**EX No: 1A Uninformed search algorithm BFS**

**Aim:**

To implement Breadth first search algorithm in python**.**

**Algorithm:**

1. **Start:** Insert the starting node into a queue. Mark the starting node as visited.
2. **Explore:** While the queue is not empty:
   * Remove a node from the queue and print its value
   * Insert all unvisited adjacent nodes of the removed node into the queue. Mark each adjacent node as visited to avoid revisiting.
3. **End:** Repeat the process until the queue is empty.

**Program:**

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 program to implement BFS algorithm is successfully executed and verified.

**EX No: 1B Uninformed search algorithm DFS**

**Aim:**

To implement Depth First Search algorithm in python**.**

**Algorithm:**

1. Start by putting any one of the graph's vertices on top of a stack.
2. Take the top item of the stack 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 top of the stack.
4. Keep repeating steps 2 and 3 until the stack is empty.

Program:

# DFS algorithm in Python

# DFS algorithm

def dfs(graph, start, visited=None):

if visited is None:

visited = set()

visited.add(start)

print(start)

for next in graph[start] - visited:

dfs(graph, next, visited)

return visited

graph = {'0': set(['1', '2']),

'1': set(['0', '3', '4']),

'2': set(['0']),

'3': set(['1']),

'4': set(['2', '3'])}

dfs(graph, '0')

**Output: 0 2 1 3 4**

**Result:**

Thus the program to implement DFS algorithm is successfully executed and verified.

**EX No: 2A Implementation of Informed search (A\*) algorithm**

**Aim:**

**To implement Informed search algorithm (A\*)**

**Algorithm:**

**Step 1: Add the beginning node to the open list**  
**Step 2: Repeat the following step**

* In the open list, find the square with the lowest F cost, which denotes the current square. Now we move to the closed square.
* Consider 8 squares adjacent to the current square and Ignore it if it is on the closed list or if it is not workable. Do the following if it is workable.
* Check if it is on the open list; if not, add it. You need to make the current square as this square’s a parent. You will now record the different costs of the square, like the F, G, and H costs.
  + If it is on the open list, use G cost to measure the better path. The lower the G cost, the better the path. If this path is better, make the current square as the parent square. Now you need to recalculate the other scores – the G and F scores of this square.**You’ll stop:**
* If you find the path, you need to check the closed list and add the target square to it.
* There is no path if the open list is empty and you cannot find the target square.

**Step 3. Now you can save the path and work backward, starting from the target square, going to the parent square from each square you go, till it takes you to the starting square. You’ve found your path now.**

**Program:**

def aStarAlgo(start\_node, stop\_node):

open\_set = set(start\_node)

closed\_set = set()

g = {} #store distance from starting node

parents = {}# parents contains an adjacency map of all nodes

#ditance of starting node from itself is zero

g[start\_node] = 0

#start\_node is root node i.e it has no parent nodes

#so start\_node is set to its own parent node

parents[start\_node] = start\_node

while len(open\_set) > 0:

n = None

#node with lowest f() is found

for v in open\_set:

if n == None or g[v] + heuristic(v) < g[n] + heuristic(n):

n = v

if n == stop\_node or Graph\_nodes[n] == None:

pass

else:

for (m, weight) in get\_neighbors(n):

#nodes 'm' not in first and last set are added to first

#n is set its parent

if m not in open\_set and m not in closed\_set:

open\_set.add(m)

parents[m] = n

g[m] = g[n] + weight

#for each node m,compare its distance from start i.e g(m) to the

#from start through n node

else:

if g[m] > g[n] + weight:

#update g(m)

g[m] = g[n] + weight

#change parent of m to n

parents[m] = n

#if m in closed set,remove and add to open

if m in closed\_set:

closed\_set.remove(m)

open\_set.add(m)

if n == None:

print('Path does not exist!')

return None

# if the current node is the stop\_node

# then we begin reconstructin the path from it to the start\_node

if n == stop\_node:

path = []

while parents[n] != n:

path.append(n)

n = parents[n]

path.append(start\_node)

path.reverse()

print('Path found: {}'.format(path))

return path

# remove n from the open\_list, and add it to closed\_list

# because all of his neighbors were inspected

open\_set.remove(n)

closed\_set.add(n)

print('Path does not exist!')

return None

#define fuction to return neighbor and its distance

#from the passed node

def get\_neighbors(v):

if v in Graph\_nodes:

return Graph\_nodes[v]

else:

return None

#for simplicity we ll consider heuristic distances given

#and this function returns heuristic distance for all nodes

def heuristic(n):

H\_dist = {

'A': 11,

'B': 6,

'C': 99,

'D': 1,

'E': 7,

'G': 0,

}

return H\_dist[n]

#Describe your graph here

Graph\_nodes = {

'A': [('B', 2), ('E', 3)],

'B': [('C', 1),('G', 9)],

'C': None,

'E': [('D', 6)],

'D': [('G', 1)],

}

aStarAlgo('A', 'G')

**Output:** **Path found: ['A', 'E', 'D', 'G']**

**Result: Thus the Informed search algorithm (A\*) is executed and verified successfully.**

**EX No: 2B Implementation of Informed search (memory-bounded A\*) algorithm**

**Aim:**

**To implement memory-bounded A\* algorithm in python**

**Algorithm:**

Step 1: Initialize Memory Bounded Search function in this that accepts an issue and a memory limit as input.

Step 2: The nodes to be investigated and the nodes that have already been explored are stored in the open and closed lists, respectively, which are initialized.

Step 3: The algorithm then goes into a loop where it keeps track of whether memory use goes above the predetermined threshold.

Step 4: If the memory is utilized beyond threshold ,then memory is freed up by pruning the open and closed lists.

Step 5: Based on a heuristic assessment, the select\_best\_node function selects the most promising node from the open list.

Step 6: The answer is sent back if this node is in the desired state. If not, the node is transferred from the open list to the closed list, and if its heirs have previously been investigated or are already on the open list, they are created and added to the open list.

**Program:**

# Define a class to represent nodes in the search tree

class Node:

def \_\_init\_\_(self, state, parent=None, action=None):

self.state = state

self.parent = parent

self.action = action

# Define the heuristic function (Manhattan distance)

def heuristic(state):

distance = 0

for i in range(9):

if state[i] != 0:

distance += abs(i // 3 - (state[i] - 1) // 3) + abs(i % 3 - (state[i] - 1) % 3)

return distance

# Define the memory usage function

def memory\_usage(open\_list, closed\_list):

return len(open\_list) + len(closed\_list)

# Define the function to prune memory

def prune\_memory(open\_list, closed\_list):

# Prune the least promising nodes from the open list

open\_list.sort(key=lambda x: heuristic(x.state), reverse=True)

open\_list[:] = open\_list[:len(open\_list) // 2] # Keep only the top half of the open list

# Define the function to select the best node

def select\_best\_node(open\_list):

return min(open\_list, key=lambda x: heuristic(x.state))

# Define the function to check if a node is the goal state

def is\_goal(node):

return node.state == goal\_state

# Define the function to generate successors

def generate\_successors(node):

successors = []

zero\_index = node.state.index(0)

for move in moves[zero\_index]:

new\_state = list(node.state)

new\_state[zero\_index], new\_state[move] = new\_state[move], new\_state[zero\_index]

successors.append(Node(tuple(new\_state), parent=node, action=move))

return successors

# Define the function to check if a successor is redundant

def redundant(successor, open\_list, closed\_list):

for node in open\_list + closed\_list:

if node.state == successor.state:

return True

return False

# Define the memory-bounded search function

def MemoryBoundedSearch(initial\_state, memory\_limit):

node = Node(initial\_state)

open\_list = [node]

closed\_list = []

while open\_list:

if memory\_usage(open\_list, closed\_list) > memory\_limit:

prune\_memory(open\_list, closed\_list)

# No solution found within memory limit

if not open\_list:

return None

current\_node = select\_best\_node(open\_list)

# Return the goal node

if is\_goal(current\_node):

return current\_node

open\_list.remove(current\_node)

closed\_list.append(current\_node)

for successor in generate\_successors(current\_node):

if not redundant(successor, open\_list, closed\_list):

open\_list.append(successor)

# No solution found within memory limit

return None

# Define the goal state

goal\_state = (1, 2, 3, 4, 5, 6, 7, 8, 0)

# Define the possible moves

moves = {

0: [1, 3],

1: [0, 2, 4],

2: [1, 5],

3: [0, 4, 6],

4: [1, 3, 5, 7],

5: [2, 4, 8],

6: [3, 7],

7: [4, 6, 8],

8: [5, 7]

}

# Example usage

initial\_state = (1, 2, 3, 4, 5, 6, 0, 7, 8) # Initial state of the puzzle

print("Case 1 with Memory Limit 1")

memory\_limit = 1 # Set memory limit

goal\_node = MemoryBoundedSearch(initial\_state, memory\_limit)

if goal\_node:

print("Solution found!")

# Print the solution path if needed

while goal\_node.parent:

print("Action:", goal\_node.action)

print("State:")

print(goal\_node.state[:3])

print(goal\_node.state[3:6])

print(goal\_node.state[6:])

print()

goal\_node = goal\_node.parent

else:

print("Memory limit exceeded. No solution found within the given memory limit.")

print("\nCase 1 with Memory Limit 10")

memory\_limit = 10 # Set memory limit

goal\_node = MemoryBoundedSearch(initial\_state, memory\_limit)

if goal\_node:

print("Solution found!")

# Print the solution path if needed

while goal\_node.parent:

print("Action:", goal\_node.action)

print("State:")

print(goal\_node.state[:3])

print(goal\_node.state[3:6])

print(goal\_node.state[6:])

print()

goal\_node = goal\_node.parent

else:

print("Memory limit exceeded. No solution found within the given memory limit.")

**Output:**

**Case 1 with Memory Limit 1**

**Memory limit exceeded. No solution found within the given memory limit.**

**Case 1 with Memory Limit 10**

**Solution found!**

**Action: 8**

**State:**

**(1, 2, 3)**

**(4, 5, 6)**

**(7, 8, 0)**

**Action: 7**

**State:**

**(1, 2, 3)**

**(4, 5, 6)**

**(7, 0, 8)**

Result:

Thus the program for memory-bounded A\* algorithm in python is implemented and verified successfully.

**EX No: 3a Implement naïve Bayes model (iris dataset)**

**Aim:**

To implement naïve Bayes model in **scikit-learn and report the outcomes for iris dataset.**

**Algorithm**

1. **Import necessary Library files**
2. **Load the iris data set**
3. The data is split into features (X) and target (y).X contains feature information of iris data set (sepal length, sepal width, petal length, petal width) and y contains iris class (Iris setosa, Iris virginica and Iris versicolor) labels
4. The data is split into training and test sets using the train\_test\_split function from the sklearn.model\_selection module.
5. Create an instance of Gaussian Naïve Bayes classfier
6. Fit the classifier to the training data X\_train and y\_train
7. Perform the prediction
8. Estimate the performance metrics such as accuracy, precision and recall for the classifier.

Program:

import pandas as pd

from sklearn import metrics

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import precision\_score

from sklearn.metrics import recall\_score

from sklearn.metrics import confusion\_matrix

from sklearn.naive\_bayes import GaussianNB

from sklearn.datasets import load\_iris

iris = load\_iris()

X = iris.data

y = iris.target

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=.3, random\_state=4)

NBclassifier = GaussianNB()

NBclassifier.fit(X\_train, y\_train)

# performing predictions on the test dataset

y\_pred = NBclassifier.predict(X\_test)

# metrics are used to find accuracy or error

print("ACCURACY OF THE MODEL:", metrics.accuracy\_score(y\_test, y\_pred))

accuracy = metrics.accuracy\_score(y\_test, y\_pred)

precision = metrics.precision\_score(y\_test, y\_pred,average='micro')

recall = metrics.recall\_score(y\_test, y\_pred,average='micro')

from sklearn.metrics import accuracy\_score, classification\_report, confusion\_matrix

class\_report = classification\_report(y\_test, y\_pred)

print("Classification Report:\n", class\_report)

Output

Classification Report:

precision recall f1-score support

0 0.38 1.00 0.55 43

1 0.00 0.00 0.00 71

accuracy 0.38 114

macro avg 0.19 0.50 0.27 114

weighted avg 0.14 0.38 0.21 114

Result:

Thus the program for naïve Bayes model in **scikit-learns and reporting the outcomes for iris dataset is successfully done.**

**EX No: 3b Implement naïve Bayes model (wine dataset)**

**Aim:**

To implement naïve Bayes model in **scikit-learn and report the outcomes for wine dataset.**

**Algorithm**

1. **Import necessary Library files**
2. **Load the wine data set**
3. The data is split into features (X) and target (y).
4. The data is split into training and test sets using the train\_test\_split function from the sklearn.model\_selection module.
5. Create an instance of Gaussian Naïve Bayes classfier
6. Fit the classifier to the training data X\_train and y\_train
7. Perform the prediction
8. Estimate the performance metrics such as accuracy, precision and recall for the classifier.

Program:

import pandas as pd

from sklearn import metrics

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import precision\_score

from sklearn.metrics import recall\_score

from sklearn.metrics import confusion\_matrix

from sklearn.naive\_bayes import GaussianNB

wine=pd.read\_csv('C:/Users/user/Desktop/NMP/6\_SCIKIT\dataset/wine.csv').dropna()

X=wine.drop("Wine",axis=1)

y=wine["Wine"]

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=.3, random\_state=4)

NBclassifier = GaussianNB()

NBclassifier.fit(X\_train, y\_train)

# performing predictions on the test dataset

y\_pred = NBclassifier.predict(X\_test)

# metrics are used to find accuracy or error

print("ACCURACY OF THE MODEL:", metrics.accuracy\_score(y\_test, y\_pred))

accuracy = metrics.accuracy\_score(y\_test, y\_pred)

precision = metrics.precision\_score(y\_test, y\_pred,average='micro')

recall = metrics.recall\_score(y\_test, y\_pred,average='micro')

from sklearn.metrics import accuracy\_score, classification\_report, confusion\_matrix

class\_report = classification\_report(y\_test, y\_pred)

print("Classification Report:\n", class\_report)

Output

Result:

Thus the program for naïve Bayes model in **scikit-learns and reporting the outcomes for wine dataset is successfully done.**

**EX No: 3c Implement naïve Bayes model (penguin dataset)**

**Aim:**

To implement naïve Bayes model in **scikit-learn and report the outcomes for** penguin **dataset.**

**Algorithm**

1. **Import necessary Library files**
2. **Load the** penguin **data set**
3. The data is split into features (X) and target (y).
4. The data is split into training and test sets using the train\_test\_split function from the sklearn.model\_selection module.
5. Create an instance of Gaussian Naïve Bayes classfier
6. Fit the classifier to the training data X\_train and y\_train
7. Perform the prediction
8. Estimate the performance metrics such as accuracy, precision and recall for the classifier.

Program:

import pandas as pd

from sklearn import metrics

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import precision\_score

from sklearn.metrics import recall\_score

from sklearn.metrics import confusion\_matrix

from sklearn.naive\_bayes import GaussianNB

penguins = pd.read\_csv('C:/Users/user/Desktop/NMP/6\_SCIKIT\dataset/penguins\_size.csv').dropna()

penguins

df = pd.DataFrame(penguins)

X = penguins.select\_dtypes(exclude=['object'])

y = penguins.species

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=.3, random\_state=4)

NBclassifier = GaussianNB()

NBclassifier.fit(X\_train, y\_train)

# performing predictions on the test dataset

y\_pred = NBclassifier.predict(X\_test)

# metrics are used to find accuracy or error

print("ACCURACY OF THE MODEL:", metrics.accuracy\_score(y\_test, y\_pred))

accuracy = metrics.accuracy\_score(y\_test, y\_pred)

precision = metrics.precision\_score(y\_test, y\_pred,average='micro')

recall = metrics.recall\_score(y\_test, y\_pred,average='micro')

from sklearn.metrics import accuracy\_score, classification\_report, confusion\_matrix

class\_report = classification\_report(y\_test, y\_pred)

print("Classification Report:\n", class\_report)

Output

Result:

Thus the program for naïve Bayes model in **scikit-learns and reporting the outcomes for penguin dataset is successfully done.**

**EX No: 4 Implement Bayesian Networks**

**Aim:**

To implement **Bayesian Networks** model using pgmpy **and report the outcomes for covid patient dataset.**

**Algorithm:**

1. Install pgmpy package using pip
2. Import pandas and BayesianNetwork library files
3. Load the covid patient data set
4. Create a data set that contains crucial features ('SR\_2 Folder thickness (mm) SR\_3 Folder thickness(mm) SR\_4 Folder thickness(mm),COVID-19 Infection)
5. Rename the column names.
6. Perform the CPT for the features
7. Perform VariableElimination of the model
8. Perform the query for infection when the evidence for a feature is certain value.
9. Print the result

Code:

pip install pgmpy

import pandas as pd

from pgmpy.models import BayesianNetwork

df = pd.read\_csv('C:/Users/user/Desktop/NMP/AIML\_LAB\_CSE/EXPERIMENTS/Patient\_details.csv')

df1=df[['SR\_2 Folder thickness(mm)', 'SR\_3 Folder thickness(mm)','SR\_4 Folder thickness(mm)','COVID-19 Infection']]

dict = {'SR\_2 Folder thickness(mm)': 'SR\_2','SR\_3 Folder thickness(mm)': 'SR\_3','SR\_4 Folder thickness(mm)': 'SR\_4','COVID-19 Infection': 'Infection'}

df1.rename(columns=dict,inplace=True)

result= {"Positive": 1,"Negative":0}

df1['Infection']=df1['Infection'].map(result)

df1.isnull()

df1.dropna

model=BayesianNetwork([("SR\_2", "SR\_3"),("SR\_3", "SR\_4"),("SR\_2", "Infection"),("SR\_3", "Infection"),("SR\_4", "Infection")])

model.fit(df1)

from pgmpy.inference import VariableElimination

infer=VariableElimination(model)

q1=infer.query(variables=['Infection'],evidence={'SR\_2':4})

print(q1)

**Output:**

**Result:**

Thus the program for Bayesian Networks model using pgmpy and report the outcomes for covid patient dataset are successfully done.

**EX No: 5** Build Regression model

**Aim:**

To implement Regression model in stats model a**nd print regression line for a given data set.**

**Algorithm**

1. **Import necessary library files**
2. **Create a data set that contains runs scored by a team against overs**
3. **Load the data set**
4. **Select ordinary least squares model to perform regression and fit the model for the data.**
5. **Print the summary**
6. **Import ploty package**
7. **Render the outcome in the browser and figure out the regression line.**

**Program**

# import packages

import numpy as np

import pandas as pd

import statsmodels.formula.api as smf

# import packages

import numpy as np

import pandas as pd

import statsmodels.formula.api as smf

# loading the csv file

df = pd.read\_csv('C:/Users/user/Desktop/NMP/AIML\_LAB\_IT/5\_REGRESSION/runs.csv')

print(df.head())

# fitting the model

df.columns = ['Overs', 'Runs']

model = smf.ols(formula='Overs ~ Runs', data=df).fit()

# model summary

print(model.summary())

import plotly.express as px

fig = px.scatter(df, x="Overs", y="Runs", trendline="ols")

fig.show()

fig.show(renderer="browser")

**Output:**

**Graphical Output:**

**Result:**

Thus the program to implement Regression model in stats model a**nd regression line for a given data set is done successfully.**

**EX No: 6 A Decision tree**

**Aim:**

To implement **Decision tree** in **scikit-learn and report the outcomes for iris dataset.**

**Algorithm**

1. **Import necessary Library files**
2. **Load the iris data set**
3. The data is split into features (X) and target (y).X contains feature information of iris data set (sepal length, sepal width, petal length, petal width) and y contains iris class (Iris setosa, Iris virginica and Iris versicolor) labels
4. The data is split into training and test sets using the train\_test\_split function from the sklearn.model\_selection module.
5. Create an instance of Decision Tree Classifier
6. Fit the classifier to the training data X\_train and y\_train
7. Perform the prediction
8. Estimate the performance metrics such as accuracy, precision and recall for the classifier.

**Program**

#Decision TREE

import pandas as pd

from sklearn import metrics

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import precision\_score

from sklearn.metrics import recall\_score

from sklearn.metrics import confusion\_matrix

from sklearn.datasets import load\_iris

from sklearn.tree import DecisionTreeClassifier

iris = load\_iris()

X = iris.data

y = iris.target

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=.3, random\_state=4)

# Create Decision Tree classifer object

clf = DecisionTreeClassifier()

# Train Decision Tree Classifer

clf = clf.fit(X\_train,y\_train)

# performing predictions on the test dataset

y\_pred = clf.predict(X\_test)

# metrics are used to find accuracy or error

print("ACCURACY OF THE MODEL:", metrics.accuracy\_score(y\_test, y\_pred))

accuracy = metrics.accuracy\_score(y\_test, y\_pred)

precision = metrics.precision\_score(y\_test, y\_pred,average='micro')

recall = metrics.recall\_score(y\_test, y\_pred,average='micro')

from sklearn.metrics import accuracy\_score, classification\_report, confusion\_matrix

class\_report = classification\_report(y\_test, y\_pred)

print("Classification Report:\n", class\_report)

**Output:**

**Result:**

Thus the program for Decision tree model in **scikit learn and performance outcomes for iris dataset are successfully done.**

**EX No: 6 B Random Forest**

**Aim:**

To implement **Random Forest** in **scikit-learn and report the outcomes for iris dataset.**

**Algorithm**

1. **Import necessary Library files**
2. **Load the iris data set**
3. The data is split into features (X) and target (y).X contains feature information of iris data set (sepal length, sepal width, petal length, petal width) and y contains iris class (Iris setosa, Iris virginica and Iris versicolor) labels
4. The data is split into training and test sets using the train\_test\_split function from the sklearn.model\_selection module.
5. Create an instance of Random Forest Classifier
6. Fit the classifier to the training data X\_train and y\_train
7. Perform the prediction
8. Estimate the performance metrics such as accuracy, precision and recall for the classifier.

**Program**

from sklearn.datasets import load\_iris

iris = load\_iris()

X = iris.data

y = iris.target

from sklearn.model\_selection import train\_test\_split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.4, random\_state=1)

from sklearn.ensemble import RandomForestClassifier

import pandas as pd

clf = RandomForestClassifier(n\_estimators = 100)

clf.fit(X\_train, y\_train)

# performing predictions on the test dataset

y\_pred = clf.predict(X\_test)

# metrics are used to find accuracy or error

from sklearn import metrics

print()

# using metrics module for accuracy calculation

print("ACCURACY OF THE MODEL:", metrics.accuracy\_score(y\_test, y\_pred))

clf.predict([[3, 3, 2, 2]])

from sklearn.metrics import accuracy\_score, classification\_report, confusion\_matrix

class\_report = classification\_report(y\_test, y\_pred)

print("Classification Report:\n", class\_report)

**Output:**

**Result:**

Thus the program for Random Forest model in **scikit learn and performance outcomes for iris dataset are successfully done.**

**EX No: 7 SVM**

**Aim:**

To implement **Support Vector Machine (SVM)** in **scikit-learn and report the outcomes for iris dataset.**

**Algorithm**

1. **Import necessary Library files**
2. **Load the iris data set**
3. The data is split into features (X) and target (y).X contains feature information of iris data set (sepal length, sepal width, petal length, petal width) and y contains iris class (Iris setosa, Iris virginica and Iris versicolor) labels
4. The data is split into training and test sets using the train\_test\_split function from the sklearn.model\_selection module.
5. Create an instance of Support Vector Machine Classifier
6. Fit the classifier to the training data X\_train and y\_train
7. Perform the prediction
8. Estimate the performance metrics such as accuracy, precision and recall for the classifier.

**Program**

from sklearn.datasets import load\_iris

iris = load\_iris()

X = iris.data

y = iris.target

from sklearn.model\_selection import train\_test\_split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.4, random\_state=1)

#Import svm model

from sklearn import svm

#Create a svm Classifier

clf = svm.SVC(kernel='linear') # Linear Kernel

#Train the model using the training sets

clf.fit(X\_train, y\_train)

#Predict the response for test dataset

y\_pred = clf.predict(X\_test)

#Import scikit-learn metrics module for accuracy calculation

from sklearn import metrics

# Model Accuracy: how often is the classifier correct?

print("Accuracy:",metrics.accuracy\_score(y\_test, y\_pred))

print("Precision:",metrics.precision\_score(y\_test, y\_pred,average='macro'))

print("Recall:",metrics.recall\_score(y\_test, y\_pred,average='macro'))

from sklearn.metrics import accuracy\_score, classification\_report, confusion\_matrix

class\_report = classification\_report(y\_test, y\_pred)

print("Classification Report:\n", class\_report)

**Output:**

**Result:**

Thus the program for Support Vector Machine (SVM) model in **scikit learn and performance outcomes for iris dataset are successfully done.**

**EX No: 8 Implement Ensembling technique- Bagging**

**Aim:**

To implement **Ensembling technique- Bagging** in **scikit-learn and report the outcomes for digits dataset.**

**Algorithm**

1. **Import necessary Library files**
2. **Load the digits data set**
3. The data is split into features (X) and target (y).
4. The data is split into training and test sets using the train\_test\_split function from the sklearn.model\_selection module.
5. Create an instance of Bagging Classifier for Decision Tree Classifier.
6. Fit the classifier to the training data X\_train and y\_train
7. Perform the prediction
8. Estimate the accuracy of the model.

**Program**

from sklearn.ensemble import BaggingClassifier

from sklearn.datasets import make\_classification

from sklearn.datasets import load\_digits

from sklearn.tree import DecisionTreeClassifier

from sklearn import metrics

# Load the dataset

digit = load\_digits()

X, y = digit.data, digit.target

# Split the data into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Create the base classifier

dc = DecisionTreeClassifier()

model = BaggingClassifier(estimator=dc, n\_estimators=10)

classifiers = model.fit(X\_train, y\_train)

# Make predictions on the test set

y\_pred = model.predict(X\_test)

# Calculate accuracy

print("Accuracy:", metrics.accuracy\_score(y\_test, y\_pred))

from sklearn.metrics import accuracy\_score, classification\_report, confusion\_matrix

class\_report = classification\_report(y\_test, y\_pred)

print("Classification Report:\n", class\_report)

**Output:**

**Result:**

Thus the program for **Ensembling technique - Bagging** in **scikit-learn and report the outcomes for digits dataset** **are successfully done.**

**EX No: 9 KMeans Clustering**

**Aim:**

To implement **KMeans Clustering** in **scikit-learn and report the outcomes for iris dataset.**

**Algorithm**

1. **Import necessary Library files**
2. **Load the iris data set**
3. Create a new data frame by dropping id and species from the original data frame. In other words the created data frame will hold the feature information (sepal length, sepal width, petal length, petal width) of iris.
4. Perform standard scaling operation.
5. Create an instance of Kmeans Classifier and fit the data.
6. Using elbow method, determine the optimum number clusters that can be formed from the iris data set.

**Program**

import pandas as pd

df=pd.read\_csv('C:/Users/user/Desktop/NMP/AIML\_LAB\_IT/DATASET/Iris.csv')

df = df.drop(["Id", "Species"], axis="columns")

from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

df = scaler.fit\_transform(df)

from sklearn.cluster import KMeans

kmeans = KMeans(n\_clusters = 4, random\_state = 0, n\_init='auto')

kmeans.fit(df)

from sklearn.metrics import silhouette\_score

K = range(2, 8)

fits = []

score = []

import seaborn as sns

for k in K:

# train the model for current value of k on training data

model = KMeans(n\_clusters = k, random\_state = 0, n\_init='auto').fit(df)

# append the model to fits

fits.append(model)

# Append the silhouette score to scores

score.append(silhouette\_score(df, model.labels\_, metric='euclidean'))

sns.lineplot(x=K, y= score).set(title='Elbow\_Method', xlabel='No\_Of\_Clusters', ylabel='Score')

**Output:**

Result:

Thus the program for K-Means clustering in scikit learn is executed and verified successfully.

**EX No: 10 Implement EM for Bayesian networks**

**Aim:**

**To implement the program for expectation maximization for Bayesian networks**

**Overview:**

EM is an iterative algorithm commonly used for estimation in the case when there are latent variables in the model. The algorithm iteratively improves the parameter estimates maximizing the likelihood of the given data.

Parameters:

* model (A pgmpy.models.BayesianNetwork instance)
* data (pandas DataFrame object) – DataFrame object with column names identical to the variable names of the network. (If some values in the data are missing the data cells should be set to numpy.NaN. Note that pandas converts each column containing numpy.NaN`s to dtype `float.)
* state\_names ([dict](https://docs.python.org/3/library/stdtypes.html#dict) (optional)) – A dict indicating, for each variable, the discrete set of states that the variable can take. If unspecified, the observed values in the data set are taken to be the only possible states.

get\_parameters(latent\_card=None, max\_iter=100, atol=1e-08, n\_jobs=1, batch\_size=1000, seed=None, init\_cpds={}, show\_progress=True)

Method to estimate all model parameters (CPDs) using Expecation Maximization.

Parameters:

* latent\_card ([*dict*](https://docs.python.org/3/library/stdtypes.html#dict) *(default: None)*) – A dictionary of the form {latent\_var: cardinality} specifying the cardinality (number of states) of each latent variable. If None, assumes *2* states for each latent variable.
* max\_iter ([*int*](https://docs.python.org/3/library/functions.html#int) *(default: 100)*) – The maximum number of iterations the algorithm is allowed to run for. If max\_iter is reached, return the last value of parameters.
* atol ([*int*](https://docs.python.org/3/library/functions.html#int) *(default: 1e-08)*) – The absolute accepted tolerance for checking convergence. If the parameters change is less than atol in an iteration, the algorithm will exit.
* n\_jobs ([*int*](https://docs.python.org/3/library/functions.html#int) *(default: 1)*) – Number of jobs to run in parallel. Using n\_jobs > 1 for small models or datasets might be slower.
* batch\_size ([*int*](https://docs.python.org/3/library/functions.html#int) *(default: 1000)*) – Number of data used to compute weights in a batch.
* seed ([*int*](https://docs.python.org/3/library/functions.html#int)) – The random seed to use for generating the intial values.
* init\_cpds ([*dict*](https://docs.python.org/3/library/stdtypes.html#dict)) – A dictionary of the form {variable: instance of TabularCPD} specifying the initial CPD values for the EM optimizer to start with. If not specified, CPDs involving latent variables are initialized randomly, and CPDs involving only observed variables are initialized with their MLE estimates.
* show\_progress (*boolean (default: True)*) – Whether to show a progress bar for iterations.

Returns : Estimated parameters (CPDs) – A list of estimated CPDs for the model.

Return type: list

Algorithm:

1. Load required libraries
2. Load the data
3. Define Bayesian network
4. Estimate all model parameters (CPDs) using Expectation Maximization(EM)
5. Get the list

Program

import numpy as np

import pandas as pd

from pgmpy.models import BayesianNetwork

from pgmpy.estimators import ExpectationMaximization

data = pd.DataFrame(np.random.randint(low=0, high=2, size=(1000, 5)), columns=['A', 'B', 'C', 'D', 'E'])

model = BayesianNetwork([('A', 'B'), ('C', 'B'), ('C', 'D'), ('B', 'E')])

estimator = ExpectationMaximization(model, data)

estimator.get\_parameters(latent\_card={'B': 2})

Output:

WARNING:pgmpy:Replacing existing CPD for B

WARNING:pgmpy:Replacing existing CPD for D

WARNING:pgmpy:Replacing existing CPD for E

WARNING:pgmpy:Replacing existing CPD for A

WARNING:pgmpy:Replacing existing CPD for C

0%| | 0/100 [00:00<?, ?it/s]

Out[3]:

[<TabularCPD representing P(B:2 | A:2, C:2) at 0x21ab7a14ca0>,

<TabularCPD representing P(D:2 | C:2) at 0x21ab7a149d0>,

<TabularCPD representing P(E:2 | B:2) at 0x21ab7994af0>,

<TabularCPD representing P(A:2) at 0x21ab7994340>,

<TabularCPD representing P(C:2) at 0x21ab79943d0>]

Result:

Thus the program for EM for Bayesian networks is successfully implemented and verified.

**EX No: 11 Implement simple NN models**

**Aim:**

To implement simple NN models in **scikit-learn and report the outcomes for** breast cancer **dataset.**

**Algorithm**

1. **Import necessary Library files**
2. **Load the** breast cancer **data set**
3. The data is split into features (X) and target (y).
4. The data is split into training and test sets using the train\_test\_split function from the sklearn.model\_selection module.
5. Perform scaling operation of data.
6. Create an instance of Multi layer Multi-layer Perceptron classifier
7. Fit the classifier to the training data X\_train and y\_train
8. Perform the prediction
9. Estimate the performance metrics such as accuracy, precision and recall for the classifier.

**Code:**

# Import necessary libraries

from sklearn.neural\_network import MLPClassifier

from sklearn.datasets import load\_breast\_cancer

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import StandardScaler

from sklearn.metrics import accuracy\_score, classification\_report, confusion\_matrix

# Load the Breast Cancer dataset

cancer\_data = load\_breast\_cancer()

X, y = cancer\_data.data, cancer\_data.target

# Split the dataset into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Standardize features by removing the mean and scaling to unit variance

scaler = StandardScaler()

X\_train = scaler.fit\_transform(X\_train)

X\_test = scaler.transform(X\_test)

# Create an MLPClassifier model

mlp = MLPClassifier(hidden\_layer\_sizes=(64, 32),max\_iter=1000, random\_state=42)

# Train the model on the training data

mlp.fit(X\_train, y\_train)

# Make predictions on the test data

y\_pred = mlp.predict(X\_test)

# Calculate the accuracy of the model

accuracy = accuracy\_score(y\_test, y\_pred)

print(f"Accuracy: {accuracy:.2f}")

#Accuracy: 0.97

# Generate a classification report

class\_report = classification\_report(y\_test, y\_pred)

print("Classification Report:\n", class\_report)

Output:

**Result:**

Thus the program to implement simple neural network model is implemented and verified successfully.

**EX No: 12 Implement Deep learning NN model**

**Aim:**

To implement Deep learning NN models using keras, tensorflow **and report the outcomes for** pimadataset.

**Algorithm**

1. **Import necessary Library files**
2. **Load the** pima **data set**
3. The data is split into features (X) and target (y).The feature Variables (X) contains:

1)Number of times pregnant, 2)Plasma glucose concentration at 2 hours in an oral glucose tolerance test,3) Diastolic blood pressure (mm Hg),4)Triceps skin fold thickness (mm),5) 2-hour serum insulin (mu U/ml),6) Body mass index (weight in kg/(height in m)^2),7)Diabetes pedigree function, 8) Age (years).The target (y) contains Class variable (0 or 1)

1. Define sequential model.
2. Create a dense layer (First input Layer) that accepts 12 input features,12 perceptrons with relu activation function.
3. Create a dense layer (Hidden Layer) that contains 8 perceptrons with relu activation function.
4. Create a dense layer (Output Layer) with sigmoid activation function.
5. Compile the model with loss= binary\_crossentropy, optimizer=adam, metrics='accuracy'.
6. Fit the model
7. Evaluate the accuracy.

Program:

from numpy import loadtxt

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense

# load the dataset

dataset = loadtxt('C:/Users/user/Desktop/NMP/pima.csv', delimiter=',')

# split into input (X) and output (y) variables

X = dataset[:,0:8]

y = dataset[:,8]

model = Sequential()

model.add(Dense(12, input\_shape=(8,), activation='relu'))

model.add(Dense(8, activation='relu'))

model.add(Dense(1, activation='sigmoid'))

# compile the keras model

model.compile(loss='binary\_crossentropy', optimizer='adam', metrics=['accuracy'])

# fit the keras model on the dataset

model.fit(X, y, epochs=150, batch\_size=10)

# evaluate the keras model

\_, accuracy = model.evaluate(X, y)

print('Accuracy: %.2f' % (accuracy\*100))

Output:

Result:

Thus the program to implement Deep learning NN model is implemented and program is executed successfully.

Appendix

Confusion Matrix:

A **confusion matrix** is a matrix that summarizes the performance of a machine learning model on a set of test data. It is a means of displaying the number of accurate and inaccurate instances based on the model’s predictions. It is often used to measure the performance of classification models, which aim to predict a categorical label for each input instance.

The matrix displays the number of instances produced by the model on the test data.

* **True positives (TP):** occur when the model accurately predicts a positive data point.
* **True negatives (TN)**: occur when the model accurately predicts a negative data point.
* **False positives (FP)**: occur when the model predicts a positive data point incorrectly.
* **False** **negatives (FN)**: occur when the model mispredicts a negative data point.

**Confusion Matrix For binary classification**

* A 2X2 Confusion matrix is shown below for the image recognition having a Dog image or Not Dog image.

|  | | **Actual** | |
| --- | --- | --- | --- |
| **Dog** | **Not Dog** |
| **Predicted** | **Dog** | True Positive (TP) | False Positive (FP) |
| **Not Dog** | False Negative (FN) | True Negative (TN) |

* **True Positive (TP):** It is the total counts having both predicted and actual values are Dog.
* **True Negative (TN):** It is the total counts having both predicted and actual values are Not Dog.
* **False Positive (FP):** It is the total counts having prediction is Dog while actually Not Dog.
* **False Negative (FN):** It is the total counts having prediction is Not Dog while actually, it is Dog.

## Metrics based on Confusion Matrix Data

### 1. Accuracy

Accuracy is used to measure the performance of the model. It is the ratio of Total correct instances to the total instances.



### 2. Precision

[Precision](https://www.geeksforgeeks.org/precision-and-recall-in-information-retrieval/) is a measure of how accurate a model’s positive predictions are. It is defined as the ratio of true positive predictions to the total number of positive predictions made by the model.



### 3. Recall

[Recall](https://www.geeksforgeeks.org/precision-and-recall-in-information-retrieval/) measures the effectiveness of a classification model in identifying all relevant instances from a dataset. It is the ratio of the number of true positive (TP) instances to the sum of true positive and false negative (FN) instances.



### 4. F1-Score

[F1-score](https://www.geeksforgeeks.org/precision-recall-and-f1-score-using-r/) is used to evaluate the overall performance of a classification model. It is the harmonic mean of precision and recall,

