Breadth First Search

Breadth First Search (BFS) algorithm starts with the initial node of the Graph G, and then goes to all adjusted node before visited to their children until the goal node is not found. The algorithm, reaches all levels until the graph to be completely explored.

The data structure which is being used in BFS is queue. In BFS, the edges that leads to an unvisited node are called discovery edges or frontier edges while the edges that leads to an already visited node are called visited edges.

BFS is a recursive algorithm for searching all the vertices of a graph. Traversal means visiting all the nodes in the graph.

Steps

A standard BFS implementation puts each vertex of the graph into one of the two categories:

1. Visited
2. Not visited

The purpose of the algorithm is to mark each vertex as visited avoiding cycles

1. Start by putting any one of the graph's vertices on the top of the queue.
2. Take the front item of the queue and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the back of the queue.
4. Keep repeating steps 2 and 3 until the queue is empty.

Ex.No: 1.a   Implementation of Uninformed search algorithms – Breadth First Search

Date:

Aim:

To implement Breadth first search using Python Programming.

Algorithm

Step 1: Start
Step 2: Initialize Graph
Step 3: Create list for visited node
Step 4: Create list to initialize queue
Step 5: Perform Breadth First Search
Step 6: print the order of node visited
Step 7: Stop

Source Code

```
graph = {
  '5' : ['3','7'],
  '3' : ['2', '4'],
  '7' : ['8'],
  '2' : [],
  '4' : ['8'],
  '8' : []
}

visited = [] # List for visited nodes.
queue = []     #Initialize a queue

def bfs(visited, graph, node): #function for BFS
  visited.append(node)
  queue.append(node)

  while queue:          # Creating loop to visit each node
    m = queue.pop(0)
    print (m, end = " ")
  for neighbour in graph[m]:
    if neighbour not in visited:
      visited.append(neighbour)
      queue.append(neighbour)
      # Driver Code
print("Following is the Breadth-First Search")
bfs(visited, graph, '5')    # function calling
```

Output

```
Following is the Breadth-First Search
5 3 7 2 4 8
```

Result

Thus the Breadth First Program is implemented and executed successfully using Python Programming.

Depth First Search

Depth First Search (DFS) algorithm starts with the initial node of the Graph G, and then goes to deeper and deeper until we find the goal node which has no children. The algorithm, then backtracks from the dead end towards the most recent node that is yet to be completely unexplored.

The data structure which is being used in DFS is stack. In DFS, the edges that leads to an unvisited node are called discovery edges or frontier edges while the edges that leads to an already visited node are called visited edges.

DFS is a recursive algorithm for searching all the vertices of a graph or tree data structure. Traversal means visiting all the nodes in the graph.

Steps

A standard DFS implementation puts each vertex of the graph into one of the two categories:

3. Visited
4. Not visited

The purpose of the algorithm is to mark each vertex as visited avoiding cycles

5. Start by putting any one of the graph's vertices on the top of the stack.
6. Take the top item of the stack and add it to the visited list.
7. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of the stack.
8. Keep repeating steps 2 and 3 until the stack is empty.

Ex.No: 1.b   Implementation of Uninformed search algorithms – Depth First Search

Date:

Aim:

To implement Depth first search using Python Programming

Algorithm

Step 1: Start
Step 2: Initialize Graph
Step 3: Create list for visited node
Step 4: Create list to initialize queue
Step 5: Perform Depth First Search
Step 6: print the order of node visited
Step 7: Stop

Source Code

```
graph = {
  '5' : ['3','7'],
  '3' : ['2', '4'],
  '7' : ['8'],
  '2' : [],
  '4' : ['8'],
  '8' : []
}

visited = set() # Set to keep track of visited nodes of graph.

def dfs(visited, graph, node):  #function for dfs
    if node not in visited:
        print (node)
        visited.add(node)
        for neighbour in graph[node]:
            dfs(visited, graph, neighbour)

# Driver Code
```

```
print("Following is the Depth-First Search")
dfs(visited, graph, '5')
```

Output

```
Following is the Depth-First Search
5
3
2
4
8
7
```

Result

Thus the Depth First Program is implemented and executed successfully using Python Programming.

A* Search

The implementation uses a heap to maintain the frontier, and a dictionary to maintain the explored set. The Node class represents a node in the search tree, and includes the state, parent node, action taken to reach the node, and heuristic cost to the goal.

The A* search function takes a problem instance as input, and returns the goal state if found, or none otherwise. The function implements the A* algorithm, which uses a heuristic value. The node which having the less heuristic value is selected.

Note that this implementation assumes that the problem instance has the following methods:

initial_state: returns the initial state of the problem

goal_test(state): returns the goal state False otherwise

actions(state): returns a list of actions that can be taken from the given state

result(state, action): returns the goal state from the given graph.

heuristic(state): returns the estimated cost of reaching a goal state from the given state

Ex.No: 2.a Implementation of Informed search algorithm - A* Search

Date:

Aim:

To implement A* search using Python Programming.

Algorithm

Step 1: Start
Step 2: Initialize heap
Step 3: Initialize heuristic value for all node
Step 4: Perform A* Search
Step 5: print the destination node heuristic value
Step 6: Stop

Source Code

```
import heapq

def astar(graph, heuristic, start, goal):
    visited = set()
    heap = [(0, start)]
    while heap:
        (cost, node) = heapq.heappop(heap)
        if node in visited:
            continue
        visited.add(node)
        if node == goal:
            return cost
        for neighbor, distance in graph[node].items():
            if neighbor not in visited:
                heapq.heappush(heap, (cost + distance + heuristic[neighbor], neighbor))
    return -1

graph = {
    'A': {'B': 2, 'C': 3},
    'B': {'D': 4},
    'C': {'D': 2},
    'D': {'E': 3},
    'E': {}
}

heuristic = {
    'A': 7,
    'B': 6,
    'C': 4,
    'D': 2,
    'E': 0
}
print("A*:")
print(astar(graph, heuristic, 'A', 'E'))
```

Output

```
A*:
14
```

Result

Thus the A* search is implemented and executed successfully using Python Programming.

Memory Bounded   A* Search

The implementation uses a priority queue to maintain the frontier, and a dictionary to maintain the explored set. The Node class represents a node in the search tree, and includes the state, parent node, action taken to reach the node, path cost to reach the node, and heuristic cost to the goal.

The memory_bounded_a_star_search function takes a problem instance and a memory limit as input, and returns the goal state if found, or None otherwise. The function implements the Memory Bounded A* algorithm, which uses a memory limit to prune nodes from the search frontier whose f_cost exceeds the memory limit.

Note that this implementation assumes that the problem instance has the following methods:

initial_state: returns the initial state of the problem

goal_test(state): returns True if the given state is a goal state, False otherwise

actions(state): returns a list of actions that can be taken from the given state

result(state, action): returns the new state that results from taking the given action in the given state

step_cost(state, action, next_state): returns the cost of taking the given action from the given state to reach the next state

heuristic(state): returns the estimated cost of reaching a goal state from the given state

Ex.No: 2.b Implementation of Informed search algorithm – Memory Bounded A* Search

Date:

Aim:

To implement Memory Bounded A* search using Python Programming.

Algorithm

Step 1: Start
Step 2: Import priority queue
Step 3: Initialize graph and heuristic value for all node
Step 4: Initialize weights for all paths
Step 5: Perform Memory Bounded heuristic search
Step 6: print the result
Step 7: Stop

Source Code

```
from queue import PriorityQueue
import sys

graph = {
    'A': {'B': 2, 'C': 3},
    'B': {'D': 4},
    'C': {'D': 2},
    'D': {'E': 3},
    'E': {}
}

heuristic = {
    'A': 7,
    'B': 6,
    'C': 4,
    'D': 2,
    'E': 0
}
```

```
class Node:
    def __init__(self, state, parent=None, action=None, path_cost=0, heuristic_cost=0):
        self.state = state
        self.parent = parent
        self.action = action
        self.path_cost = path_cost
        self.heuristic_cost = heuristic_cost
        self.f_cost = path_cost + heuristic_cost

    def __lt__(self, other):
        return self.f_cost < other.f_cost

def memory_bounded_a_star_search(problem, memory_limit=sys.maxsize):
    start_node = Node(problem.initial_state)
    if problem.goal_test(start_node.state):
        return start_node.state

    frontier = PriorityQueue()
    frontier.put(start_node)
    explored = {}

    while frontier:
        node = frontier.get()
        state = node.state

        if problem.goal_test(state):
            return state

        if state in explored and explored[state].f_cost <= node.f_cost:
            continue

        explored[state] = node

        if node.f_cost > memory_limit:
            continue

        for action in problem.actions(state):
            child_state = problem.result(state, action)
            child_path_cost = node.path_cost + problem.step_cost(state, action, child_state)
            child_heuristic_cost = problem.heuristic(child_state)
            child_node = Node(child_state, node, action, child_path_cost, child_heuristic_cost)

            if child_state not in explored or child_node.f_cost < explored[child_state].f_cost:
                frontier.put(child_node)

    return None
```

Output

```
Memory Bounded A*:
34
```

Result

Thus the Memory Bounded A* search is implemented and executed successfully using Python Programming.

Machine learning

Machine learning is a subset of artificial intelligence in the field of computer science that often uses statistical techniques to give computers the ability to "learn" (i.e., progressively improve performance on a specific task) with data, without being explicitly programmed. In the past decade, machine learning has given us self-driving cars, practical speech recognition, effective web search, and a vastly improved understanding of the human genome.

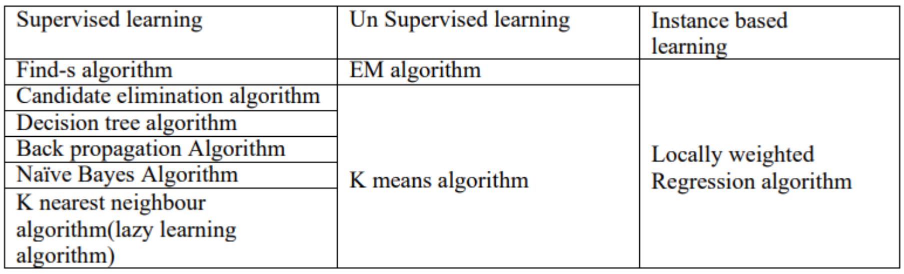
Machine learning tasks Machine learning tasks are typically classified into two broad categories, depending on whether there is a learning "signal" or "feedback" available to a learning system: Supervised learning:

The computer is presented with example inputs and their desired outputs, given by a "teacher", and the goal is to learn a general rule that maps inputs to outputs. As special cases, the input signal can be only partially available, or restricted to special feedback: Semi-supervised learning: the computer is given only an incomplete training signal: a training set with some (often many) of the target outputs missing.

Active learning: the computer can only obtain training labels for a limited set of instances (based on a budget), and also has to optimize its choice of objects to acquire labels for. When used interactively, these can be presented to the user for labeling.

Reinforcement learning: training data (in form of rewards and punishments) is given only as feedback to the program's actions in a dynamic environment, such as driving a vehicle or playing a game against an opponent. Unsupervised learning: No labels are given to the learning algorithm, leaving it on its own to find structure in its input.

Unsupervised learning can be a goal in itself (discovering hidden patterns in data) or a means towards an end (feature learning).



Supervised learning	Un Supervised learning	Instance based learning
Find-s algorithm	EM algorithm	Locally weighted Regression algorithm
Candidate elimination algorithm	K means algorithm	
Decision tree algorithm		
Back propagation Algorithm		
Naïve Bayes Algorithm		
K nearest neighbour algorithm(lazy learning algorithm)		

Machine learning applications

In classification, inputs are divided into two or more classes, and the learner must produce a model that assigns unseen inputs to one or more (multi-label classification) of these classes. This is typically tackled in a supervised manner. Spam filtering is an example of classification, where the inputs are email (or other) messages and the classes are "spam" and "not spam". In regression, also a supervised problem, the outputs are continuous rather than discrete.

In clustering, a set of inputs is to be divided into groups. Unlike in classification, the groups are not known beforehand, making this typically an unsupervised task. Density estimation finds the distribution of inputs in some space. Dimensionality reduction simplifies inputs by mapping them into a lower- dimensional space. Topic modeling is a related problem, where a program is given a list of human language documents and is tasked with finding out which documents cover similar topics.

Ex.No: 3 **Implement Naïve Bayes models**

Naive Bayes classifiers are a family of simple probabilistic classifiers based on applying Bayes' theorem with strong (naive) independence assumptions between the features in machine learning. Basically we can use above theories and equations for classification problem.

What are Bayesian Networks (BN) ?

Bayesian Network is used to represent the graphical model for probability relationship among a set of variables. Bayes' theorem is a way to figure out conditional probability. Conditional probability is the probability of an event happening, given that it has some relationship to one or more other events. For example, your probability of getting a parking space is connected to the time of day you park, where you park, and what conventions are going on at any time. Bayes' theorem is slightly more nuanced. In a nutshell, it gives you the actual probability of an event given information about tests.

"Events" Are different from "tests."

For example, there is a test for liver disease, but that's separate from the event of actually having liver disease.

• Tests are flawed: just because you have a positive test does not mean you actually have the disease. Many tests have a high false positive rate. Rare events tend to have higher false positive rates than more common events. We're not just talking about medical tests here. For example, spam filtering can have high false positive rates. Bayes' theorem takes the test results and calculates your real probability that the test has identified the event.

Real time example using Bayes' Theorem (liver disease).

You might be interested in finding out a patient's probability of having liver disease if they are an alcoholic. "Being an alcoholic" is the test (kind of like a litmus test) for liver disease.

• A could mean the event "Patient has liver disease." Past data tells you that 10% of patients entering your clinic have liver disease. P(A) = 0.10.

• B could mean the litmus test that "Patient is an alcoholic." Five percent of the clinic's patients are alcoholics. P(B) = 0.05.

• You might also know that among those patients diagnosed with liver disease, 7% are alcoholics.

This is your B|A: the probability that a patient is alcoholic, given that they have liver disease, is 7%. Bayes' theorem tells you:

P(A|B)=(0.07*0.1)/0.05=0.14

In other words, if the patient is an alcoholic, their chances of having liver disease is 0.14 (14%). This is a large increase from the 10% suggested by past data. But it's still unlikely that any particular patient has liver disease.

For implementation the iris dataset is taken to found whether Naive Bias model efficiently classifies the labels based four features namely sepal length, sepal width, petal length, and petal width.

Some instance from the dataset

Ex.No: 3 Implement Naïve Bayes models to classify the Labels in Iris Dataset

Date:

Aim:
To implement Naive Bayes to accurately classify the label based on given features using Python Programming.

Algorithm

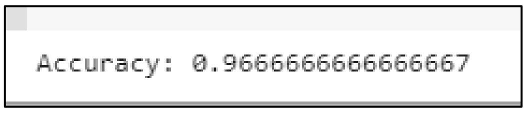
Step 1: Start
Step 2: Import sklearn
Step 3: Import iris dataset from sklearn datasets
Step 4: Import all required packages to perform naive bias classification in machine learning
Step 5: Display the result
Step 6: Stop

Source Code

```
from sklearn.datasets import load_iris

from sklearn.model_selection import train_test_split

from sklearn.naive_bayes import GaussianNB

from sklearn.metrics import accuracy_score


# Load the iris dataset

iris = load_iris()


# Split the dataset into training and testing sets

X_train, X_test, y_train, y_test = train_test_split(iris.data, iris.target, test_size=0.2,
random_state=0)


# Train a Gaussian Naive Bayes classifier
```

```
classifier = GaussianNB()

classifier.fit(X_train, y_train)


# Predict the test set labels

y_pred = classifier.predict(X_test)


# Calculate the accuracy score

accuracy = accuracy_score(y_test, y_pred)

print("Accuracy:", accuracy)
```

Output



```
Accuracy: 0.9666666666666667
```

Result

Thus the Naive Bias model for iris dataset is implemented and executed successfully using Python Program

Ex.No: 4 **Implement Bayesian Network**

A Bayesian network is a directed acyclic graph in which each edge corresponds to a conditional dependency, and each node corresponds to a unique random variable.

Bayesian network consists of two major parts: a directed acyclic graph and a set of conditional probability distributions

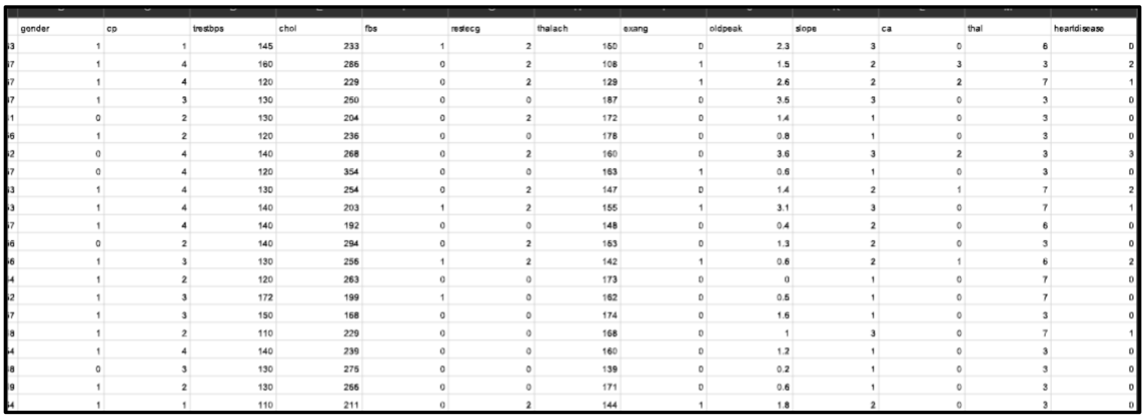
• The directed acyclic graph is a set of random variables represented by nodes.

• The conditional probability distribution of a node (random variable) is defined for every possible outcome of the preceding causal node(s).

The goal is to calculate the posterior conditional probability distribution of each of the possible unobserved causes given the observed evidence, i.e. P [Cause | Evidence].

Data Set: Title: Heart Disease Databases The Cleveland database contains 76 attributes, but all published experiments refer to using a subset of 14 of them. In particular, the Cleveland database is the only one that has been used by ML researchers to this date. The "Heartdisease" field refers to the presence of heart disease in the patient. It is integer valued from 0 (no presence) to 4.

Some instance from the dataset



	gender	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal	heartdisease
3	1	1	145	233	1	2	150	0	2.3	3	0	6	0
7	1	4	160	286	0	2	108	1	1.5	2	3	3	2
7	1	4	120	229	0	2	129	1	2.6	2	2	7	1
7	1	3	130	250	0	0	187	0	3.5	3	0	3	0
1	0	2	130	204	0	2	172	0	1.4	1	0	3	0
6	1	2	120	236	0	0	178	0	0.8	1	0	3	0
2	0	4	140	268	0	2	160	0	3.6	3	2	3	3
7	0	4	120	354	0	0	163	1	0.6	1	0	3	0
3	1	4	130	254	0	2	147	0	1.4	2	1	7	2
3	1	4	140	203	1	2	155	1	3.1	3	0	7	1
7	1	4	140	192	0	0	148	0	0.4	2	0	6	0
6	0	2	140	294	0	2	153	0	1.3	2	0	3	0
6	1	3	130	256	1	2	142	1	0.6	2	1	6	2
4	1	2	120	263	0	0	173	0	0	1	0	7	0
2	1	3	172	199	1	0	162	0	0.5	1	0	7	0
7	1	3	150	168	0	0	174	0	1.6	1	0	3	0
8	1	2	110	229	0	0	168	0	1	3	0	7	1
4	1	4	140	239	0	0	160	0	1.2	1	0	3	0
8	0	3	130	275	0	0	139	0	0.2	1	0	3	0
9	1	2	130	266	0	0	171	0	0.6	1	0	3	0
4	1	1	110	211	0	2	144	1	1.8	2	0	3	0

Ex.No: 4 Implement Bayesian Networks to Predict Patient Probability Rate to having Heartdisease

Date:

Aim:

To implement Bayesian Network to find the probability rate of patient having heart disease using Python Programming.

Algorithm

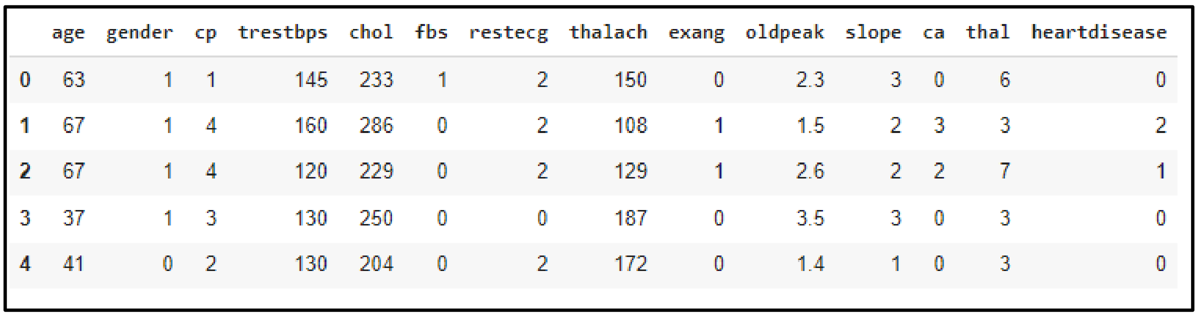
Step 1: Start
Step 2: Import sklearn
Step 3: Import heardisease dataset as csv
Step 4: Import all required packages to perform Bayesian Network in machine learning
Step 5: Display the result
Step 6: Stop

Source Code – Import packages

```
import numpy as np

import pandas as pd

import csv

from pgmpy.estimators import MaximumLikelihoodEstimator

from pgmpy.models import BayesianModel

from pgmpy.inference import VariableElimination
```

Source Code – Loading dataset

```
from google.colab import drive

drive.mount('/content/drive')

heartDisease = pd.read_csv("/content/drive/My Drive/Bayesiannetworkdataset.csv")

heartDisease.head()
```



	age	gender	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal	heartdisease
0	63	1	1	145	233	1	2	150	0	2.3	3	0	6	0
1	67	1	4	160	286	0	2	108	1	1.5	2	3	3	2
2	67	1	4	120	229	0	2	129	1	2.6	2	2	7	1
3	37	1	3	130	250	0	0	187	0	3.5	3	0	3	0
4	41	0	2	130	204	0	2	172	0	1.4	1	0	3	0

Source Code – Filling sparse values

heartDisease = heartDisease.replace('?',np.nan)

print('\n Attributes and datatypes')

print(heartDisease.dtypes)

Source Code – Fitting model in Bayesian Network

model= BayesianModel([('age','heartdisease'),('gender','heartdisease'),('exang','heartdisease'),('cp','heartdisease'),('heartdisease','restecg'),('heartdisease','chol')])

print('\nLearning CPD using Maximum likelihood estimators')

model.fit(heartDisease,estimator=MaximumLikelihoodEstimator)

Source Code – Reduce redundant variables

print('\n Inferencing with Bayesian Network:')

HeartDiseasetest_infer = VariableElimination(model)

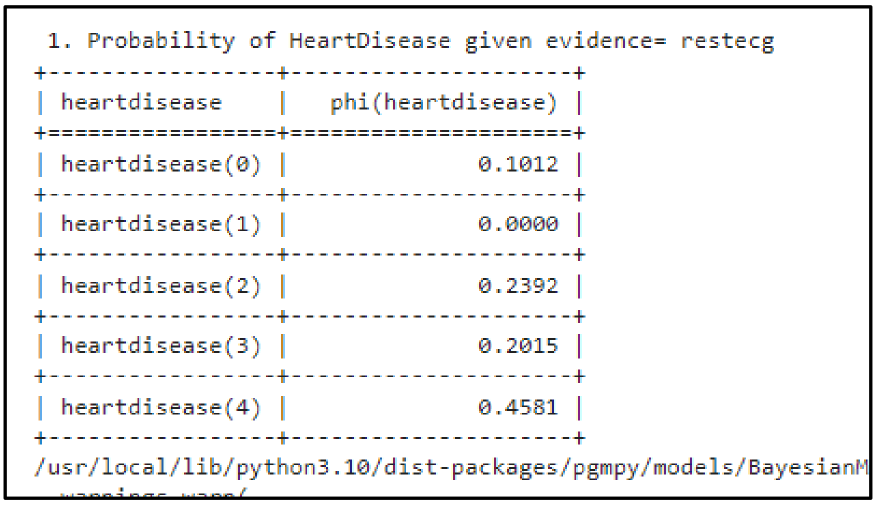
Source Code – Predicting Probability value of heartdisease for evidence restecg

print('\n 1. Probability of HeartDisease given evidence= restecg')

q1=HeartDiseasetest_infer.query(variables=['heartdisease'],evidence={'restecg':1})

print(q1)

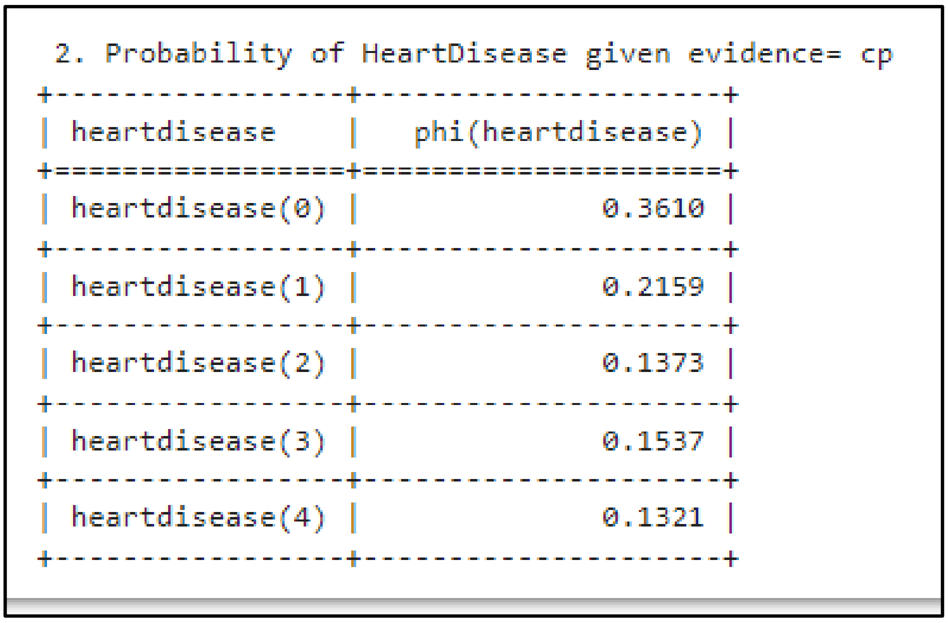
Output



```
 1. Probability of HeartDisease given evidence= restecg
+-----------------+---------------------+
| heartdisease    |   phi(heartdisease) |
+=================+=====================+
| heartdisease(0) |              0.1012 |
+-----------------+---------------------+
| heartdisease(1) |              0.0000 |
+-----------------+---------------------+
| heartdisease(2) |              0.2392 |
+-----------------+---------------------+
| heartdisease(3) |              0.2015 |
+-----------------+---------------------+
| heartdisease(4) |              0.4581 |
+-----------------+---------------------+
/usr/local/lib/python3.10/dist-packages/pgmpy/models/BayesianM
  warnings.warn(
```

Source Code – Predicting Probability value of heartdisease for evidence cp

```
print('\n 2. Probability of HeartDisease given evidence= cp ')

q2=HeartDiseasetest_infer.query(variables=['heartdisease'],evidence={'cp':2})

print(q2)
```



```
 2. Probability of HeartDisease given evidence= cp
+-----------------+---------------------+
| heartdisease    |   phi(heartdisease) |
+=================+=====================+
| heartdisease(0) |              0.3610 |
+-----------------+---------------------+
| heartdisease(1) |              0.2159 |
+-----------------+---------------------+
| heartdisease(2) |              0.1373 |
+-----------------+---------------------+
| heartdisease(3) |              0.1537 |
+-----------------+---------------------+
| heartdisease(4) |              0.1321 |
+-----------------+---------------------+
```

Result

Thus the Bayesian model for heart disease dataset is implemented and executed successfully using Python Programming.

Ex.No: 5 **Build Regression models**

Regression

Regression analysis is one of the most important fields in statistics and machine learning. There are many regression methods available. Linear regression is one of them what Is Regression? Regression analysis is one of the most important fields in statistics and machine learning. There are many regression methods available. Linear regression is one of them. Regression searches for relationships among variables. For example, you can observe several employees of some company and try to understand how their salaries depend on the features, such as experience, level of education, role, city they work in, and so on. This is a regression problem where data related to each employee represent one observation. The presumption is that the experience, education, role, and city are the independent features, while the salary depends on them. Generally, in regression analysis, you usually consider some phenomenon of interest and there have a number of observations. Each observation has two or more features.

Following the assumption that (atleast) one of the features depends on the others, you try to establish a relation among them. You need to find a function that maps some features or variables to others sufficiently well. The dependent features are called the dependent variables, outputs, or responses. The independent features are called the independent variables, inputs, or predictors. Linear Regression: Linear regression is probably one of the most important and widely used regression techniques. It’s among the simplest regression methods. One of its main advantages is the ease of interpreting results. When implementing linear regression of some dependent variable $y$on the set of independent variables x=(x1…xr$_r$), where r is the number of predictors, you assume a linear relationship between y and x:

$$y= \beta 0 + \beta 1x1 + \ldots\ldots+ \beta rxr + \epsilon.$$

This equation is the regression equation. β0,, βr$_r$are the regression coefficients, and ε is the random error. Linear regression calculates the estimators of the regression coefficients or simply the predicted weights, denoted with b1…br. They define the estimated regression function

f(x) = b0 +b1x1 + ….+brxr

This function should capture the dependencies between the inputsand output sufficiently well.

Ex.No: 5. a **To Implement Single Linear Regression**

Date:

Aim:
To implement Single Linear Regression using Python Programming

Algorithm

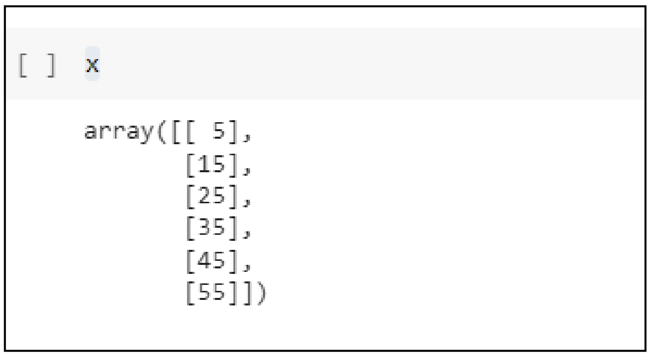
Step 1: Start
Step 2: Import sklearn
Step 3: Initialize features and labels
Step 4: Import all required packages to perform Linear regression in machine learning
Step 5: Display the result
Step 6: Stop

Source Code – Import packages

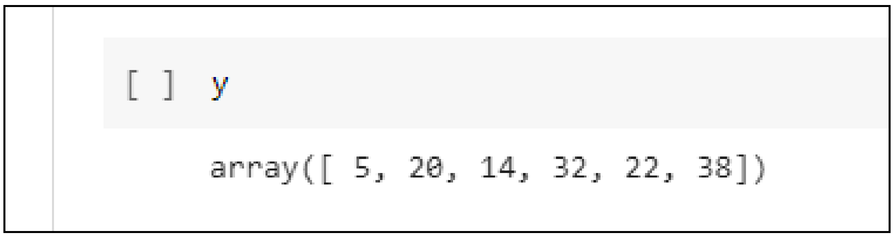
```
import numpy as np
from sklearn.linear_model import LinearRegression
```

Source code - Data and Reshape of x and display of y value

```
x = np.array([2, 15, 25, 35, 45, 55]).reshape((-1, 1))
y = np.array([5, 20, 14, 32, 22, 38])
x
y
```



```
[ ]  x

     array([[ 5],
            [15],
            [25],
            [35],
            [45],
            [55]])
```



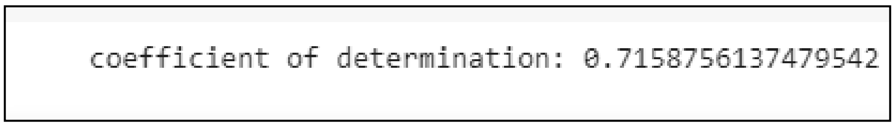
```
[ ]  y

     array([ 5, 20, 14, 32, 22, 38])
```

Source code - Create Model

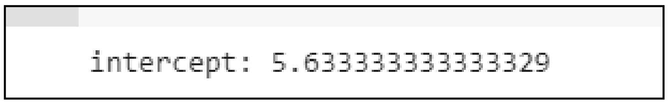
model = LinearRegression()
model.fit(x, y)

Source code - Get results

r_sq = model.score(x, y)
print(f"coefficient of determination: {r_sq}")
print(f"intercept: {model.intercept_}")

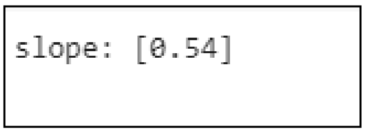


```
coefficient of determination: 0.7158756137479542
```



```
intercept: 5.633333333333329
```

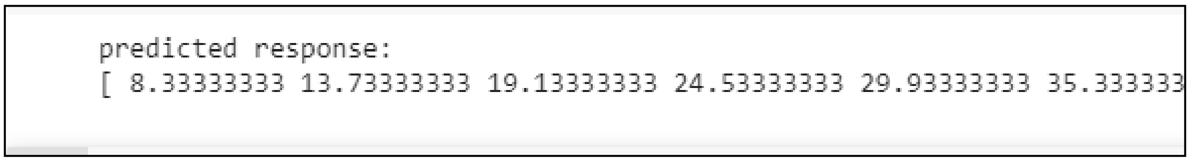
print(f"slope: {model.coef_}")



```
slope: [0.54]
```

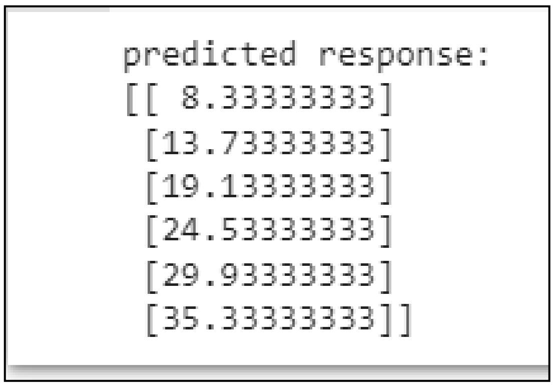
Source code - Predict Response

y_pred = model.predict(x)
print(f"predicted response:\n{y_pred}")

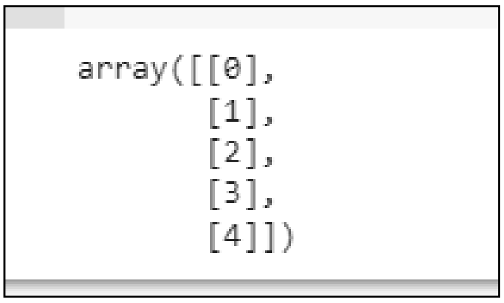


```
predicted response:
[ 8.33333333 13.73333333 19.13333333 24.53333333 29.93333333 35.333333
```

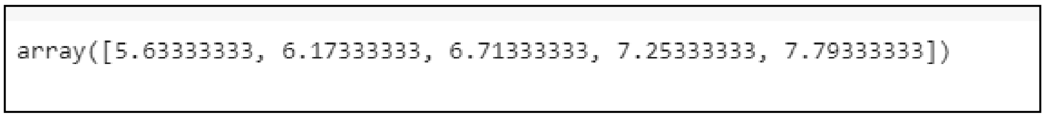
y_pred = model.intercept_ + model.coef_ * x
print(f"predicted response:\n{y_pred}")


```
predicted response:
[[ 8.33333333]
 [13.73333333]
 [19.13333333]
 [24.53333333]
 [29.93333333]
 [35.33333333]]
```

x_new = np.arange(5).reshape((-1, 1))
x_new


```
array([[0],
       [1],
       [2],
       [3],
       [4]])
```

y_new = model.predict(x_new)
y_new


```
array([5.63333333, 6.17333333, 6.71333333, 7.25333333, 7.79333333])
```

Result

Thus the Single Linear Regression for is implemented and executed successfully using Python Programming.

Ex.No: 5 .b   To Implement Multiple Linear Regressions to predict Unemployment Rate

Date:

Aim:

To implement Multiple Linear Regression to predict Unemployment Rate based on given features using Python Programming.

Algorithm

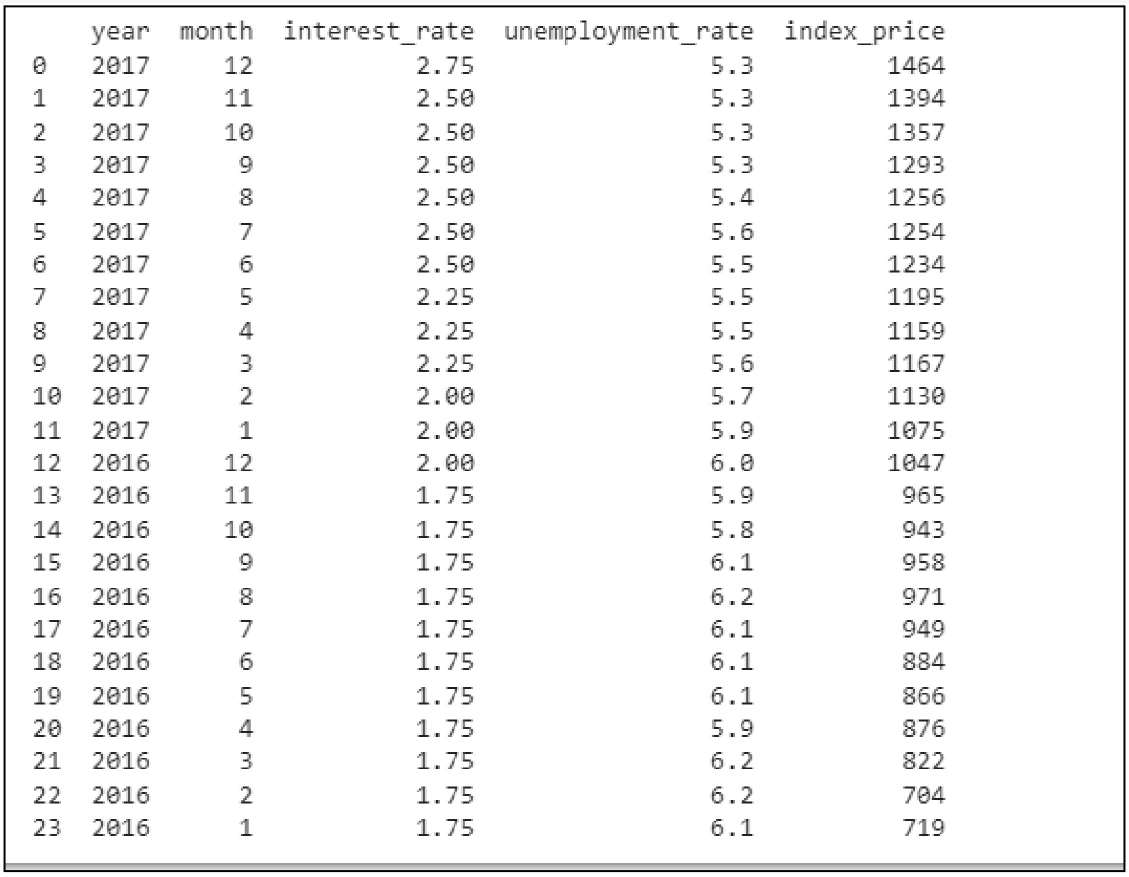
Step 1: Start
Step 2: Import sklearn
Step 3: Initialize features and labels
Step 4: Import all required packages to perform Multiple Linear regression in machine learning
Step 5: Display the result
Step 6: Stop

Source Code -

```
import pandas as pd
data = {'year':
[2017,2017,2017,2017,2017,2017,2017,2017,2017,2017,2017,2017,2016,2016,2016,2016,2016,2016,2016,2016,2016,2016,2016,2016],
        'month': [12,11,10,9,8,7,6,5,4,3,2,1,12,11,10,9,8,7,6,5,4,3,2,1],
        'interest_rate':
[2.75,2.5,2.5,2.5,2.5,2.5,2.5,2.25,2.25,2.25,2,2,2,1.75,1.75,1.75,1.75,1.75,1.75,1.75,1.75,1.75,1.75,1.75],
        'unemployment_rate':
[5.3,5.3,5.3,5.3,5.4,5.6,5.5,5.5,5.5,5.6,5.7,5.9,6,5.9,5.8,6.1,6.2,6.1,6.1,6.1,5.9,6.2,6.2,6.1],
        'index_price':
[1464,1394,1357,1293,1256,1254,1234,1195,1159,1167,1130,1075,1047,965,943,958,971,949,884,866,876,822,704,719]
        }
df = pd.DataFrame(data)
```

```
print(df)
```

Output

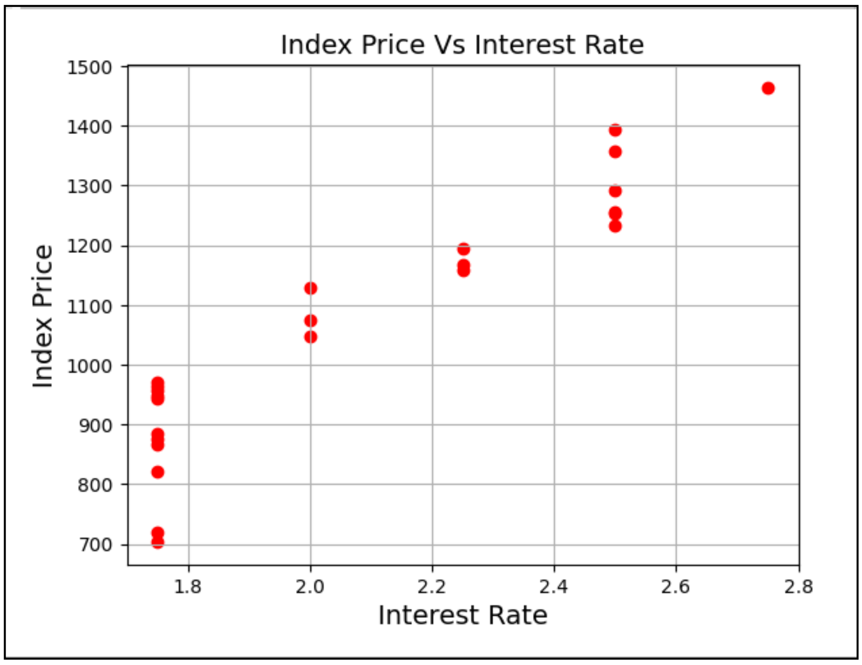


```
    year  month  interest_rate  unemployment_rate  index_price
0   2017     12           2.75                5.3         1464
1   2017     11           2.50                5.3         1394
2   2017     10           2.50                5.3         1357
3   2017      9           2.50                5.3         1293
4   2017      8           2.50                5.4         1256
5   2017      7           2.50                5.6         1254
6   2017      6           2.50                5.5         1234
7   2017      5           2.25                5.5         1195
8   2017      4           2.25                5.5         1159
9   2017      3           2.25                5.6         1167
10  2017      2           2.00                5.7         1130
11  2017      1           2.00                5.9         1075
12  2016     12           2.00                6.0         1047
13  2016     11           1.75                5.9          965
14  2016     10           1.75                5.8          943
15  2016      9           1.75                6.1          958
16  2016      8           1.75                6.2          971
17  2016      7           1.75                6.1          949
18  2016      6           1.75                6.1          884
19  2016      5           1.75                6.1          866
20  2016      4           1.75                5.9          876
21  2016      3           1.75                6.2          822
22  2016      2           1.75                6.2          704
23  2016      1           1.75                6.1          719
```

Source Code – Index price vs Unemployment rate

```
import pandas as pd

import matplotlib.pyplot as plt

data = {'year': [2017,2017,2017,2017,2017,2017,2017,2017,2017,2017,2017,2017,2016,2016,2016,2016,2016,2016,2016,2016,2016,2016,2016,2016],

        'month': [12,11,10,9,8,7,6,5,4,3,2,1,12,11,10,9,8,7,6,5,4,3,2,1],

        'interest_rate': [2.75,2.5,2.5,2.5,2.5,2.5,2.5,2.25,2.25,2.25,2,2,2,1.75,1.75,1.75,1.75,1.75,1.75,1.75,1.75,1.75,1.75,1.75],

        'unemployment_rate': [5.3,5.3,5.3,5.3,5.4,5.6,5.5,5.5,5.5,5.6,5.7,5.9,6,5.9,5.8,6.1,6.2,6.1,6.1,6.1,5.9,6.2,6.2,6.1],
        'index_price':
[1464,1394,1357,1293,1256,1254,1234,1195,1159,1167,1130,1075,1047,965,943,958,971,94
9,884,866,876,822,704,719]

        }

df = pd.DataFrame(data)

plt.scatter(df['interest_rate'], df['index_price'], color='red')

plt.title('Index Price Vs Interest Rate', fontsize=14)

plt.xlabel('Interest Rate', fontsize=14)

plt.ylabel('Index Price', fontsize=14)

plt.grid(True)

plt.show()
```

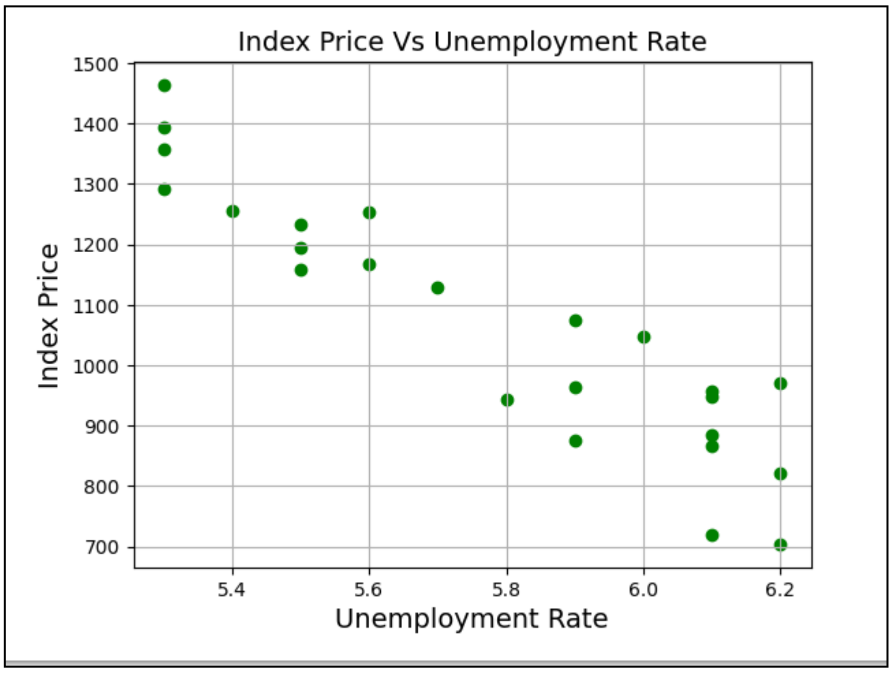
Output



Source Code – Index price vs Unemployment rate

```
import pandas as pd
import matplotlib.pyplot as plt

data = {'year':
[2017,2017,2017,2017,2017,2017,2017,2017,2017,2017,2017,2017,2016,2016,2016,2016,20
16,2016,2016,2016,2016,2016,2016,2016],
        'month': [12,11,10,9,8,7,6,5,4,3,2,1,12,11,10,9,8,7,6,5,4,3,2,1],
        'interest_rate':
[2.75,2.5,2.5,2.5,2.5,2.5,2.5,2.25,2.25,2.25,2,2,2,1.75,1.75,1.75,1.75,1.75,1.75,1.75,1.75,1.75,
1.75,1.75],
        'unemployment_rate':
[5.3,5.3,5.3,5.3,5.4,5.6,5.5,5.5,5.5,5.6,5.7,5.9,6,5.9,5.8,6.1,6.2,6.1,6.1,6.1,5.9,6.2,6.2,6.1],
        'index_price':
[1464,1394,1357,1293,1256,1254,1234,1195,1159,1167,1130,1075,1047,965,943,958,971,94
9,884,866,876,822,704,719]
        }

df = pd.DataFrame(data)

plt.scatter(df['unemployment_rate'], df['index_price'], color='green')
plt.title('Index Price Vs Unemployment Rate', fontsize=14)
plt.xlabel('Unemployment Rate', fontsize=14)
plt.ylabel('Index Price', fontsize=14)
plt.grid(True)
plt.show()
```

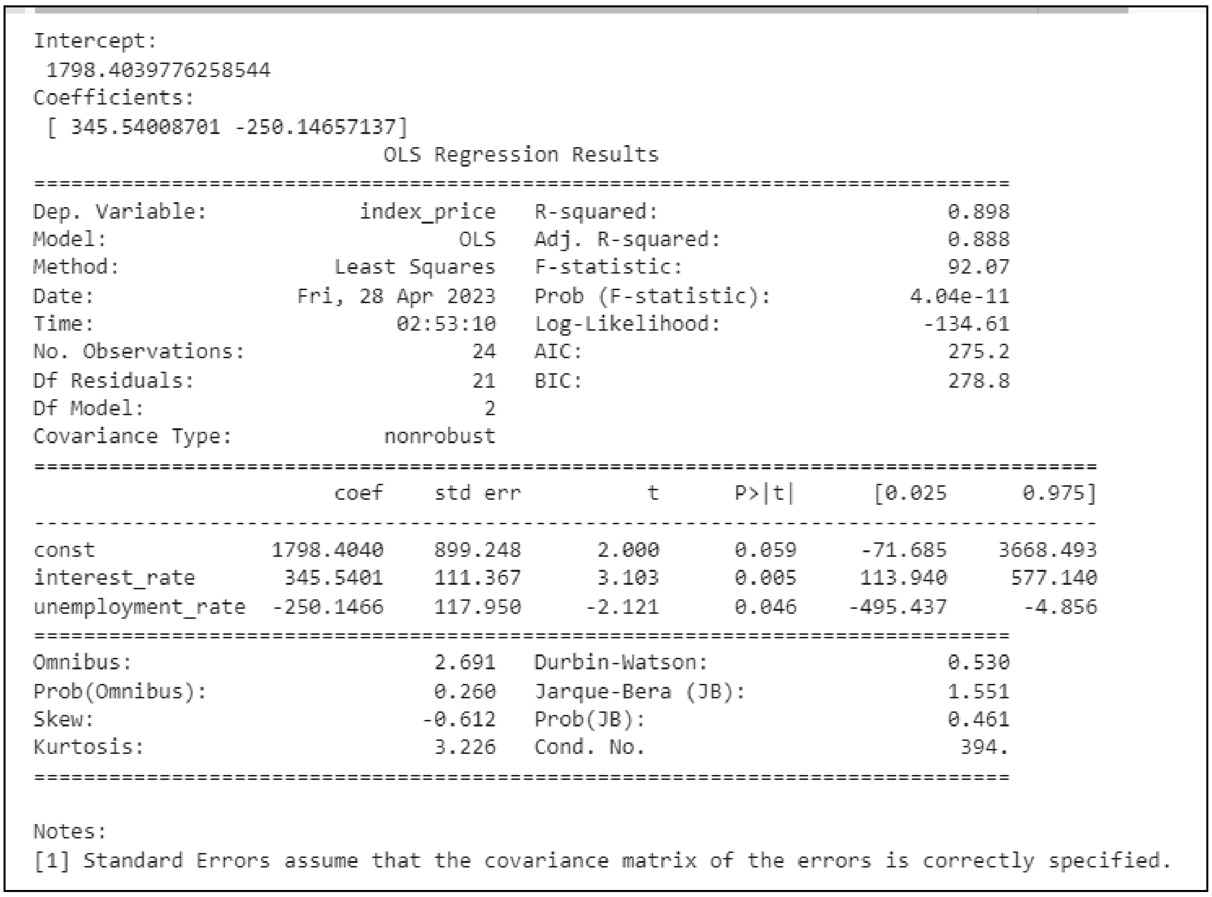
Output



Source Code – Display Statistical Measure

```
import statsmodels.api as sm

data = {'year':
[2017,2017,2017,2017,2017,2017,2017,2017,2017,2017,2017,2017,2016,2016,2016,2016,2016,2016,2016,2016,2016,2016,2016,2016],
        'month': [12,11,10,9,8,7,6,5,4,3,2,1,12,11,10,9,8,7,6,5,4,3,2,1],
        'interest_rate':
[2.75,2.5,2.5,2.5,2.5,2.5,2.5,2.25,2.25,2.25,2,2,2,1.75,1.75,1.75,1.75,1.75,1.75,1.75,1.75,1.75,1.75,1.75],
        'unemployment_rate':
[5.3,5.3,5.3,5.3,5.4,5.6,5.5,5.5,5.5,5.6,5.7,5.9,6,5.9,5.8,6.1,6.2,6.1,6.1,6.1,5.9,6.2,6.2,6.1],
        'index_price':
[1464,1394,1357,1293,1256,1254,1234,1195,1159,1167,1130,1075,1047,965,943,958,971,94
9,884,866,876,822,704,719]
        }

df = pd.DataFrame(data)

x = df[['interest_rate','unemployment_rate']]
y = df['index_price']

# with sklearn
regr = linear_model.LinearRegression()
regr.fit(x, y)

print('Intercept: \n', regr.intercept_)
print('Coefficients: \n', regr.coef_)

# with statsmodels
x = sm.add_constant(x) # adding a constant

model = sm.OLS(y, x).fit()
predictions = model.predict(x)

print_model = model.summary()
print(print_model)
```

Output



```
Intercept:
 1798.4039776258544
Coefficients:
 [ 345.54008701 -250.14657137]
                            OLS Regression Results
==============================================================================
Dep. Variable:            index_price   R-squared:                       0.898
Model:                            OLS   Adj. R-squared:                  0.888
Method:                 Least Squares   F-statistic:                     92.07
Date:                Fri, 28 Apr 2023   Prob (F-statistic):           4.04e-11
Time:                        02:53:10   Log-Likelihood:                -134.61
No. Observations:                  24   AIC:                             275.2
Df Residuals:                      21   BIC:                             278.8
Df Model:                           2
Covariance Type:            nonrobust
=====================================================================================
                        coef    std err          t      P>|t|      [0.025      0.975]
-------------------------------------------------------------------------------------
const              1798.4040    899.248      2.000      0.059     -71.685    3668.493
interest_rate       345.5401    111.367      3.103      0.005     113.940     577.140
unemployment_rate  -250.1466    117.950     -2.121      0.046    -495.437      -4.856
==============================================================================
Omnibus:                        2.691   Durbin-Watson:                   0.530
Prob(Omnibus):                  0.260   Jarque-Bera (JB):                1.551
Skew:                          -0.612   Prob(JB):                        0.461
Kurtosis:                       3.226   Cond. No.                         394.
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
```

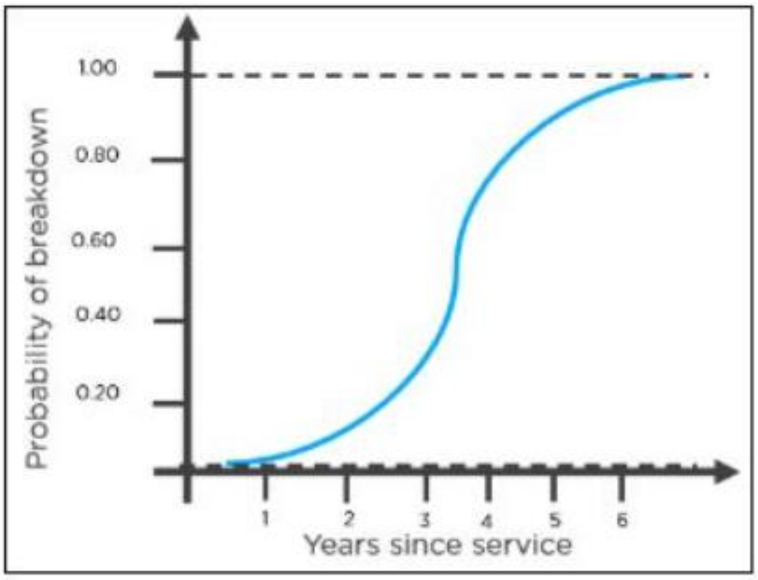
Result

Thus the Multiple Linear Regression to predict unemployment rate is implemented and executed successfully using Python Programming.

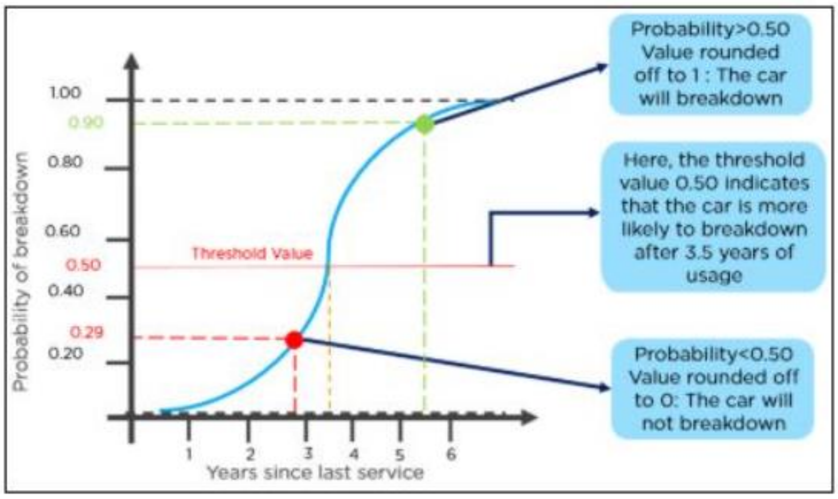
Ex.No.5c **Logistic Regression**

Logistic regression is a statistical method that is used for building machine learning models where the dependent variable is dichotomous: i.e. binary. Logistic regression is used to describe data and the relationship between one dependent variable and one or more independent variables. The independent variables can be nominal, ordinal, or of interval type. The name “logistic regression” is derived from the concept of the logistic function that it uses. The logistic function is also known as the sigmoid function. The value of this logistic function lies between zero and one.

The following is an example of a logistic function we can use to find the probability of a vehicle breaking down, depending on how many years it has been since it was serviced last.



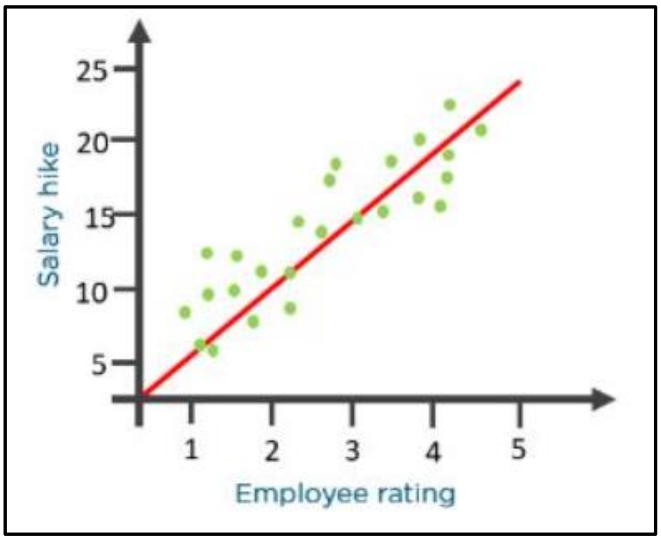
interpret the results from the graph to decide whether the vehicle will break down or not.



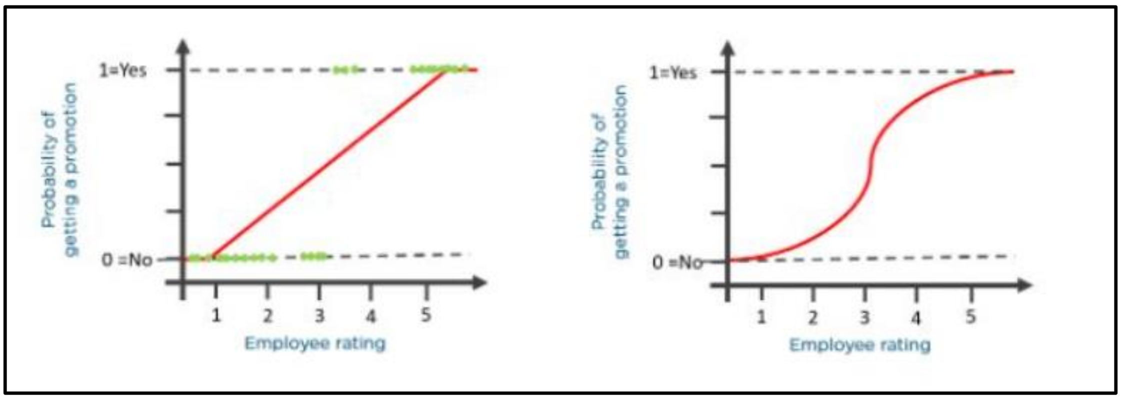
Working of Logistic Regression Algorithm

Consider the following example: An organization wants to determine an employee's salary increase based on their performance.

For this purpose, a linear regression algorithm will help them decide. Plotting a regression line by considering the employee's performance as the independent variable, and the salary increase as the dependent variable will make their task easier.



Now, what if the organization wants to know whether an employee would get a promotion or not based on their performance? The above linear graph won't be suitable in this case. As such, we clip the line at zero and one, and convert it into a sigmoid curve (S curve).



Based on the threshold values, the organization can decide whether an employee will get a salary increase or not.

To understand logistic regression, let's go over the odds of success.

Odds (θ) = Probability of an event happening / Probability of an event not happening

$$\boxed{\theta = p / 1 - p}$$

The values of odds range from zero to ∞ and the values of probability lies between zero and one.

Consider the equation of a straight line:

$$\boxed{y = \beta 0 + \beta 1 * x}$$

Here, $\beta 0$ is the y-intercept

$\beta 1$ is the slope of the line

x is the value of the x coordinate

y is the value of the prediction

Now to predict the odds of success, we use the following formula:

$$\log\left(\frac{p(x)}{1-P(x)}\right) = \beta_0 + \beta_1 x$$

Exponentiating both the sides, we have:

$$e^{ln\left(\frac{p(x)}{1-p(x)}\right)} = e^{\beta_0+\beta_1 x}$$

$$\left(\frac{p(x)}{1-p(x)}\right) = e^{\beta_0+\beta_1 x}$$

Let Y = e β0+β1 * x

Then p(x) / 1 - p(x) = Y

p(x) = Y(1 - p(x))

p(x) = Y - Y(p(x))

p(x) + Y(p(x)) = Y

p(x)(1+Y) = Y

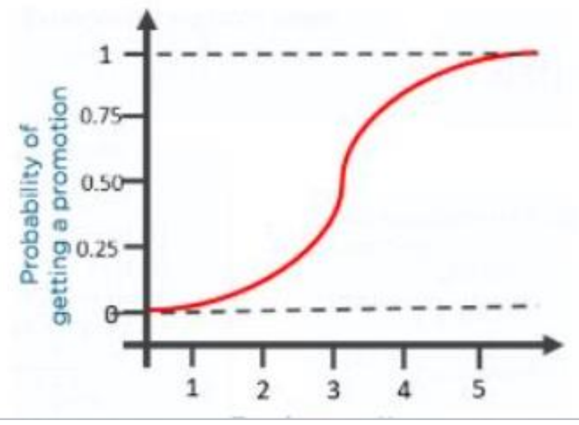
p(x) = Y / 1+Y

$$p(x) = \frac{e^{\beta_0+\beta_1 x}}{1 + e^{\beta_0+\beta_1 x}}$$

The equation of the sigmoid function is:

$$p(x) = \frac{1}{1 + e^{-(\beta_0+\beta_1 x)}}$$

The sigmoid curve obtained from the above equation is as follows:



Predict the Digits in Images Using a Logistic Regression Classifier in Python

Ex.No: 5 .c To Implement Logistic Regressions to predict predict digit values from images

Date:

Aim:
To implement Logistic Regression library to predict digit values from images using Python Programming

Algorithm

Step 1: Start
Step 2: Import sklearn
Step 3: Import image dataset
Step 4: Import all required packages to perform Logistic regression in machine learning
Step 5: Display the result
Step 6: Stop

Source Code – Import packages

```
import re
```

Importing libraries

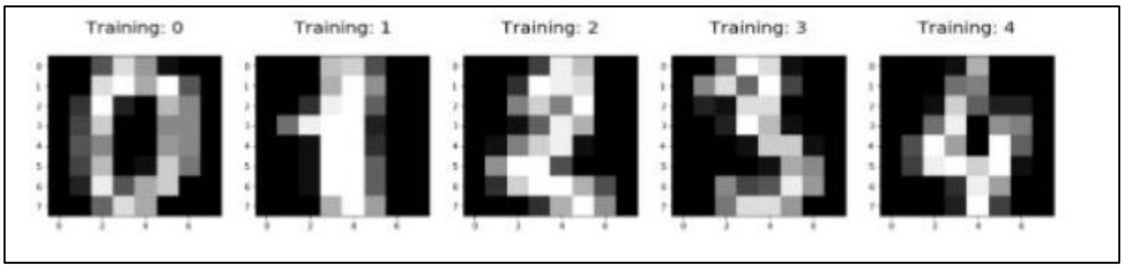
```
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
import numpy as np
import matplotlib.pyplot as ply
import seaborn as sns
from sklearn import metrics
%matplotlib inline
digits = load_digits()
```

Determining the total number of images and labels

```
print("Image Data Shape", digits.data.shape)
print("Label Data Shape", digits.target.shape)
```

Displaying some of the images and their labels

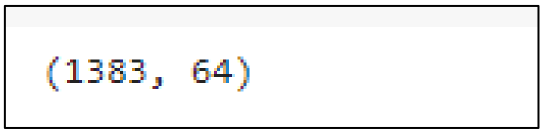
```
import numpy as np
import matplotlib.pyplot as plt
plt.figure(figsize=(20, 4))
for index, (image, label) in enumerate(zip(digits.data[0:5], digits.target[0:5])):
  plt.subplot(1, 5, index+1)
  plt.imshow(np.reshape(image, (8, 8)), cmap=plt.cm.gray)
  plt.title("Training : %i\n" %label, fontsize=20)
```



Dividing dataset into "training" and "test" set

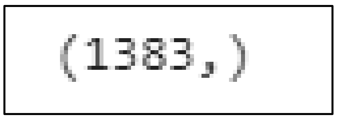
```
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test = train_test_split(digits.data, digits.target, test_size=0.23, random_state=2)
```

```
print(x_train.shape)
```



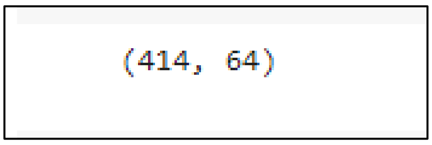
```
(1383, 64)
```

```
print(y_train.shape)
```



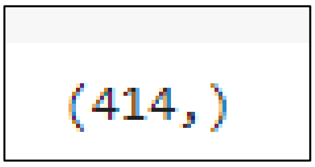
```
(1383,)
```

```
print(x_test.shape)
```



```
(414, 64)
```

print(y_test.shape)



```
(414,)
```

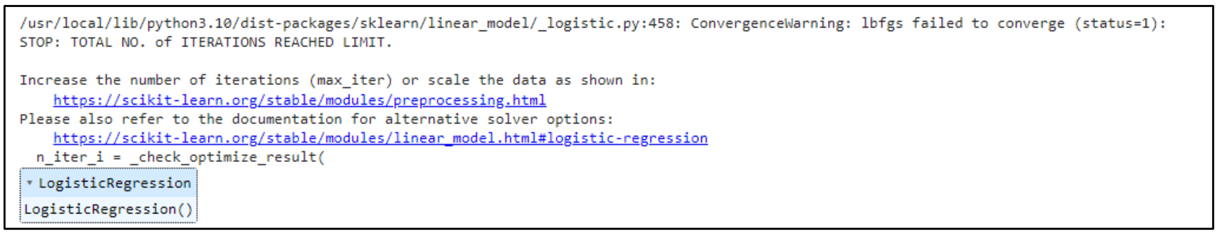
Importing the logistic regression model

from sklearn.linear_model import LogisticRegression

Making an instance of the model and training

logisticRegr = LogisticRegression()

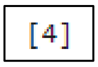
logisticRegr.fit(x_train, y_train)



```
/usr/local/lib/python3.10/dist-packages/sklearn/linear_model/_logistic.py:458: ConvergenceWarning: lbfgs failed to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  n_iter_i = _check_optimize_result(
▾ LogisticRegression
LogisticRegression()
```

Predicting the output of the first element of the test set

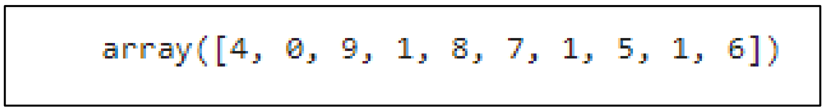
print(logisticRegr.predict(x_test[0].reshape(1, -1)))



```
[4]
```

Predicting the output of the first 10 elements of the test set

logisticRegr.predict(x_test[0:10])



```
array([4, 0, 9, 1, 8, 7, 1, 5, 1, 6])
```

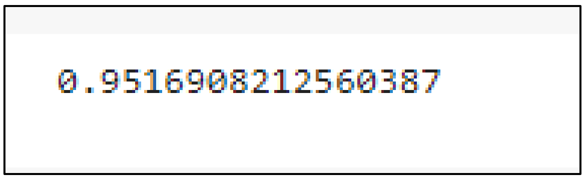
Prediction for the entire dataset

predictions = logisticRegr.predict(x_test)

Determining the accuracy of the model

score=logisticRegr.score(x_test, y_test)

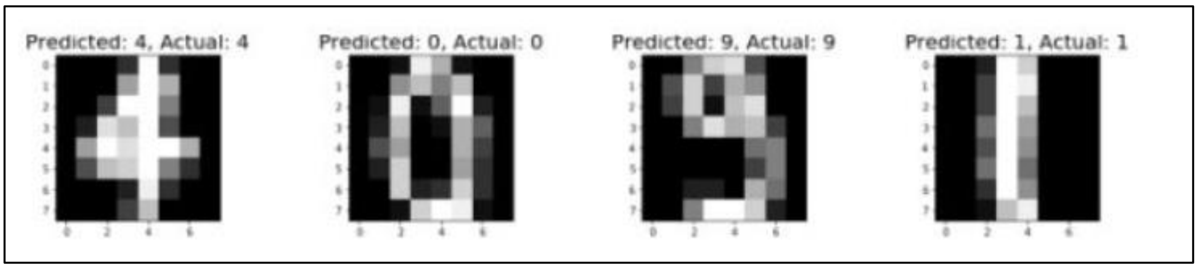
print(score)



```
0.9516908212560387
```

Presenting predictions and actual output

```
index=0
misclassifiedIndex=[]
for predict, actual in zip(predictions, y_test):
  if predict==actual:
    misclassifiedIndex.append(index)
  index+=1
plt.figure(figsize=(20,3))
for plotIndex, wrong in enumerate(misclassifiedIndex[0:4]):
  plt.subplot(1, 4, plotIndex +1)
  plt.imshow(np.reshape(x_test[wrong], (8,8)), cmap=plt.cm.gray)
  plt.title("predicted: {}, Actual: {}" .format(predictions[wrong], y_test[wrong]), fontsize=20)
```



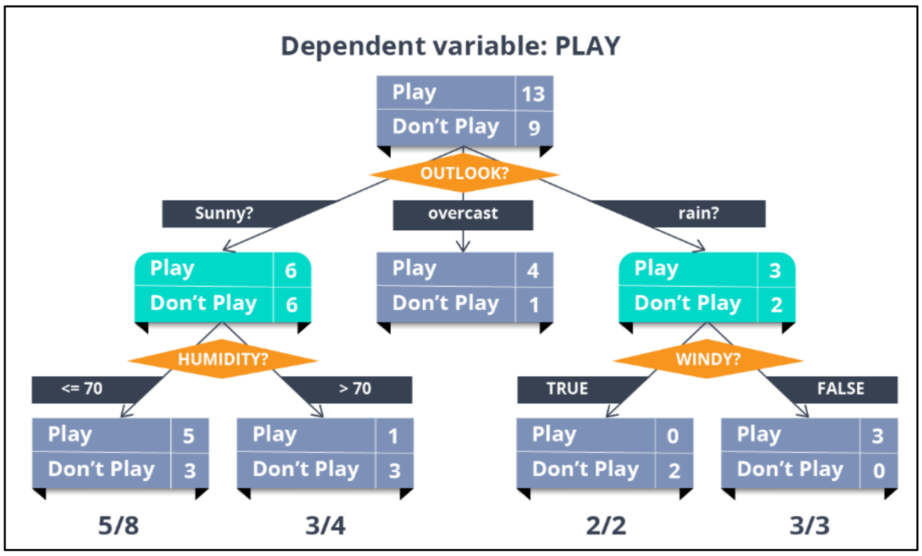
Result

Thus the Logistic Regression to predict digit values from images is implemented and executed successfully using Python Program

Ex.No: 6 To Implement Decision Tree and Random Forest

A decision tree is a flowchart-like tree structure where an internal node represents a feature(or attribute), the branch represents a decision rule, and each leaf node represents the outcome.

The topmost node in a decision tree is known as the root node. It learns to partition on the basis of the attribute value. It partitions the tree in a recursive manner called recursive partitioning. This flowchart-like structure helps you in decision-making. It's visualization like a flowchart diagram which easily mimics the human level thinking. That is why decision trees are easy to understand and interpret.



Attribute Selection Measures

Attribute selection measure is a heuristic for selecting the splitting criterion that partitions data in the best possible manner. It is also known as splitting rules because it helps us to determine breakpoints for tuples on a given node. ASM provides a rank to each feature (or attribute) by explaining the given dataset. The best score attribute will be selected as a splitting attribute (**Source**). In the case of a continuous-valued attribute, split points for branches also need to define. The most popular selection measures are Information Gain, Gain Ratio, and Gini Index.

Information Gain

Claude Shannon invented the concept of entropy, which measures the impurity of the input set. In physics and mathematics, entropy is referred to as the randomness or the impurity in a system. In information theory, it refers to the impurity in a group of examples. Information gain is the decrease in entropy. Information gain computes the difference between entropy before the split and average entropy after the split of the dataset based on given attribute values. ID3 (Iterative Dichotomiser) decision tree algorithm uses information gain.

$$\text{Info}(D) = -\sum_{i=1}^{m} pi \log_2 pi$$

Where Pi is the probability that an arbitrary tuple in D belongs to class Ci.

$$\text{Info}_A(D) = \sum_{j=1}^{v} \frac{|Dj|}{|D|} \ \text{X Info}(D_j)$$

$$\text{Gain}(A) = \text{Info}(D) - \text{Info}_A(D)$$

Where:

- Info(D) is the average amount of information needed to identify the class label of a tuple in D.
- |Dj|/|D| acts as the weight of the jth partition.
- InfoA(D) is the expected information required to classify a tuple from D based on the partitioning by A.

The attribute A with the highest information gain, Gain(A), is chosen as the splitting attribute at node N().

Gain Ratio

Information gain is biased for the attribute with many outcomes. It means it prefers the attribute with a large number of distinct values. For instance, consider an attribute with a unique identifier, such as customer_ID, that has zero info(D) because of pure partition. This maximizes the information gain and creates useless partitioning.

C4.5, an improvement of ID3, uses an extension to information gain known as the gain ratio. Gain ratio handles the issue of bias by normalizing the information gain using Split Info. Java implementation of the C4.5 algorithm is known as J48, which is available in WEKA data mining tool.

$$SplitInfo_A(D) = -\sum_{j=1}^{v} \frac{|D_j|}{|D|} \times \log_2\left(\frac{|D_j|}{|D|}\right)$$

Where:

- |Dj|/|D| acts as the weight of the jth partition.
- v is the number of discrete values in attribute A.

The gain ratio can be defined as

$$GainRatio(A) = \frac{Gain(A)}{SplitInfo_A(D)}$$

The attribute with the highest gain ratio is chosen as the splitting attribute (**Source**).

Gini index

Another decision tree algorithm CART (Classification and Regression Tree) uses the Gini method to create split points.

$$\text{Gini(D)}=1-\sum_{i=1}^{m} \text{Pi}^2$$

Where pi is the probability that a tuple in D belongs to class Ci.

The Gini Index considers a binary split for each attribute. You can compute a weighted sum of the impurity of each partition. If a binary split on attribute A partitions data D into D1 and D2, the Gini index of D is:

$$\text{Gini}_A(\text{D})=\frac{|D1|}{|D|}\text{Gini}(\text{D}_1)+\frac{|D2|}{|D|}\text{Gini}(\text{D}_2)$$

In the case of a discrete-valued attribute, the subset that gives the minimum gini index for that chosen is selected as a splitting attribute. In the case of continuous-valued attributes, the strategy is to select each pair of adjacent values as a possible split point, and a point with a smaller gini index is chosen as the splitting point.

$$\Delta Gini(A) = Gini(D) - Gini_A(D).$$

The attribute with the minimum Gini index is chosen as the splitting attribute.

Ex.No: 6.a To Implement Decision Tree to Predict Diabetes

Date:

Aim:

To implement Decision Tree to predict diabetes based on given features using Python Programming

Algorithm

Step 1: Start
Step 2: Import sklearn
Step 3: Initialize features and labels
Step 4: Import all required packages to perform Decision tree in machine learning
Step 5: Display the result
Step 6: Stop

Source Code - Importing Required Libraries

Let's first load the required libraries.

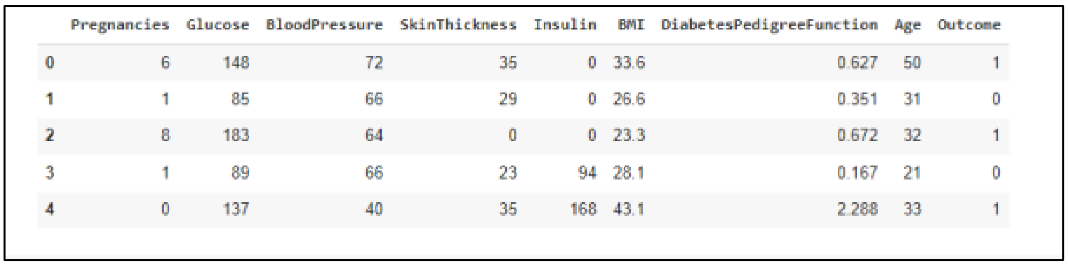
```
import pandas as pd

from sklearn.tree import DecisionTreeClassifier # Import Decision Tree Classifier

from sklearn.model_selection import train_test_split # Import train_test_split function

from sklearn import metrics #Import scikit-learn metrics module for accuracy calculation
```

Loading Data

Pima Indian Diabetes dataset

```
col_names = ['pregnant', 'glucose', 'bp', 'skin', 'insulin', 'bmi', 'pedigree', 'age', 'label']

# load dataset

pima = pd.read_csv("diabetes.csv", header=None, names=col_names)
```

pima.head()



	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
0	6	148	72	35	0	33.6	0.627	50	1
1	1	85	66	29	0	26.6	0.351	31	0
2	8	183	64	0	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1

Feature Selection

Here, you need to divide given columns into two types of variables dependent(or target variable) and independent variable(or feature variables).

```
#split dataset in features and target variable

feature_cols = ['pregnant', 'insulin', 'bmi', 'age','glucose','bp','pedigree']

X = pima[feature_cols] # Features

y = pima.label # Target variable
```

Splitting Data

To understand model performance, dividing the dataset into a training set and a test set is a good strategy.

Let's split the dataset by using the function train_test_split(). You need to pass three parameters features; target, and test_set size.

```
# Split dataset into training set and test set

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=1) # 70% training and 30% test
```

Building Decision Tree Model

Let's create a decision tree model using Scikit-learn.

```
# Create Decision Tree classifer object

clf = DecisionTreeClassifier()

# Train Decision Tree Classifer
```

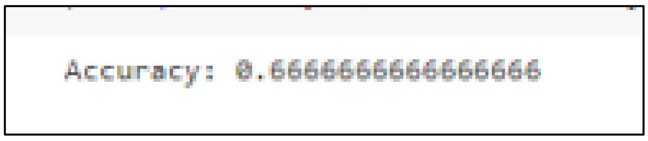
```
clf = clf.fit(X_train,y_train)

#Predict the response for test dataset

y_pred = clf.predict(X_test)
```

Evaluating the Model

Let's estimate how accurately the classifier or model can predict the type of cultivars.

Accuracy can be computed by comparing actual test set values and predicted values.

```
# Model Accuracy, how often is the classifier correct?

print("Accuracy:",metrics.accuracy_score(y_test, y_pred))
```



```
Accuracy: 0.6666666666666666
```

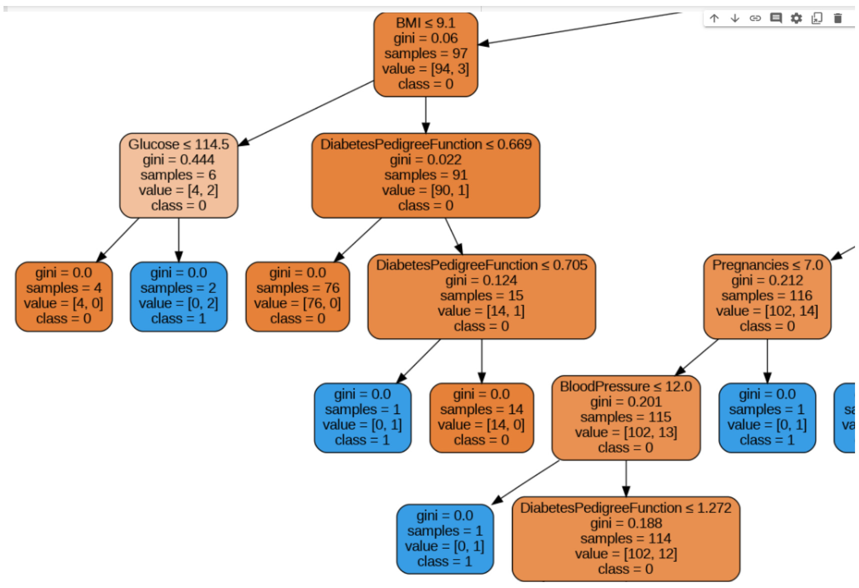
Visualizing Decision Trees

```
pip install graphviz

pip install pydotplus
```

The export_graphviz function converts the decision tree classifier into a dot file, and pydotplus converts this dot file to png or displayable

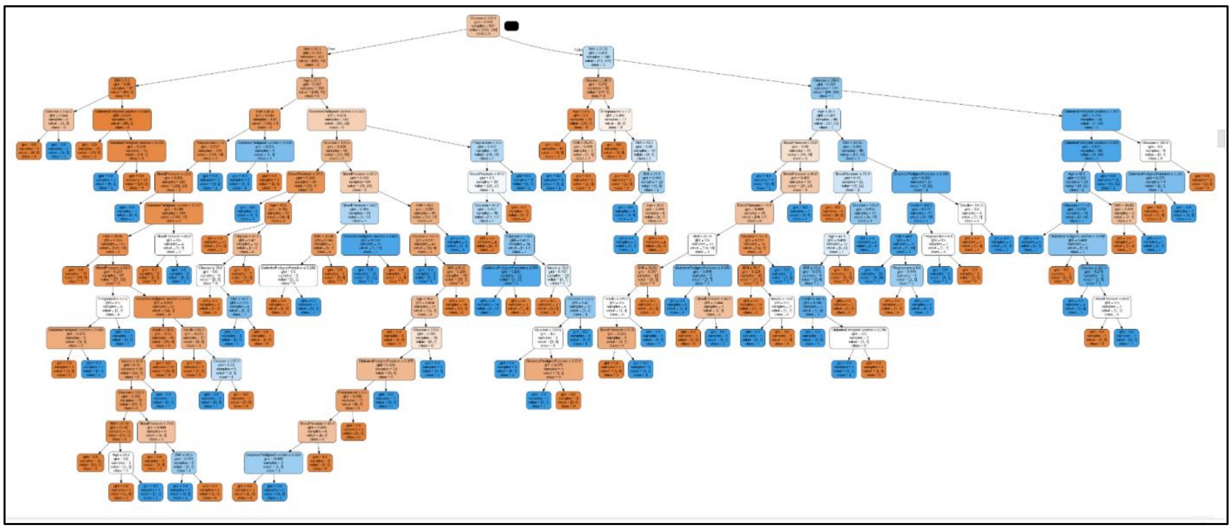
```
from sklearn.tree import export_graphviz

from sklearn.externals.six import StringIO

from IPython.display import Image

import pydotplus

dot_data = StringIO()

export_graphviz(clf, out_file=dot_data,

                filled=True, rounded=True,
special_characters=True,feature_names = feature_cols,class_names=['0','1'])

graph = pydotplus.graph_from_dot_data(dot_data.getvalue())

graph.write_png('diabetes.png')

Image(graph.create_png())
```

Example



BMI ≤ 9.1 gini = 0.06 samples = 97 value = [94, 3] class = 0
Glucose ≤ 114.5 gini = 0.444 samples = 6 value = [4, 2] class = 0
DiabetesPedigreeFunction ≤ 0.669 gini = 0.022 samples = 91 value = [90, 1] class = 0
gini = 0.0 samples = 4 value = [4, 0] class = 0
gini = 0.0 samples = 2 value = [0, 2] class = 1
gini = 0.0 samples = 76 value = [76, 0] class = 0
DiabetesPedigreeFunction ≤ 0.705 gini = 0.124 samples = 15 value = [14, 1] class = 0
Pregnancies ≤ 7.0 gini = 0.212 samples = 116 value = [102, 14] class = 0
gini = 0.0 samples = 1 value = [0, 1] class = 1
gini = 0.0 samples = 14 value = [14, 0] class = 0
BloodPressure ≤ 12.0 gini = 0.201 samples = 115 value = [102, 13] class = 0
gini = 0.0 samples = 1 value = [0, 1] class = 1
gini = 0.0 samples = 1 value = [0, 1] class = 1
DiabetesPedigreeFunction ≤ 1.272 gini = 0.188 samples = 114 value = [102, 12] class = 0

Decision Tree



In the decision tree chart, each internal node has a decision rule that splits the data. Gini, referred to as Gini ratio, measures the impurity of the node. You can say a node is pure when all of its records belong to the same class, such nodes known as the leaf node.

Here, the resultant tree is unpruned. This unpruned tree is unexplainable and not easy to understand. In the next section, let's optimize it by pruning.

Optimizing Decision Tree Performance

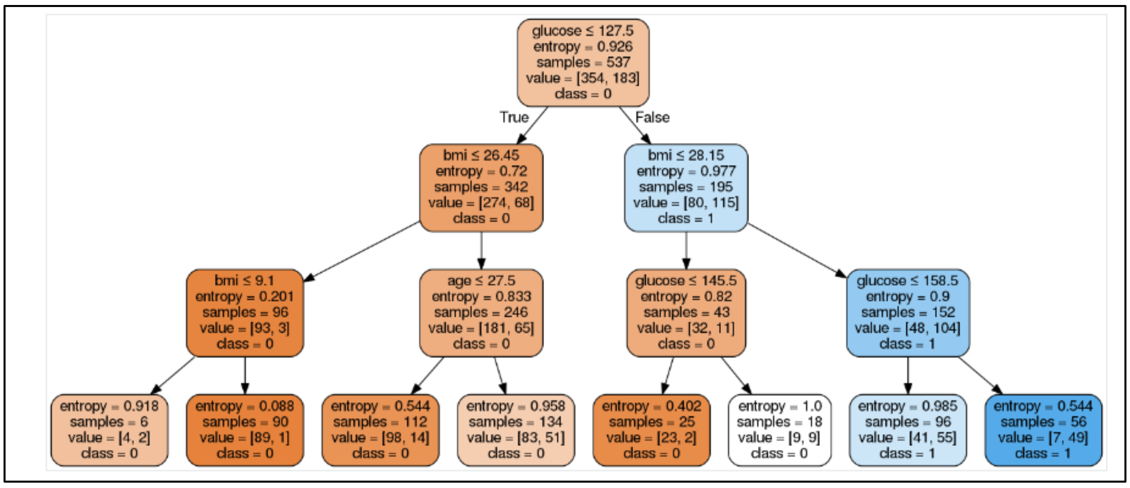
- **criterion : optional (default="gini") or Choose attribute selection measure.** This parameter allows us to use the different-different attribute selection measure. Supported criteria are "gini" for the Gini index and "entropy" for the information gain.
- **splitter : string, optional (default="best") or Split Strategy.** This parameter allows us to choose the split strategy. Supported strategies are "best" to choose the best split and "random" to choose the best random split.
- **max_depth : int or None, optional (default=None) or Maximum Depth of a Tree.** The maximum depth of the tree. If None, then nodes are expanded until all the leaves contain less than min_samples_split samples. The higher value of maximum depth causes overfitting, and a lower value causes underfitting (**Source**).

In Scikit-learn, optimization of decision tree classifier performed by only pre-pruning. Maximum depth of the tree can be used as a control variable for pre-pruning. In the following the example, you can plot a decision tree on the same data with max_depth=3. Other than pre-pruning parameters, You can also try other attribute selection measure such as entropy.

```
# Create Decision Tree classifer object

clf = DecisionTreeClassifier(criterion="entropy", max_depth=3)

# Train Decision Tree Classifer

clf = clf.fit(X_train,y_train)

#Predict the response for test dataset

y_pred = clf.predict(X_test)

# Model Accuracy, how often is the classifier correct?

print("Accuracy:",metrics.accuracy_score(y_test, y_pred))

Accuracy: 0.7705627705627706
```

Visualizing Decision Trees

```
from six import StringIO from IPython.display import Image
from sklearn.tree import export_graphviz

import pydotplus

dot_data = StringIO()

export_graphviz(clf, out_file=dot_data,

                filled=True, rounded=True,

                special_characters=True, feature_names = feature_cols,class_names=['0','1'])

graph = pydotplus.graph_from_dot_data(dot_data.getvalue())

graph.write_png('diabetes.png')

Image(graph.create_png())
```


glucose ≤ 127.5 entropy = 0.926 samples = 537 value = [354, 183] class = 0
True
False
bmi ≤ 26.45 entropy = 0.72 samples = 342 value = [274, 68] class = 0
bmi ≤ 28.15 entropy = 0.977 samples = 195 value = [80, 115] class = 1
bmi ≤ 9.1 entropy = 0.201 samples = 96 value = [93, 3] class = 0
age ≤ 27.5 entropy = 0.833 samples = 246 value = [181, 65] class = 0
glucose ≤ 145.5 entropy = 0.82 samples = 43 value = [32, 11] class = 0
glucose ≤ 158.5 entropy = 0.9 samples = 152 value = [48, 104] class = 1
entropy = 0.918 samples = 6 value = [4, 2] class = 0
entropy = 0.088 samples = 90 value = [89, 1] class = 0
entropy = 0.544 samples = 112 value = [98, 14] class = 0
entropy = 0.958 samples = 134 value = [83, 51] class = 0
entropy = 0.402 samples = 25 value = [23, 2] class = 0
entropy = 1.0 samples = 18 value = [9, 9] class = 0
entropy = 0.985 samples = 96 value = [41, 55] class = 1
entropy = 0.544 samples = 56 value = [7, 49] class = 1

Result

Thus the Decision Tree to predict Diabetes from the given feature is implemented and executed successfully using Python Program

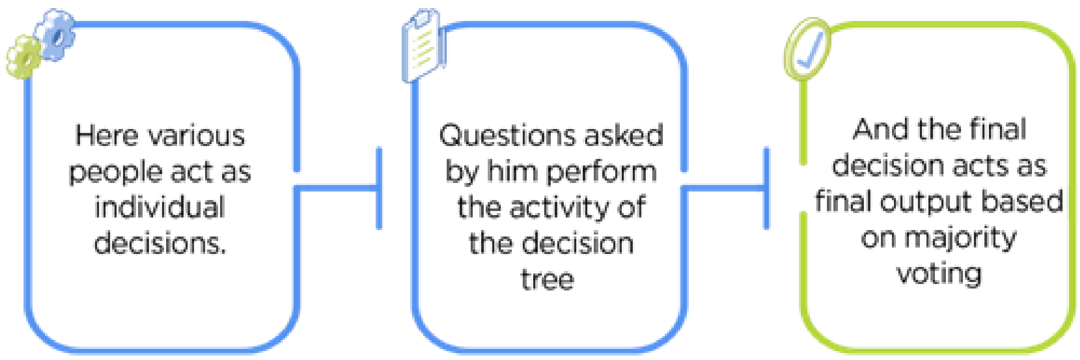
Random Forests

Random Forest is one of the most popular and commonly used algorithms by Data Scientists. Random forest is a Supervised Machine Learning Algorithm that is used widely in Classification and Regression problems. It builds decision trees on different samples and takes their majority vote for classification and average in case of regression.

One of the most important features of the Random Forest Algorithm is that it can handle the data set containing continuous variables, as in the case of regression, and categorical variables, as in the case of classification. It performs better for classification and regression tasks. In this tutorial, we will understand the working of random forest and implement random forest on a classification task.

Real-Life Analogy of Random Forest

Let's dive into a real-life analogy to understand this concept further. A student named X wants to choose a course after his 10+2, and he is confused about the choice of course based on his skill set. So he decides to consult various people like his cousins, teachers, parents, degree students, and working people. He asks them varied questions like why he should choose, job opportunities with that course, course fee, etc. Finally, after consulting various people about the course he decides to take the course suggested by most people.


Here various people act as individual decisions.
Questions asked by him perform the activity of the decision tree
And the final decision acts as final output based on majority voting

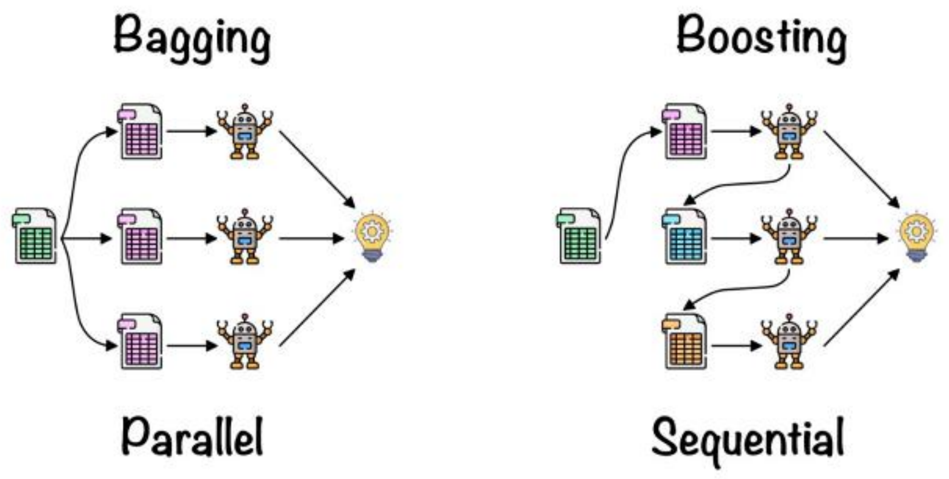
Working of Random Forest Algorithm

Before understanding the working of the random forest algorithm in machine learning, we must look into the ensemble learning technique. *Ensemble* simplymeans combining multiple models. Thus a collection of models is used to make predictions rather than an individual model.

Ensemble uses two types of methods:

1. Bagging– It creates a different training subset from sample training data with replacement & the final output is based on majority voting. For example, Random Forest.

2. Boosting– It combines weak learners into strong learners by creating sequential models such that the final model has the highest accuracy. For example, ADA BOOST, XG BOOST.



Random forest works on the Bagging principle. Now let's dive in and understand bagging in detail.

Bagging

Bagging, also known as Bootstrap Aggregation, is the ensemble technique used by random forest.Bagging chooses a random sample/random subset from the entire data set. Hence each model is generated from the samples (Bootstrap Samples) provided by the Original Data with replacement known as row sampling. This step of row sampling with replacement is called bootstrap. Now each model is trained independently, which generates results. The final output is based on majority voting after combining the results of all models. This step which involves combining all the results and generating output based on majority voting, is known as aggregation.

Steps Involved in Random Forest Algorithm

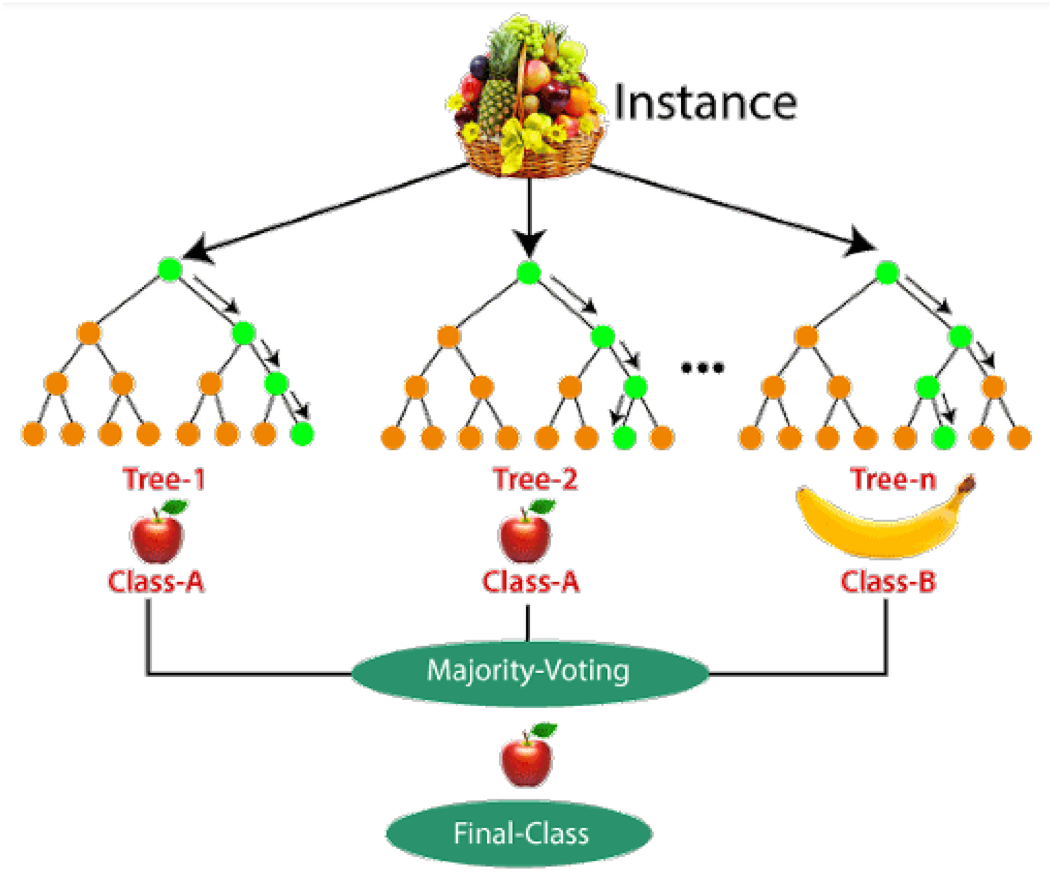
Step 1: In the Random forest model, a subset of data points and a subset of features is selected for constructing each decision tree. Simply put, n random records and m features are taken from the data set having k number of records.

Step 2: Individual decision trees are constructed for each sample.

Step 3: Each decision tree will generate an output.

Step 4: Final output is considered based on *Majority Voting or Averaging* for Classification and regression, respectively.

For example: consider the fruit basket as the data as shown in the figure below. Now n number of samples are taken from the fruit basket, and an individual decision tree is constructed for each sample. Each decision tree will generate an output, as shown in the figure. The final output is considered based on majority voting. In the below figure, you can see that the majority decision tree gives output as an apple when compared to a banana, so the final output is taken as an apple.



Hyperparameters in Random Forest

Hyperparameters are used in random forests to either enhance the performance and predictive power of models or to make the model faster.
Hyperparameters to Increase the Predictive Power

n_estimators: Number of trees the algorithm builds before averaging the predictions.
max_features: Maximum number of features random forest considers splitting a node.
mini_sample_leaf: Determines the minimum number of leaves required to split an internal node.
criterion: How to split the node in each tree? (Entropy/Gini impurity/Log Loss)
max_leaf_nodes: Maximum leaf nodes in each tree

Hyperparameters to Increase the Speed

n_jobs: it tells the engine how many processors it is allowed to use. If the value is 1, it can use only one processor, but if the value is -1, there is no limit.
random_state: controls randomness of the sample. The model will always produce the same results if it has a definite value of random state and has been given the same hyperparameters and training data.
oob_score: *OOB* means out of the bag. It is a random forest cross-validation method. In this, one-third of the sample is not used to train the data; instead used to evaluate its performance. These samples are called out-of-bag samples.

Ex.No: 6.b To Implement Random Forest to Predict Heart Disease

Date:

Aim:

To implement Random Forest to predict heart disease using Random Forest

Algorithm

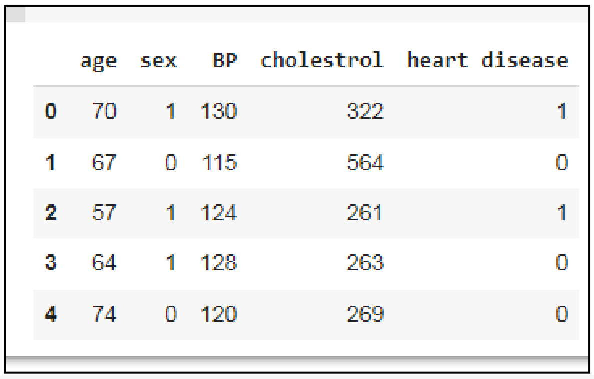
Step 1: Start
Step 2: Import sklearn
Step 3: Initialize features and labels
Step 4: Import all required packages to perform Random Forest in machine learning
Step 5: Display the result
Step 6: Stop

Source Code – Import Libraries

```
import pandas as pd, numpy as np
import matplotlib.pyplot as plt, seaborn as sns
%matplotlib inline
```

Source Code – Import Dataset

```
df=pd.read_csv("https://raw.githubusercontent.com/soumya-mishra/Heart-
Disease_DT/main/heart_v2.csv")
df.head()
df.to_csv("heart_v2.csv")
```



```
   age  sex   BP  cholestrol  heart disease
0   70    1  130         322              1
1   67    0  115         564              0
2   57    1  124         261              1
3   64    1  128         263              0
4   74    0  120         269              0
```

Source Code – Putting Feature Variable to X and Target variable to y

```
X=df.drop('heart disease', axis=1)
y=df['heart disease']
```

```
((189, 5), (81, 5))
```

Source Code – Train-Test-Split

```
X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.7, random_state=42)
X_train.shape, X_test.shape
```

```
pip install -U scikit-learn
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Requirement already satisfied: scikit-learn in /usr/local/lib/python3.10/dist-packages (1.2.2)
Requirement already satisfied: numpy>=1.17.3 in /usr/local/lib/python3.10/dist-packages (from scikit-learn) (1.22.4
Requirement already satisfied: scipy>=1.3.2 in /usr/local/lib/python3.10/dist-packages (from scikit-learn) (1.10.1)
Requirement already satisfied: joblib>=1.1.1 in /usr/local/lib/python3.10/dist-packages (from scikit-learn) (1.2.0)
Requirement already satisfied: threadpoolctl>=2.0.0 in /usr/local/lib/python3.10/dist-packages (from scikit-learn)
```

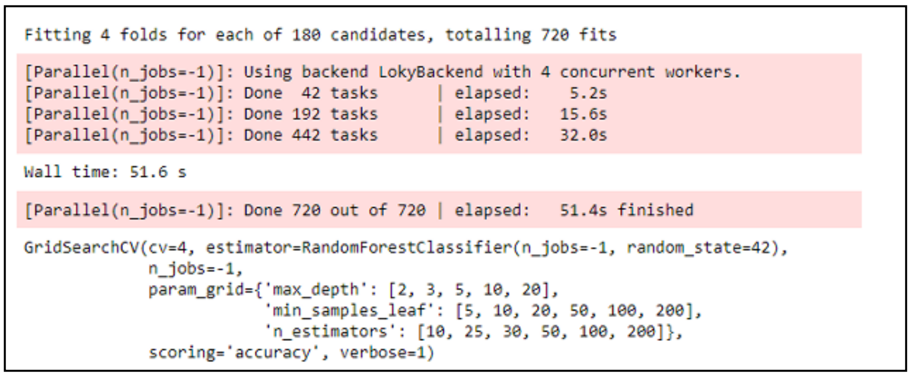
Source Code – Import RandomForestClassifier and fit the data

```
from sklearn.ensemble import RandomForestClassifier

classifier_rf = RandomForestClassifier(random_state=42, n_jobs=-1, max_depth=5,
n_estimators=100, oob_score=True)
classifier_rf.fit(X_train, y_train)
classifier_rf.oob_score_
```

```
0.6402116402116402
```

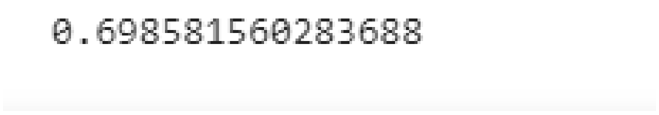
Source Code – Hyperparameter tuning for Random Forest using GridSearchCV and fit the data

```
rf = RandomForestClassifier(random_state=42, n_jobs=-1)
params = {
    'max_depth': [2,3,5,10,20],
    'min_samples_leaf': [5,10,20,50,100,200],
    'n_estimators': [10,25,30,50,100,200]
}

from sklearn.model_selection import GridSearchCV
# Instantiate the grid search model
grid_search = GridSearchCV(estimator=rf,
                           param_grid=params,
                           cv = 4,
                           n_jobs=-1, verbose=1, scoring="accuracy")
grid_search.fit(X_train, y_train)
```



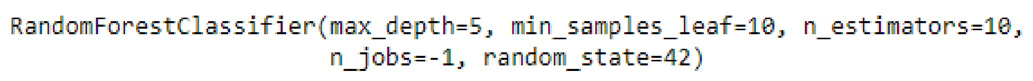
```
Fitting 4 folds for each of 180 candidates, totalling 720 fits
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=-1)]: Done  42 tasks      | elapsed:    5.2s
[Parallel(n_jobs=-1)]: Done 192 tasks      | elapsed:   15.6s
[Parallel(n_jobs=-1)]: Done 442 tasks      | elapsed:   32.0s
Wall time: 51.6 s
[Parallel(n_jobs=-1)]: Done 720 out of 720 | elapsed:   51.4s finished
GridSearchCV(cv=4, estimator=RandomForestClassifier(n_jobs=-1, random_state=42),
             n_jobs=-1,
             param_grid={'max_depth': [2, 3, 5, 10, 20],
                         'min_samples_leaf': [5, 10, 20, 50, 100, 200],
                         'n_estimators': [10, 25, 30, 50, 100, 200]},
             scoring='accuracy', verbose=1)
```

```
grid_search.best_score_
```



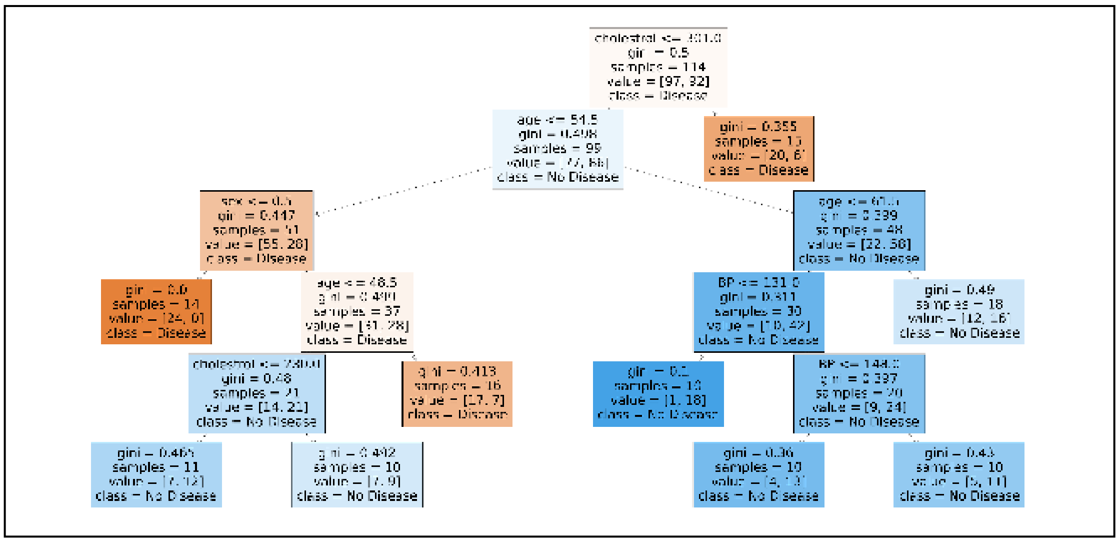
```
0.698581560283688
```

```
rf_best = grid_search.best_estimator_
rf_best
```

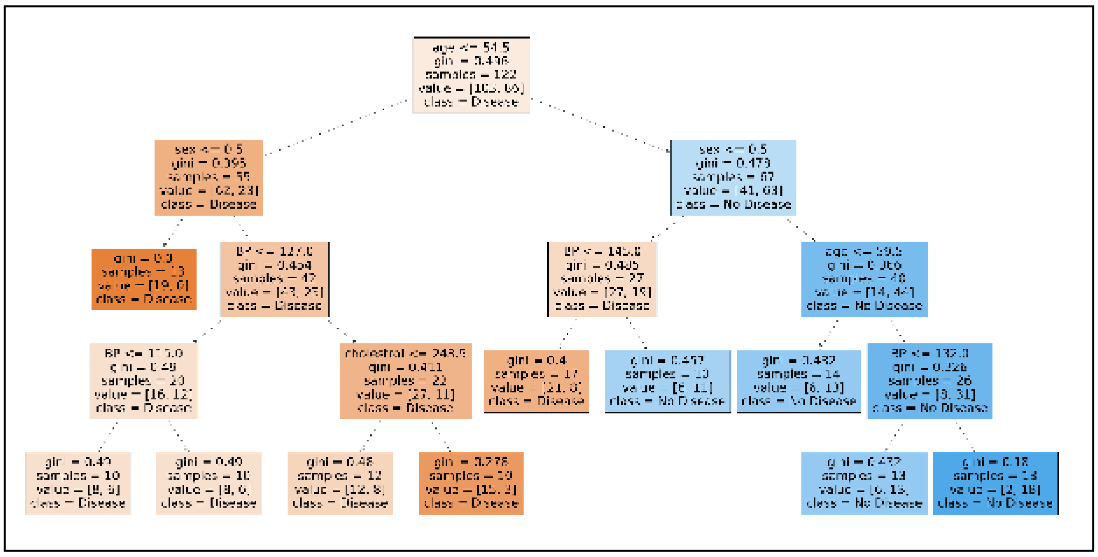


```
RandomForestClassifier(max_depth=5, min_samples_leaf=10, n_estimators=10,
                       n_jobs=-1, random_state=42)
```

```
from sklearn.tree import plot_tree
plt.figure(figsize=(80,40))
plot_tree(rf_best.estimators_[5], feature_names = X.columns,class_names=['Disease', "No Disease"],filled=True);
```

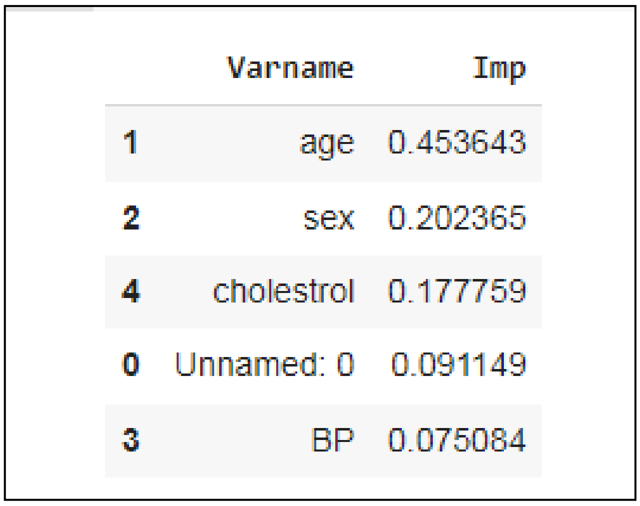


from sklearn.tree import plot_tree
plt.figure(figsize=(80,40))
plot_tree(rf_best.estimators_[5], feature_names = X.columns,class_names=['Disease', "No Disease"],filled=True);



Source Code – Feature importance

```
rf_best.feature_importances_
imp_df = pd.DataFrame({
    "Varname": X_train.columns,
    "Imp": rf_best.feature_importances_
})
imp_df.sort_values(by="Imp", ascending=False)
```



	Varname	Imp
1	age	0.453643
2	sex	0.202365
4	cholestrol	0.177759
0	Unnamed: 0	0.091149
3	BP	0.075084

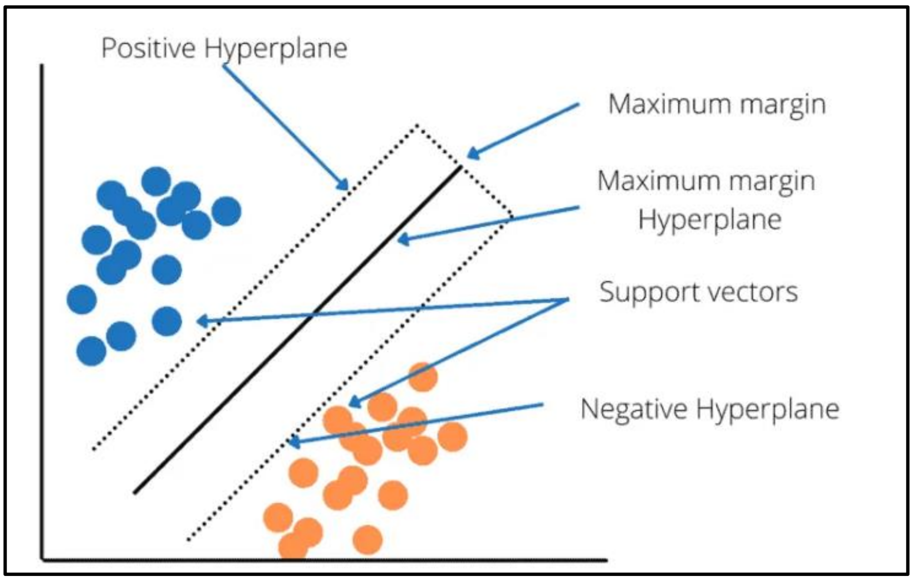
Result

Thus the Random Forest to predict heart disease from the given feature in dataset is implemented and executed successfully using Python Programming.

Ex.No: 7 **To Implement SVM Model**

Support Vector Machine (SVM), also known as Support Vector Classification, is a supervised and linear Machine Learning technique typically used to solve classification problems. SVR stands for Support Vector Regression and is a subset of SVM that uses the same ideas to tackle regression problems. SVM also supports the kernel method called the kernel SVM, which allows us to tackle non-linearity.

The primary use case for SVM is classification, but it can solve classification and regression problems. SVM constructs a hyperplane (see the picture below) in multidimensional space to separate different classes. It iteratively generates the best hyperplane to minimize classification error. The goal of SVM is to find a maximum marginal hyperplane (MMH) that splits a dataset into classes as evenly as possible.

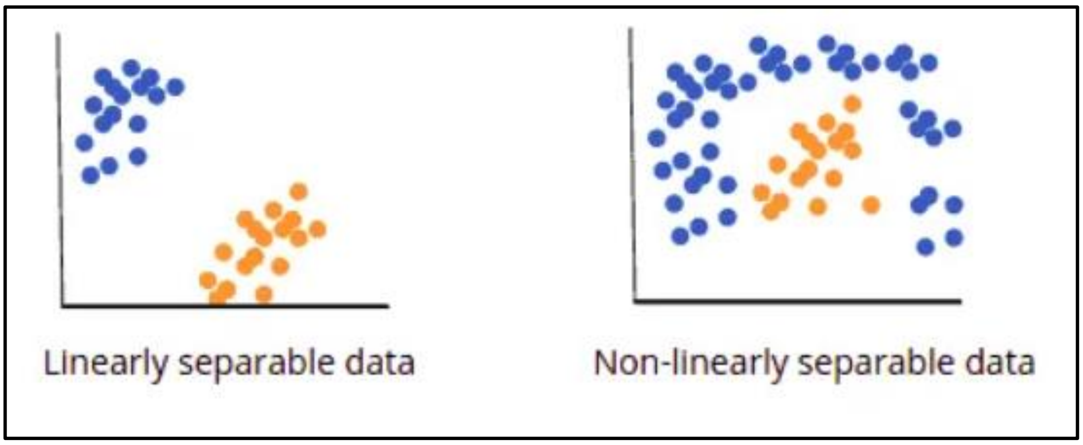


Support Vectors are data points closest to the hyperplane called support vectors. These points will define the separating line better by calculating margins and are more relevant to the construction of the classifier.

A hyperplane is a decision plane that separates objects with different class memberships.
Margin is the distance between the two lines on the class points closest to each other. It is calculated as the perpendicular distance from the line to support vectors or nearest points. The bold margin between the classes is good, whereas a thin margin is not good.
Depending on the type of data, there are two types of Support Vector Machines:

- Linear SVM or Simple SVM is used for data that is linearly separable. A dataset is termed linearly separable data if it can be classified into two classes using a single straight line, and the classifier is known as the linear SVM classifier. It’s most commonly used for tasks involving linear regression and classification.

- Nonlinear SVM or Kernel SVM also known as Kernel SVM, is a type of SVM that is used to classify nonlinearly separated data, or data that cannot be classified using a straight line. It has more flexibility for nonlinear data because more features can be added to fit a hyperplane instead of a two-dimensional space.



The objective of SVM is to draw a line that best separates the two classes of data points. SVM produces a line that cleanly divides the two classes. There are many other ways to construct a line that separates the two classes, but in SVM, the margins and support vectors are used.

Ex.No: 7   To implement SVM model to predict product is purchased by customer or not based on given features

Date:

Aim:

To implement SVM model to predict customer is purchase the product or not based on given features using python programming.

Algorithm

Step 1: Start
Step 2: Import sklearn
Step 3: Initialize features and labels
Step 4: Import all required packages to perform Support Vector Machine in machine learning
Step 5: Import Grid Search to find optimal Hyper parameters for SVM
Step 6: Display the result
Step 7: Stop

Source code - Install Packages

```
% pip install sklearn

% pip install pandas

% pip install seaborn

% pip install matplotlib

% pip install numpy
```

Source Code – Import Libraries

```
# importing the libraries

import matplotlib.pyplot as plt

import pandas as pd

import seaborn as sns
```

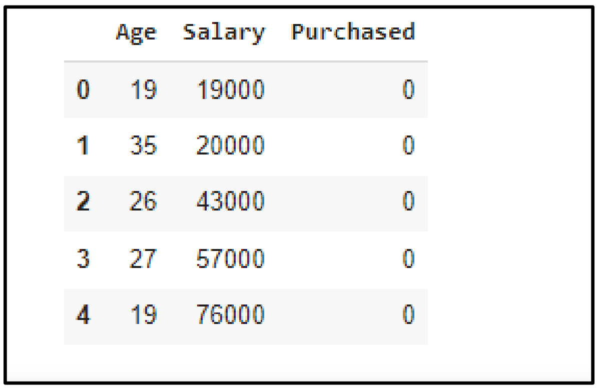
Source Code – Import the data set and divide it into input and output variables

from google.colab import drive

drive.mount('/content/drive')

df = pd.read_csv("/content/drive/My Drive/customer_purchases.csv")

df.head()



	Age	Salary	Purchased
0	19	19000	0
1	35	20000	0
2	26	43000	0
3	27	57000	0
4	19	76000	0

Source Code

# split the data into inputs and outputs

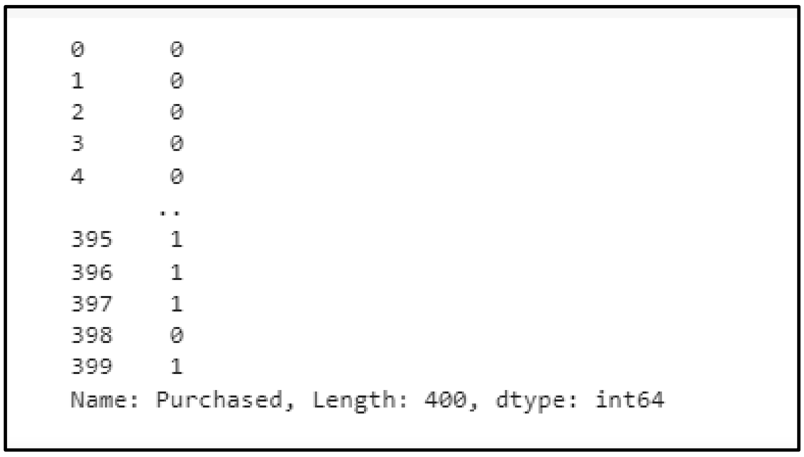
X = dataset.iloc[:, [0,1]].values

y = dataset.iloc[:, 2].values

print out the target/output class to verify that our data is a binary set

# printing the target values

print(dataset.Purchased)



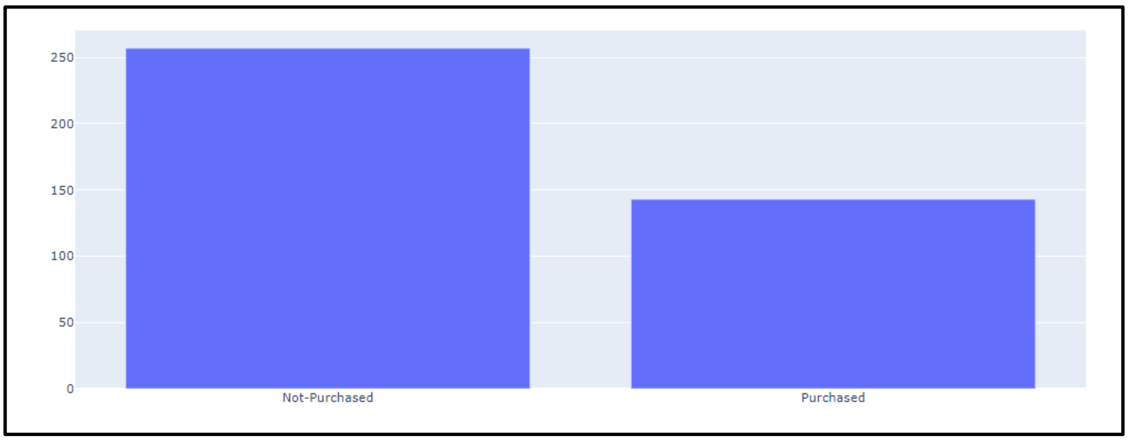
```
0      0
1      0
2      0
3      0
4      0
      ..
395    1
396    1
397    1
398    0
399    1
Name: Purchased, Length: 400, dtype: int64
```

the output class contains either 1 or 0, showing whether the customer had purchased the product or not. The next thing we can do as a part of data pre-processing is visually seen the number of those output classes.

```
pip install chart_studio
```

Source Code - importing the required modules for data visualization

```
import matplotlib.pyplot as plt

import chart_studio.plotly as py

import plotly.graph_objects as go

import plotly.offline as pyoff

import pandas as pd

# importing the dats set

df = pd.read_csv("/content/drive/My Drive/customer_purchases.csv")

# counting the total output data from purchased column

target_balance = df['Purchased'].value_counts().reset_index()

# dividing the output classes into two sections

target_class = go.Bar(

    name = 'Target Balance',

    x = ['Not-Purchased', 'Purchased'],

    y = target_balance['Purchased']

)

# ploting the output classes

fig = go.Figure(target_class)

pyoff.iplot(fig)
```



Source Code – Training And Testing Linear SVM Model

Once we are done with the pre-processing of the data, we can move into the splitting part to divide the data into the testing and training parts.

```
# training and testing data

from sklearn.model_selection import train_test_split

# assign test data size 25%

X_train, X_test, y_train, y_test =train_test_split(X, y, test_size=0.25, random_state=0)
```

Source Code - importing StandardScaler

```
from sklearn.preprocessing import StandardScaler

# scalling the input data

sc_X = StandardScaler()

X_train = sc_X.fit_transform(X_train)

X_test = sc_X.fit_transform(X_test)
```

Source Code - importing SVM module

```
from sklearn.svm import SVC

# kernel to be set linear as it is binary class

classifier = SVC(kernel='linear')
```

```
# traininf the model

classifier.fit(X_train, y_train)
```

Testing the model

```
# testing the model

y_pred = classifier.predict(X_test)
```

# importing accuracy score

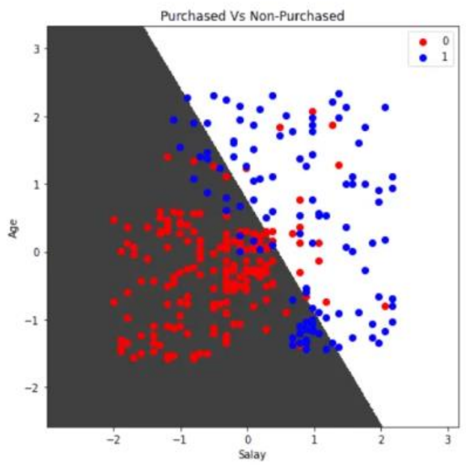
```
from sklearn.metrics import accuracy_score

# printing the accuracy of the model

print(accuracy_score(y_test, y_pred))
```

```
Output:

0.88
```

Source Code –Visualize Training Data

```
import matplotlib.pyplot as plt

from matplotlib.colors import ListedColormap

# plotting the fgiure

plt.figure(figsize = (7,7))

# assigning the input values

X_set, y_set = X_train, y_train

# ploting the linear graph

X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 1, stop = X_set[:, 0].max() + 1, step = 0.01), np.arange(start = X_set[:, 1].min() - 1, stop = X_set[:, 1].max() + 1, step = 0.01))

plt.contourf(X1, X2, classifier.predict(np.array([X1.ravel(), X2.ravel()]).T).reshape(X1.shape), alpha = 0.75, cmap = ListedColormap(('black', 'white')))
plt.xlim(X1.min(), X1.max())

plt.ylim(X2.min(), X2.max())

# ploting scattered graph for the values

for i, j in enumerate(np.unique(y_set)):

    plt.scatter(X_set[y_set == j, 0], X_set[y_set == j, 1], c = ListedColormap(('red', 'blue'))(i),
label = j)

# labeling the graph

plt.title('Purchased Vs Non-Purchased')

plt.xlabel('Salay')

plt.ylabel('Age')

plt.legend()

plt.show()
```

Output:



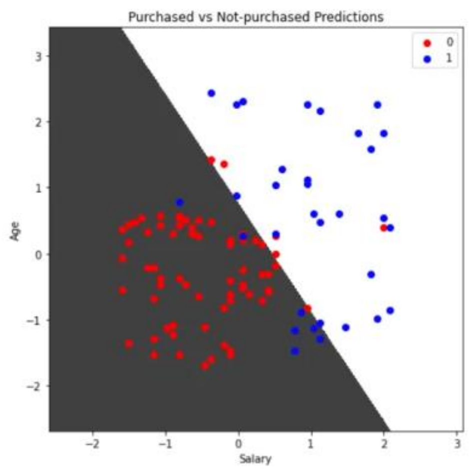
Notice that there is a linear boundary between the two classes because we have specified the Kernel to be linear.

Source Code - ploting graph of size 7,7

```
plt.figure(figsize = (7,7))

# assigning the testing dataset

X_set, y_set = X_test, y_test

# ploting the predicted graph
```

```
X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 1, stop = X_set[:, 0].max() + 1, step = 0.01),np.arange(start = X_set[:, 1].min() - 1, stop = X_set[:, 1].max() + 1, step = 0.01))

plt.contourf(X1, X2, classifier.predict(np.array([X1.ravel(), X2.ravel()]).T).reshape(X1.shape),alpha = 0.75, cmap = ListedColormap(('black', 'white')))

plt.xlim(X1.min(), X1.max())

plt.ylim(X2.min(), X2.max())

# plorting scattred graph for the testing values

for i, j in enumerate(np.unique(y_set)):

    plt.scatter(X_set[y_set == j, 0], X_set[y_set == j, 1],c = ListedColormap(('red', 'blue'))(i), label = j)

# labelling the graphe

plt.title('Purchased vs Not-purchased Predictions')

plt.xlabel('Salary')

plt.ylabel('Age')

plt.legend()

plt.show()
```

Output:

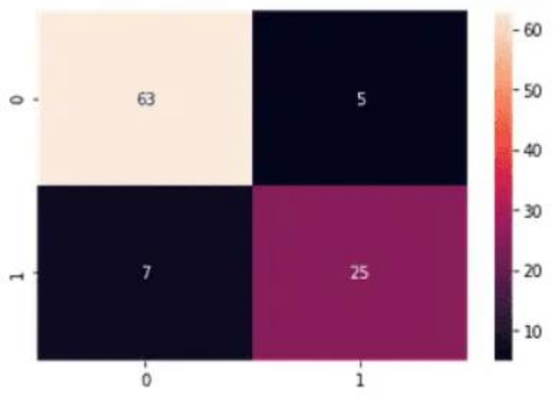


You can consider any testing point in the black area as Not-purchased and any point in the while area as Purchased.

Evaluation of SVM Algorithm Performance For Binary Classification
Source Code - importing the required modules

```
import seaborn as sns

from sklearn.metrics import confusion_matrix

# passing actual and predicted values

cm = confusion_matrix(y_test, y_pred, labels=classifier.classes_)

# true Write data values in each cell of the matrix

sns.heatmap(cm, annot=True)

plt.savefig('confusion.png')
```

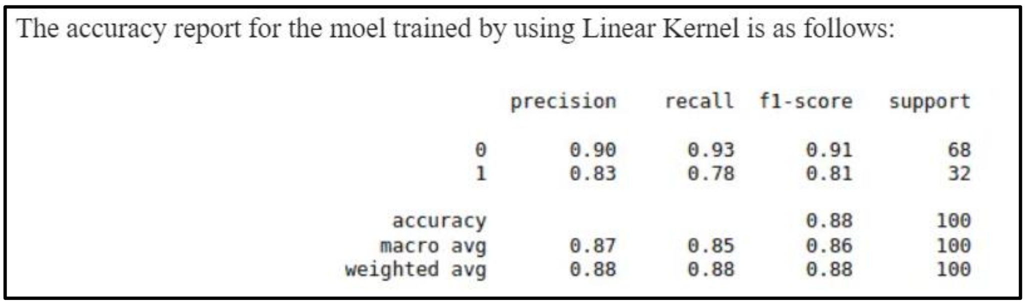
Output:



This output shows that 63 of the Non-purchased class were classified correctly, and 25 of the purchased were classified correctly.

Source Code - importing classification report

```
from sklearn.metrics import classification_report

# printing the report

print(classification_report(y_test, y_pred))
```


The accuracy report for the moel trained by using Linear Kernel is as follows:

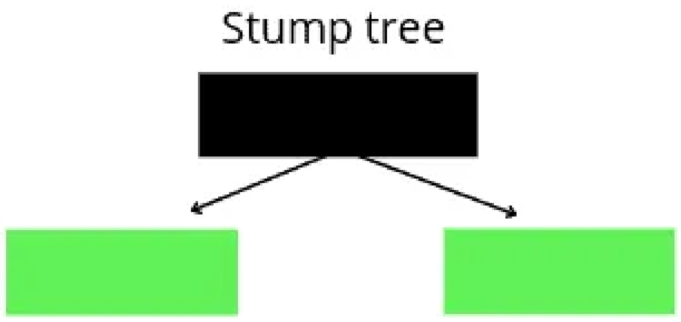
```
              precision    recall  f1-score   support

           0       0.90      0.93      0.91        68
           1       0.83      0.78      0.81        32

    accuracy                           0.88       100
   macro avg       0.87      0.85      0.86       100
weighted avg       0.88      0.88      0.88       100
```

Result

Thus the Support Vector Machine to predict customer purchase the product or not from the given feature in dataset is implemented and executed successfully using Python Programming.

Ex.No: 8 Implement Ensembling Techniques - AdaBoosting

AdaBoost (also called Adaptive Boosting) is an **ensemble method** technique in Machine Learning that combines several base models in order to produce one optimal predictive model. The AdaBoost can use any classifier making weak predictions and combine them to build a strong predictive model. The most popular classifier used by AdaBoost algorithm is Decision Trees with one level (the Decision Trees does only 1 split). These trees are called **Decision Stumps** which are similar to Random Forest trees, but not "fully grown."



The three most important ideas behind the AdaBoost algorithm using Decision Trees are:

- AdaBoost combines a lot of week learners to make better predictions which are known as stumps
- Some stumps have more contribution to the predictions than others
- Each stump tree is considering the previous stump's mistakes into account

To understand how the AdaBoost algorithm works, we will take a sample dataset that contains data about whether a patient has heart disease or not depending on some input variables. And we will also restrict the AdaBoost algorithm to be trained on only one stump tree.

Ex.No: 8 Implement Ensembling Techniques - AdaBoosting to Classify the Flower Species

Date:

Aim:

To implement AdaBoost Ensembling model to predict classify flower species based on given dataset using Python Programming

Algorithm

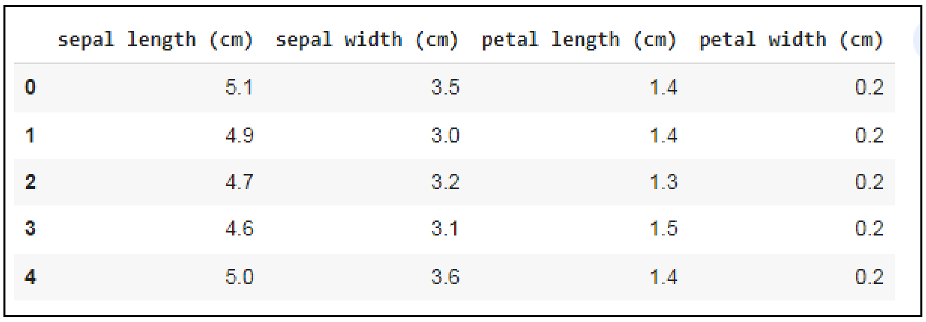
Step 1: Start
Step 2: Import sklearn
Step 3: Initialize features and labels
Step 4: Import all required packages to perform AdaBoost Ensembling model in machine learning
Step 5: Display the result
Step 6: Stop

Source code - Install Packages

```
import pandas as pd

import numpy as np

from sklearn import datasets

# loading the dataset

iris = datasets.load_iris()
```

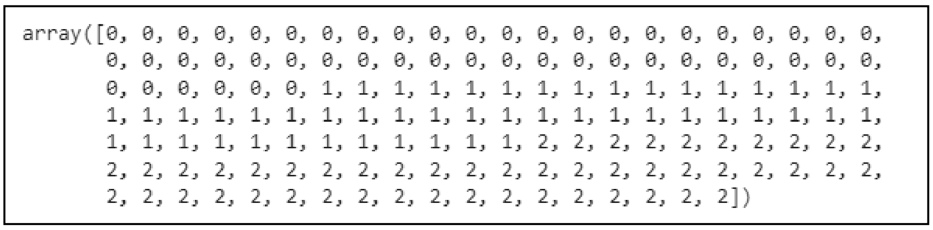
Source code - converting the dataset into pandas dataframe

```
data = pd.DataFrame(iris.data, columns=iris.feature_names)

# head

data.head()
```



	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)
0	5.1	3.5	1.4	0.2
1	4.9	3.0	1.4	0.2
2	4.7	3.2	1.3	0.2
3	4.6	3.1	1.5	0.2
4	5.0	3.6	1.4	0.2

iris.target



```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2])
```

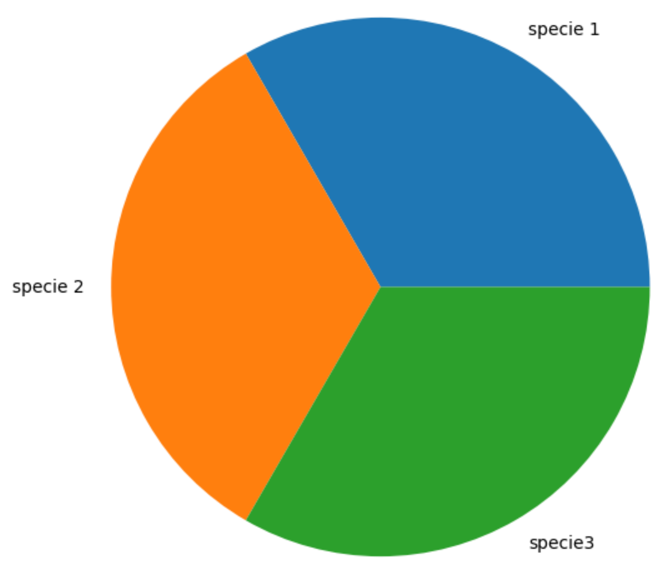
Source code - creating variables

```
specie1=0

specie2 = 0

specie3 = 0

labels=["specie 1", 'specie 2', 'specie3']

# for loop to count the outputs

for i in iris.target:

    if i ==0:

        specie1+=1

    elif i ==1:

        specie2+=1

    else:

        specie3+=1
```

Source code - importing required module

```
import matplotlib.pyplot as plt
fig = plt.figure()
# Creating plot
fig = plt.figure(figsize =(10, 7))
plt.pie([specie1, specie2, specie3], labels = labels)
 # show plot
plt.show()
```

```
<Figure size 640x480 with 0 Axes>
```



```
# input and output
Input, output = datasets.load_iris(return_X_y=True)
```

Source code - importing the module

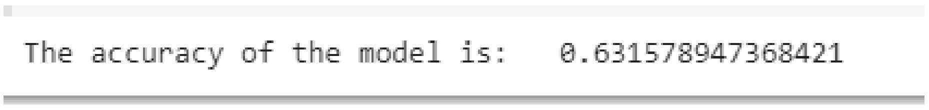
```
from sklearn.model_selection import train_test_split
# splitting the dataset
X_train, X_test, y_train, y_test = train_test_split(Input, output, test_size=0.25)
```

Source code - Import the AdaBoost classifier

```
from sklearn.ensemble import AdaBoostClassifier

# Create adaboost classifer with 1 stump trees

Ada_classifier = AdaBoostClassifier(n_estimators=1)

# Train Adaboost Classifer

AdaBoost = Ada_classifier.fit(X_train, y_train)

#Predict the response for test dataset

AdaBoost_pred = AdaBoost.predict(X_test)
```

Source code - importing the module

```
from sklearn.metrics import accuracy_score

# printing

print("The accuracy of the model is:  ", accuracy_score(y_test, AdaBoost_pred))
```



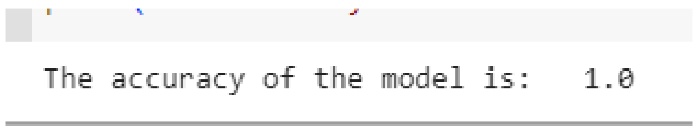
```
The accuracy of the model is:   0.631578947368421
```

Source code - Create adaboost classifer with 20 stump trees

```
Ada_classifier = AdaBoostClassifier(n_estimators=20)

# Train Adaboost Classifer

AdaBoost = Ada_classifier.fit(X_train, y_train)

#Predict the response for test dataset

AdaBoost_pred = AdaBoost.predict(X_test)
```

# printing

print("The accuracy of the model is: ", accuracy_score(y_test, AdaBoost_pred))



```
The accuracy of the model is:   1.0
```

Result

Thus the AdaBoost to predict the class of the flower from the given feature in dataset is implemented and executed successfully using Python Programming.

Ex.No: 9 **Implement Clustering Algorithms**

K-Means Clustering

Clustering represents a set of unsupervised machine learning algorithms belonging to different categories such as prototype-based clustering, hierarchical clustering, density-based clustering etc. **K-means** is one of the most popular clustering algorithm belong to **prototype-based clustering** category. The idea is to create **K clusters (**hence, K-means name) of data where data in each of the **K clusters** have greater similarity with other data in the same cluster. The algorithm works by iteratively assigning each data point to the cluster whose centroid is closest to it, and then updating the centroids based on the new assignments. The algorithm terminates when the assignment of data points to clusters no longer changes.

In K-Means clustering, the goal is to divide a given dataset into K clusters, where each data point belongs to the cluster with the nearest mean value. The algorithm works by iteratively updating the cluster centroids until convergence is achieved.

Key steps of K-Means clustering algorithm

The following represents the key steps of K-means clustering algorithm:

- Define **number of clusters, K,** which need to be found out. Randomly select **K cluster data points (cluster centers) or cluster centroids.** The goal is to optimise the position of the K centroids.
- For each observation, find out the **Euclidean distance** between the observation and all the K cluster centers. Of all distances, find the nearest distance between the observation and one of the K cluster centroids (cluster centers) and assign the observation to that cluster.
- Move the K-centroids to the center of the points assigned to it.
- Repeat the above two steps until there is no change in the cluster centroids or maximum number of iterations or user-defined tolerance is reached.

What is the objective function in K-means which get optimized?

K-means clustering algorithm is an optimization problem where the goal is to minimise the within-cluster **sum of squared errors** (**SSE**). At times, SSE is also termed as **cluster inertia.** SSE is the sum of the squared differences between each observation and the cluster centroid. At each stage of cluster analysis the total SSE is minimised with $SSE_{total} = SSE_1 + SSE_2 + SSE_3 + SSE_4 \ldots + SSE_n$.

$$SSE = \sum_{i=1}^{n}\sum_{j=1}^{k} w^{(i,j)} \left\| x^{(i)} - \mu^{(j)} \right\|_2^2$$

key features of K-means algorithm

The following are some of the **key features of K-means clustering algorithm**:

- One needs to define the number of clusters (K) beforehand. This is unlike other clustering algorithms related to hierarchical clustering or density-based clustering algorithms. The need to define the number of clusters, K, a priori can be considered as a **disadvantage** because for the real-world applications, it may not always be evident as to how many clusters can the data be partitioned into.
- K-means clusters do not overlap and are not hierarchical.
- **Produces hard clustering**: K-Means clustering produces **hard clustering**, which means that each data point is assigned to a single cluster.
- It is an **unsupervised learning technique** that does not require labeled data.
- **Can handle different data types**: K-Means clustering can handle both continuous and categorical data types, although it is typically used with continuous data.

How to find most optimal value of K?

One of the key challenges in using K-Means clustering is determining the optimal number of clusters (K) to use. Choosing the right value of K is important, as it can significantly affect the quality of the clustering results. There are several methods for determining the optimal value of K, including:

Elbow method

The technique used to find the most optimal value of K is draw a reduction in variation vs number of clusters (K) plot. Alternatively, one could draw the squared sum of error (SSE) vs number of clusters (K) plot. Here is the diagram representing the plot of SSE vs K (no. of clusters). In the diagram below, the point representing the optimal number of clusters can also be called as **elbow point.** The elbow point can be seen as the point after which the distortion/cluster inertia/SSE start decreasing in a linear fashion.

Silhouette method

The silhouette method involves computing the average silhouette score for each value of K and selecting the value of K with the highest silhouette score. The silhouette score measures the quality of clustering based on the distance between data points within a cluster compared to the distance between data points in different clusters.

Gap Statistics Method
The gap statistic method involves computing the gap statistic for each value of K and selecting the value of K with the largest gap statistic. The gap statistic measures the difference between the within-cluster sum of squares for a given value of K and the expected within-cluster sum of squares for a random sample.

Ex.No: 9 Implement K-Means Clustering to Cluster Similar Flower Species

Date:

Aim:

To implement K-means Clustering to predict species of the flowers based on given features using Python Programming

Algorithm

Step 1: Start
Step 2: Import sklearn
Step 3: Initialize features and labels
Step 4: Import all required packages to perform K-means clustering
Step 5: Display the result
Step 6: Stop

Source code - Install Packages

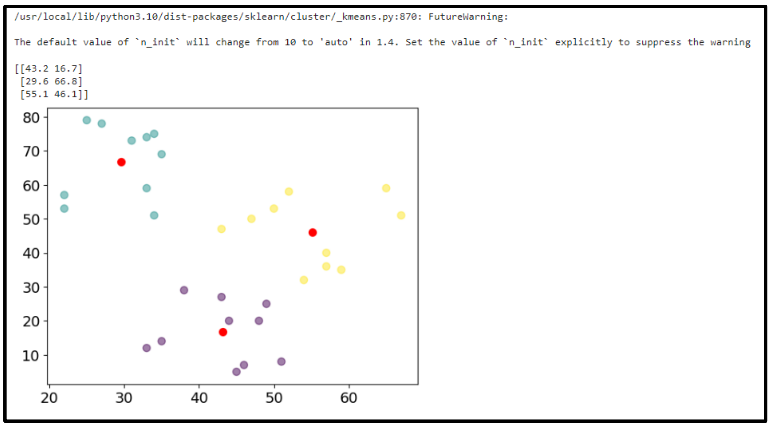
K-Means Clustering

Implement K-Means Clustering using Scikit-Learn for IRIS dataset, which contains information about different species of flowers.

Source Code

```
import pandas as pd

import matplotlib.pyplot as plt

from sklearn.cluster import KMeans

data = {

   'x': [25, 34, 22, 27, 33, 33, 31, 22, 35, 34, 67, 54, 57, 43, 50, 57, 59, 52, 65, 47, 49, 48, 35, 33, 44, 45, 38,

        43, 51, 46],

   'y': [79, 51, 53, 78, 59, 74, 73, 57, 69, 75, 51, 32, 40, 47, 53, 36, 35, 58, 59, 50, 25, 20, 14, 12, 20, 5, 29, 27,

        8, 7]

   }
```

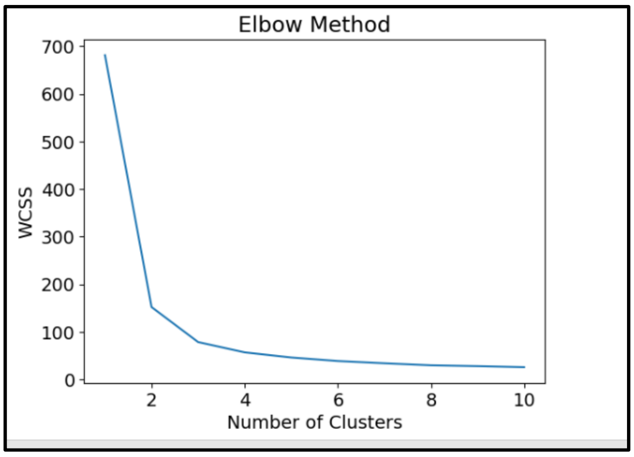
```python
df = pd.DataFrame(data)

kmeans = KMeans(n_clusters=3).fit(df)

centroids = kmeans.cluster_centers_

print(centroids)


plt.scatter(df['x'], df['y'], c=kmeans.labels_.astype(float), s=50, alpha=0.5)

plt.scatter(centroids[:, 0], centroids[:, 1], c='red', s=50)

plt.show()
```



```
/usr/local/lib/python3.10/dist-packages/sklearn/cluster/_kmeans.py:870: FutureWarning:

The default value of `n_init` will change from 10 to 'auto' in 1.4. Set the value of `n_init` explicitly to suppress the warning

[[43.2 16.7]
 [29.6 66.8]
 [55.1 46.1]]
```

Source Code - To find optimal value of K using Elbow Method

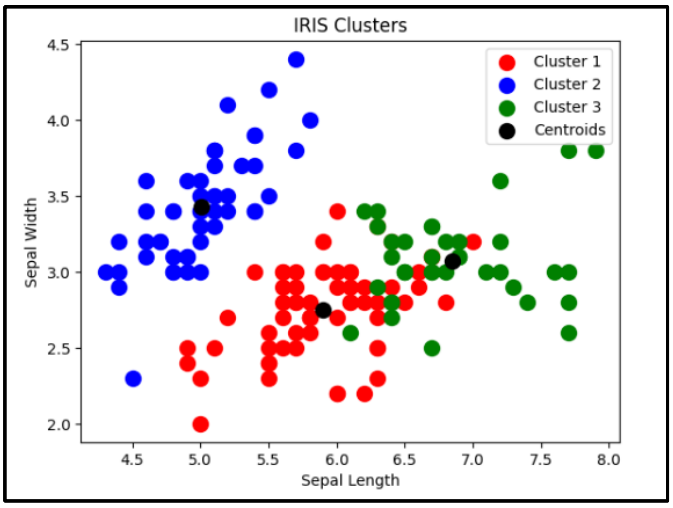
```python
from sklearn.datasets import load_iris

import matplotlib.pyplot as plt

from sklearn.cluster import KMeans

 # Load IRIS Dataset

iris = load_iris()

X = iris.data


# Create the WCSS Plot against no. of clusters

wcss = []

for i in range(1, 11):
   kmeans = KMeans(n_clusters=i, init='k-means++', max_iter=300, n_init=10, random_state=0)

   kmeans.fit(X)

   wcss.append(kmeans.inertia_)lt.plot(range(1, 11), wcss)

plt.title('Elbow Method')

plt.xlabel('Number of Clusters')

plt.ylabel('WCSS')

plt.show()
```



Create a K-Means model with 3 clusters and fit it to the IRIS dataset. The **fit_predict** method returns the cluster labels for each data point in the IRIS dataset, which we can use to visualize the clusters.

Source Code - Train K-Means Clusters

```
kmeans = KMeans(n_clusters=3, init='k-means++', max_iter=300, n_init=10, random_state=0)

y_kmeans = kmeans.fit_predict(X)

 # Create the visualization plot of the clusters

plt.scatter(X[y_kmeans == 0, 0], X[y_kmeans == 0, 1], s = 100, c = 'red', label = 'Cluster 1')

plt.scatter(X[y_kmeans == 1, 0], X[y_kmeans == 1, 1], s = 100, c = 'blue', label = 'Cluster 2')

plt.scatter(X[y_kmeans == 2, 0], X[y_kmeans == 2, 1], s = 100, c = 'green', label = 'Cluster 3')
plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1], s = 100, c = 'black', label = 'Centroids')

plt.title('IRIS Clusters')

plt.xlabel('Sepal Length')

plt.ylabel('Sepal Width')

plt.legend()

plt.show()
```



Result

Thus the K-means Cluster to predict the species of the flower from the given feature in dataset is implemented and executed successfully using Python Programming.

Ex.No: 10 Implement EM for Bayesian networks

Bayesian Network with Hidden Variables

bayesnet_em predicts values for a hidden variable in a Bayesian network by implementing the expectation maximization algorithm. It works as an extension to the Beysian network implementation in Pomegranade.

A Bayesian network is a probabilistic graphical model that represents relationships between random variables as a direct acyclic graph. Each node in the network represents a random variable whearas each edge represents a conditional dependency. Bayesian networks provides an efficient way to construct a full joint probability distribution over the variables. The random varibles can either be observed variables or unobserved variables, in which case they are called hidden (or latent) variables.

Pomegranade currently supports a discrete Baysian network. Each node represents a categorical variable, which means it can take on a discrete number of values. The model parameters can be learned from data. However (at least as of now), it does not support a network with hidden variables. The purpose of bayesnet_em is to work with the pomegranade Bayesian network model to predict values of the hidden variables.

bayesnet_em takes an already constructed and initialized BayesianNetwork object, a data array, and the index of the hidden node, and returns a complete data set.

This simple relationship describes a naive Bayes model. The full joint probability distribution is

$$P(F,C,T,S) = P(F)*P(C|F)*P(T|F)*P(S|F)$$

Ex.No: 10 **Implement EM for Bayesian networks**

Date:

Aim:

To implement Expectation Maximization for Bayesian Network using Python Programming

Algorithm

Step 1: Start
Step 2: Initialize features and labels
Step 3: Assign weight for features
Step 4: Import all required packages to perform Neural Computation
Step 5: Display the result
Step 6: Stop

Source code - Install Packages

Expectation Maximization

```
from .em import *

import
numpy
as np

        from pomegranate import *

        import itertools

        from .mb import *

        def em_bayesnet(model, data, ind_h, max_iter = 50, criteria = 0.005):

            """Returns the data array with the hidden node filled in.

            (model is not modified.)


            Parameters

            ----------

            model : a BayesianNetwork object
        an already baked BayesianNetwork object with initialized parameters

    data : an ndarray
        each column is the data for the node in the same order as the nodes in the model
        the hidden node should be a column of NaNs

    ind_h : int
        index of the hidden node

    max_iter : int
        maximum number of iterations

    criteria : float between 0 and 1
        the change in probability in consecutive iterations, below this value counts as convergence

    Returns
    -------
    data : an ndarray
        the same data arary with the hidden node column filled in
    """

    # create the Markov blanket object for the hidden node
    mb = MarkovBlanket(ind_h)
    mb.populate(model)
    mb.calculate_prob(model)
```

```
# create the count table from data

items = data[:, mb.parents + mb.children + mb.coparents]

ct = CountTable(model, mb, items)


# create expected counts

expected_counts = ExpectedCounts(model, mb)

expected_counts.update(model, mb)


# ---- iterate over the E-M steps

i = 0

previous_params = np.array(mb.prob_table[mb.hidden].values())

convergence = False


while (not convergence) and (i < max_iter):

    mb.update_prob(model, expected_counts, ct)

    expected_counts.update(model, mb)

    # print 'Iteration',i,mb.prob_table


    # convergence criteria

    hidden_params = np.array(mb.prob_table[mb.hidden].values())

    change = abs(hidden_params - previous_params)

    convergence = max(change) < criteria

    previous_params = np.array(mb.prob_table[mb.hidden].values())


    i += 1
```

```
        if i == max_iter:
            print 'Maximum iterations reached.'

        # ---- fill in the hidden node data by sampling the distribution
        labels = {}
        for key, prob in expected_counts.counts.items():
            try:
                labels[key[1:]].append((key[0], prob))
            except:
                labels[key[1:]] = [(key[0], prob)]

        for key, counts in ct.table.items():
            label, prob = zip(*labels[key])
            prob = tuple(round(p,5) for p in prob)
            if not all(p == 0 for p in prob):
                samples = np.random.choice(label, size=counts, p=prob)
                data[ct.ind[key], ind_h] = samples

        return data
```

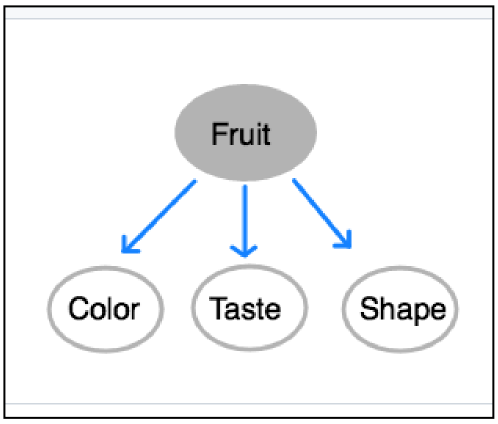
Markov Blanket Model

```
def
search_hidden(data):

            Parameters
            ----------
            data : An ndarray (n_sample, n_nodes)
```

```
    Returns
    -------
    ind_h : the index of the hidden node column
    """
    is_col_nan = np.all(np.isnan(data), axis=0)
    ind = np.where(is_col_nan)
    if np.size(ind)==1:
        ind_h = ind[0][0]
    else:
        raise ValueError('Data contains more than one hidden nodes or no hidden
node')
    return ind_h
class MarkovBlanket():
    """
    An object for storing info on nodes within the markov blanket of the hidden
node

    Parameters
    ----------
    ind_h : int
        index of the hidden node within the model

    Attributes
    ----------
    hidden : int
        index of the hidden node

    parents : list of int
    a list of indices of the parent nodes

children : list of int
    a list of indices of the children nodes

coparents : list of int
    a list of indices of the coparent nodes

prob_table : dict
    a dict of probabilities table of nodes within the Markov blanket

"""

def __init__(self, ind_h):
    self.hidden = ind_h
    self.parents = []
    self.children = []
    self.coparents = []
    self.prob_table = {}

def populate(self, model):
    """populate the parents, children, and coparents nodes
    """
    state_indices = {state.name : i for i, state in enumerate(model.states)}

    edges_list = [(parent.name, child.name) for parent, child in model.edges]
    edges_list = [(state_indices[parent],state_indices[child])
              for parent, child in edges_list]
```

```
        self.children = list(set([child for parent, child in edges_list if parent==self.hidden]))
        self.parents = list(set([parent for parent, child in edges_list if child==self.hidden]))
        self.coparents = list(set([parent for parent, child in edges_list if child in self.children]))
        try:
            self.coparents.remove(self.hidden)
        except ValueError:
            pass


    def calculate_prob(self, model):
        """Create the probability table from nodes
        """
        for ind_state in [self.hidden]+self.children:
            distribution = model.states[ind_state].distribution


            if isinstance(distribution, ConditionalProbabilityTable):
                table = distribution.parameters[0]
                self.prob_table[ind_state] = {
                    tuple(row[:-1]) : row[-1] for row in table}
            else:
                self.prob_table[ind_state] = distribution.parameters[0]


    def update_prob(self, model, expected_counts, ct):
        """Update the probability table using expected counts
        """
        ind = {x : i for i, x in enumerate([self.hidden] + self.parents + self.children + self.coparents)}
        mb_keys = expected_counts.counts.keys()

        for ind_state in [self.hidden] + self.children:
            distribution = model.states[ind_state].distribution

            if isinstance(distribution, ConditionalProbabilityTable):
                idxs = distribution.column_idxs
                table = self.prob_table[ind_state] # dict

                # calculate the new parameter for this key
                for key in table.keys():
                    num = 0
                    denom = 0

                    # marginal counts
                    for mb_key in mb_keys:
                        # marginal counts of node + parents
                        if tuple([mb_key[ind[x]] for x in idxs]) == key:
                            num                                          +=
    ct.table[mb_key[1:]]*expected_counts.counts[mb_key]

                        # marginal counts of parents
                        if tuple([mb_key[ind[x]] for x in idxs[:-1]]) == key[:-1]:
                            denom                                        +=
    ct.table[mb_key[1:]]*expected_counts.counts[mb_key]

                    try:
                        prob = num/denom
                    except ZeroDivisionError:
                        prob = 0

                    # update the parameter
                    table[key] = prob

            else: # DiscreteProb
                table = self.prob_table[ind_state] # dict

                # calculate the new parameter for this key
                for key in table.keys():
                    prob = 0
                    for mb_key in mb_keys:
                        if mb_key[ind[ind_state]] == key:
                            prob                                                  +=
    ct.table[mb_key[1:]]*expected_counts.counts[mb_key]

                    # update the parameter
                    table[key] = prob

    class ExpectedCounts():
        """Calculate the expected counts using the model parameters

        Parameters
        ----------
        model : a BayesianNetwork object

        mb : a MarkovBlanket object
        Attributes
        ----------
        counts : dict
            a dict of expected counts for nodes in the Markov blanket
        """

        def __init__(self, model, mb):
            self.counts = {}

            self.populate(model, mb)

        def populate(self, model, mb):
            #create combinations of keys
            keys_list = [model.states[mb.hidden].distribution.keys()]
            for ind in mb.parents + mb.children + mb.coparents:
            keys_list.append(model.states[ind].distribution.keys())

            self.counts = {p:0 for p in itertools.product(*keys_list)}

        def update(self, model, mb):
            ind = {x : i for i, x in enumerate([mb.hidden] + mb.parents + mb.children
    + mb.coparents)}

            marginal_prob = {}

            # calculate joint probability and marginal probability
            for i, key in enumerate(self.counts.keys()):
            prob = 1
            for j, ind_state in enumerate([mb.hidden] + mb.children):
                distribution = model.states[ind_state].distribution

                if isinstance(distribution, ConditionalProbabilityTable):
                    idxs = distribution.column_idxs
                    state_key = tuple([key[ind[x]] for x in idxs])
                else:
                    state_key = key[ind[ind_state]]

                prob = prob*mb.prob_table[ind_state][state_key]
                self.counts[key] = prob
            try:
                marginal_prob[key[1:]] += prob
            except KeyError:
                marginal_prob[key[1:]] = prob

        # divide the joint prob by the marginal prob to get the conditional
        for i, key in enumerate(self.counts.keys()):
            try:
                self.counts[key] = self.counts[key]/marginal_prob[key[1:]]
            except ZeroDivisionError:
                self.counts[key] = 0

class CountTable():
    """Counting the data"""

    def __init__(self, model, mb, items):
```

```
    """
    Parameters
    ----------
    model : BayesianNetwork object

    mb : MarkovBlanket object

    items : ndarray
        columns are data for parents, children, coparents

    """
    self.table ={}
    self.ind = {}

    self.populate(model, mb, items)

def populate(self, model, mb, items):
    keys_list = []
    for ind in mb.parents + mb.children + mb.coparents:
        keys_list.append(model.states[ind].distribution.keys())

    # init
    self.table = {p:0 for p in itertools.product(*keys_list)}
    self.ind = {p:[] for p in itertools.product(*keys_list)}

    # count
    for i, row in enumerate(items):
        try:
            self.table[tuple(row)] += 1
            self.ind[tuple(row)].append(i)
        except KeyError:
            print 'Items in row', i, 'does not match the set of keys.'
            raise KeyError
```



Bayesian Network

```
import numpy as np
from pomegranate import *
data = np.array([[np.nan, 'yellow', 'sweet', 'long'],
                 [np.nan, 'green', 'sour', 'round'],
                 [np.nan, 'green', 'sour', 'round'],
                 [np.nan, 'yellow', 'sweet', 'long'],
                 [np.nan, 'yellow', 'sweet', 'long'],
                 [np.nan, 'green', 'sour', 'round'],
                 [np.nan, 'green', 'sweet', 'long'],
                 [np.nan, 'green', 'sweet', 'round']])
```

The columns represent the nodes in a specified order (fruit, color, taste, shape). The order of the columns have to match the order of the nodes (states) when constructing the Bayesian network. The first column with the nan values is the hidden node. Next, create the distributions of all the nodes and initialize the probabilities to some non-uniform values. The first node is just P(F). The other three nodes are described by conditional probabilities.

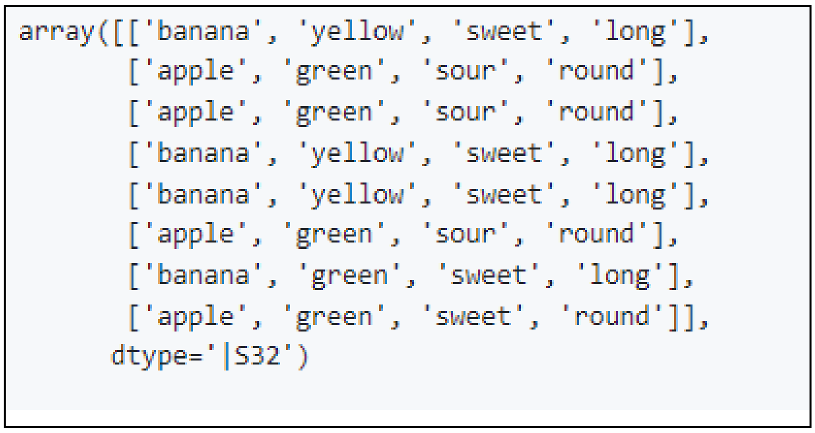
```
Fruit = DiscreteDistribution({'banana':0.4, 'apple':0.6})
Color = ConditionalProbabilityTable([['banana', 'yellow', 0.6],
                        ['banana', 'green', 0.4],
                        ['apple', 'yellow', 0.6],
                        ['apple', 'green', 0.4]], [Fruit] )
Taste = ConditionalProbabilityTable([['banana', 'sweet', 0.6],
                        ['banana', 'sour', 0.4],
                        ['apple', 'sweet', 0.4],
                        ['apple', 'sour', 0.6]], [Fruit])
Shape = ConditionalProbabilityTable([['banana', 'long', 0.6],
                        ['banana', 'round', 0.4],
                        ['apple', 'long', 0.4],
                        ['apple', 'round', 0.6]], [Fruit])
```

Create the state (node) objects and BayesianNetwork object.

```
s_fruit = State(Fruit, 'fruit')
s_color = State(Color, 'color')
s_taste = State(Taste, 'taste')
s_shape = State(Shape, 'shape')
model = BayesianNetwork('fruit')
```

Add states and edges to the network

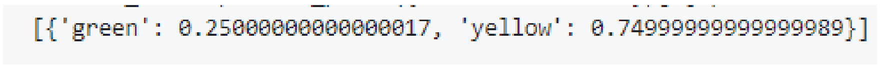
```
model.add_states(s_fruit, s_color, s_taste, s_shape)
model.add_transition(s_fruit, s_color)
model.add_transition(s_fruit, s_taste)
model.add_transition(s_fruit, s_shape)
model.bake()

from bayesnet_em import *
hidden_node_index = 0
new_data = em_bayesnet(model, data, hidden_node_index)
new_data
```



```
array([['banana', 'yellow', 'sweet', 'long'],
       ['apple', 'green', 'sour', 'round'],
       ['apple', 'green', 'sour', 'round'],
       ['banana', 'yellow', 'sweet', 'long'],
       ['banana', 'yellow', 'sweet', 'long'],
       ['apple', 'green', 'sour', 'round'],
       ['banana', 'green', 'sweet', 'long'],
       ['apple', 'green', 'sweet', 'round']],
      dtype='|S32')
```

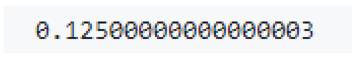
new_model = model.fit(new_data)

new_model.predict_proba({'fruit':'banana'})[1].parameters



```
[{'green': 0.25000000000000017, 'yellow': 0.74999999999999989}]
```

new_model.probability(['apple', 'green', 'sweet', 'round'])



```
0.12500000000000003
```

Result

Thus the EM for Bayesian Network is implemented and executed successfully using Python Programming.

Ex.No: 11 **Build simple NN models**

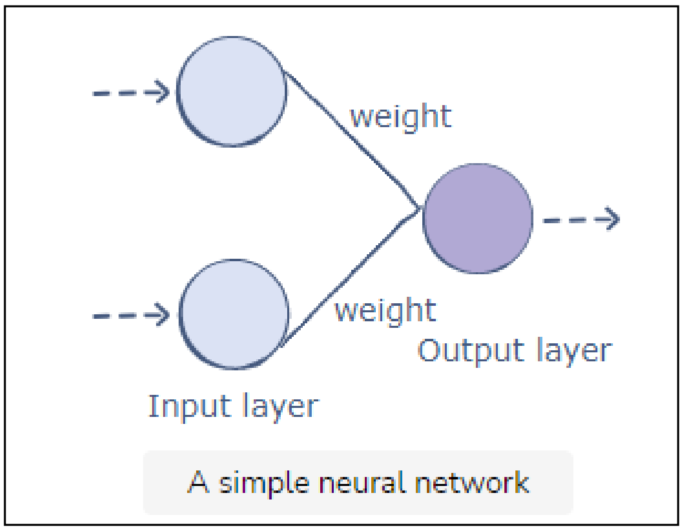
Much like the human brain, a simple neural network consists of interconnected neurons transferring information to each other. Each neuron multiplies its input with its weight(s), applies the activation function on the result, and passes its output on to other neurons. With the help of examples in the training process, a neural network adjusts its weights such that it correctly classifies an unseen input.

A neural network consists of three main layers:

- **input layer**: the initial layer of the network which takes in an input.
- **hidden layer(s)**: the middle optional layer(s) needed for complex tasks.
- **output layer**: the final layer of the network which gives the output.

The following are some important functions that will be used in the implementation:

1. activation function: $1/(1 + e^{-x})$
2. error function: $(target - output)2/2$
3. derivate of the error function: $(target{-}output)$
4. partial derivative of the activation function: $output * (1{-}output)$



Ex.No: 11 **Build simple NN models**

Date:

Aim:

To build simple Neuron Network using Python Programming

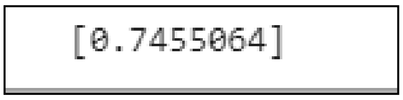
Algorithm

Step 1: Start
Step 2: Initialize features and labels
Step 3: Assign weight for features
Step 4: Import all required packages to perform Neural Computation
Step 5: Display the result
Step 6: Stop

Source code - Install Packages

```
# importing dependancies
import numpy as np

# The activation function
def activation(x):
    return 1 / (1 + np.exp(-x))

# A 2 x 1 matrix of randomly generated weights in the range -1 to 1
weights = np.random.uniform(-1,1,size = (2, 1))

# The training set divided into input and output. Notice that
# we are trying to train our neural network to predict the output
# of the logical OR.
training_inputs = np.array([[0, 0, 1, 1, 0, 1]]).reshape(3, 2)
training_outputs = np.array([[0, 1, 1]]).reshape(3,1)

for i in range(15000):
    # forward pass
    dot_product = np.dot(training_inputs, weights)
    output = activation(dot_product)
    # backward pass.
    temp2 = -(training_outputs - output) * output * (1 - output)
    adj = np.dot(training_inputs.transpose(), temp2)
    # 0.5 is the learning rate.
    weights = weights - 0.5 * adj

# The testing set
test_input = np.array([1, 0])
test_output = activation(np.dot(test_input, weights))
# OR of 1, 0 is 1
print(test_output)
```

Output



```
[0.7455064]
```

Result

Thus the simple Neural Network is implemented and executed successfully using Python Programming.

Ex.No: 12 **Build deep learning NN models**

An artificial Neural Network is a sub-field of Artificial Intelligence compiled under Deep Learning Neural Networks which attempt to mimic the network of neurons that makes the human brain which allows them to understand and respond like a human.

Neural Network consists of a larger set of neurons, which are termed units arranged in layers. In simple words, Neural Network is designed to perform a more complex task where Machine Learning algorithms do not find their use and fail to achieve the required performance.

Neural Networks are used to perform many complex tasks including Image Classification, Object Detection, Face Identification, Text Summarization, speech recognition, and the list is endless.

How neural networks learn complex features? A neural network has many layers and each layer performs a specific function and complex the network. The more the layers are more performance is received. That’s why the neural network is also called a multi-layer perceptron.

Introduction

An artificial Neural Network is a sub-field of Artificial Intelligence compiled under Deep Learning Neural Networks which attempt to mimic the network of neurons that makes the human brain which allows them to understand and respond like a human.

Brief Overview of Neural Network

Neural Network consists of a larger set of neurons, which are termed units arranged in layers. In simple words, Neural Network is designed to perform a more complex task where Machine Learning algorithms do not find their use and fail to achieve the required performance.

Neural Networks are used to perform many complex tasks including Image Classification, Object Detection, Face Identification, Text Summarization, speech recognition, and the list is endless.

How neural networks learn complex features? A neural network has many layers and each layer performs a specific function and complex the network. The more the layers are more performance is received. That’s why the neural network is also called a multi-layer perceptron.

Introduction to Kears Library

Keras is a fast, open-source, and easy-to-use Neural Network Library written in Python that runs at top of Theano or Tensorflow. Tensorflow provides low-level as well as high-level API, indeed Keras only provide High-level API.

As a beginner, it is recommended to work with Keras first and then move to TensorFlow. The reason is using Tensorflow functions as a beginner is a little bit complex to understand and interpret but Keras functionality is simple.

Ex.No: 12 **Build deep learning NN models to predict Diabetes**

Date:

Aim:

To build deep Neuron Network to predict diabetes using Python Programming

Algorithm

Step 1: Start
Step 2: Import keras
Step 3: Initialize features and labels
Step 4: Initialize weights, bias values, optimizers, and hidden layers
Step 4: Import all required packages to perform Neural computation
Step 5: Display the result
Step 6: Stop

Source code - Import Packages and load dataset

```
import pandas as pd
from google.colab import drive
drive.mount('/content/drive')
data = pd.read_csv("/content/drive/My Drive/diabetes.csv")
data.head()
x = data.drop("Outcome", axis=1)
y = data["Outcome"]
```

Define Keras Model

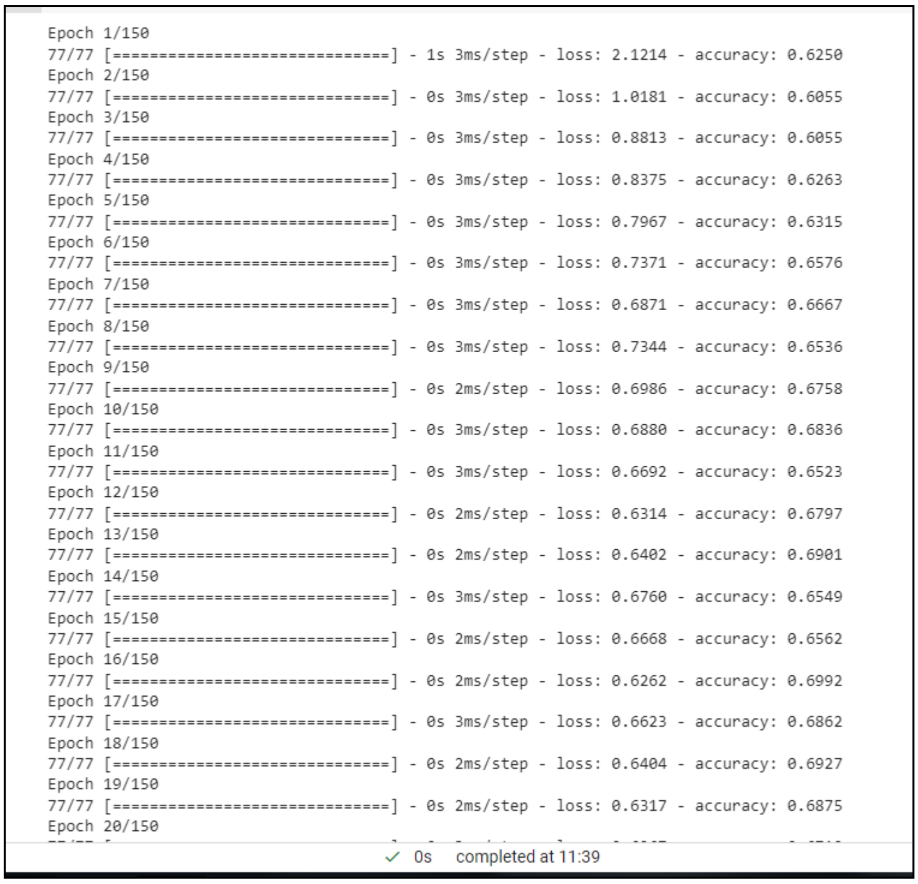
```
from keras.models import Sequential
from keras.layers import Dense
model = Sequential()
model.add(Dense(12, input_dim=8, activation="relu"))
model.add(Dense(12, activation="relu"))
model.add(Dense(1, activation="sigmoid"))
```

Compile The Keras Model

```
model.compile(loss="binary_crossentropy", optimizer="adam", metrics=["accuracy"])
```

Start Training (Fit the Model)

model.fit(x,y, epochs=150, batch_size=10)

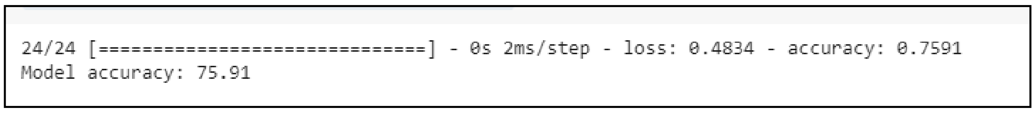


```
Epoch 1/150
77/77 [==============================] - 1s 3ms/step - loss: 2.1214 - accuracy: 0.6250
Epoch 2/150
77/77 [==============================] - 0s 3ms/step - loss: 1.0181 - accuracy: 0.6055
Epoch 3/150
77/77 [==============================] - 0s 3ms/step - loss: 0.8813 - accuracy: 0.6055
Epoch 4/150
77/77 [==============================] - 0s 3ms/step - loss: 0.8375 - accuracy: 0.6263
Epoch 5/150
77/77 [==============================] - 0s 3ms/step - loss: 0.7967 - accuracy: 0.6315
Epoch 6/150
77/77 [==============================] - 0s 3ms/step - loss: 0.7371 - accuracy: 0.6576
Epoch 7/150
77/77 [==============================] - 0s 3ms/step - loss: 0.6871 - accuracy: 0.6667
Epoch 8/150
77/77 [==============================] - 0s 3ms/step - loss: 0.7344 - accuracy: 0.6536
Epoch 9/150
77/77 [==============================] - 0s 2ms/step - loss: 0.6986 - accuracy: 0.6758
Epoch 10/150
77/77 [==============================] - 0s 3ms/step - loss: 0.6880 - accuracy: 0.6836
Epoch 11/150
77/77 [==============================] - 0s 3ms/step - loss: 0.6692 - accuracy: 0.6523
Epoch 12/150
77/77 [==============================] - 0s 2ms/step - loss: 0.6314 - accuracy: 0.6797
Epoch 13/150
77/77 [==============================] - 0s 2ms/step - loss: 0.6402 - accuracy: 0.6901
Epoch 14/150
77/77 [==============================] - 0s 3ms/step - loss: 0.6760 - accuracy: 0.6549
Epoch 15/150
77/77 [==============================] - 0s 2ms/step - loss: 0.6668 - accuracy: 0.6562
Epoch 16/150
77/77 [==============================] - 0s 2ms/step - loss: 0.6262 - accuracy: 0.6992
Epoch 17/150
77/77 [==============================] - 0s 3ms/step - loss: 0.6623 - accuracy: 0.6862
Epoch 18/150
77/77 [==============================] - 0s 2ms/step - loss: 0.6404 - accuracy: 0.6927
Epoch 19/150
77/77 [==============================] - 0s 2ms/step - loss: 0.6317 - accuracy: 0.6875
Epoch 20/150
```

Evaluate the Model

_, accuracy = model.evaluate(x, y)

print("Model accuracy: %.2f"% (accuracy*100))

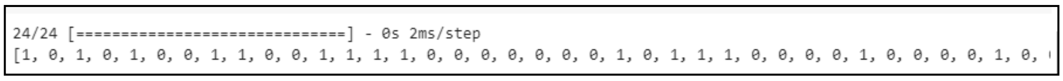


```
24/24 [==============================] - 0s 2ms/step - loss: 0.4834 - accuracy: 0.7591
Model accuracy: 75.91
```

Making Predictions

predictions = model.predict(x)

print([round(x[0]) for x in predictions])



```
24/24 [==============================] - 0s 2ms/step
[1, 0, 1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0,
```

Result

Thus the deep Neural Network to predict diabetes based on given features is implemented and executed successfully using Python Programming.