**Machine Learning**

1. **Unsupervised Machine Learning**

*In this Unsupervised machine learning we give only Input data to train the model and we do not have any Output data to compare the predictions. By giving the input data to the suitable model it will gives the predictions accordingly.*

*Model finds the Hidden pattens from the data and make predictions out of it.*

*There are some situations where we do not have any output data so we will use this unsupervised technique.*

*We can compare this technique with human brains where we learn and make assumptions without knowing the result or fact.*

Here the model is Trained using the Imbalanced Dataset

* Clustering
* Association

**Clustering: -** It is a method used to group the values based on similarities so that it can make the output values of it.

If there are multiple shares in the it used to separate the shapes like triangles are at one place and circles are at one place so that it forms the groups a classification output to it

Sometimes rectangle and square may be coming under same group and leads to error or reduction in the accuracy…

**K means Clustering**

|  |  |
| --- | --- |
| Import | from sklearn.cluster import KMeans |
| loop to get intersection point values in the elbow graph | inter = []  for i in range(1,11):  algo = KMeans(n\_clusters = i)  algo.fit(X)  inter.append(algo.inertia\_) |
| elbow graph | pd.DataFrame(inter).plot()  plt.show() |
| Model building | k\_model = KMeans(n\_clusters = 3)  k\_model.fit(X) |
| clusters | l1 = k\_model.labels\_ |
| Dataset with clusters | df = clusters.assign(cluster = l1) |
| plotting the values with clusters | sns.scatterplot('Age','Spending Score (1-100)',hue = 'cluster', data = clusters,palette = 'bright')  plt.show() |

**K-Means Cluster: -**It is an Iterative algorithm which means it repeats its work multiple times to get the divided unlabelled dataset in to K different clusters or groups

K = number of groups or clusters those data can divided.

How K means clustering works. Below

Ex: - K = 2 for below steps.

* 1st plot the values using the scatter plot for 2d data, if its 3d data or multidimensional data still works same as how 2d data works. But only 2d data is visible in scatter plot.
* Select two random values as a centroid draw a line to connect those two points
* Draw a median line to cut the centroid line equally and make a partition to show there are two groups.
* (K = those centroid points to be selected so that we can cut those to separated and make each individual group)
* Take the Euclidean distance from each point near to the centroid of that particular group
* Euclidean distance: -
* Euclidean distance: - it is a distance from each point in the group to the centroid
* by taking all the values find the mean and draw the point of the mean in that group
* mean can be a point which already exist or we need to create/draw
* once the mean is created in both sides takes those mean points as the centroid and draw the centroid line which connect these both the points
* after forming the centroid line cut the line equally with median line and takes the Euclidean distance from all the values to make new mean point
* same way this process do multiple times in background of this algorithm to get the best fit and makes the similar values at one side and from the group
* that’s how the clusters are formed by this algorithm.

**How to take the K value:**

K means clustering algorithm is highly depends on the number of K values

The number of K value is highly depending on the number of clusters to form.

The clusters are separated by the similarities of the data. To know the K value, we will use the elbow curve method.

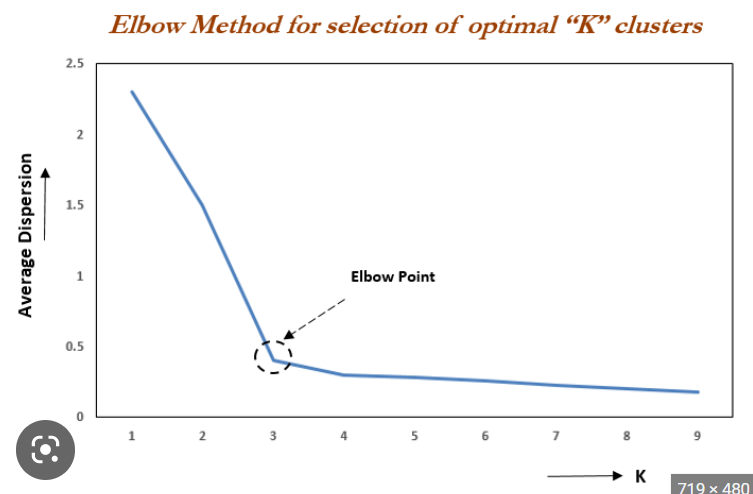
Elbow curve Method: -

* the Elbow curve method works on WCSS
* WCSS: - With in the Cluster Sum of Squared Error
* SSE: - (Sum of Squared Error) is used in liner regression.
* WCSS = SSE + SSE

+ + …

* xi= actual value; c1=cluster value; dis= distance from;
* sum of all the SSE values together is WCSS
* if K = the number of datapoints in the scatter plot then the WCSS = 0 and its overfitting the model.
* If K value is very less then the WCSS value will become very high so its underfitting the model.
* So, we have to select the optimum value

How to plot the elbow curve: -

* Default taking the range from 1 to 10 in code we take range (1,11)
* K means algorithm apply k value for every value in the range
* For every k value we will get the WCSS score
* Plot the WCSS values from 1 to 10 in order to see elbow curve
* 
* Form points 3 we can see the most deviation in the curve so point 3 will be the elbow point.
* Which means k=3; so, there will be 3 clusters for this dataset.

**PCA**

**Principle Component Analysis**

|  |  |
| --- | --- |
| Normalisation | sc = StandardScaler() |
| Only for Input data: X | x\_sc = sc.fit\_transform(X) |
| Covariance Matrix | cov\_mat = np.cov(x\_sc.T) |
| eig\_vals = eigen values  eig\_vecs = eigen vecters  np.linalg.eig(cov\_mat) = numpy. linear algibra.eigen(covariance matrix) | eig\_vals,eig\_vecs = np.linalg.eig(cov\_mat) |
| Explained Variance (Individual)  tot = variable refers total  list comprehension(var\_exp) | tot = sum(eig\_vals)  var\_exp = [(i/tot)\*100 for i in sorted(eig\_vals,  reverse = True)] |
| Cumulative Variance Explained | cum\_var\_exp = np.cumsum(var\_exp) |
| Plotting Bar plot and Step plot in 1 single plot to visualise the Principal Components to find the n\_components for the PCA | plt.figure(figsize = (10,5)) # size of the plot  plt.bar(range(len(var\_exp)), var\_exp, label = 'individual explained variance', color = 'red')  plt.step(range(len(cum\_var\_exp)), cum\_var\_exp, label = 'cumulative explained variance')  plt.ylabel('Cum Explained Variance')  plt.xlabel('Principle Components')  plt.legend()  plt.show() |
| Data Split | X\_train,X\_test,y\_train,y\_test = train\_test\_split(x\_sc,y,train\_size = 0.8) |
| Import PCA | from sklearn.decomposition import PCA |
| PCA (Principal Component Analysis) | pca = PCA(n\_components = 20)  pca\_X\_train = pca.fit\_transform(X\_train) pca\_X\_test = pca.transform(X\_test) |

Dimensionality Reduction (Curse of Dimensionality)

1. Feature Selection (Supervised Learning)
   1. Multicollinearity
2. Feature Extraction (Un Supervised Learning)
   1. PCA

**Feature Extraction:** It is a process of transforming the original data into a lower-dimensional space by identifying the most important features that contribute to the variance (spread) in the data. This is typically done using techniques like PCA, LDA, or t-SNE. Feature extraction is used to reduce the dimensionality of the data, eliminate irrelevant features, and improve the performance of machine learning algorithms.

|  |  |
| --- | --- |
| Feature Selection | Feature Extraction |
| Selecting the features from Existing columns | Here we will create new features from existing columns |

**PCA: Principal Component Analysis: -** (1000D to 3D)

Ex: - 60 columns to 20 columns depending on the n\_components.

Ex: - In Cricket Live Game actually happens in 3d but we can able to see in tv as 2d.

Ex: - if there is house price prediction dataset

|  |  |  |
| --- | --- | --- |
| Rooms | Bathrooms | Price |

|  |  |
| --- | --- |
| Flats | Price |

In the above dataset no. rooms = no. bathrooms so there is no need of 2 columns we can merge and make 1 column in same way PCA works. It reduces the columns by maintaining the essence of the dataset and removing the noise of the dataset.

**Mean Vs Variance: -**

**Mean: -**

0

-5

5

0

-10

10

Mean = 0

Mean = 0

M = sum of the elements / no. of the elements

Sum of the elements = 0

no. of the elements = 0

So, in above Ex. Mean = 0

**Variance: -** spread of the values w.r.t mean in standard deviation

0

-5

5

0

-10

10

variance = 50/3

variance = 200/3

n = 3 (total number of points or elements)

xi = points spread in the graph, here -5 or -10

V =

In above ex. Variance =

= = 50/3

Coordinate system: - 2d graph or 3d graph which constrains x, y, and z axis in it.

**Covariance: -** it speaks about the relation between 2 columns; but variance never speak. Only covariance speaks about 2 columns.

(0,0)

(-1,1)

(1, -1)

(0,0)

(1, -1)

(-1,1)

1st plot = =

2nd plot = =

From the above

Both the columns are indirectly proportional to each other, if one increases another one decreases

These two columns are having negative corelation

Sign are different but the magnitude is same for both

**Covariance Matrix: - let’s say columns X1 and X2**

X1

X2

cov (x1, x1)

cov (x2, x1)

cov (x2, x2)

cov (x1, x2)

X1

X1

X2

X2

var(x1)

cov (x2, x1)

var (x2)

cov (x1, x2)

X1

X1

X2

X2

var(x1)

cov (x2, x1)

var (x2)

cov (x1, x2)

X1

X1

X2

X2

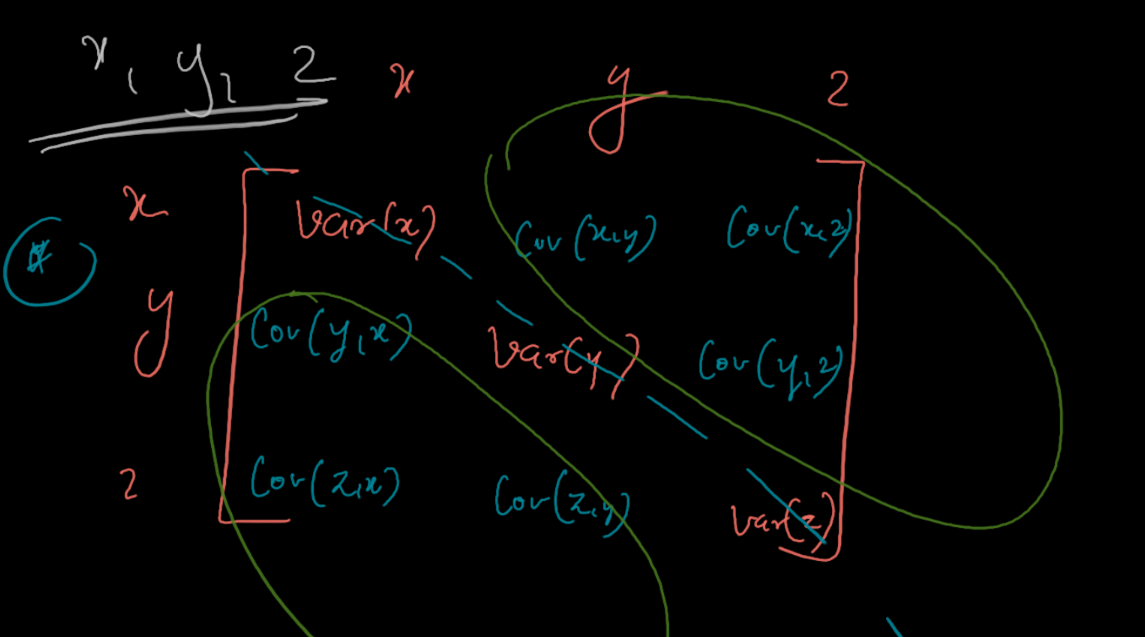
cov (x1, x1) = var(x1)

cov (x2, x2) = var (x2)

Covariance Matrix will always show the mirror image w.r.t diagonal.

Covariance in the format of matrix is called covariance matrix.

For 3d: - X, Y, and Z



**Vector: -** any point in the coordinate system called as vector point, though it could be in 2d or 3d. (vector is a quantity which contains magnitude and direction)

\* A

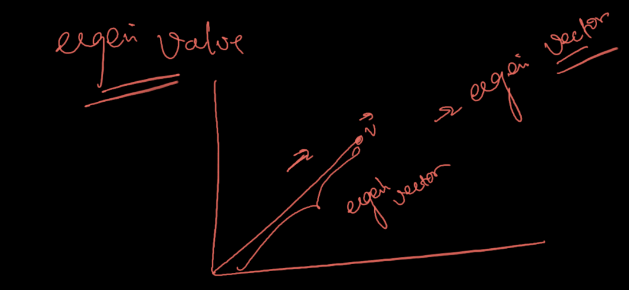
when the vector multiplied with a matrix the magnitude and direction of the matrix will change

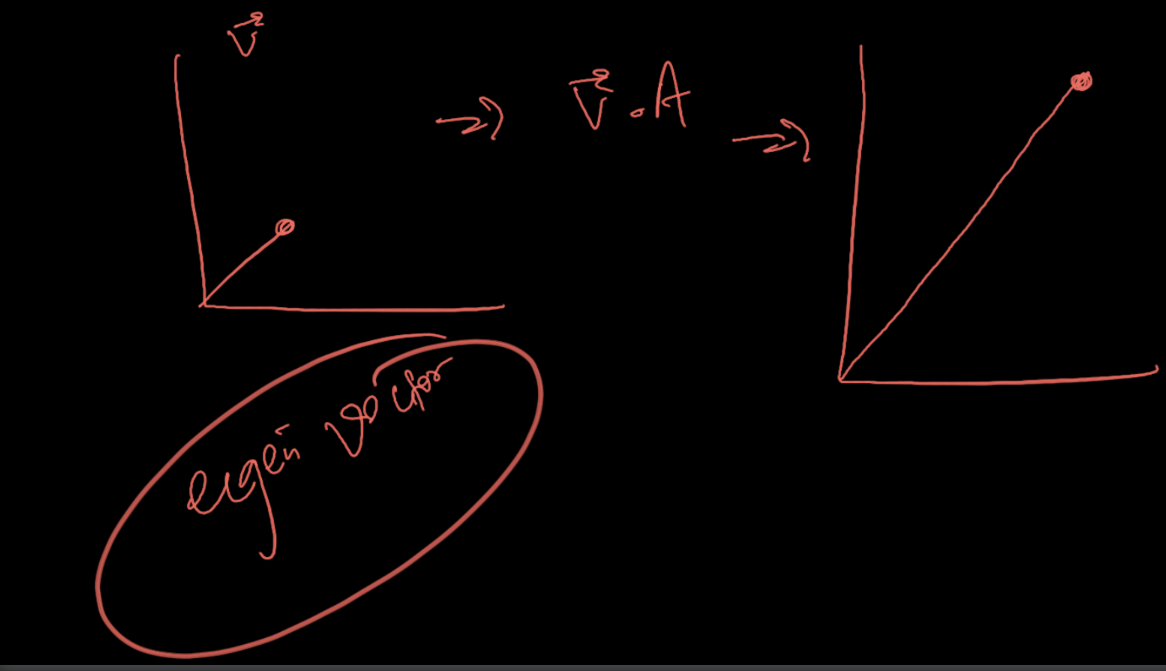
= Vector

A = Covariance Matrix

The process of multiplying the vector values with matrix called **Transformation** or **Linear Transformation.**

**Eigen Vector: -** this vector values are also called special vectors.

When we wanted to transform the eigen vectors (when we multiply with matrix) magnitude remain same for the vectors but the direction changes.



Formula = \* A = λ\*A

λ = eigen value

A = matrix

= Vector

How the formula is made:

V = vector but = vector with the direction so, when the vector with direction and magnitude is multiply with the matrix the values of the magnitude is change in the eigen vectors. So, the change in the magnitude can say as eigen value λ form the above 2nd picture.

Dimensionality reduction: -

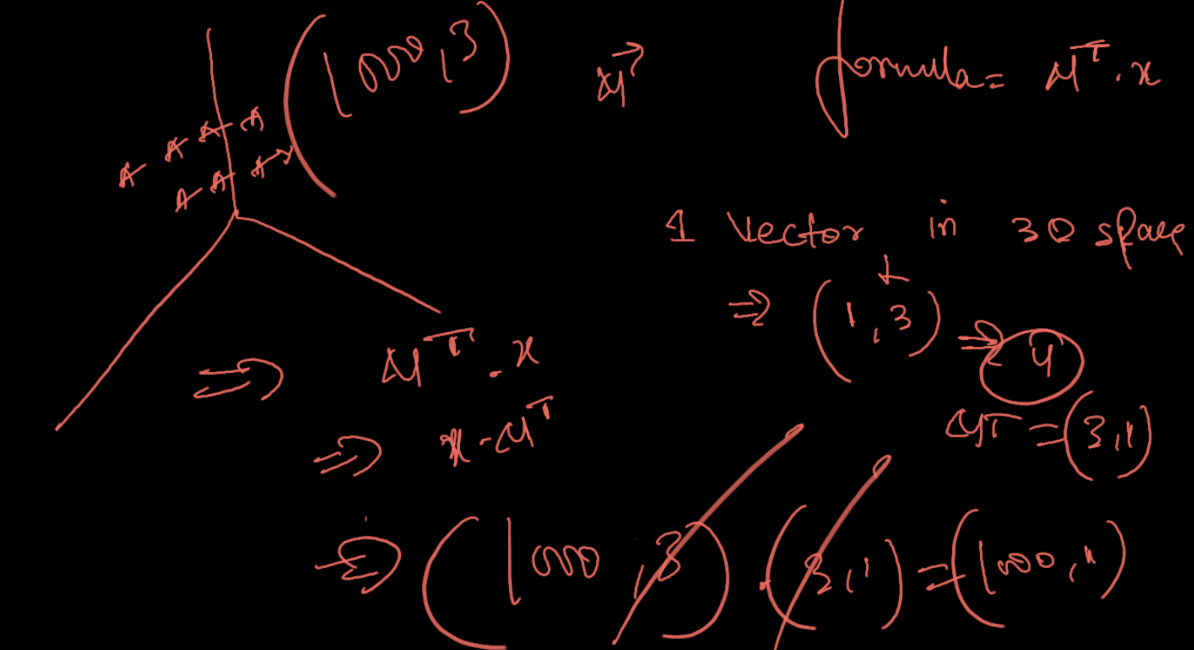
3D Space

(1000,3)

Converting 3D space in to 1D space

(1000,3) 🡪 is the shape of the data set.

So, there are 1000 rows and 3 columns and we are converting it into 1D so it becomes 1 column with same number of rows.



1 Vector in 3D Space so the formula is

= unit vector

= 🡪changing the positions

So, 1 vector in 3D Space = µ = (1,3)

=

= (1000,3) \* (3,1)

*x* = (100,3)

= (3,1); T in the exponential place mentions the Transpose of the element which means reverse the value.

= (1000,3) \* (3,1)

3 and 3 get cancel and forms (1000,1)

Same for 2 vectors in 3D Space becomes (1000,2) in the end.

eigen values and eigen vectors finding mathematics YouTube video: <https://youtu.be/ZcMyUftHpVY>

Java point: theory explained:

<https://www.javatpoint.com/principal-component-analysis>

Krish naik YouTube: <https://youtu.be/H99JRtDDnvk>

PCA In coding: -

1st step: - we have to Normalise the input data coz we don’t have any output data in the dataset in unsupervised learning so,

x\_sc.shape

**2nd step: -** finding the covariance matrix of that data

cov\_mat = np.cov(x\_sc.T)

cov\_mat = co variance matrix variable in code

np.cov() = it is a numpy function used to find the covariance of the data

x\_sc.T = Input data in transpose just like

**3rd step: -** finding the eigen values and eigen vectors of the covariance matrix

eig\_val s= eigen values

eig\_vecs = eigen vectors

np.linalg.eig() = numpy function linear algebra.eigen

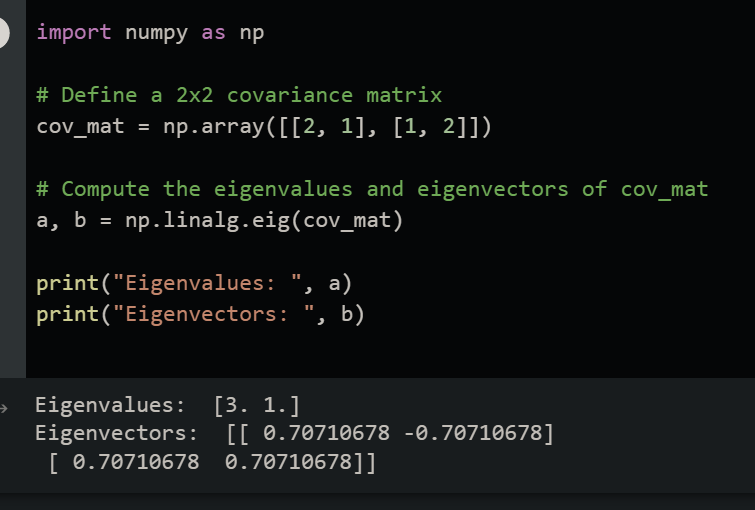
eig\_vals,eig\_vecs = np.linalg.eig(cov\_mat)

np.linalg.eig(cov\_mat) = this code gives the output in tuple in which there will be tow arrays 1. Eigen values and 2. Eigen vectors

eigen values are in 1D array in tuple and eigen vectors in 2D array in tuple.

These are eigen values and eigen vectors of covariance matrix

**Ex: -**



**4th step: -** find the percentage of the total variance (spread) in a dataset that is explained by each principal component called variance explained

tot = sum(eig\_vals)

var\_exp = [(i/tot)\*100 for i in sorted(eig\_vals, reverse = True)]

tot = total or sum of the eigen values

var\_exp = variance explained

which is the percentage of the each eigen values

[(i/tot)\*100 for i in sorted(eig\_vals, reverse = True)] = this is the list comprehension which contains for loop in side and says that I in eigen values are converting into the percentage of **the variance explained in descending order (reverse = True).**

**5th step: -** now cumulative percentage of the variance percentage means

cum\_var\_exp = np.cumsum(var\_exp)

cum\_var\_exp = variable refers to cumulative sum of the variance explained

np.cumsum() = numpy function for cumulative sum

**cumulative sum: -**

input list or array = [1 2 3 4 5]

output list or array = [ 1 3 6 10 15]

so, it is working like [1, 1+2, 1+2+3, 1+2+3+6, 1+2+3+6+10, 1+3+6+10+15]

so, here in the original code the cumulative sum happening for the variance explained which means happening of the variance percentage

**6th step: -** plotting the bar chat and step chat (bar plot and step plot)

Bar plot = (x = length of the variable explained, y = variable explained)

Step plot = (x = length of the cumulative var\_exp, y = cumulative var\_exp)

To visualise the n-components for the PCA.

7th step: - train test split (x train, x test, y train, y test)

8th step: - applying the PCA on the dataset

Selected n\_components as 20 here (here the dataset will reduce its features to 20)

pca = PCA(n\_components = 20)

then we have to transform the input data

pca\_X\_train = pca.fit\_transform(X\_train)

fit and transform train dataset just like normalisation on both input data (train and test)

pca\_X\_test = pca.transform(X\_test)

then final step

9th step: - model building

applying the pca\_x\_train and pca\_x\_test data to suitable model or algorithm

then predicting the values then accuracy score

**Gradient Decent**

|  |
| --- |
| Gradient Decent function code |
| import numpy as np  def gradient\_descent(x,y):  m\_curr = b\_curr = 0  iteration = 100  learning\_rate = 0.001  n = len(x)  for i in range(iteration):  yp = m\_curr \* x + b\_curr # y = mx + c  cost = (1/n) \* sum([val\*\*2 for val in y-yp])  md = -(2/n) \* sum(x\*(y - yp)) # slope  bd = -(2/n) \* sum(y - yp) # slope  # new values of parameter at each iteration  m\_curr = m\_curr - learning\_rate \* md  b\_curr = b\_curr - learning\_rate \* bd  print("cost:", cost, "m:" , m\_curr, "b:" , b\_curr)  x = np.array([1,2,3,4,5])  y = np.array([5,7,9,11,13])  gradient\_descent(x,y) |
| Visualisation Gradient descent |
| import numpy as np  import matplotlib.pyplot as plt  %matplotlib inline  def gradient\_descent(x,y):  m\_curr = b\_curr = 0  rate = 0.01  n = len(x)  plt.figure(figsize = (10,10))  plt.scatter(x,y,color='red',marker='+',linewidth=5)  for i in range(100):  y\_predicted = m\_curr \* x + b\_curr  # print (m\_curr,b\_curr, i)  plt.plot(x,y\_predicted,color='green')  md = -(2/n)\*sum(x\*(y-y\_predicted))  yd = -(2/n)\*sum(y-y\_predicted)  m\_curr = m\_curr - rate \* md  b\_curr = b\_curr - rate \* yd  x = np.array([1,2,3,4,5])  y = np.array([5,7,9,11,13])  # calling the function  gradient\_descent(x,y) |

It is a technique in ML or DL to reduce the cost function by changing the valuers.

Cost function = loss function = error percentage

If accuracy was 98% means there is 2% error or loss the process of reducing this loss is called gradient decent.

**Formula: -** **y = mx + c**

Y = dependent variable

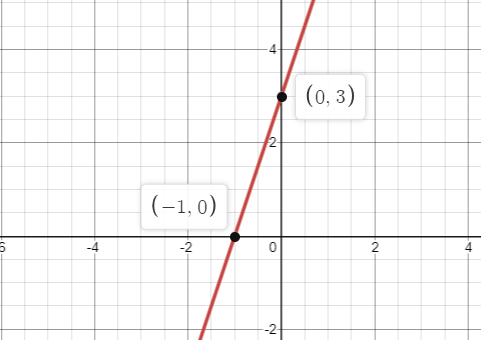
M = slope

X = point in the graph

C = y-Intercept

It is the equation of a straight line where m is the slope, c is the y-intercept (the point where the line crosses the vertical y-axis).

**For example:** 3x−y=−3 can be written as y=3x+3 where slope of the equation is 3 and y-intercept is also 3 (By putting x=0)



Example: - for gradient decent

You: - hey, I got good marks in Exam

Friend: - how much

You: - give it a guess

Friend: - 70

You: - little closer

Friend: - 80

You: - little lesser

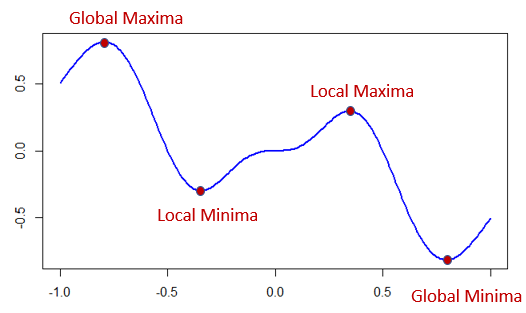
Friend: - 73

You: - little more

Friend: - 75

You: - Exactly

You make your friend to reach your exact marks by optimising the parameters.



Global Maxima: - it is the highest point values of a cost function in the problem and it contains maximum loss compared with any other points.

Global Minima: - it is the minimum loss value in the problem (it may be the required point of position in most of the problems)

Local Maxima: - it is the maximum value of the loss function at a specific range

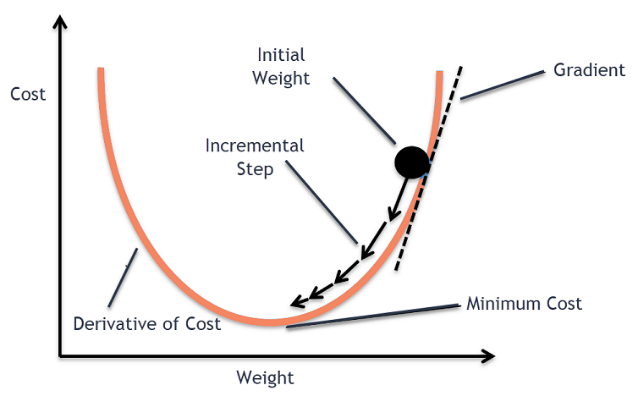
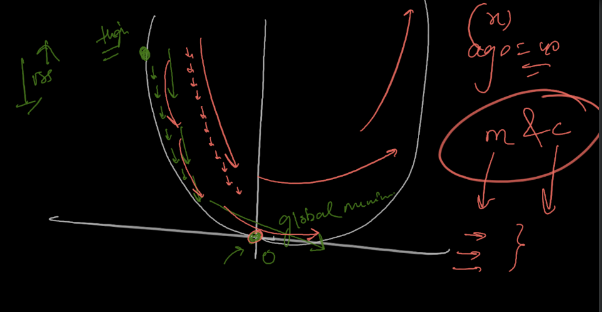
Local Minima: - it is the minimum value of the loss function at a specific range

Why to reach global minima?

to reduce the cost function

how to reach global minima?

By the rate of change in your parameter m and c and we can achieve this with learning rate concept.





Above graph is the slope of the loss function and the learning rate is shown with arrow marks.

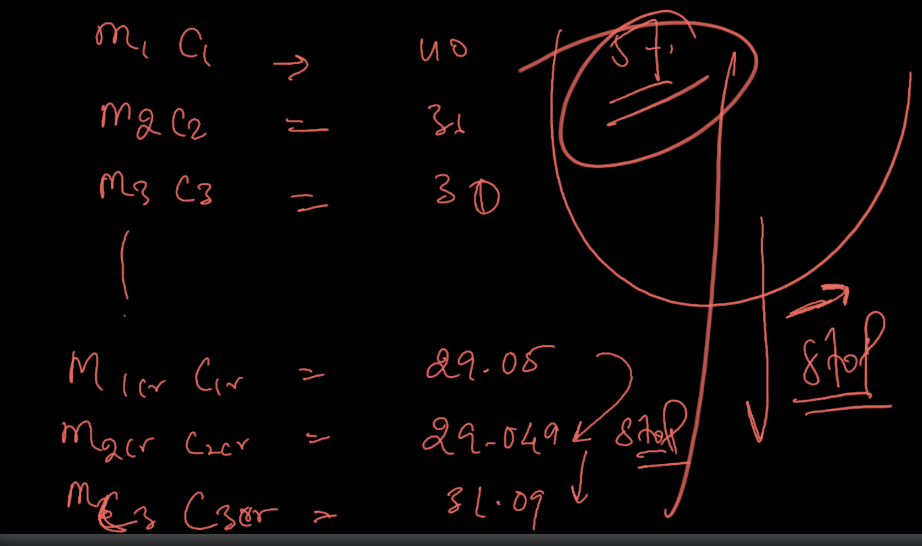
Minimum cost = global minima

If the learning rate is too (less) slow then model may achieve the global minima but it takes more time to execute

And if we take the (high) faster running rate then the model may take less time to execute but it may lead to miss the global minima

So, we need optimal learning rate.

We have to change the m and c values multiple times by iterating to reduce the cost function



At any one point or number of iterations the error or loss may not give the huge difference so it concludes that to stop iterating for reducing the loss function. In above example its m 2 crore, c 2 crore it’s the point where we can see no big change in the cost function and if still, we do try we may lead to get the higher cost function value again which is not good.

We have to do iteration until the cost function showing the good change.

Formula: -

New value = old value – step size

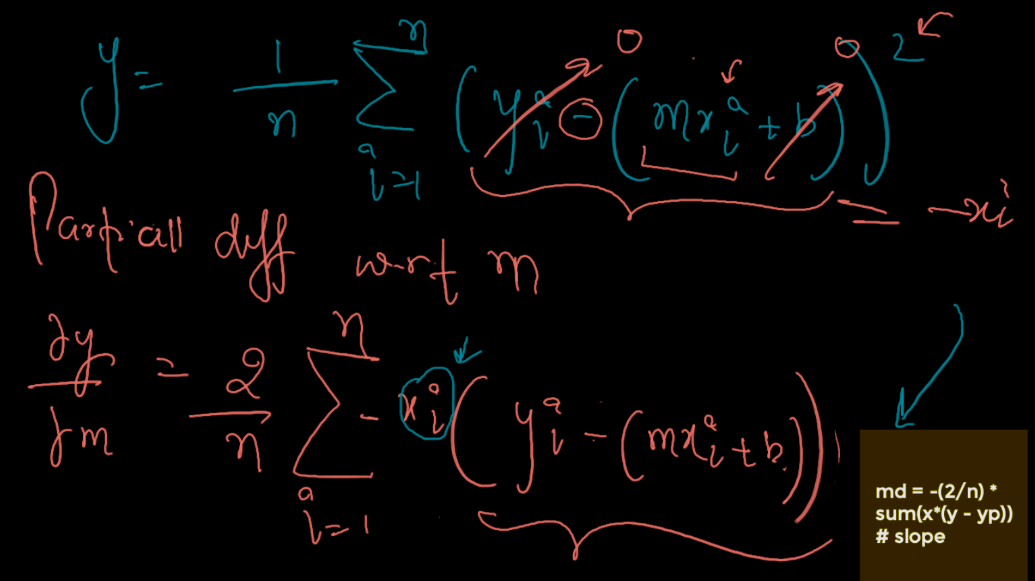
Step size = learning rate \* slope

**MSE: Mean Squared Error: -**

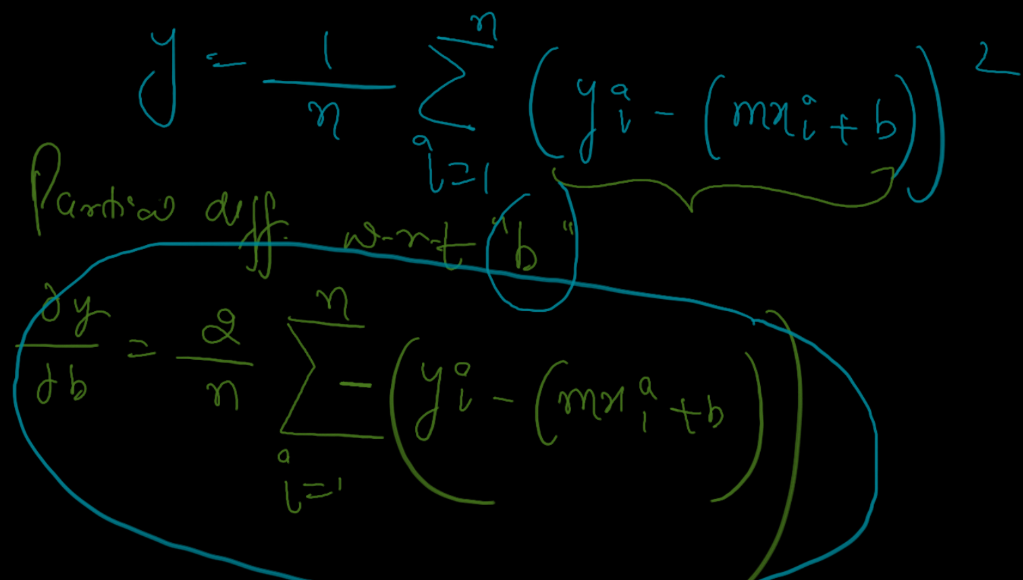
Here ypred =

Partial difference with respect to value: m

Here in the below picture -xi extracted by partial differentiation of



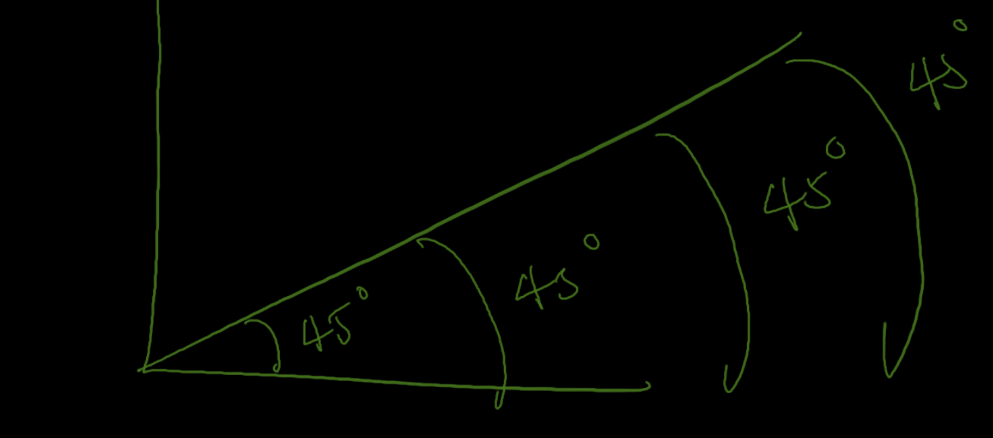
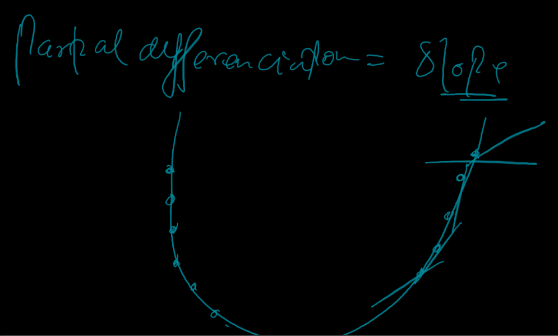
Partial difference with respect to value: b



What is slop:

Slop is a fit line of predicted values in the graph

Slop = dy/dx



Differentiating anything becomes slope.

From above 2 pictures 1st picture is for differentiation and 2nd picture is about partial differentiation

In differentiation slope is in straight line but in partial differentiation slope is in U shape with taking multiple tangent points to form U shape slope.

**CODE: -**

After importing the numpy and matplotlib libraries

def gradient\_descent(x,y):

create a function gradient descent name and give x and parameters to refer the dataset

    m\_curr = b\_curr = 0

    rate = 0.01

    n = len(x)

this are the variable which we are going to use those in for loop later

m\_curr = current value of m (slope in y predicted value)

b\_curr = current value of b (intercept in y predicted value)

rate = learning rate

n = number of rows (here you can give y also instead of x coz both have same no. of rows in dataset)

    plt.figure(figsize =  (10,10))

    plt.scatter(x,y,color='red',marker='+',linewidth=5)

scatter plot to visualise the gradient descent lines (best fit line among the actual line with learning rate)

    for i in range(100):

now the for loop start here we have given 100 iteration which means 100 times the loop going to repeat with different increasing values w.r.t code me mention below

and also, we can give any number of iterations like more than 100 or 100000 or any to reduce the cost function but if we gave a greater number of iteration there may leads to increase the cost function(error) so we need to give optimal iteration to reach the y pred value to the global minima point.

y\_predicted = m\_curr \* x + b\_curr

above is the formula for y pred value just like

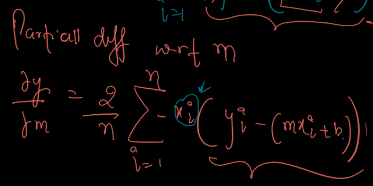
ypred =

but here it is in loop and previously mentioned before loop that these 2 values are 0 (zero) so, there values are changes multiple times in the code in for loop.

plt.plot(x,y\_predicted,color='green')

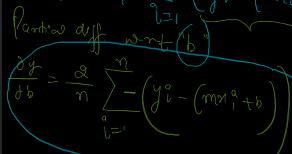
this is the plot here x is the input values and the y predicted is the predicted values. And colour = green means the lines in the plot will display in green colour.

md = -(2/n)\*sum(x\*(y-y\_predicted))

md = differentiation of m value.

Just like 🡪

yd = -(2/n)\*sum(y-y\_predicted)

yd = bd = differentiation of b value

Just like 🡪

        m\_curr = m\_curr - rate \* md

        b\_curr = b\_curr - rate \* yd

as I have mentioned above that m and b values will change multiple times in for loop to get best fit with iteration so here is the code which will make the m and b values to change multiple times.

x = np.array([1,2,3,4,5])

y = np.array([5,7,9,11,13])

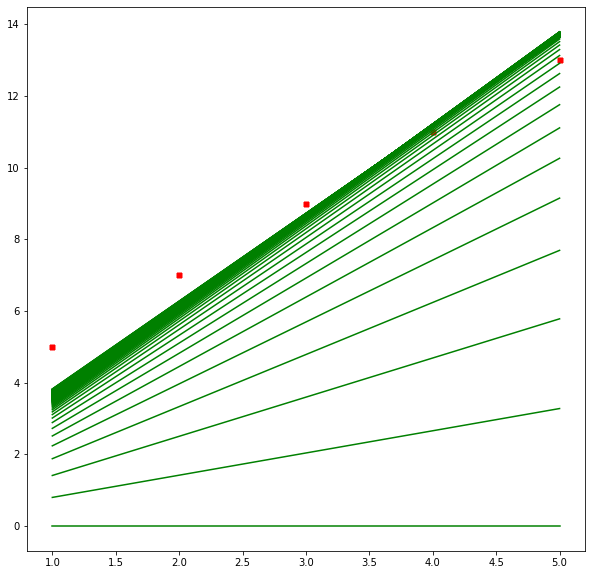
sample data x is the input example and y is the output example

the above arrays just represent the dataset for easy understanding

gradient\_descent(x,y)

calling the function in the end of the code.

This is how the gradient descent plot look like



……………………………………………………………….