LAB 1

Implement A* Search Algorithm

A* Search Algorithm is a Path Finding Algorithm. It is similar to Breadth First Search(BFS). It will search shortest path using heuristic value assigned to node and actual cost from Source_node to Dest_node

Real-life Examples

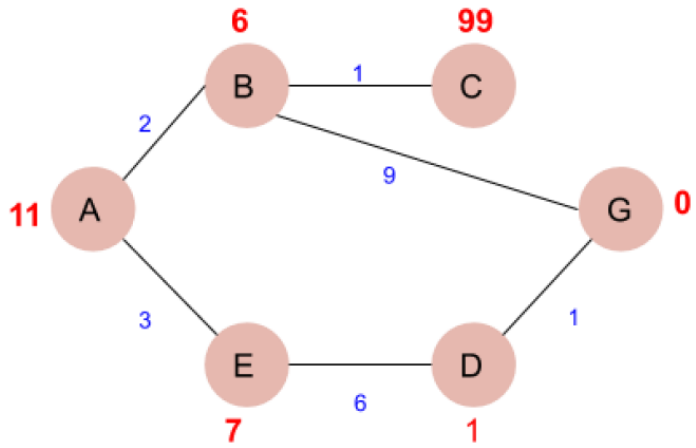
- Maps
- Games

Formula for A* Algorithm

h(n) =heuristic_value
g(n) =actual_cost
f(n) =actual_cost+heursitic_value

f(n) = g(n) + h(n)

EXAMPLE



Numbers written on edges represent the distance between nodes. Numbers written on nodes represent the heuristic value.

Given the graph, find the cost-effective path from A to G. That is A is the source node and G is the goal node.

Now from A, we can go to point B or E, so we compute f(x) for each of them,

A → B = g(B) + h(B) = 2 + 6 = 8

A → E = g(E) + h(E) = 3 + 7 = 10

Since the cost for A → B is less, we move forward with this path and compute the f(x) for the children nodes of B.

See also Problem Characteristics in Artificial Intelligence
Now from B, we can go to point C or G, so we compute f(x) for each of them,

$A \rightarrow B \rightarrow C = (2 + 1) + 99 = 102$

$A \rightarrow B \rightarrow G = (2 + 9) + 0 = 11$

Here the path $A \rightarrow B \rightarrow G$ has the least cost but it is still more than the cost of $A \rightarrow E$, thus we explore this path further.

Now from E, we can go to point D, so we compute f(x),

$A \rightarrow E \rightarrow D = (3 + 6) + 1 = 10$

Comparing the cost of $A \rightarrow E \rightarrow D$ with all the paths we got so far and as this cost is least of all we move forward with this path.

Now compute the f(x) for the children of D

$A \rightarrow E \rightarrow D \rightarrow G = (3 + 6 + 1) + 0 = 10$

Now comparing all the paths that lead us to the goal, we conclude that $A \rightarrow E \rightarrow D \rightarrow G$ is the most cost-effective path to get from A to G.

EXPLANATION

1. **aStarAlgo(start_node, stop_node)**: This is the main function that takes two arguments, **start_node** and **stop_node**, representing the starting and stopping nodes for the pathfinding algorithm.
2. Initialize some data structures:
    - **open_set**: A set containing nodes to be explored.
    - **closed_set**: A set containing nodes that have been explored.
    - **g**: A dictionary to store the distance from the starting node to each node encountered.
    - **parents**: A dictionary to store the parent node for each node encountered.
3. Set the initial values for the starting node in **g** and **parents**.
4. Enter a **while** loop that continues until **open_set** is not empty.
5. Inside the loop:
    - Find the node **n** from **open_set** with the lowest total cost (**g** + heuristic).
    - Check if **n** is the **stop_node** or if it has no neighbors (represented in **Graph_nodes**).
    - If neither of the above conditions is met, iterate through the neighbors of **n** using the **get_neighbors** function.
    - For each neighbor**m**, calculate its tentative **g** score from the current node **n**.
    - If **m** is not in **open_set** or **closed_set**, add it to **open_set**, update its **g** score, and set **n** as its parent.

- If **m** is already in **open_set** or **closed_set**, compare the new **g** score with the existing one. If the new score is better, update **g**, change the parent to **n**, and move **m** from **closed_set** to **open_set**.

6. If **n** is still **None** after the loop, it means there's no path from the **start_node** to the **stop_node**, and the function returns **None**.
7. If **n** is equal to the **stop_node**, a path has been found. Reconstruct the path from the **stop_node** to the **start_node** using the **parents** dictionary.
8. Finally, print the path and return it. If there's no path, it prints a message and returns **None**.
9. There are also three helper functions defined:
    - **get_neighbors(v)**: Returns the neighbors of a node **v** from the **Graph_nodes** dictionary.
    - **heuristic(n)**: Returns the heuristic distance for a given node **n** based on the **H_dist** dictionary.
    - **Graph_nodes**: Defines the graph with nodes and their neighbors along with edge weights.

The code then calls the **aStarAlgo** function with 'A' as the starting node and 'J' as the stopping node to find the shortest path in the provided graph.

LAB 2

Implement AO* Algorithm

AO* Search Algorithm is a Path Finding Algorithm and it is similar to A* star, other than AND is used between two nodes along with OR. After getting shortest path it will return back to root node and it will update it's heuristic value. It is similar to Depth First Search(DFS). It will search shortest path using heuristic value assigned to node and actual cost from Source_node to Dest_node

What is difference between A * and AO * algorithm?

An A* algorithm represents an OR graph algorithm that is used to find a single solution (either this or that). An AO* algorithm represents an AND-OR graph algorithm that is used to find more than one solution by ANDing more than one branch.

Real-life Examples

- Maps
- Games

Formula for AO* Algorithm

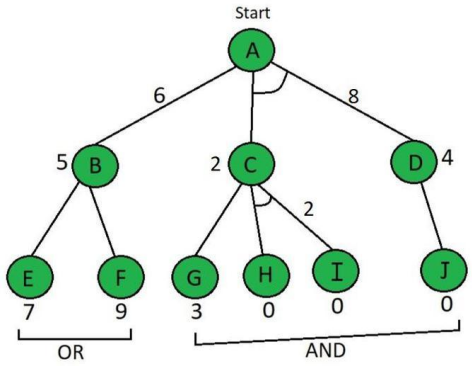
h(n) =heuristic_value
g(n) =actual_cost
f(n) =actual_cost+heursitic_value

f(n) = g(n) + h(n)

Difference between the A* Algorithm and AO* algorithm

- A* algorithm and AO* algorithm both works on the **best first search**.
- They are both **informed search** and works on given heuristics values.
- **A*** always **gives** the **optimal solution** but AO* doesn't guarantee to give the optimal solution.
- Once AO* got a solution **doesn't explore** all possible paths but A* explores all paths.
- When compared to the A* algorithm, the AO* algorithm uses **less memory.**
- opposite to the A* algorithm, the AO* algorithm cannot go into an endless **loop.**

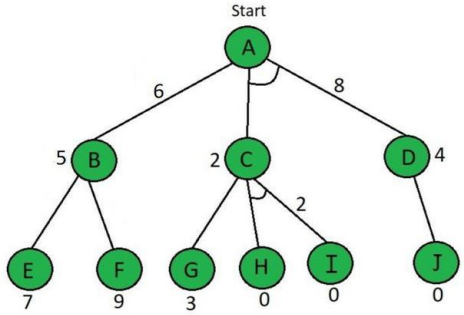
Example:



AO Algorithm – Question tree*

Here in the above example below the Node which is given is the heuristic value i.e **h(n)**. Edge length is considered as **1**.

Step 1



AO Algorithm (Step-1)*

With help of **f(n) = g(n) + h(n)** evaluation function,

Start from node A,

f(A⇢B) = g(B) + h(B)

= 1 + 5 ……here **g(n)=1** is taken by default for path cost

= 6

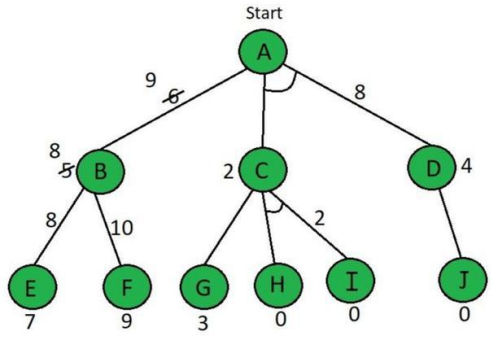
f(A⇢C+D) = g(c) + h(c) + g(d) + h(d)

= 1 + 2 + 1 + 4 ……here we have added **C & D** because they are in **AND**

= 8

So, by calculation **A⇢B** path is chosen which is the minimum path, i.e **f(A⇢B)**

Step 2



AO Algorithm (Step-2)*

According to the answer of step 1, explore node B

Here the value of E & F are calculated as follows,

f(B⇢E) = g(e) + h(e)

f(B⇢E) = 1 + 7

= 8

f(B⇢f) = g(f) + h(f)

f(B⇢f) = 1 + 9

= 10

So, by above calculation B⇢E path is chosen which is minimum path, i.e **f(B⇢E)** because **B's** heuristic value is different from its actual value The heuristic is updated and the minimum cost path is selected. The minimum value in our situation is **8**. Therefore, the heuristic for **A** must be updated due to the change in **B's** heuristic. So we need to calculate it again.

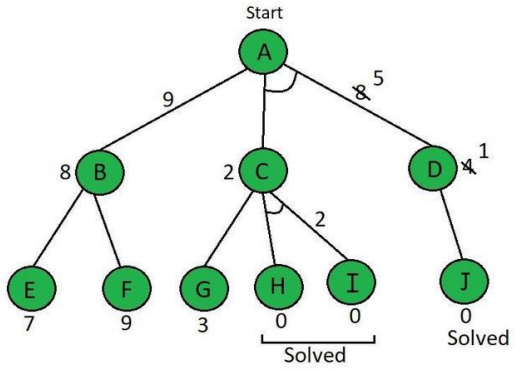
f(A⇢B) = g(B) + updated h(B)

= 1 + 8

= 9

We have Updated all values in the above tree.

Step 3



By comparing **f(A⇢B)** & **f(A⇢C+D)**

f(A⇢C+D) is shown to be **smaller**. i.e 8 < 9

Now explore f(A⇢C+D)

So, the current node is **C**

f(C⇢G) = g(g) + h(g)

f(C⇢G) = 1 + 3

= 4

f(C⇢H+I) = g(h) + h(h) + g(i) + h(i)

f(C⇢H+I) = 1 + 0 + 1 + 0 ……here we have added **H & I** because they are in **AND**

= 2

f(C⇢H+I) is selected as the path with the lowest cost and the heuristic is also left unchanged because it matches the actual cost. Paths H & I are solved because the heuristic for those paths is **0,** but Path **A⇢D** needs to be calculated because it has an **AND**.

f(D⇢J) = g(j) + h(j)

f(D⇢J) = 1 + 0

= 1

the heuristic of node D needs to be updated to 1.

f(A⇢C+D) = g(c) + h(c) + g(d) + h(d)

= 1 + 2 + 1 + 1

= 5

as we can see that path **f(A⇢C+D)** is get solved and this **tree has become a solved tree** now.

In simple words, the main flow of this algorithm is that we have to find **firstly level 1st** heuristic value and **then level 2nd** and after that **update the values** with going **upward** means towards the root node.

In the above tree diagram, we have updated all the values.

LAB 3

For a given set of training data examples stored in a .CSV file, implement and demonstrate the Candidate□Elimination algorithm to output a description of the set of all hypotheses consistent with the training example

The candidate elimination algorithm incrementally builds the version space given a hypothesis space H and a set E of examples. The examples are added one by one; each example possibly shrinks the version space by removing the hypotheses that are inconsistent with the example. The candidate elimination algorithm does this by updating the general and specific boundary for each new example.

- We can consider this as an extended form of the Find-S algorithm.
- Consider both positive and negative examples.
- Actually, positive examples are used here as the Find-S algorithm (Basically they are generalizing from the specification).
- While the negative example is specified in the generalizing form.

Terms:

- Concept learning: Concept learning is basically the learning task of the machine (Learn by Train data)
- General Hypothesis: Not Specifying features to learn the machine.
- G = {'?', '?','?','?'...}: Number of attributes
- Specific Hypothesis: Specifying features to learn machine (Specific feature)
- S= {'pi','pi','pi'...}: The number of pi depends on a number of attributes.

- Version Space: It is an intermediate of general hypothesis and Specific hypothesis. It not only just writes one hypothesis but a set of all possible hypotheses based on training data-set.

Algorithm:
Step1: Load Data set
Step2: Initialize General Hypothesis and Specific Hypothesis.
Step3: For each training example
Step4: If example is positive example
    if attribute_value == hypothesis_value:
      Do nothing
    else:
      replace attribute value with '?' (Basically generalizing it)
Step5: If example is Negative example
    Make generalize hypothesis more specific.

Example:

Consider the dataset given below:

Sky	Temperature	Humid	Wind	Water	Forest	Output
sunny	warm	normal	strong	warm	same	yes
sunny	warm	high	strong	warm	same	yes
rainy	cold	high	strong	warm	change	no
sunny	warm	high	strong	cool	change	yes

Algorithmic steps:

Initially : G = [[?, ?, ?, ?, ?, ?], [?, ?, ?, ?, ?, ?], [?, ?, ?, ?, ?, ?],
[?, ?, ?, ?, ?, ?], [?, ?, ?, ?, ?, ?], [?, ?, ?, ?, ?, ?]]
S = [Null, Null, Null, Null, Null, Null]

For instance 1 : <'sunny','warm','normal','strong','warm ','same'> and positive output.
G1 = G
S1 = ['sunny','warm','normal','strong','warm ','same']

For instance 2 : <'sunny','warm','high','strong','warm ','same'> and positive output.
G2 = G
S2 = ['sunny','warm',?,'strong','warm ','same']

For instance 3 : <'rainy','cold','high','strong','warm ','change'> and negative output.
G3 = [['sunny', ?, ?, ?, ?, ?], [?, 'warm', ?, ?, ?, ?], [?, ?, ?, ?, ?, ?],
[?, ?, ?, ?, ?, ?], [?, ?, ?, ?, ?, ?], [?, ?, ?, ?, ?, 'same']]
S3 = S2

For instance 4 : <'sunny','warm','high','strong','cool','change'> and positive output.

G4 = G3

S4 = ['sunny','warm',?,'strong', ?, ?]

At last, by synchronizing the G4 and S4 algorithm produce the output.

Output :

G = [['sunny', ?, ?, ?, ?, ?], [?, 'warm', ?, ?, ?, ?]]

S = ['sunny','warm',?,'strong', ?, ?]

The Candidate Elimination Algorithm (CEA) is an improvement over the Find-S algorithm for classification tasks. While CEA shares some similarities with Find-S, it also has some essential differences that offer advantages and disadvantages. Here are some advantages and disadvantages of CEA in comparison with Find-S:

Advantages of CEA over Find-S:

1. Improved accuracy: CEA considers both positive and negative examples to generate the hypothesis, which can result in higher accuracy when dealing with noisy or incomplete data.
2. Flexibility: CEA can handle more complex classification tasks, such as those with multiple classes or non-linear decision boundaries.
3. More efficient: CEA reduces the number of hypotheses by generating a set of general hypotheses and then eliminating them one by one. This can result in faster processing and improved efficiency.
4. Better handling of continuous attributes: CEA can handle continuous attributes by creating boundaries for each attribute, which makes it more suitable for a wider range of datasets.

Disadvantages of CEA in comparison with Find-S:

1. More complex: CEA is a more complex algorithm than Find-S, which may make it more difficult for beginners or those without a strong background in machine learning to use and understand.
2. Higher memory requirements: CEA requires more memory to store the set of hypotheses and boundaries, which may make it less suitable for memory-constrained environments.
3. Slower processing for large datasets: CEA may become slower for larger datasets due to the increased number of hypotheses generated.
4. Higher potential for overfitting: The increased complexity of CEA may make it more prone to overfitting on the training data, especially if the dataset is small or has a high degree of noise.

LAB 4

Write a program to demonstrate the working of the decision tree based ID3 algorithm. Use an appropriate data set for building the decision tree and apply this knowledge to classify a new sample.

What is a Decision Tree?
A Supervised Machine Learning Algorithm, used to build classification and regression models in the form of a tree structure.
A decision tree is a tree where each -
Node - a feature(attribute)
Branch - a decision(rule)
Leaf - an outcome(categorical or continuous)

What is an ID3 Algorithm?

ID3 stands for **Iterative Dichotomiser 3**

It is a classification algorithm that follows a greedy approach by selecting a best attribute that yields maximum Information Gain(IG) or minimum Entropy(H).

What is Entropy and Information gain?

Entropy is a measure of the amount of uncertainty in the dataset S. Mathematical Representation of Entropy is shown here -

$$H(S) = \sum_{c \in C} -p(c) log_2 p(c)$$

Where,

S - The current dataset for which entropy is being calculated(changes every iteration of the ID3 algorithm).

C - Set of classes in S {example - C ={yes, no}}

p(c) - The proportion of the number of elements in class c to the number of elements in set S.

In ID3, entropy is calculated for each remaining attribute. The attribute with the smallest entropy is used to split the set S on that particular iteration.

Entropy = 0 implies it is of pure class, that means all are of same category.

Information Gain IG(A) tells us how much uncertainty in S was reduced after splitting set S on attribute A. Mathematical representation of Information gain is shown here -

$$IG(A,S) = H(S) - \sum_{t \in T} p(t) H(t)$$

Where,

H(S) - Entropy of set S.

T - The subsets created from splitting set S by attribute A such that

$$S = \bigcup_{t \in T} t$$

p(t) - The proportion of the number of elements in t to the number of elements in set S.

H(t) - Entropy of subset t.

In ID3, information gain can be calculated (instead of entropy) for each remaining attribute. The attribute with the largest information gain is used to split the set S on that particular iteration.

The steps in ID3 algorithm are as follows:

1. Calculate entropy for dataset.
2. For each attribute/feature.
   2.1. Calculate entropy for all its categorical values.
   2.2. Calculate information gain for the feature.
3. Find the feature with maximum information gain.
4. Repeat it until we get the desired tree.

Problem

Build a decision tree using ID3 algorithm for the given training data in the table (Buy Computer data), and predict the class of the following new example: age<=30, income=medium, student=yes, credit-rating=fair

age	income	student	Credit rating	Buys computer
<=30	high	no	fair	no
<=30	high	no	excellent	no
31...40	high	no	fair	yes
>40	medium	no	fair	yes
>40	low	yes	fair	yes
>40	low	yes	excellent	no
31...40	low	yes	excellent	yes
<=30	medium	no	fair	no
<=30	low	yes	fair	yes
>40	medium	yes	fair	yes
<=30	medium	yes	excellent	yes
31...40	medium	no	excellent	yes
31...40	high	yes	fair	yes
>40	medium	no	excellent	no

Solution:
First, check which attribute provides the highest Information Gain in order to split the training set based on that attribute. We need to calculate the expected information to classify the set and the entropy of each attribute.

The information gain is this mutual information minus the entropy:

The mutual information of the two classes,

Entropy(S)= E(9,5)= -9/14 log2(9/14) – 5/14 log2(5/14)=0.94
Now Consider the Age attribute
For Age, we have three values age<=30 (2 yes and 3 no), age31..40 (4 yes and 0 no), and age>40 (3 yes and 2 no)

Entropy(age) = 5/14 (-2/5 log2(2/5)-3/5log2(3/5)) + 4/14 (0) + 5/14 (-3/5log2(3/5)-2/5log2(2/5))
= 5/14(0.9709) + 0 + 5/14(0.9709) = 0.6935

Gain(age) = 0.94 – 0.6935 = 0.2465

Next, consider Income Attribute
For Income, we have three values incomehigh (2 yes and 2 no), incomemedium (4 yes and 2 no), and incomelow (3 yes 1 no)
Entropy(income) = 4/14(-2/4log2(2/4)-2/4log2(2/4)) + 6/14 (-4/6log2(4/6)-2/6log2(2/6)) + 4/14 (-3/4log2(3/4)-1/4log2(1/4))
= 4/14 (1) + 6/14 (0.918) + 4/14 (0.811)

= 0.285714 + 0.393428 + 0.231714 = 0.9108

Gain(income) = 0.94 – 0.9108 = 0.0292

Next, consider Student Attribute
For Student, we have two values studentyes (6 yes and 1 no) and studentno (3 yes 4 no)

Entropy(student) = 7/14(-6/7log2(6/7)-1/7log2(1/7)) + 7/14(-3/7log2(3/7)-4/7log2(4/7)

= 7/14(0.5916) + 7/14(0.9852)

= 0.2958 + 0.4926 = 0.7884

Gain (student) = 0.94 – 0.7884 = 0.1516

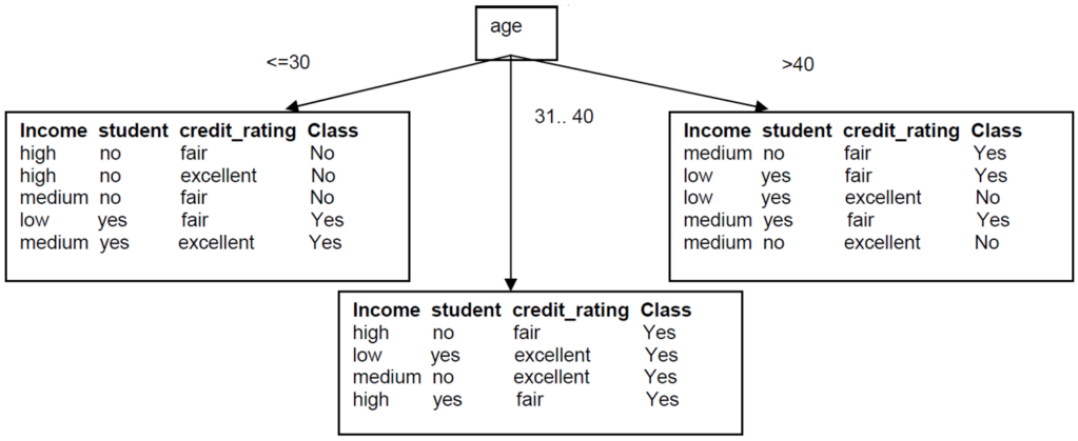
Finally, consider Credit_Rating Attribute
For Credit_Rating we have two values credit_ratingfair (6 yes and 2 no) and credit_ratingexcellent (3 yes 3 no)

Entropy(credit_rating) = 8/14(-6/8log2(6/8)-2/8log2(2/8)) + 6/14(-3/6log2(3/6)-3/6log2(3/6))
= 8/14(0.8112) + 6/14(1)

= 0.4635 + 0.4285 = 0.8920

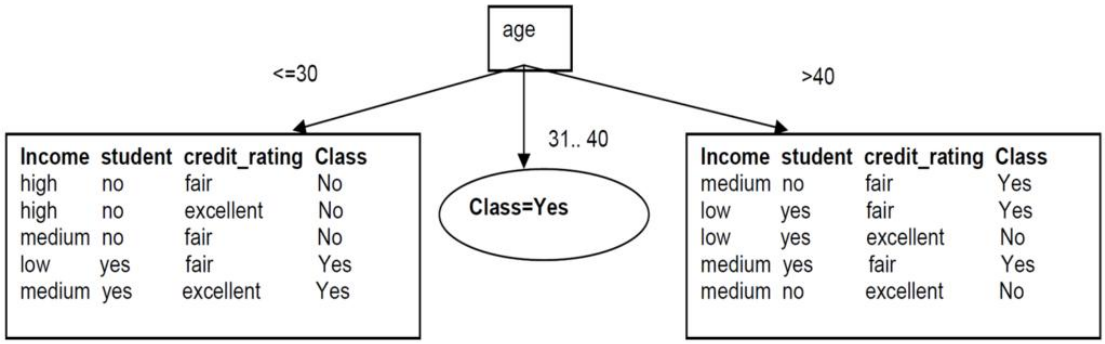
Gain(credit_rating) = 0.94 – 0.8920 = 0.479

Since Age has the highest Information Gain we start splitting the dataset using the age attribute.



Decision Tree after step 1

Since all records under the branch age31..40 are all of the class, Yes, we can replace the leaf with Class=Yes



Decision Tree after step 1_1

Now build the decision tree for the left subtree

The same process of splitting has to happen for the two remaining branches.

Income	student	credit_rating	Class
high	no	fair	No
high	no	excellent	No
medium	no	fair	No
low	yes	fair	Yes
medium	yes	excellent	Yes

Left sub-branch

For branch age<=30 we still have attributes income, student, and credit_rating. Which one should be used to split the partition?

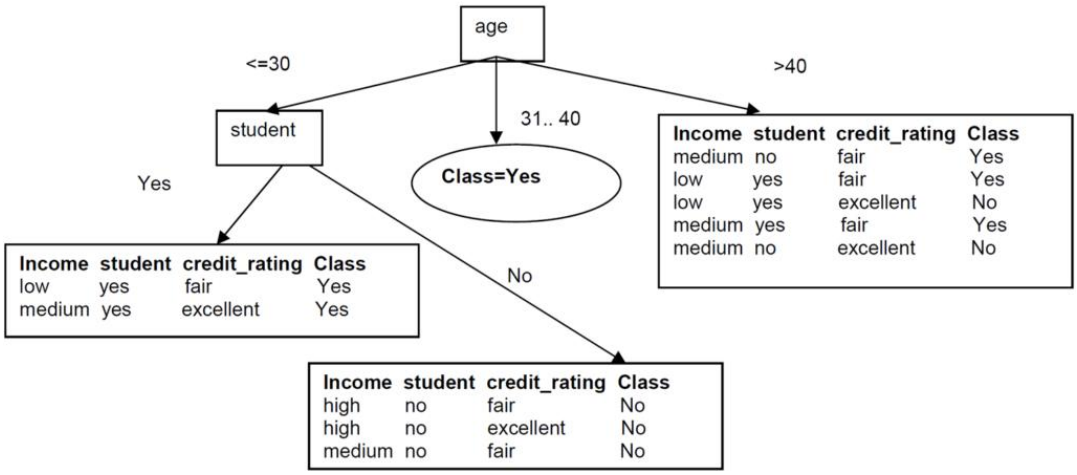
The mutual information is E(Sage<=30)= E(2,3)= -2/5 log2(2/5) – 3/5 log2(3/5)=0.97
For **Income,** we have three values incomehigh (0 yes and 2 no), incomemedium (1 yes and 1 no) and incomelow (1 yes and 0 no)
Entropy(income) = 2/5(0) + 2/5 (-1/2log2(1/2)-1/2log2(1/2)) + 1/5 (0) = 2/5 (1) = 0.4

Gain(income) = 0.97 – 0.4 = 0.57

For **Student,** we have two values studentyes (2 yes and 0 no) and studentno (0 yes 3 no)
Entropy(student) = 2/5(0) + 3/5(0) = 0

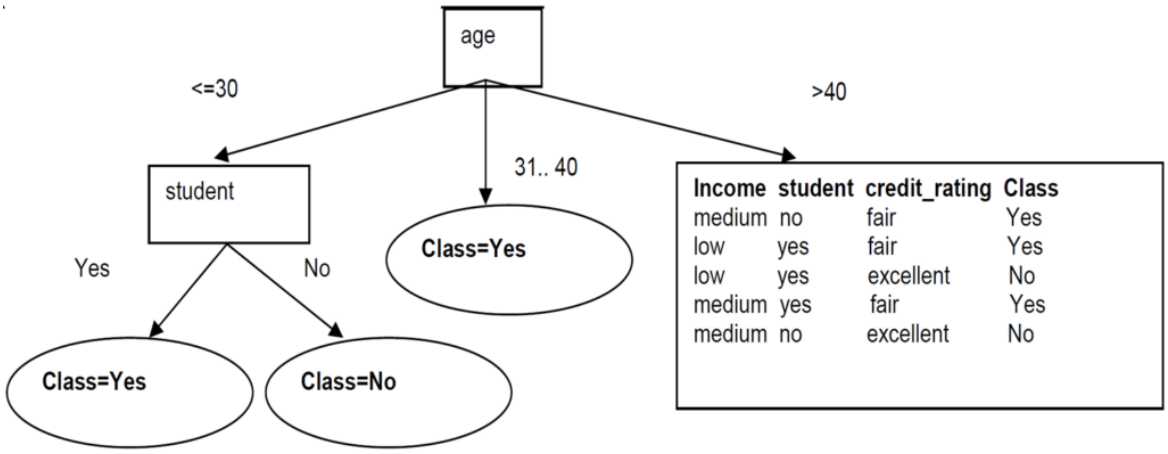
Gain (student) = 0.97 – 0 = 0.97

We can then safely split on attribute student without checking the other attributes since the information gain is maximized.



Decision Tree after step 2

Since these two new branches are from distinct classes, we make them into leaf nodes with their respective class as label:



Decision Tree after step 2_2

Now build the decision tree for right left subtree

Income	student	credit_rating	Class
medium	no	fair	Yes
low	yes	fair	Yes
low	yes	excellent	No
medium	yes	fair	Yes
medium	no	excellent	No

Right sub-branch

The mutual information is Entropy(Sage>40)= I(3,2)= -3/5 log2(3/5) – 2/5 log2(2/5)=0.97

For **Income**, we have two values incomemedium (2 yes and 1 no) and incomelow (1 yes and 1 no)

Entropy(income) = 3/5(-2/3log2(2/3)-1/3log2(1/3)) + 2/5 (-1/2log2(1/2)-1/2log2(1/2))

= 3/5(0.9182)+2/5 (1) = 0.55+0. 4= 0.95

Gain(income) = 0.97 – 0.95 = 0.02

For **Student**, we have two values studentyes (2 yes and 1 no) and studentno (1 yes and 1 no)

Entropy(student) = 3/5(-2/3log2(2/3)-1/3log2(1/3)) + 2/5(-1/2log2(1/2)-1/2log2(1/2)) = 0.95

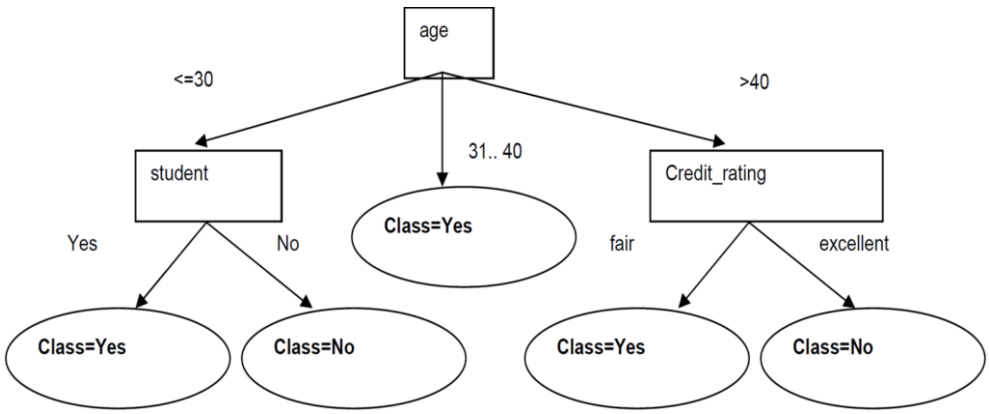
Gain (student) = 0.97 – 0.95 = 0.02

For **Credit_Rating**, we have two values credit_ratingfair (3 yes and 0 no) and credit_ratingexcellent (0 yes and 2 no)

Entropy(credit_rating) = 0

Gain(credit_rating) = 0.97 – 0 = 0.97

We then split based on credit_rating. These splits give partitions each with records from the same class. We just need to make these into leaf nodes with their class label attached:



Decision Tree for Buys Computer

New example: age<=30, income=medium, student=yes, credit-rating=fair

Follow branch(age<=30) then student=yes we predict Class=yes

Buys_computer = yes

LAB 5

Build an Artificial Neural Network by implementing the Back propagation algorithm and test the same using appropriate data sets..

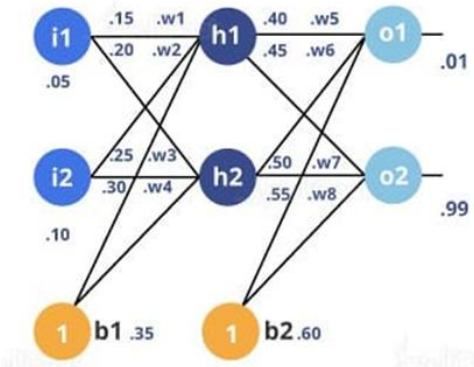
The goal of the back propagation algorithm is to optimize the weights so that the neural network can learn how to correctly map arbitrary inputs to outputs. Here, we will understand the complete scenario of back propagation in neural networks with the help of a single training set.

n order to have some numbers to work with, here are initial weights, biases, and training input and output.

Inputs(i1**): 0.05** Output (o1): **0.01**

Inputs(i2): **0.10** Output(o2):**0.99**

Step 1: The Forward Pass:



The total net input for h1: The net input for h1 (the next layer) is calculated as the sum of the product of each weight value and the corresponding input value and, finally, a bias value added to it.

$$net\ h1 = w_1 * i_1 + w_2 * i_2 + b_1 * 1 \quad \text{..... (Equation 1)}$$

$$net\ h1 = 0.15 * 0.05 + 0.2 * 0.1 + 0.35 * 1 = 0.3775$$

The output for h1: The output for h1 is calculated by applying a sigmoid function to the net input Of h1.

The sigmoid function pumps the values for which it is used in the range of 0 to 1.

It is used for models where we have to predict the probability. Since the probability of any event lies between 0 and 1, the sigmoid function is the right choice.

$$out\ h1 = 1/(1 + e^{-net\ h1}) = 1/(1 + e^{-0.3775}) = 0.593269992 \quad \text{..... (Equation 2)}$$

Carrying out the same process for h2:

out h2 = 0.596884378

The output for o1 is:

$$net\ o1 = w_5 * out\ h1 + w_6 * out\ h2 + b_2 * 1 \quad \text{..... (Equation 3)}$$

$$net\ o1 = 0.4 * 0.593269992 + 0.45 * 0.596884378 + 0.6 * 1 = 1.105905967$$

$$out\ o1 = 1/(1 + e^{-net\ o1}) = 1/(1 + e^{-1.105905967}) = 0.75136507 \quad \text{..... (Equation 4)}$$

Carrying out the same process for o2:

out o2 = 0.772928465

Calculating the Total Error:

We can now calculate the error for each output neuron using the squared error function and sum them up to get the total error: E total = Σ1/2(target – output)2

The target output for o1 is 0.01, but the neural network output is 0.75136507; therefore, its error is:

E o1 = 1/2(target o1 - out o1)2 = 1/2(0.01 - 0.75136507)2 = 0.27481108 …….…….. (Equation 5)

By repeating this process for o2 (remembering that the target is 0.99), we get:

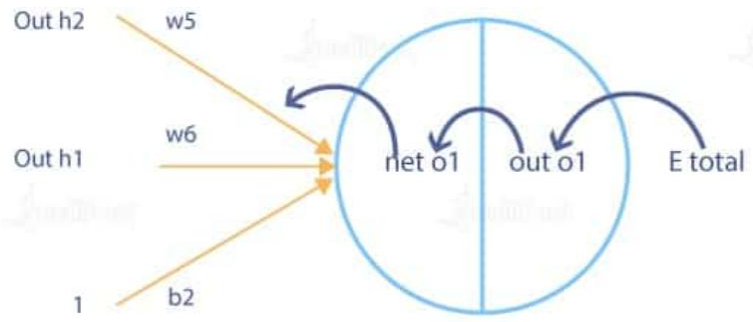
E o2 = 0.023560026

Then, the total error for the neural network is the sum of these errors:

E total = E o1 + E o2 = 0.274811083 + 0.023560026 = 0.298371109

Step 2: Backward Propagation:

Our goal with the backward propagation algorithm is to update each weight in the network so that the actual output is closer to the target output, thereby minimizing the error for each neuron and the network as a whole.



Consider w5; we will calculate the rate of change of error w.r.t the change in the weight w5:

$$(\partial E\ total)/\partial w_5 = (\partial E\ total)/(\partial out\ o1) * (\partial out\ o1)/(\partial net\ o1) * (\partial net\ o1)/\partial w_5$$

Since we are propagating backward, the first thing we need to calculate the change in total errors w.r.t the outputs o1 and o2:

$$E\ total = (1/2)*(target\ o1 - out\ o1)^2 + (1/2)*(target\ o2 - out\ o2)^2$$
$$(\partial E\ total)/(\partial out\ o1) = -(target\ o1 - out\ o1) = -(0.01 - 0.75136507) = 0.74136507$$

Now, we will propagate further backward and calculate the change in the output o1 w.r.t to its total net input:

$$out\ o1 = 1/(1 + e^{-net\ o1}) \quad (from\ Equation\ 4)$$

$$(\partial out\ o1)/(\partial net\ o1) = out\ o1(1-out\ o1) = 0.75136507(1 - 0.75136507) = 0.186815602$$

How much does the total net input of o1 change w.r.t w5?

$$net\ o1 = w_5*out\ h1 + w_6*out\ h2 + b_2*1 \quad (from\ Equation\ 3)$$

$$(\partial net\ o1)/\partial w_5 = 1*out\ h1*w_5^{(1-1)} + 0 + 0 = out\ h1 = 0.593269992$$

Putting all values together and calculating the updated weight value:

$$(\mathbf{\partial E\ total})/\partial w_5 = (\partial E\ total)/(\partial out\ o1) * (\partial out\ o1)/(\mathbf{\partial net}\ o1) * (\mathbf{\partial net}\ o1)/\partial w_5 = 0.082167041$$

Let's calculate the updated value of w5.

$$W_5^+ = W_5 - n*(\mathbf{\partial E\ total})/\partial w_5 = 0.4 - 0.5 * 0.082167041 = 0.35891648$$ *(This is gradient descent)*

We can repeat this process to get the new weights w6, w7, and w8.

$$W_6^+ = 0.408666186$$

$$W_7^+ = 0.511301270$$

$$W_8^+ = 0.561370121$$

We perform the actual updates in the neural network after we have the new weights leading into the hidden layer neurons.

We'll continue the backward pass by calculating new values for w1, w2, w3, and w4:

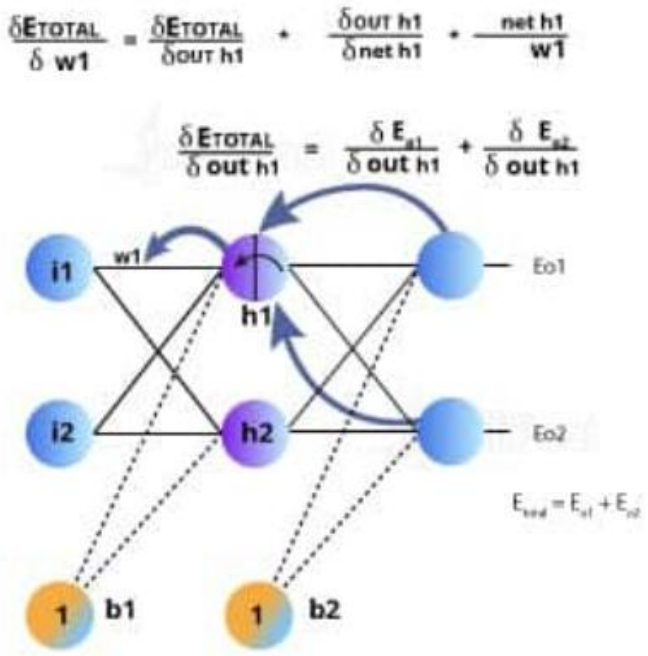
Starting with w1:

$$(\partial E\ total)/\partial w_1 = (\partial E\ total)/(\partial out\ h1) * (\partial out\ h1)/(\partial net\ h1) * (\partial net\ h1)/\partial w_1$$

$$(\partial E\ total)/(\partial out\ h1) = (\partial E\ o1)/(\partial out\ h1) + (\partial E\ o2)/(\partial out\ h1)$$

We're going to use a similar process as we did for the output layer, but slightly different to account for the fact that the output of each hidden layer neuron contributes to the final output. Thus, we need to take Eo1 and Eo2 into consideration.

We can visualize it as follows:



Starting with h1:

(∂E total)/(∂out h1) = (∂E o1)/(∂out h1) + (∂E o2)/(∂out h1)

(∂E o1)/(∂out h1) = (∂E o1)/(∂net o1) * (∂net o1)/(∂out h1)

We can calculate (∂E o1)/(∂net o1) using the values calculated earlier.

(∂E o1)/(∂net o1) = (∂E o1)/(∂out h1) * (∂out h1)/(∂net o1)

= 0.74136507 * 0.186815602 = 0.138498562

net o1 = w_5*out h1 + w_6*out h2 + b_2*1

(∂net o1)/(∂out h1) = w_5 = 0.40

Let's put the values in the equation.

(∂E o1)/(∂out h1) = (∂E o1)/(∂net o1) * (∂net o1)/(∂out h1)

= 0.138498562 * 0.40 = 0.055399425

Following the same process for (∂E o2)/(∂out h1), we get:

(∂E o2)/(∂out h1) = -0.019049119

We can calculate:

$(\partial E \text{ total})/(\partial \text{out h1}) = (\partial E \text{ o1})/(\partial \text{out h1}) + (\partial E \text{ o2})/(\partial \text{out h1})$

$= 0.055399425 + (-0.019049119) = 0.036350306$

Now that we have $(\partial E \text{ total})/(\partial \text{out h1})$, we need to figure out $(\partial \text{out h1})/(\partial \text{net h1})$and $(\partial \text{net h1})/\partial w$ *for each weight*

$\text{out h1} = 1/(1 + e^{-\text{net h1}})$

$(\partial \text{out h1})/(\partial \text{net h1}) = \text{out h1}(1- \text{out h1}) = 0.59326999(1 - 0.59326999) = 0.241300709$

We will calculate the partial derivative of the total net input of h1 w.r.t w1 the same way as we did for the output neuron.

$\text{net h1} = w_1 * i_1 + w_3 * i_2 + b_1 * 1$

$(\partial \text{net h1})/\partial w1 = i_1 = 0.05$

Let's put it all together.

$(\partial E \text{ total})/\partial w_1 = (\partial E \text{ total})/(\partial \text{out h1}) * (\partial \text{out h1})/(\partial \text{net h1}) * (\partial \text{net h1})/ \partial w_1$

$(\partial E \text{ total})/\partial w_1 = 0.036350306 * 0.241300709 * 0.05 = 0.000438568$

We can now update w1.

$w_1^+ = w_1 - n * (\partial E \text{ total})/\partial w_1 = 0.15 - 0.5 * 0.000438568 = 0.149780716$

Let's update other weights similarly.

$w_2^+ = 0.19956143$

$w_3^+ = 0.24975114$

$w_4^+ = 0.29950229$

- When we originally fed forward 0.05 and 0.1 inputs, the error on the network was 0.298371109.
- After the first round of backpropagation, the total error is now down to 0.291027924.

It might not seem like much, but after repeating this process 10,000 times, for example, the error plummets to 0.0000351085. At this point, when we feedforward 0.05 and 0.1, the two output neurons will generate 0.015912196 (vs. 0.01 target) and 0.984065734 (vs. 0.99 target).

LAB 6 Write a program to implement the naïve Bayesian classifier for a sample training data set stored as a .CSV file. Compute the accuracy of the classifier, considering few test data sets.

- Naïve Bayes algorithm is a supervised learning algorithm, which is based on Bayes theorem and used for solving classification problems.
- It is mainly used in text classification that includes a high-dimensional training dataset.
- Naïve Bayes Classifier is one of the simple and most effective Classification algorithms which helps in building the fast machine learning models that can make quick predictions.
- It is a probabilistic classifier, which means it predicts on the basis of the probability of an object.
- Some popular examples of Naïve Bayes Algorithm are spam filtration, Sentimental analysis, and classifying articles.

The Naïve Bayes algorithm is comprised of two words Naïve and Bayes, Which can be described as:

- **Naïve**: It is called Naïve because it assumes that the occurrence of a certain feature is independent of the occurrence of other features. Such as if the fruit is identified on the bases of color, shape, and taste, then red, spherical, and sweet fruit is recognized as an apple. Hence each feature individually contributes to identify that it is an apple without depending on each other.
- **Bayes**: It is called Bayes because it depends on the principle of Bayes' Theorem.

Bayes' Theorem:

- Bayes' theorem is also known as **Bayes' Rule** or **Bayes' law**, which is used to determine the probability of a hypothesis with prior knowledge. It depends on the conditional probability.
- The formula for Bayes' theorem is given as:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

Where,

P(A|B) is Posterior probability: Probability of hypothesis A on the observed event B.

P(B|A) is Likelihood probability: Probability of the evidence given that the probability of a hypothesis is true.

P(A) is Prior Probability: Probability of hypothesis before observing the evidence.

P(B) is Marginal Probability: Probability of Evidence.

LAB 7

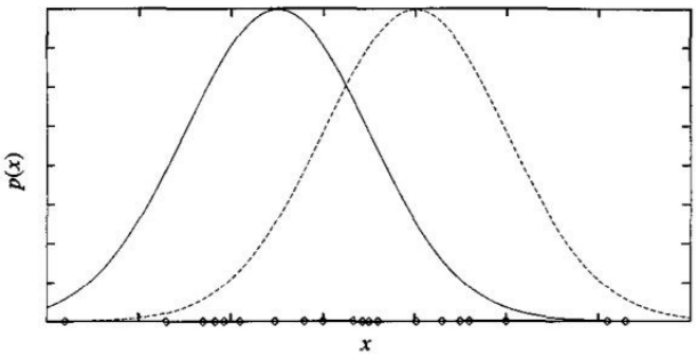
Apply EM algorithm to cluster a set of data stored in a .CSV file. Use the same data set for clustering using k□Means algorithm. Compare the results of these two algorithms and comment on the quality of clustering. You can add Java/Python ML library classes/API in the program.

THE EM ALGORITHM

The EM algorithm can be used even for variables whose value is never directly observed, provided the general form of the probability distribution governing these variables is known.

Estimating Means of k Gaussians

- Consider a problem in which the data D is a set of instances generated by a probability distribution that is a mixture of k distinct Normal distributions.



- This problem setting is illustrated in Figure for the case where k = 2 and where the instances are the points shown along the x axis.
- Each instance is generated using a two-step process.
  - First, one of the k Normal distributions is selected at random.
  - Second, a single random instance x_i is generated according to this selected distribution.
- This process is repeated to generate a set of data points as shown in the figure.

- To simplify, consider the special case
  - The selection of the single Normal distribution at each step is based on choosing each with uniform probability
  - Each of the k Normal distributions has the same variance σ^2, known value.
- The learning task is to output a hypothesis h = (μ_1 , . . . ,μ_k) that describes the means of each of the k distributions.
- We would like to find a maximum likelihood hypothesis for these means; that is, a hypothesis ***h*** that maximizes p(D |h).

$$\mu_{ML} = \underset{\mu}{\operatorname{argmin}} \sum_{i=1}^{m} (x_i - \mu)^2 \qquad (1)$$

In this case, the sum of squared errors is minimized by the sample mean

$$\mu_{ML} = \frac{1}{m} \sum_{i=1}^{m} x_i \qquad (2)$$

- Our problem here, however, involves a mixture of k different Normal distributions, and we cannot observe which instances were generated by which distribution.
- Consider full description of each instance as the triple (x_i, z_{i1}, z_{i2}),
  - where x_i is the observed value of the ith instance and
  - where z_{i1} and z_{i2} indicate which of the two Normal distributions was used to generate the value x_i
- In particular, z_{ij} has the value 1 if x_i was created by the j^{th} Normal distribution and 0 otherwise.
- Here x_i is the observed variable in the description of the instance, and z_{il} and z_{i2} are hidden variables.
- If the values of z_{il} and z_{i2} were observed, we could use following Equation to solve for the means p_1 and p_2
- Because they are not, we will instead use the EM algorithm

EM algorithm

Step 1: Calculate the expected value $E[z_{ij}]$ of each hidden variable z_{ij}, assuming the current hypothesis $h = \langle \mu_1, \mu_2 \rangle$ holds.

Step 2: Calculate a new maximum likelihood hypothesis $h' = \langle \mu'_1, \mu'_2 \rangle$, assuming the value taken on by each hidden variable z_{ij} is its expected value $E[z_{ij}]$ calculated in Step 1. Then replace the hypothesis $h = \langle \mu_1, \mu_2 \rangle$ by the new hypothesis $h' = \langle \mu'_1, \mu'_2 \rangle$ and iterate.

Let us examine how both of these steps can be implemented in practice. Step 1 must calculate the expected value of each z_{ij}. This $E[z_{ij}]$ is just the probability that instance x_i was generated by the jth Normal distribution

$$E[z_{ij}] = \frac{p(x = x_i|\mu = \mu_j)}{\sum_{n=1}^{2} p(x = x_i|\mu = \mu_n)}$$
$$= \frac{e^{-\frac{1}{2\sigma^2}(x_i-\mu_j)^2}}{\sum_{n=1}^{2} e^{-\frac{1}{2\sigma^2}(x_i-\mu_n)^2}}$$

Thus the first step is implemented by substituting the current values $\langle \mu_1, \mu_2 \rangle$ and the observed x_i into the above expression.

In the second step we use the $E[z_{ij}]$ calculated during Step 1 to derive a new maximum likelihood hypothesis $h' = \langle \mu'_1, \mu'_2 \rangle$.
maximum likelihood hypothesis in this case is given by

$$\mu_j \leftarrow \frac{\sum_{i=1}^{m} E[z_{ij}]\ x_i}{\sum_{i=1}^{m} E[z_{ij}]}$$

LAB 8

Write a program to implement K-Nearest Neighbour algorithm to classify the iris data set. Print both correct and wrong predictions. Java/Python ML library classes can be used for this problem.

K-Nearest Neighbor Algorithm

Training algorithm:

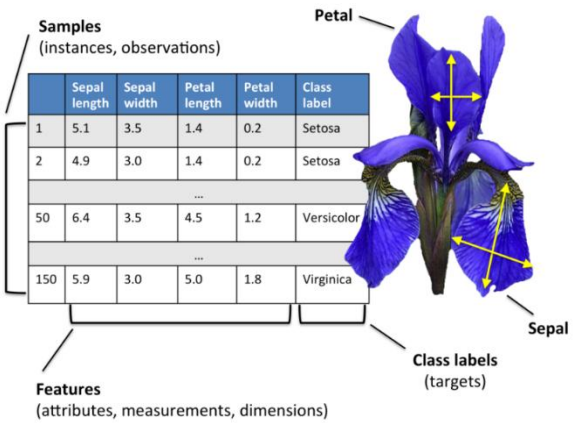
- For each training example (x, f (x)), add the example to the list training examples Classification algorithm:
    - Given a query instance x_q to be classified,
        - Let $x_1 \ldots x_k$ denote the k instances from training examples that are nearest to x_q
        - Return

$$\hat{f}(x_q) \leftarrow \frac{\sum_{i=1}^{k} f(x_i)}{k}$$

- Where, $f(x_i)$ function to calculate the mean value of the k nearest training examples.

Data Set:

Iris Plants Dataset: Dataset contains 150 instances (50 in each of three classes) Number of Attributes: 4 numeric, predictive attributes and the Class.



	Sepal length	Sepal width	Petal length	Petal width	Class label
1	5.1	3.5	1.4	0.2	Setosa
2	4.9	3.0	1.4	0.2	Setosa
...					
50	6.4	3.5	4.5	1.2	Versicolor
...					
150	5.9	3.0	5.0	1.8	Virginica

Sample Data

	sepal-length	sepal-width	petal-length	petal-width	Class
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa

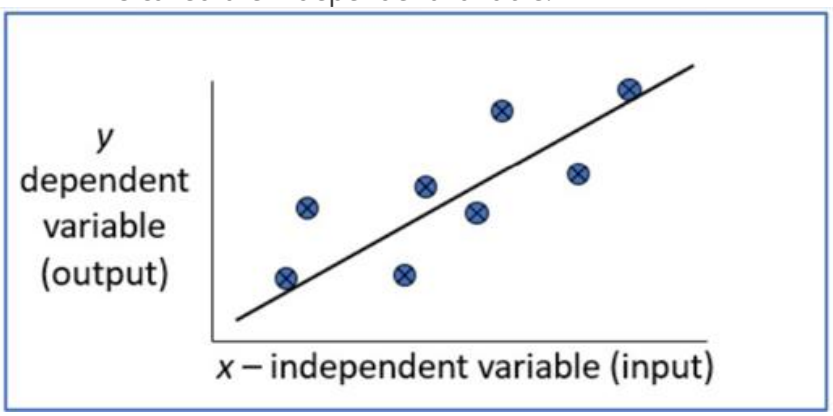
LAB 9

Implement the non-parametric Locally Weighted Regression algorithm in Python in order to fit data points. Select the appropriate data set for your experiment and draw graphs.

Locally Weighted Regression Algorithm

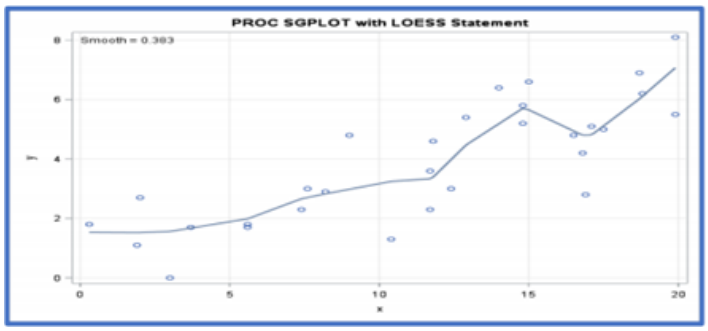
Regression:

- ✓ Regression is a technique from statistics that are used to predict values of the desired target quantity when the target quantity is continuous.
- ✓ In regression, we seek to identify (or estimate) a continuous variable y associated with a given input vector x.
- ✓ y is called the dependent variable.
- ✓ x is called the independent variable.



Loess/Lowess Regression:

Loess regression is a nonparametric technique that uses local weighted regression to fit a smooth curve through points in a scatter plot.



Lowess Algorithm:

Locally weighted regression is a very powerful nonparametric model used in statistical learning.

Given a dataset X, y, we attempt to find a model parameter β(x) that minimizes residual sum of weighted squared errors.

The weights are given by a kernel function (k or w) which can be chosen arbitrarily

Algorithm

1. Read the Given data Sample to X and the curve (linear or non linear) to Y
2. Set the value for Smoothening parameter or Free parameter say τ
3. Set the bias /Point of interest set x0 which is a subset of X
4. Determine the weight matrix using :

$$w(x, x_o) = e^{-\frac{(x-x_o)^2}{2\tau^2}}$$

5. Determine the value of model term parameter β using:

$$\hat{\beta}(x_o) = (X^T W X)^{-1} X^T W y$$

6. Prediction = x0*β