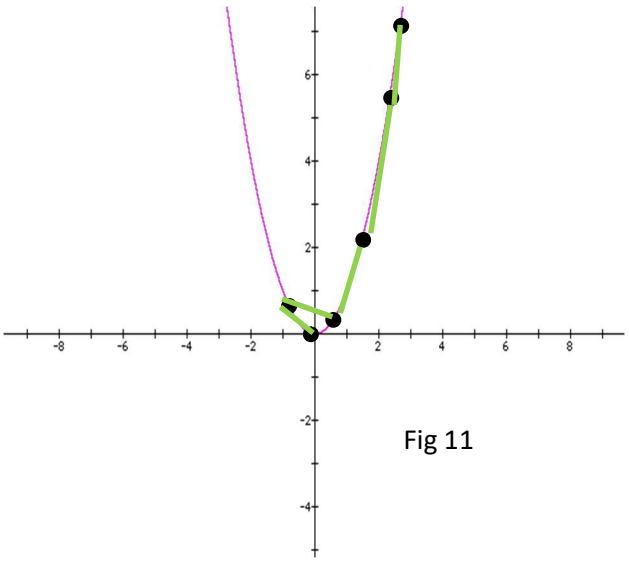

Fig 11

Though Learning Rate dampens the oscillations, it does not completely remove them. The optimizer can still oscillate around the optimal coefficient but the magnitude of the oscillations will be much less than it would without Learning Rate. Instead of having a constant learning rate since the beginning of the training iteration, will not reducing the learning rate as we inch towards the minima help? It will. For that we use a concept of decay. The LR decays with iterations and as a result, as the optimizer moves the coefficients towards the optimal values, it reduces the learning rate by a pre-fixed decay factor according to a predefined schedule. However, set incorrectly, use of decay factor may make the training extremely slow. We will deal with the various techniques developed to control the LR in part 2 of this article.

5. Impact of Data Distribution on Convexity

To understand the impact of data distribution on convexity, one needs to know that the use of convex cost function does not guarantee a convex problem. Cost functions such as cross entropy, absolute loss, least squares are mathematically convex, however, when we employ them in the context of a data set and algorithm, the process of finding the optimal coefficients may not be convex.

For e.g. SSE function is mathematically a strong convex function but when applied in Logistic Regression ends up being non convex! There are many sources of information on this for e.g. https://stats.stackexchange.com/questions/267400/logistic-regression-cost-surface-not-convex

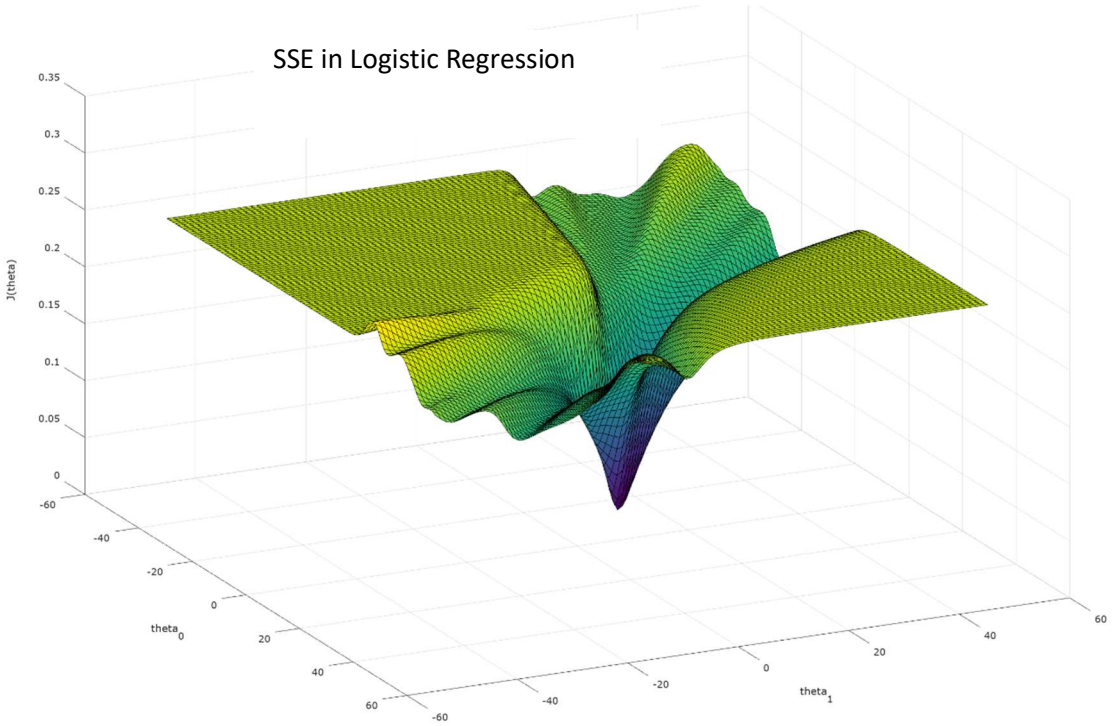

Fig 12

Image source : https://stats.stackexchange.com/questions/267400/logistic-regression-cost-surface-not-convex

To understand how the distribution characteristics of data can influence the shape of the error function, let us take binary classification. Objective is to find the threshold that does the perfect classification with zero errors. Let the error function be Log-Loss. Theoretically, this function is continuous, smooth and monotonically decreasing as shown below for positive class.

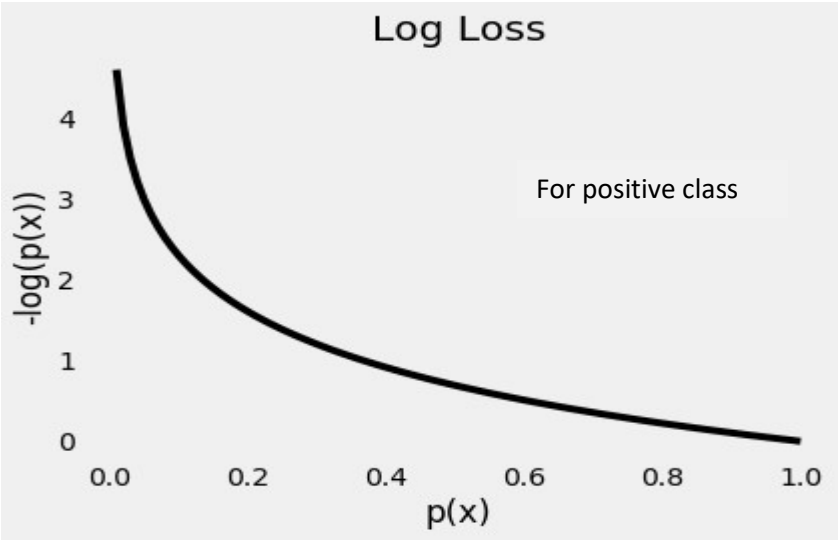

Fig 13

Source: https://towardsdatascience.com/understanding-binary-cross-entropy-log-loss-a-visual-explanation-a3ac6025181a

When a model predicts a data point to belong to positive class (predicted probability is closer to 1) and the data point actually belongs to positive class, the contribution to error will be close to zero.

On the other hand, if the model predicts a data point to belong to negative class (probability of belonging to positive class is less than .5 and closer to zero) for a point which actually positive, contribution to error will be large. Similar diagram can be drawn for negative class. In both the cases, the error function will in theory be smooth continuous and monotonically decreasing.

However, in real datasets where classes are not linearly separable, the loss function on the given data may not go down to a global minimum. For e.g., following dataset (XOR pattern) is not linearly separable in terms of the two classes.

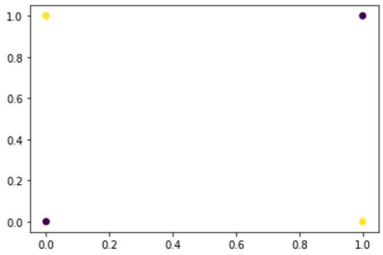

Fig 14

When we run a Logistic Regression on this with Binary Cross Entropy loss, it gives the same score (max of .75) for various combinations of the coefficients ([0.60304732 -1.1151906], [-0.37243906 -0.55488681], [-0.50994143 0.40254018]). Obviously, being linear in nature, Logistic Regression will not be able to separate the two classes and the best it can do is classify three out of four points properly (3/4 = .75). It is able to achieve this best possible score with multiple combination of coefficients as shown. Which indicates that the Cross-Entropy loss function, which is convex in nature, in the context of the given data is not strictly convex. There are multiple minima of the error function that give same accuracy score. More generically, the error function could have multiple equal and unequal minima.

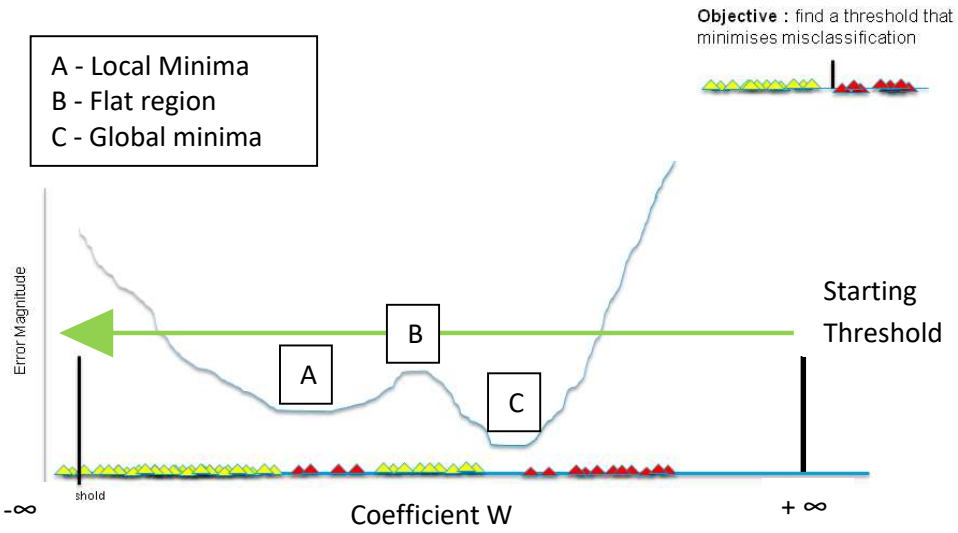

Fig 15

In the diagram above, the red class is the positive class while the yellow is the negative class. To separate the classes, we put a vertical boundary (black vertical line) that dichotomizes the space i.e. (represents the .5 threshold of the Sigmoid function which is not shown). One section of the space for negative class (left side of the threshold in the example) and other section for positive class. The position of the threshold in the feature space is decided by the position and the orientation of the

Sigmoid curve which in itself is a function of the coefficients of the linear function. When we modify the coefficients of the linear function (including the bias), the Sigmoid function reorients which includes change in slope, and lateral movement. This is beautifully demonstrated at the following site http://neuralnetworksanddeeplearning.com/chap4.html .

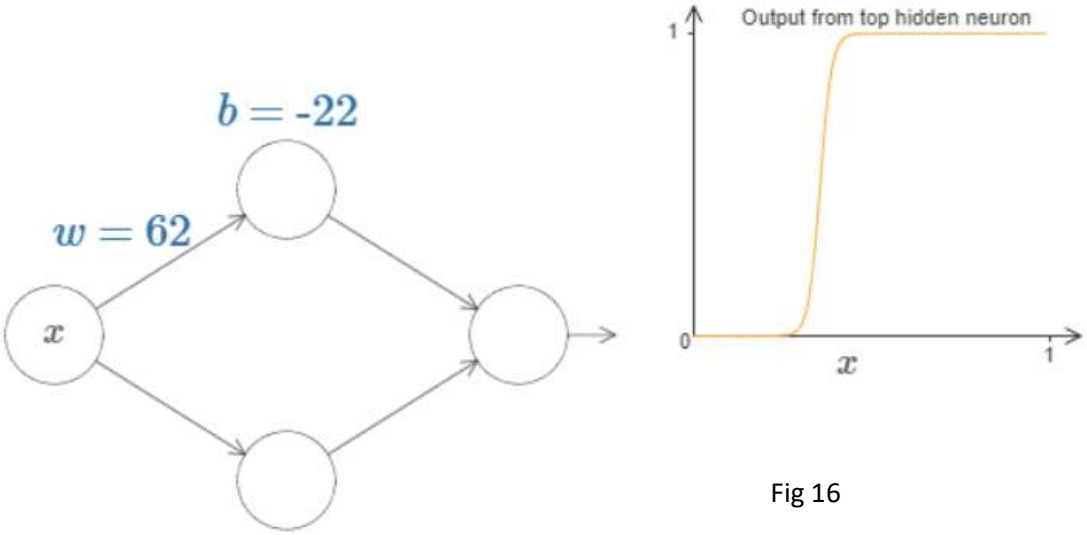

Fig 16

Image source: http://neuralnetworksanddeeplearning.com/chap4.html

Let the threshold shift from extreme right to left (as shown by the arrow in Fig 15) by modifying the coefficients thru gradient descent during the training. As a result of the starting threshold (result of some random coefficient values), all positive classes are misclassified as negative class and hence the error is very high in the beginning.

As the threshold moves due to gradient descent based modification of the coefficients, the error falls down because some of the positive class data points (on the extreme right) are classified correctly as positive with the shift in threshold. However, further shift to the left misclassify negative class data points (red ones in the centre) as positive and hence error goes up. As a result of this, the EF does not resemble the theoretical smooth continuous convex function. The error function has regions like local minima, flat regions, saddle points (not shown) and a global minimum. In the context of these deformities, the optimizer algorithm (OL) has to discover the global minima, the threshold at which the SSE is the least.

While evaluating the error at different combination of coefficients, if the optimizer gets into flat regions of the error surface where the gradient is almost nil (≈ 0) and it is multiplied with learning rate (a fraction) that will result in a very small value and as a result $W_{new} \approx W_{old}$.

$$W_{new} = W_{old} - \eta * \nabla J(w)$$

$\nabla J(w)$ represent gradient of the error function (J) w.r.t. coefficient W at the point. On flat regions of the error function the gradient at a point is zero. That $W_{new} \approx W_{old,}$ the learning slows down to almost a halt. Thus the model slows down or even stop learning. On the other hand, if the optimizer gets into a local minima region, it will find the error gradient at that point to be zero and the error

gradient around this point higher than zero and stop any further training as it would believe this is the absolute minima where the error gradient is zero.

Similarly, in Regression case, the SSE error function may not be strictly convex given the data. Which means there can be multiple solutions to the given regression problem. For e.g., the given data below has multiple solutions.... Ref: https://stats.stackexchange.com/questions/144080/can-there-be-multiple-local-optimum-solutions-when-we-solve-a-linear-regression

X (Independent variable)	Y (Independent Variable)	Z (Dependent Variable)
1	-1	0
2	-2	-1
3	-3	-2

Table 1

Z = 1 - 1X + 0Y (bias = 1, W1 = -1 , W2 = 0)
Z = 1 + 1y + 0X (bias = 1 , W1 = 0 , W2 = 1)
Z = 1 - 0.5y + 0.5Y (bias = 1, W1 = -.5, W2 = .5)

All three equations are best fit for the given data. The effect of the three set of coefficients is shown below in XL. All three are best fit as the error across the four data points is 0 in all the cases.

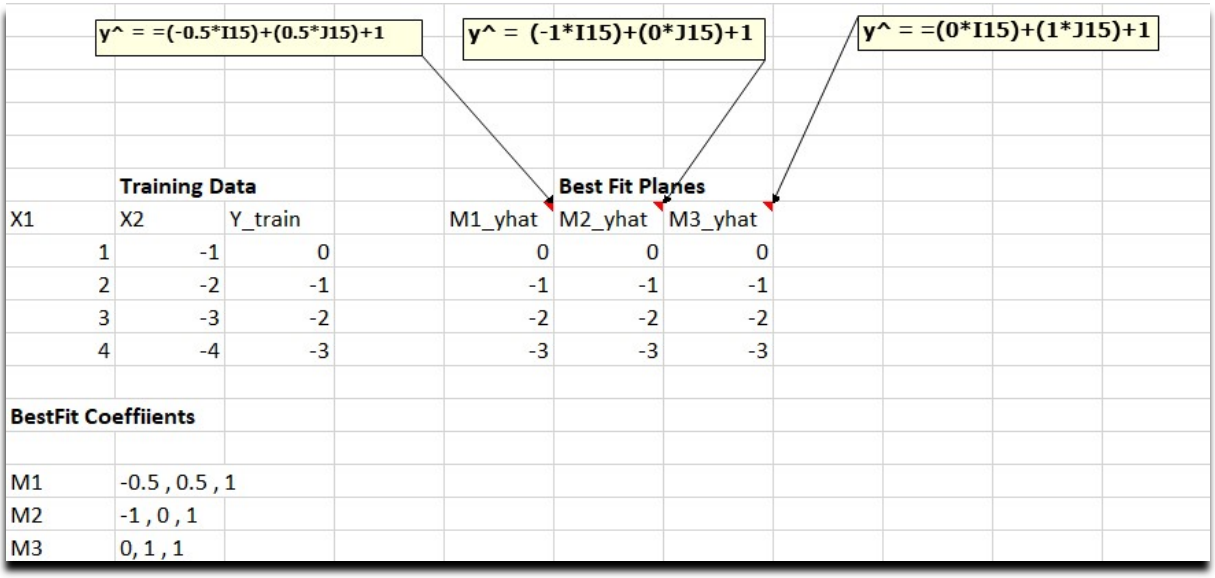

Fig 17

If there are multiple sets of coefficients giving the best fit that means the plot of Error Vs coefficients has multiple minima. Thus, we have a case of convex but not strictly convex error function in the context of the given data and the algorithm.

To add to the challenge of SSE based approach not being strictly convex, the gradient descent algorithm itself is not designed to overcome the challenges. For e.g. It cannot distinguish between a minima and global minimum; it will crawl on flat regions of the error function. It will oscillate in

saddle points, it may grind to a halt in flat regions. All this put together makes training models difficult even when the error function is convex. This difficulty becomes apparent when working with deep neural networks.

In Part 2 we will discuss these challenges in the context of deep neural networks and how the various optimizer algorithms try to address the same. However, to understand all that, we first need to understand the error function in terms of contour graph.

6. Contour Graph for Error Function

Though discussing gradient descent using the 3D and 2D graphs of the error function is an easy way of explaining the concept of gradient descent, it is not the suitable to discuss the challenges that optimizer faces while working on the non-convex situations. For that, we need to understand the Contour Graph.

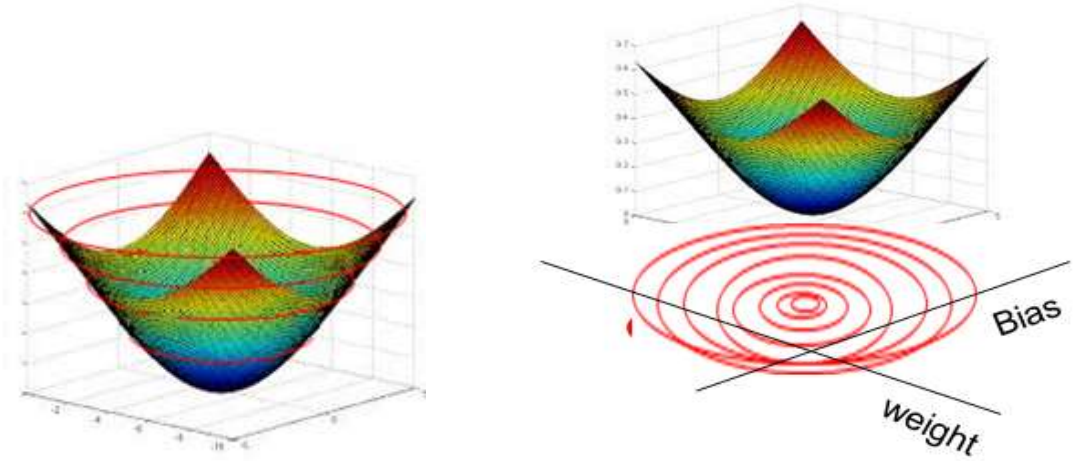

Fig 18

Let us look at a strictly convex error function in 3d where the vertical axis is measure of error and the two horizontal axes represent all the coefficients in the model. In the first picture above, each ring in the error function represents a combination of coefficients that lead to same error. The larger the ring, higher the error. The innermost ring (a point) represents the global minima of the error function. Gradient at a point other than the global minimum, represents the direction of next outer ring. The gap between rings is a fixed quantum increase in error. In the second picture, where these rings are dropped on to the floor (input space of coefficients), we get he contour graph of the EF. In contour graph of the EF, we do not need the third dimensions as shown in the picture below.

In every training iteration the optimizer will drive the coefficients to values towards innermost ring where error will be the lowest (not necessarily zero). When the EF is strictly convex, the contour graph rings are concentric and symmetric.

All the optimizer needs to do is, from whatever current combination of coefficients is, find the gradient and go opposite to the gradient. Given they symmetry of the contour graph, this move will take the optimizer closer to the bull's eye in the graph as shown in Fig 19.

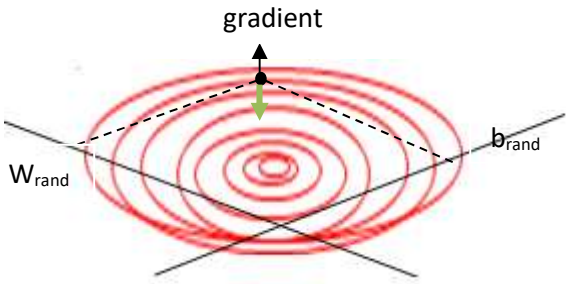

Fig 19

The concentric rings will be tightly packed for sharp parabolic error functions and loosely packed not-so-sharp parabolic error functions as shown below.

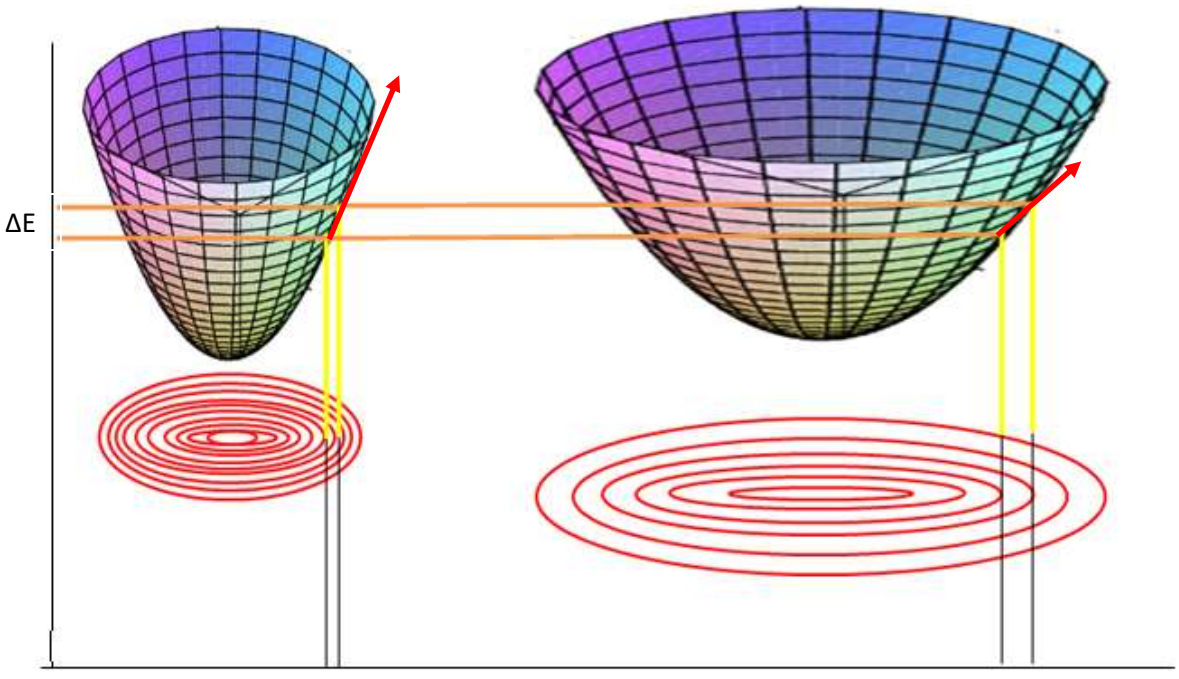

Fig 20

Note: In the fig 20 above, only one parameter (X) is shown though the contour graph is two dimensional. The other parameter is not shown to keep the drawing simple. The Error dimensions is suppressed and it's values are represented by the size of the rings.

In the figure above, the ΔE/ΔX for first error function is relatively larger than for the second error function. This fact is reflected in form of the tightness of the rings which is higher in the first case relative to second. This is also represented by the steep and large gradient vector (red arrow) in the first case than second.

Outer rings represent higher error than the inner rings. From a given position on the graph, direction of the gradient with be the nearest next larger circle and the gradient vector will be perpendicular to the current ring. The length of the gradient vector will be decided by the ration ΔE/ΔX, not by the distance between the rings. For e.g. if in the first figure above, ΔE = 1 and ΔX=.5 then gradient magnitude is 1/.5 = 2 while in the second case where ΔE = 1 and assume ΔX = 1, then gradient magnitude is 1/1 =1.

In some situations, the contour graph of the error function may have concentric circles but not necessarily symmetric. For e.g. in the diagram below

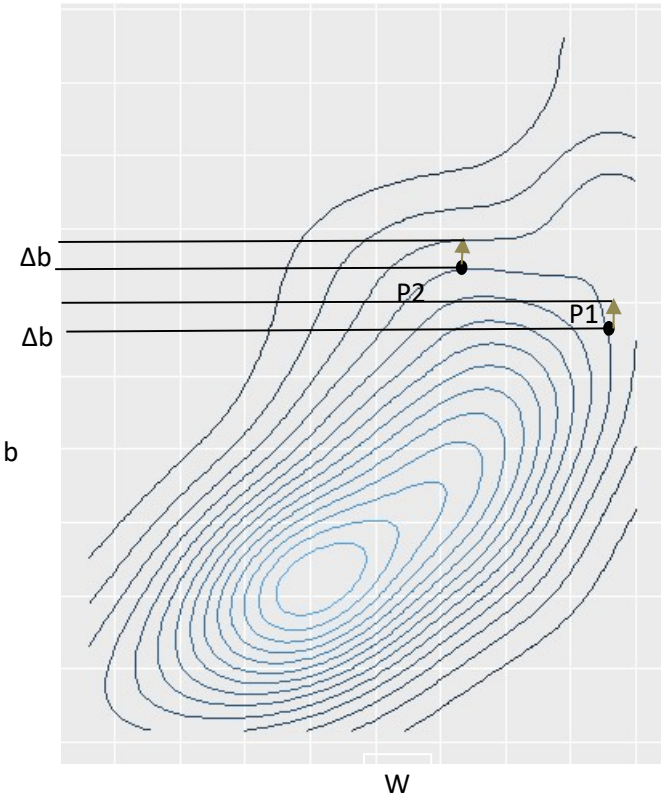

Fig 21

Image Source: R Data Visualization Recipes, Vitor Bianchi Lanzetta, Packt publications

If we compare point P1 and P2 on the contour graph, at P1, keeping w constant, Δb leads to change in error $\Delta E1$. At point P2, keeping w constant same Δb increases error by $\Delta E2$. Change Δb in P2 leads us to the next outer error ring whereas, the same Δb in P1 leads to relatively small increase in error (much less than the next outer ring). **The two arrows are not gradient vectors**. They are only to show the shift in w,b combination. Thus, in a contour graph, when rings are tightly packed, it represents a sharp increase in gradient in that region and when the rings are loosely packed as in case of point P1, the gradient is relatively small in that region. This makes the contour graph circles concentric (error circles will never touch or cut each other) but the symmetry is lost. This can result in the optimizer move haphazardly over the contour graph during the training instead of smoothly sailing towards the bull's eye!

Given the impact of data distribution on the convexity of the error function, the error functions may contain other deformities such as local minima, flat regions and saddle points which result in weird shapes in terms of the contour graph as shown below.

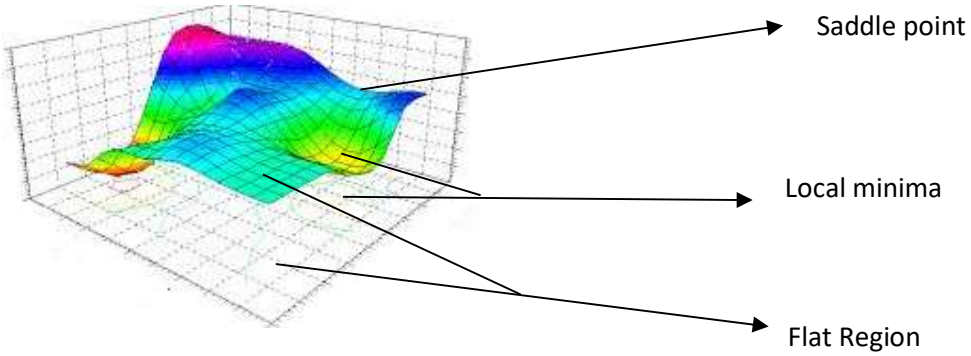

Fig 22

Image Source: Cfd-online.com

In the above diagram there are multiple minima (dip in the surface) and the corresponding contour graph with deformed error rings. Also notice the flat region where the corresponding contour rings are widely separated. Saddle point, the region where in one direction error goes up and in other it goes down.

During the training, if the optimizer reaches a combination of W, and b which is in the flat region of the error function, the training (modification to the W and b towards optimal values) will become very slow as the gradient in this region is close to zero. Instead, if the combination of W,b lead the algorithm to the local minima, the training will stop because any combination of coefficients around the local minima will only lead to higher error.

In Part 2 we will discuss how variants of gradient descent-based optimizer try to address these challenges and attempt to make training efficient and successful. The discussion will be in context for Deep Neural Networks.

References:

1. https://stackoverflow.com/questions/39943968/local-and-global-minima-of-the-cost-function-in-logistic-regression
2. https://arxiv.org/pdf/1909.12830.pdf
3. https://stats.stackexchange.com/questions/73165/logistic-regression-maximizing-true-positives-false-positives/73364#73364
4. https://datascience.stackexchange.com/questions/24260/logistic-regression-solving-the-cross-entropy-cost-function-analytically
5. https://arxiv.org/pdf/2009.09043.pdf
6. https://towardsdatascience.com/understanding-binary-cross-entropy-log-loss-a-visual-explanation-a3ac6025181a
7. https://stats.stackexchange.com/questions/267400/logistic-regression-cost-surface-not-convex
8. R Data Visualization Recipes, Vitor Bianchi Lanzetta, Packt publications
9. Cfd-online.com
10. http://neuralnetworksanddeeplearning.com/chap4.html

Code & Data

Linear Regression & Convexity

Data –

```
X =  np.array([[1, -1],[2, -2],[3, -3],[4, -4]])
T = np.array([[0], [-1], [-2], [-3]])
```

Code –

```
%matplotlib inline

import matplotlib.pyplot as plt
import numpy as np
from sklearn.linear_model import LinearRegression

# Data as X for X_train and T for y_train

X =  np.array([[1, -1],[2, -2],[3, -3],[4, -4]])
T = np.array([[0], [-1], [-2], [-3]])

plt.scatter(X[:,0], X[:,1], c=T)



# join the X_train and y_train

train_data = np.array(np.hstack((X,T)))   # To plot in 3D
train_data

# Plot in 3D

fig = plt.figure(figsize=(10,15))
ax = plt.axes(projection='3d')

# Data for three-dimensional scattered points

xdata = train_data[:,0]
ydata = train_data[:,1]
zdata = train_data[:,2]


ax.scatter3D(xdata, ydata, zdata, c=zdata);
ax.view_init(10, 25)

X = train_data[:,0:2]
y = train_data[:,2]

# print training data
print(X)
print(y)
```

```
# Fit the model

regression_model = LinearRegression()
regression_model.fit(X, y)

# Predict on training data itself

yhat = regression_model.predict(X)
print("\n predictions :",yhat)

# print regression coefficients

print("\n Coefficients Learnt")
print("Weights :" , regression_model.coef_)
print("")
print("Intercept :", regression_model.intercept_)
```

Logistic Regression & Convexity

Data -

```
X =  np.array([[0, 0],[0, 1],[1, 0],[1, 1]])
T = np.array([0, 1, 1, 0])
```

Code -

```
# To enable plotting graphs in Jupyter notebook
%matplotlib inline

import pandas as pd
import numpy as np

from sklearn.linear_model import LogisticRegression

# importing ploting libraries
import matplotlib.pyplot as plt

N = 4  # No of data points
D = 2  # Dimensionality


# Data Set & visualization

X =  np.array([[0, 0],[0, 1],[1, 0],[1, 1]])
T = np.array([0, 1, 1, 0])


def cross_entropy(T, Y):
```

```
    E = 0
    for i in range(4):
        if T[i] == 1:
            E -= np.log(Y[i])
        else:
            E -= np.log(1 - Y[i])
        return E

w = np.random.randn(D)  #Initialize coefficients with random values

z = X.dot(w) # Compute weighted summation of weights and X

def sigmoid(z):    #Sigmoid function on the weighted summation
    return 1/(1 + np.exp(-z))

Y = sigmoid(z)   # capture the probability of postive and negative class into an array


learning_rate = 0.001  # fix the learning steps for gradient descent to prevent oscillations around minima

error = []

for i in range(5000):    #execute 5000 loops. Could be any number but make sure they are sufficient for logistic regression to converge

    e = cross_entropy(T, Y)
    error.append(e)
    w += learning_rate * ( np.dot((T - Y).T, X) - 0.01*w)
    Y = sigmoid(X.dot(w))

plt.plot(error)
plt.title("Cross-entropy per iteration")
print ("Final classification rate: ", 1 - np.abs(T - np.round(Y)).sum() / N)
print ("Final w: ", w)
```

==========

Other blogs – (All on LinkedIn)

1. Long Short Term Memory (LSTM) - The Way I Understand It
2. Recurrent Neural Networks - The Way I Understand It
3. Convolution & Pooling - The Way I Understand it
4. Principal Component Analysis for All
5. PCA Code with Pitfalls
6. What I Do with PairPlot Analysis
7. Null Hypothesis and P Values… Introduction for Budding Data Scientists
8. Layman Introduction to Bias Variance Errors