AI
Notes

Context

Exam :

Monday 30th , 9:30am – 12:00pm

4 Questions :

1. **Clustering**
2. **Supervised Learning**
3. **Search**
4. **Optimisation**

Each 15 marks,
total 60 marks
**note: no GMM and weka

Content :

- **Clustering** **(Week 1 & 2)**
- **Supervised Learning** **(Week 3 - 6)**
- **Search** **(Week 7 & 8)**
- **Optimisation** **(Week 9 & 10)**

Clustering :

- **Hierarchical Clustering**
- **K-means**
- **GMM/EM**
- **DBSCAN**

Supervised Learning :

- **Linear Regression**
- **Gradient Descent**
- **Logistic Regression**
- **Neural Network**
- **Evaluate & Hyperparameter Tuning**
- **Naive Bayes**
- **KNN-algorithm**

Search :

- **Search Problem**
- **BFS**
- **DFS**
- **A***

Optimisation :

- **Hill Climbing**
- **Simulated Annealing**
- **Formulation**
- **Constraint handling**

Introduction

Basic Introduction:

Assuming a data set with $\boldsymbol{n}$ samples, and each sample has $\boldsymbol{d}$ dimensions :

$$[(x_1^1, x_2^1, \dots x_d^1), (x_1^2, x_2^2, \dots x_d^2), \dots, (x_1^n, x_2^n, \dots x_d^n)] = 1,2..n$$

Instance : One data sample
Dimension : Number of coordinates to specify the samples
Feature / Attribute : The value at each dimension

you can think of $\boldsymbol{n}$ samples as data inputs, and $\boldsymbol{d}$ dimensions as attributes
(or in Objects terms,
n : Objects
d : properties)

There are 3 types of Machine Learning:

- **Supervised Learning**

  (Labelled Data)

- **Unsupervised Learning**

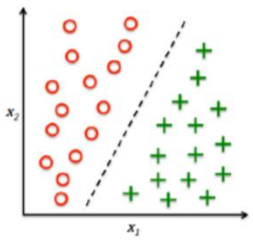
  (Unlabeled Data, may be due to data being too big / unclear)

- **Reinforced Learning**

  (Learns from rewards, rewards changes by time)

Classification

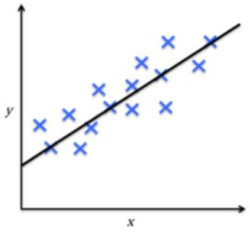
Outputs are **categorical**



eg: handwritten numbers → [1,2,3,..]

Regression

Outputs are **continuous numbers**



eg: Student Score Marks → 83.2%

Clustering

Hierarchical Clustering:

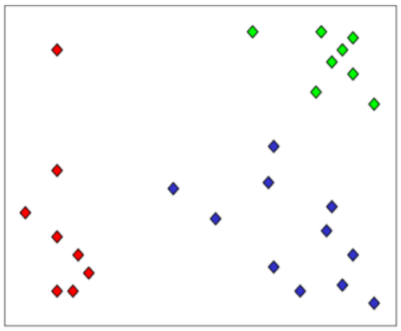
Terminology

Singleton : one data point

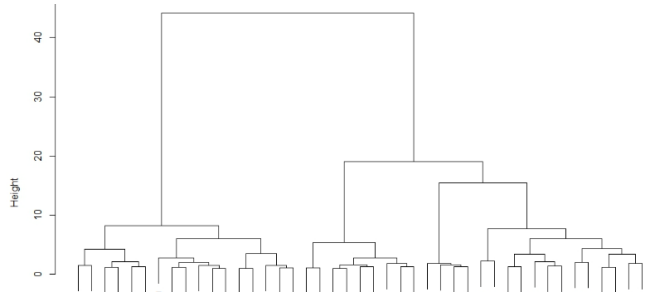
Cluster : multiple data points

Segments data into clusters such that there is a "natural grouping" among objects, it creates a hierarchical decomposition of the set of objects using some criterions

eg :



And produces a ***dendrogram*** *:*



Note : the distance between clusters are noted as ***height***

In the end, there will be **only 1 cluster** all grouped together

Measuring Distance

we will be using **Euclidean Distance** only

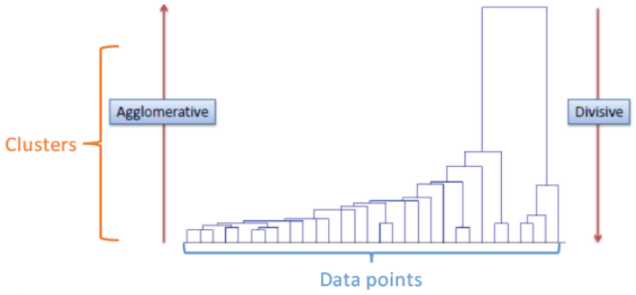
$$distance(a,b) = \sqrt{(a_x - b_x)^2 + (a_y - b_y)^2}$$

extra : it is also Minkowski with dimension = 2

There are 2 ways of Hierarchical clustering:

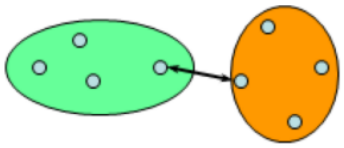
- **Agglomerative** *(bottom-up merging)*
- **Divisive** *(top-down merging)*

Divisive *is not recommended as it can be computationally expensive*



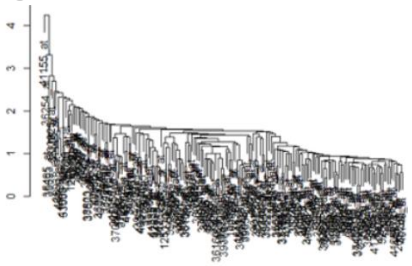
There are 3 ways to measure distance between clusters:

- **Single Linkage**



- Distance of the **closest pair**
- Produces **long chain shape** of clusters

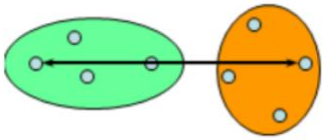
Eg:



$$c_1, c_2 = min\left(\forall distance\left(p_{c_1}, p_{c_2}\right)\right)$$

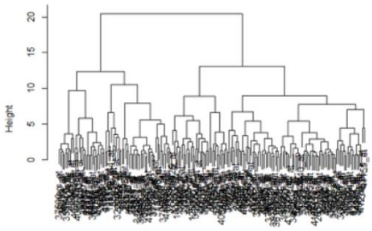
Where c_1 , c_2 are clusters and return minimum distance between 2 points in $c_1$and c_2

- **Complete Linkage**



  - Distance of the **furthest pair**
  - Produces **compact shape** clusters
  - *Sensitive to noise*

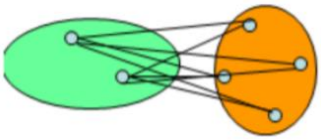
Eg:



$$c_1, c_2 = max\left(\forall distance\left(p_{c_1}, p_{c_2}\right)\right)$$

return maximum distance between 2 points in c_1 and c_2

- **Group Average**



  - **Average of all distances**
  - Most used
  - *Robust against noise*

$$c_1, c_2 = \frac{\sum \forall distance\left(p_{c_1}, p_{c_2}\right)}{Num\ p_{c_1} + Num\ p_{c_2}}$$

return Average distance of **all the points** in c_1 and c_2

Advantages

- Deterministic results (same result for any run)
- Does not need to specify number of clusters
- Can create cluster of arbitrary shapes

Disadvantages

- Does not scale up to larger datasets, time complexity is $O(n^2)$
- Will impose hierarchical structure even though data is not appropriate for it
- Once a decision is made, cant be undone

Algorithm (agglomerative):

Step 1 : Find every Euclidean distance

(create distance matrix)

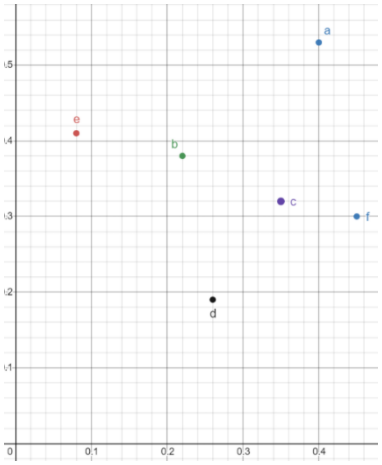
Step 2 : Get the **minimum** distance , which the 2 points/cluster joins becoming the new cluster

Step 3 : Recalculate the distance matrix with the new cluster

- Complete Linkage → Max
- Single Linkage → Min
- Group Average → Average

Step 4 : Repeat **Step 2**, until only 1 cluster left , Done!

Example of a run (Complete Linkage):



point	x-axis	y-axis
a	0.40	0.53
b	0.22	0.38
c	0.35	0.32
d	0.26	0.19
e	0.08	0.41
f	0.45	0.30

Step 1 – Use Euclidean distance to create distance matrix

(in this case we are only calculating point a and b)

$$distance(a,b) = \sqrt{(a_x - b_x)^2 + (a_y - b_y)^2}$$

$$distance(a,b) = \sqrt{(0.40 - 0.22)^2 + (0.53 - 0.38)^2}$$

$$= \sqrt{(0.18)^2 - (0.15)^2}$$

$$= \sqrt{0.0549}$$

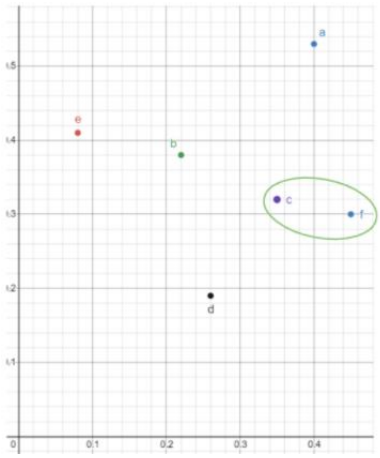
$$= 0.23$$

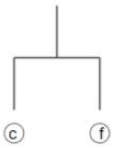
Distance Matrix:

	a	b	c	d	e	f
a	0					
b	0.23	0				
c	0.22	0.15	0			
d	0.37	0.20	0.15	0		
e	0.34	0.14	0.28	0.29	0	
f	0.23	0.25	0.11	0.22	0.39	0

Step 2 – get minimum value, forms the cluster

	a	b	c	d	e	f
a	0					
b	0.23	0				
c	0.22	0.15	0			
d	0.37	0.20	0.15	0		
e	0.34	0.14	0.28	0.29	0	
f	0.23	0.25	0.11	0.22	0.39	0





Step 3 – Recalculate the distance matrix (Complete Link)

to update the table:

$$\text{NewValue} = \max\big(distance(p_1, n), distance(p_2, n)\big)$$

Which concludes:

$$\max\big(distance(c, a), distance(f, a)\big) = 0.23$$

$$\max\big(distance(c, b), distance(f, b)\big) = 0.25$$

$$\max\big(distance(c, d), distance(f, d)\big) = 0.22$$

$$\max\big(distance(c, e), distance(f, e)\big) = 0.39$$

Updated Table :

	a	b	c,f	d	e
a	0				
b	0.23	0			
c,f	0.23	0.25	0		
d	0.37	0.20	0.15	0	
e	0.34	0.14	0.28	0.29	0

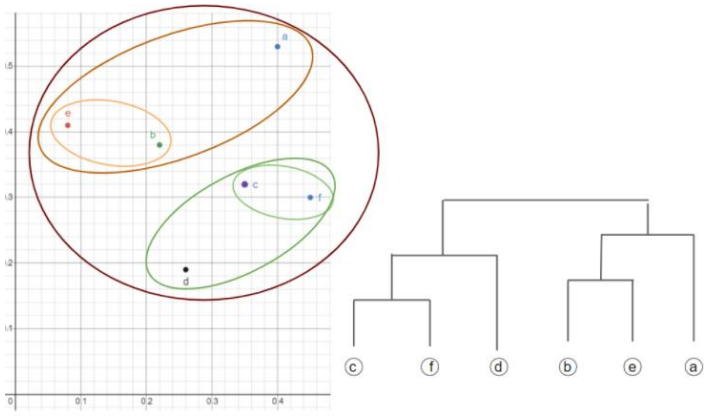
Step 4 – Repeat Step 2 until 1 cluster left

Will give an output of the distance of the final 2 cluster

**many more steps here but will skip ahead till 1 cluster left as it's just a repeat

Clustering Order

→1 (c , f)(a)(b)(d)(e) →2 (c, f)(a)(d)(e, b) →3 (c, f, d)(a)(e, b) → 4 (c, f, d)(e, b, a) →5 (c, f, d, e, b, a)



Done!

K-Means:

Terminology

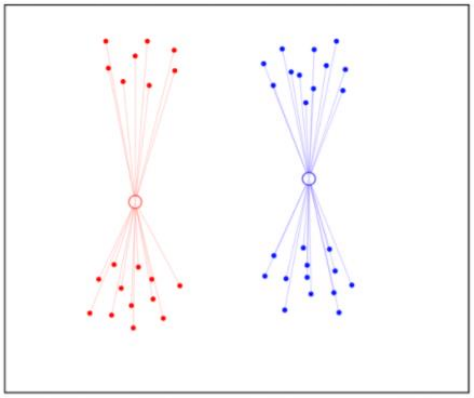
K : number of clusters

Centroid : the center of uniform density

Centroid Based – Describes each cluster by its mean

Its objective is to *minimize the within cluster variances* of all clusters that you set, ***K***

*eg (**K** = 2) :*



Advantages

- Easy to implement
- Efficient

Disadvantages

- Non-deterministic results (different result for every run)
- May result in local optima (multiple restart to get global optima)
- Require number of clusters in advance

Algorithm :

Step 1 : Set number of cluster , **K**

(K will usually have coordinates, if not → set random)

Step 2 : Calculate the distance matrix of every points to **K centroid**

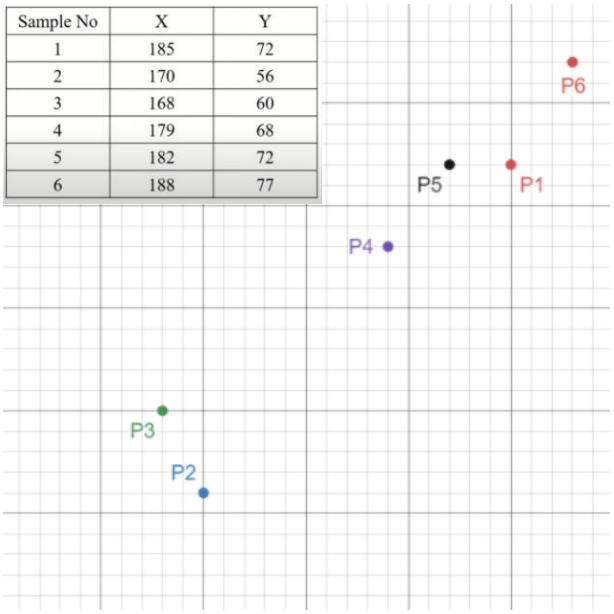
Step 3 : Assign the points to the closest **K centroid** (minimum)

Step 4 : Recalculate the cluster centroid mean

Step 5 : Repeat **Step 2**, until **K**-mean stop changing, Done!

Example of a run :

Sample No	X	Y
1	185	72
2	170	56
3	168	60
4	179	68
5	182	72
6	188	77



Step 1 : Set number of cluster , K

Set $K = 2$ and *in this case:* $K_1 = P_1, K_2 = P_2$

Initial Centroid

Cluster	X	Y
k1	185	72
k2	170	56

Step 2 : Calculate the distance matrix of every points to K centroid

Using Euclidean Distance formula :

$$distance(a,b) = \sqrt{(a_x - b_x)^2 + (a_y - b_y)^2}$$

Gives :

Sample No	K1	K2
3	20.809	4.472
4	7.211	15
5	3	20
6	5.831	27.659

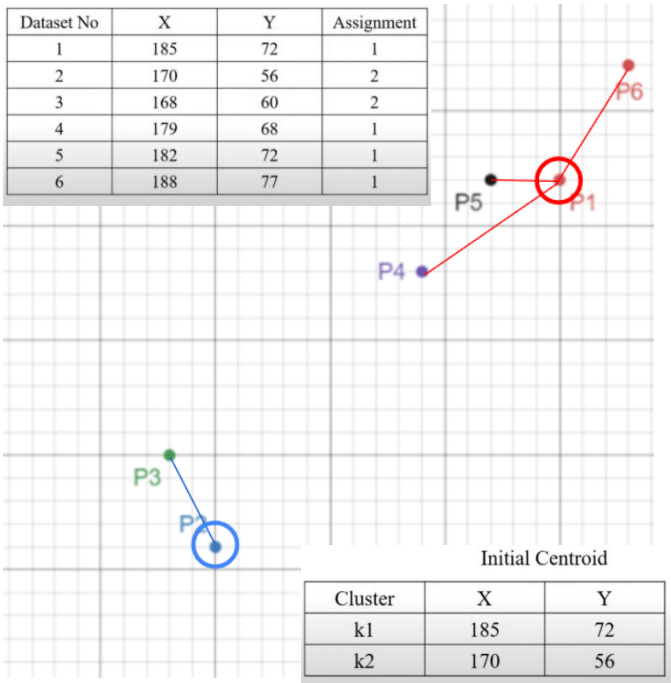
Step 3 : Assign the points to the closest K centroid

Using formula :

$$\min\big(distance(P_n, K_1), distance(P_n, K_2)\big)$$

"assign to **K** closest to it"

Dataset No	X	Y	Assignment
1	185	72	1
2	170	56	2
3	168	60	2
4	179	68	1
5	182	72	1
6	188	77	1



Initial Centroid

Cluster	X	Y
k1	185	72
k2	170	56

Step 4 : Recalculate the cluster centroid mean

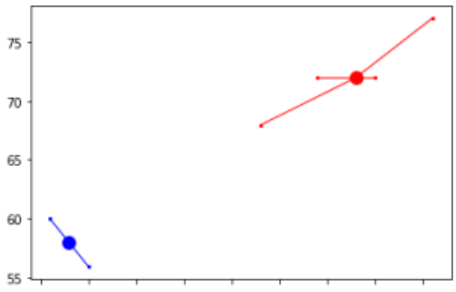
Formula for new K-mean if $\text{dimension} = 2$:

$$\frac{\sum P_x}{n}, \frac{\sum P_y}{n}$$

n = number of points in that cluster

Cluster	X	Y
k1	= (185+179+ 182+188) / 4 = 183	= (72+68+ 72+77) / 4 = 72
k2	= (170 + 168) / 2 = 169	= (60 + 59) / 2 = 58

Step 5 : Repeat Step 2, until K-mean stop changing



Done!

GMM/EM:

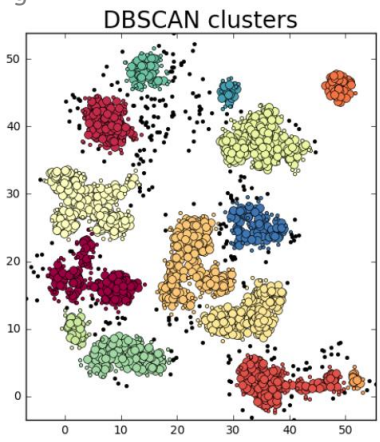
NO (K-means 2.0 , instead of centroid, it's a big +1 Dimension radius, something like a ripple)

DBSCAN:

Density Based Spatial Clustering of Application with Noise (you do not need to memorize the name)

Discover clusters by ***density*** of regions

Eg:



Usually, there are 2 parameters :

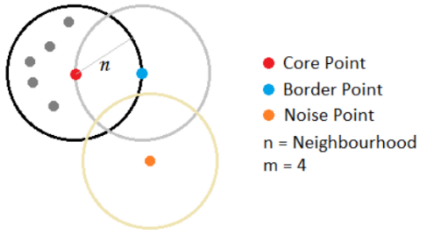
- Radius of circle (***ϵ***)
- Minimum number of points in that circle (*minPts*)

Points are categorized into 3 terms**:**

Core : point has *minPts* and within radius ***ϵ***

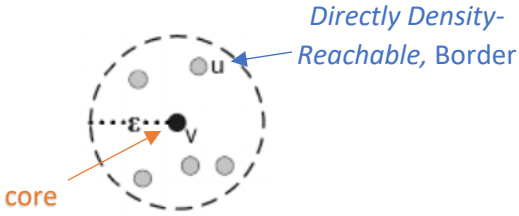
Border : does not have *minPts* and point lies within radius ***ϵ*** *of* core

Noise/Outlier : point which are not border or core



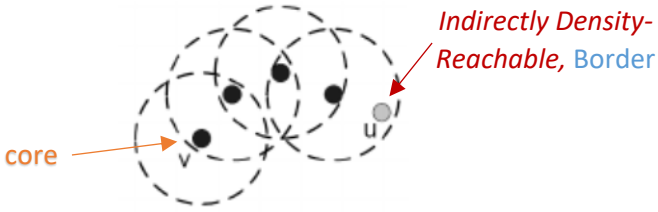
Forming Clusters

Directly Density-Reachable



A point, u is *Directly Density-Reachable* if it is within a core's radius of point v

Indirectly Density-Reachable



A point, u is *Indirectly Density-Reachable* if it is a border but **not within** a core's radius of point v

---

Advantages

- Can discover arbitrary shape clusters
- Robust to noise/outliers
- Doesn't need number of cluster

Disadvantages

- Non-deterministic results (border points may have different clusters depending on code)
- If data is not well-explained, choosing ***ϵ*** and *minPts* may be difficult
- May fail if data too sparse

Algorithm :

Step 1 : Label all the points , **core/border/noise**

Step 2 : Remove all the noise/outlier points

(or set them as *Noise*)

Step 3 : ***For every core point***, Assign points that are *density-Reachable* from that core and make it a cluster

Step 4 : Reassign border points belonging to more than 1 cluster to be the one closest to it

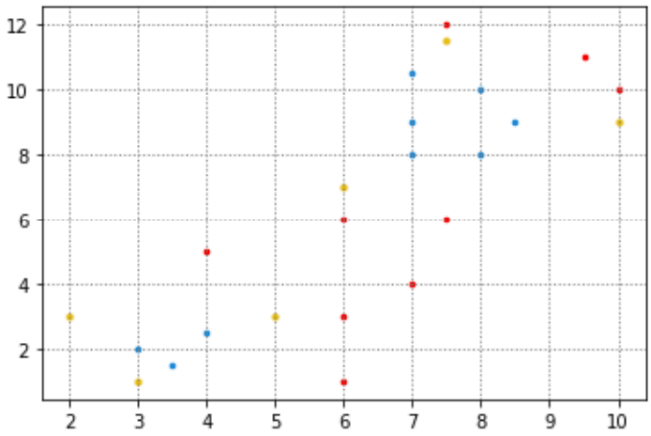
Done!

Example of a run :

Step 1 : Label all the points , **core/border/noise**

have set:

- Radius of Circle (ϵ) → *1.5*
- Number of Neighbors (minPts) → *4*

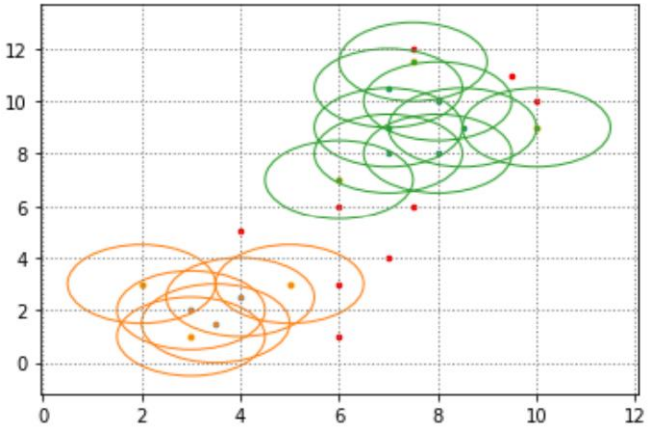


Step 2 : Remove all the noise/outlier points

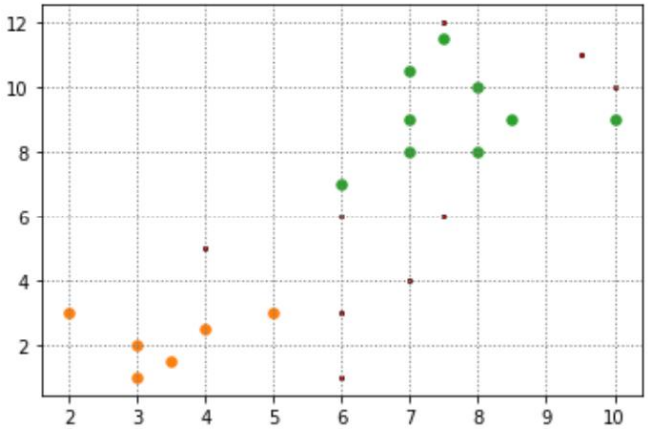
(in code, we can just ignore them to increase efficiency of program)

```
onlyList = core + border
```

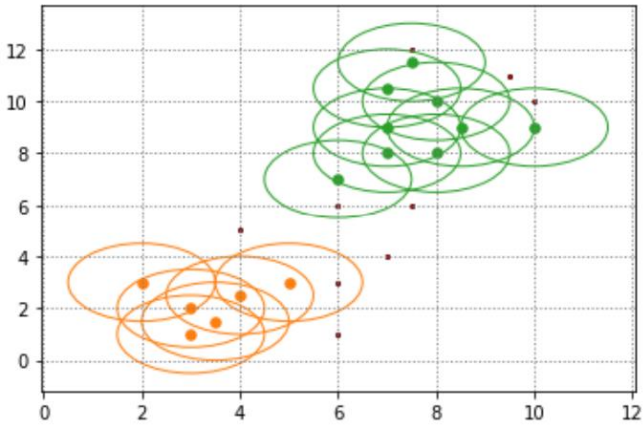
Step 3 : ***For every core point***, Assign points that are *density-Reachable* from that core and make it a cluster



Without radius shown :



Step 4 : Reassign border points belonging to more than 1 cluster to be the one closest to it



(In this case, we don't need to as border doesn't belong to more than 1 cluster)

Done!

Stuff I yoinked online

Model	Pros	Cons	Use Cases
K means	Quickest centroid based algorithm	Suffers when there is noise in the data	Even cluster size, flat geometry, not too many clusters and general-purpose
	Very lucid and can scale up for large amount of data sets	Outliers can never be identified	
	Reduces intra-cluster variance measure	Even though it reduces intra-cluster variance, it faces local minimum problem	
		Not ideal for data sets of non-convex shapes	
		Complicated to predict best K value	
Agglomerative Clustering	Embedded flexibility regarding level of granularity using dedrogram	Computationaly expensive	Possibly connectivity constraints, non Euclidean distances and many clusters
	Can handle of any forms of similarity or distance	Can't handle outliers	
		Ward's algorithm usually generates equal size clusters	
DBSCAN	Resistant to outliers	Highly sensitive to the two parameters- Eps and Min points	Uneven cluster sizes and non-flat geometry
	Can handle clusters of different shapes and sizes	DBSCAN cannot cluster data sets well with large variances in densities	
	Not required to specify the number of clusters		
GMM	Robust to outiers	The algorithm is highly complex and can be slow	Good for density estimation and flat geometry
	Provides the BIC score for selecting paramteres		
	Converges fast given good initialisation		

Supervised Learning

Supervised Learning:

Learning with *Labelled* Data (labelled by humans)

Terminology

Input : Attributes / features / independent variable

Output : target / response / dependent variable

Function : hypothesis / predictor

How it works :

Needs *labelled* data in the form of (Input , Output) and then it begins to train

Example of a labelled data:

$$\left((x_1^n, x_2^n, \dots x_d^n), y^n\right)$$

Can be any number of inputs(n), with any number of attributes(d)

Annotated training data

Examples

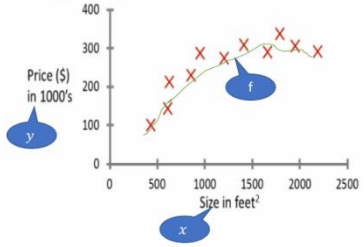
	Attributes				Labels
Number	Lines	Line types	Rectangles	Colours	Mondrian?
1	6	1	10	4	No
2	4	2	8	5	No
3	5	2	7	4	Yes
4	5	1	8	4	Yes
5	5	1	10	5	No
6	6	1	8	6	Yes
7	7	1	14	5	No
Number	Lines	Line types	Rectangles	Colours	Mondrian?
8	7	2	9	4	???

After training, present new input and it will provide an Output based on the new input

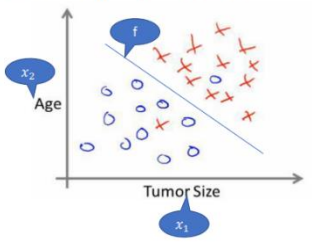
Supervised Learning is finding a good function, ***f***

Some other examples :

• Regression problem
Housing price prediction.

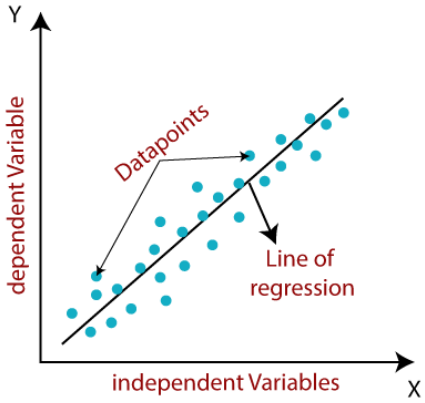


• Classification problem
Breast cancer prediction



Linear Regression:

Learning a function, f that captures a *"trend"* between input and output in the form of a straight line



Mathematically :

$$y = w_0 + w_1 x$$

Where:

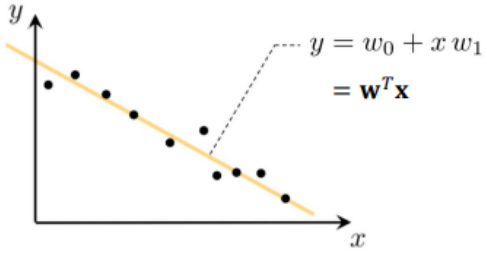
$$y = Dependent\ Variable$$
$$x = Independent\ Variable$$
$$w_0 = Intercept\ of\ the\ line$$
$$w_1 = Linear\ Regression\ coefficient$$

Types of Linear Regression

There are mainly 2 types :

- Univariate Linear Regression

Only 1 independent variable, x used to predict the value of dependent variable, y



Some examples :

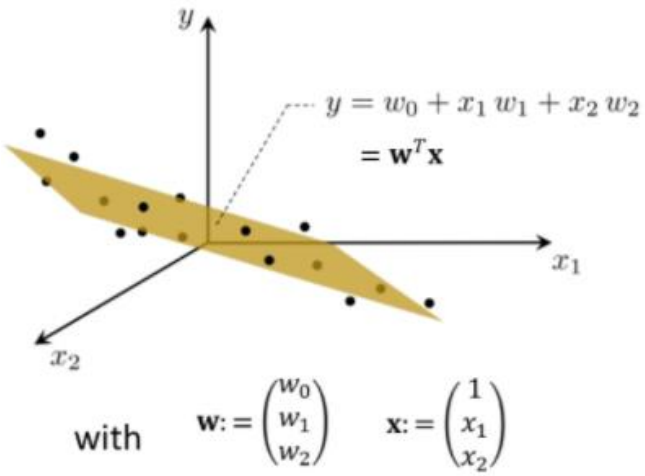
$$y = w_0 + w_1 x_1$$

$$y^n = w_0 + w_1 x^1 + w_2 x^2 + w_3 x^3 + \cdots + w_n x^n$$

Note: this is nonlinear (polynomial regression model)

- **Multivariate Linear Regression**

More than 1 independent variables, x used to predict the value of dependent variable, y



Some examples :

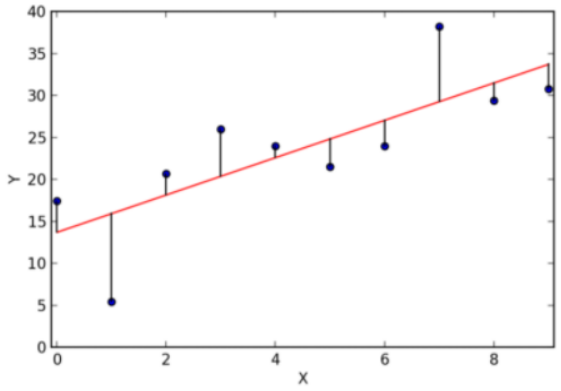
$$y = w_0 + w_1x_1 + w_2x_2 + w_3x_3 + \cdots + w_nx_n$$

$$y = w_0 + w_1x_1 + w_2x_2 + w_3x_3x_2$$

Note: this is nonlinear

Loss/Cost Function

Used to find *"line of best fit"* ,so needed a function to determine the best value for w_0 and w_1



We use ***MSE , Mean Square Error*** :

(which also can be interpreted as "Average of losses")

$$Cost = \frac{1}{N}\sum_{i=0}^{N}\left(y_{f(x)_i} - y_{(P)_i}\right)^2$$

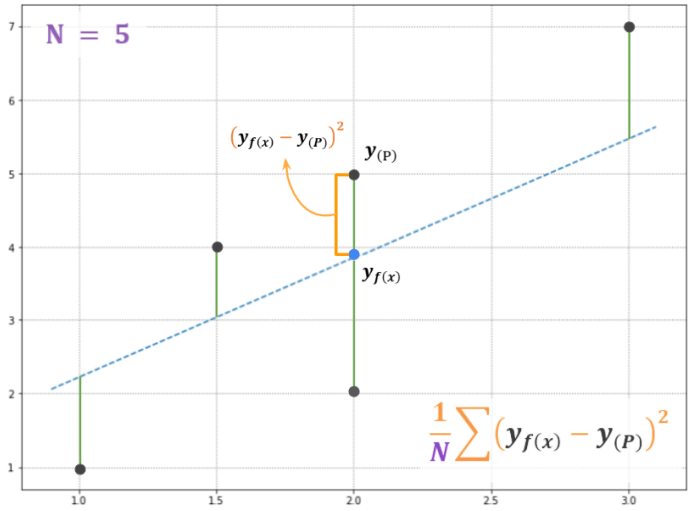
Where:

$$N = Number\ of\ Points/Data$$

$$y_{f(x)} = current\ line\ y - value$$

$$y_{(P)} = Point\ y - value$$

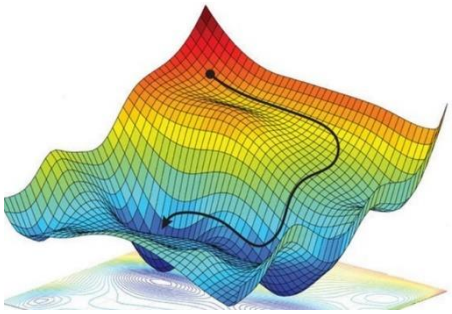
Example of finding MSE :



And our goal is to **minimize** MSE *(will talk more below in Gradient Descent)*

Gradient Descent:

A general strategy to **minimize** cost function



It does this by finding its *steepest descent* , $\Delta g(w)$

(I will not explain this in detail)

General Idea :

Step 1 : **Initialize** *random values* for *w* and Learning Rate

(0 or random for w, Learning Rate > 0)

Step 2 : **Take the gradient** (partial derivations) of the Loss function for each parameters, *w* in it

Step 3 : **Calculate** the Slope using the parameters *(the partial derivations)*

Step 4 : **Calculate** the Step size

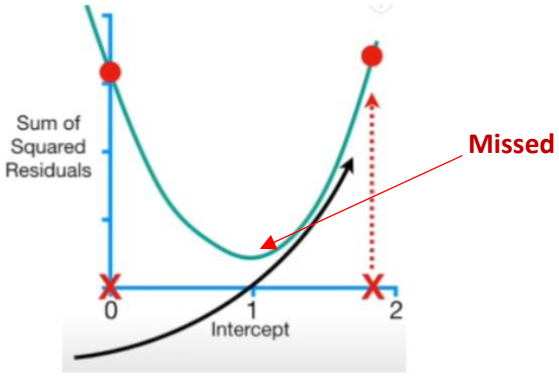
$$Step\ Size = \text{Slope} \times \text{LearningRate}$$

Step 5 : **Calculate** the new parameters, *w*

$$w_{new} = w_{\text{old}} - \text{StepSize}$$

Step 6 : **Repeat Step 3** until no or small changes/ max iteration reached

If *step size* is *too high*, will *never reach minimum* :



Algorithm (Linear Regression):

Step 1 : Initialize Learning Rate (α) , w_0 and w_1

(w_0 and w_1 can be anything, 0 or random)

(Learning Rate (α) must be small and > 0 , *eg: 0.001*)

Step 2 : Calculate the Slope , S_w *(the partial derivations)* using the parameters, w for each parameters (w_0 and w_1)

Since Cost is :

$Cost = \frac{1}{N}\sum_{i=0}^{N}\left(y_{f(x)_i} - y_{(P)_i}\right)^2$ and $y_{f(x)_i} = w_1 x_{(P)_i} + w_0$:

$$Cost = \frac{1}{N}\sum_{i=0}^{N}\left(\left(w_1 x_{(P)_i} + w_0\right) - y_{(P)_i}\right)^2$$

Partial Derivations or Slope , S_w for w_0 and w_1 *is* :

$$S_{w0} = \frac{2}{N}\sum_{i=0}^{N}\left(y_{f(x)_i} - y_{(P)_i}\right)$$

$$S_{w1} = \frac{2}{N}\sum_{i=0}^{N}\left(\left(y_{f(x)_i} - y_{(P)_i}\right) \times x_{(P)_i}\right)$$

Not needed

$$Cost = Cost + \left(y_{f(x)} - y_{(P)}\right)^2$$

Step 3 : Calculate the Step Size

$$StepSize = S_w \times \alpha$$

Step 4 : Calculate the new parameters , , w_0 and w_1

$$w_{0(new)} = w_0 - StepSize$$

$$w_{1(new)} = w_1 - StepSize$$

Step 5 : Assign new parameters as the old parameters

$$w_{old} = w_{new}$$

Step 6 : Repeat Step 2 until no or small changes/ max iterations reached

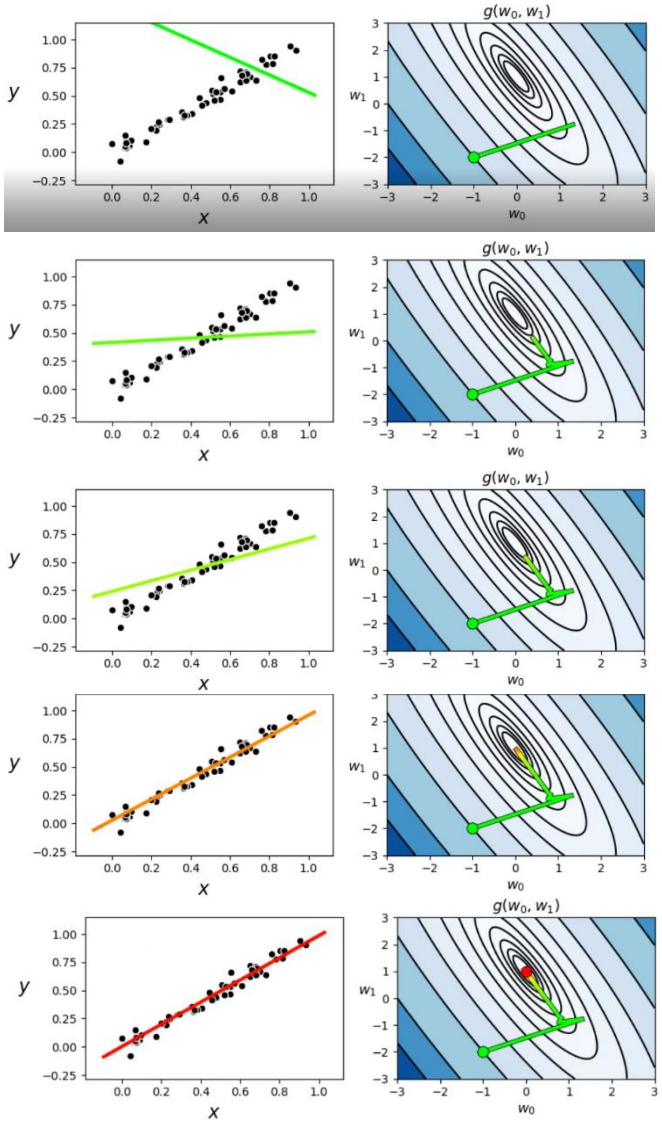
$$Cost = \frac{Cost}{N}$$ Not needed

Done!

Code:

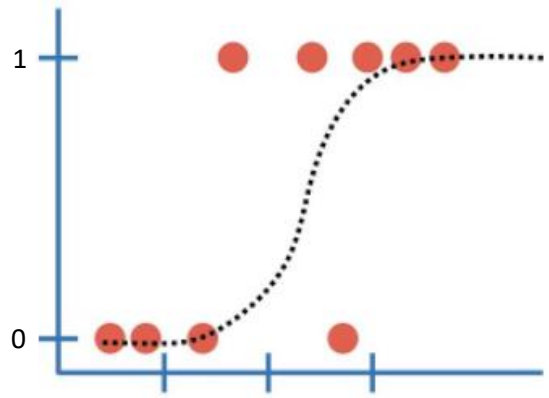
```
w0 = 0; w1 = 0; learningRate = 0.001;   # Step 1
testDataSet = [(1,1), (2,5), (3,11)]
size = len(testDataSet)                 # Number of Points

for epoch in range(5):                  # Number of times it runs
    cost = 0
    Sum_w0 , Sum_w1 = 0,0
    for point in (testDataSet):         # Finding Sum for Slope

        x,y = point
        y_fx = w0 + w1*x

        cost += ((y_fx - y)**2)         # Step 2
        Sum_w0 += (y_fx-y)
        Sum_w1 += (y_fx-y)*x

    Sw0 = (2/size) * Sum_w0   # getting the Slope/partial derivation
    Sw1 = (2/size) * Sum_w1

    w0 -= learningRate*Sw0    # Step 3,4,5
    w1 -= learningRate*Sw1

    cost = cost/size          # getting MSE

    print("epoch {} :".format(epoch),cost,w0,w1)
```

Example of a run :



Logistic Regression

Similar to Linear Regression, but a better Linear Model for Classification and predictions are usually Binary (True,1/False,0)



Can achieve this by passing Linear model through nonlinearity , which is the **Sigmoid Function**

Mathematically :

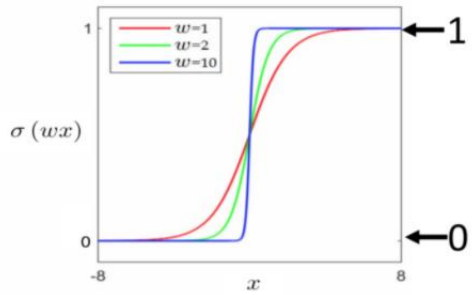
$$y = \sigma(w_0 + w_1 x)$$

Where:

$\sigma = Sigmoid\ Function$
$y = Dependent\ Variable$
$x = Independent\ Variable$
$w_0 = Intercept\ of\ the\ line$
$w_1 = Linear\ Regression\ coefficient$

Sigmoid Function

Also known as the Logistic Function



Function is :

$$\sigma(u) = \frac{1}{1 + e^{-u}}$$

- Values are always between 0 and 1
- $Constant$, w_0 : *shifts* the function (left & right)
- Gradient, w_n : determine the "*steepness*" (0 is flat)

More generally :

$$\sigma(u) = \frac{1}{1 + e^{-(w_0 + w_1 x_1 \ldots + w_n x_n)}}$$

Understanding the Sigmoid Function

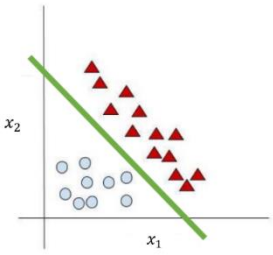
Our goal is to put a Boundary between 2 class

- If x has *1 attribute*, determined by a point

$$u = w_0 + w_1 x_1$$

- If x has *2 attributes*, determined by a line

$$u = w_0 + w_1 x_1 + w_2 x_2$$



- If x has *3 attributes*, determined by a plane

$$u = w_0 + w_1 x_1 + w_2 x_2 + w_3 x_3$$

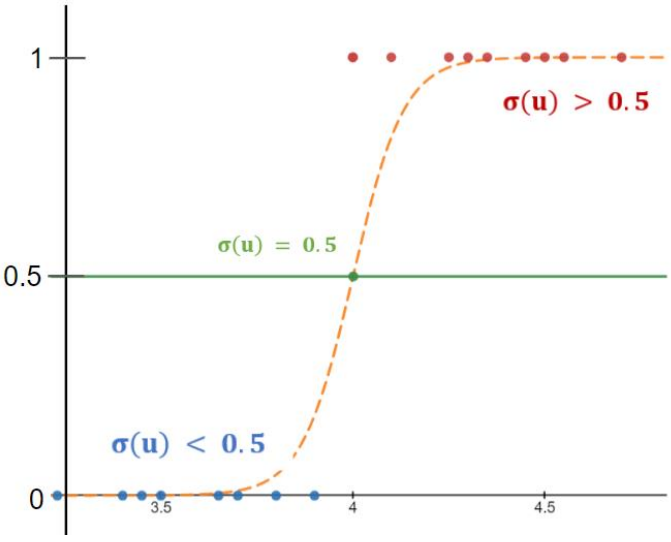
- If x has *4+ attributes*, determined by a hyperplane

$$u = w_0 + w_1 x_1 + w_2 x_2 + w_3 x_3 + \cdots + w_n x_n$$

Where y will usually be either be 1 / 0 (discrete values)

Since Sigmoid function always return a number between 0 and 1, so we can predict classes as :

- If $\sigma(u) < 0.5$, *Predict label 0*
  (This only happens if ***u < 0*** / Negative)
- If $\sigma(u) > 0.5$, *Predict label 1*
  (This only happens if ***u > 0*** / Positive)
- If $\sigma(u) = 0.5$, *Decision Boundary*
  (This only happens if ***u = 0***)



Note : *Decision Boundary* is **always** Linear

Example: (x has 2 attributes)

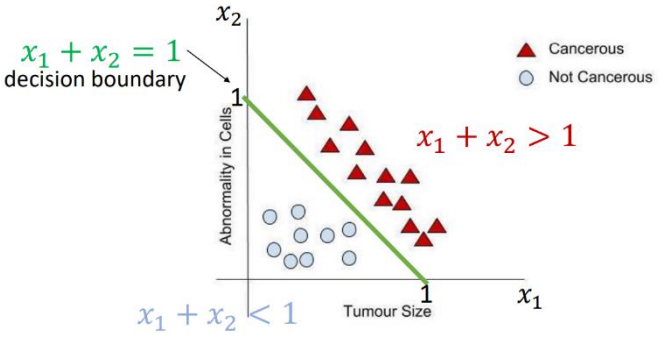
Suppose we have : $y = \sigma(w_0 + w_1x_1 + w_2x_2)$

And $w_0 = -1, w_1 = 1, w_2 = 1$:

We get → $u = (-1 + x_1 + x_2)$

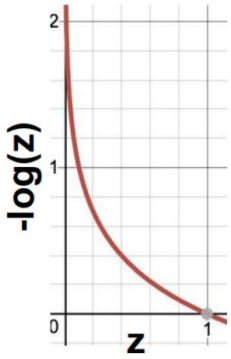
Since Decision Boundary is $u = 0$, we can now determine the labeling :

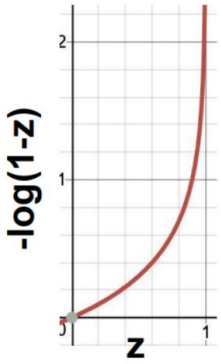
- Decision Boundary → $x_1 + x_2 = 1$
- Label 0 (negative) → $x_1 + x_2 < 1$
- Label 1 (positive) → $x_1 + x_2 > 1$



Loss/Cost Function

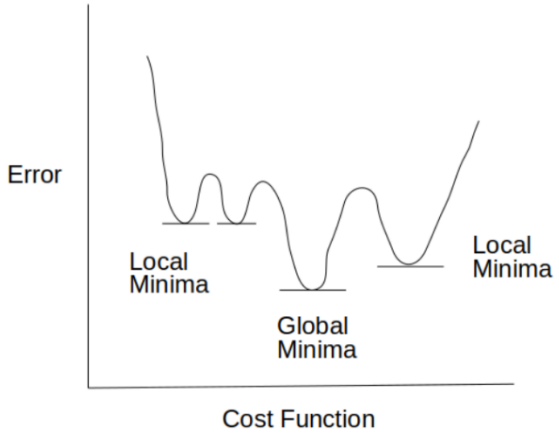
Similar to Linear Regression, requires a function to determine the best value for the parameters, *w*





Why not MSE?

May *get stuck in local optima* as logistic regression provides *non-convex outcome*



We use ***Cross-Entropy*** :

Assume $z = \sigma(w_0 + w_n x_n)$ which is between 0 and 1

$$PerCost_i = \begin{cases} -\log(1 - z) & \text{if } y = 0 \\ -\log(z) & \text{if } y = 1 \end{cases}$$

Examples :

When $y = 0$:

- Prediction is = 1 , $PerCost = \infty$
- Prediction is = 0.3 , $PerCost = -\log(1 - 0.3)$
- Prediction is = 0 , $PerCost = 0$

When $y = 1$:

- Prediction is = 1 , $PerCost = 0$
- Prediction is = 0.3 , $PerCost = -\log(0.3)$
- Prediction is = 0 , $PerCost = \infty$

Cross Entropy is the Average of these lost:

$$\mathbf{Cost} = \frac{1}{N}\sum_{i=0}^{N}(\mathbf{PerCost_i})$$

Or:

$$\mathbf{Cost} = -\frac{1}{N}\sum_{i=0}^{N}((y_i)(\log(z)) + (1 - y_i)(log(1 - z)))$$

Where:

N = *Number of Points/Data*

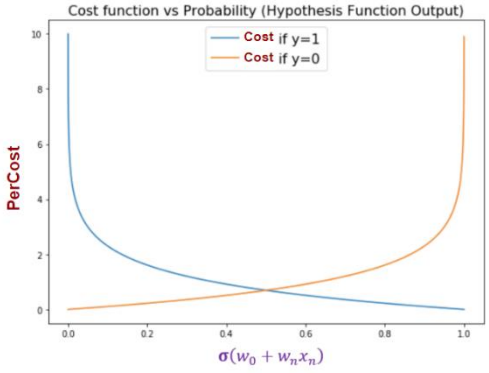
$\mathbf{PerCost_i}$ = current *Cost for that point* i

$y_i = 0$ *or* 1

Idea of ***Cross Entropy*** :

As *Predicted labelled **deviates** from the actual label*, cost increases significantly

(cost = 0 for correctly labelled , cost = ∞ for completely incorrectly labelled)



Same as before, our goal is to minimize **Cross-Entropy**

Gradient Descent:

(Could refer back to Linear Regression on concept of gradient descent) Slope is now

$$\nabla g(w) = -\frac{1}{N}\sum_{i=1}^{N}(y_i - \sigma(\mathbf{w_0} + \mathbf{w_i x_i} \dots + \mathbf{w_n x_n}))\boldsymbol{x_i}$$

Algorithm (Logistic Regression):

Step 1 : Initialize Learning Rate (α) , w_0 and w_1

(w_0 and w_1 can be anything, 0 or random)

(Learning Rate (α) must be small and > 0 , *eg: 0.5*)

Step 2 : Calculate the **Slope** , S_w *(the partial derivations)* using the parameters, w for each parameters (w_0 and w_1)

Since Cost is :

$$\mathbf{Cost} = -\frac{1}{N}\sum_{i=0}^{N}((y_i)(\log(z)) + (1 - y_i)(log(1-z)))$$

and $z = \sigma(w_1 x_{(P)_i} + w_0)$:

$$\mathbf{Cost} = -\frac{1}{N}\sum_{i=0}^{N}((y_i)(\log(\sigma(w_1 x_{(P)_i} + w_0))) + (1 - y_i)(log(1 - \sigma(w_1 x_{(P)_i} + w_0))))$$

Partial Derivations or **Slope** , S_w for w_0 and w_1 *is* :

$$S_{w0} = -\frac{1}{N}\sum_{i=0}^{N}\left(y_{(P)_i} - \sigma(w_1 x_{(P)_i} + w_0)\right)$$

$$S_{w1} = -\frac{1}{N}\sum_{i=0}^{N}\left(y_{(P)_i} - \sigma(w_1 x_{(P)_i} + w_0)\right)x_{(P)_i}$$

Step 3 : Calculate the **Step Size**

$$StepSize = S_w \times \alpha$$

Step 4 : Calculate the new parameters , , w_0 and w_1

$$w_{0(new)} = w_0 - StepSize$$

$$w_{1(new)} = w_1 - StepSize$$

Step 5 : Assign new parameters as the old parameters

$$w_{old} = w_{new}$$

Step 6 : Repeat Step 2 until no or small changes/ max step size set (iterations finish)

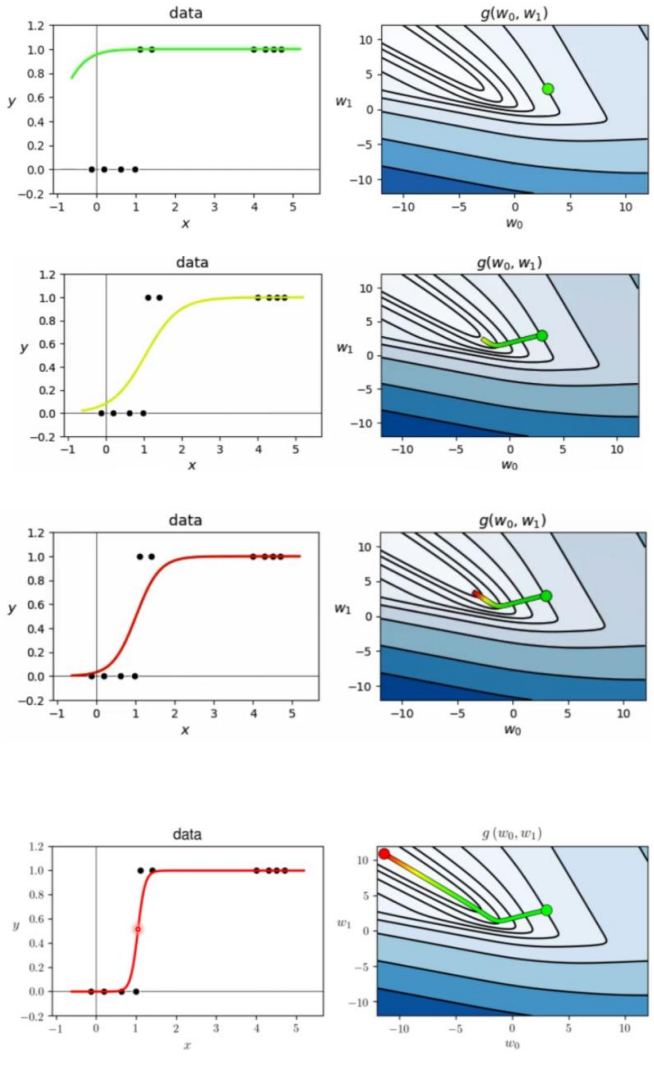
$$Cost = -\frac{Cost}{N}$$

Cost is negative initially

Done

```python
import math

w0 = 0; w1 = 0; learningRate = 0.5;            # Step 1
testDataSet = [(1,1),(3,0),(2,1)]
size = len(testDataSet)

# finding z
def sigmoid(w0,w1,x):
    return (1/(1+ math.exp(-(w0+w1*x))))


for epoch in range(25):
  cost = 0
  Sum_w0 , Sum_w1 = 0,0
  for point in (testDataSet):              # Step 2
    x,y = point
    z = sigmoid(w0,w1,x)
    #finding cost
    cost = cost + (((y)*(math.log(z)))+((1-y)*(math.log(1-z))))

    #getting Sum for Slope
    Sum_w0 += (y-z)
    Sum_w1 += (y-z)*x

  #getting Slope/partial derivation
  Sw0 = -(1/size) * Sum_w0
  Sw1 = -(1/size) * Sum_w1

  w0 -= learningRate*Sw0               # Step 3,4,5
  w1 -= learningRate*Sw1

  cost = -(cost/size)                 #getting Cross-Entropy

  print("epoch {} :".format(epoch),cost,w0,w1)
```

Example of a run :



Neural Networks

***note: I have no clue what this is and its very vague but here you go*

What is it, types of NN cost function , Gradient descent

,overfitting , Dealing with overfitting