AIML Manual

1) I)To write a Python program to implement Breadth First Search (BFS).

```
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
# Breadth-First Search
```

Ouput :

```
"C:\Program Files\Python311\python.exe" "C:\Users\Abishek\Py
Following is the Breadth-First Search
5 3 7 2 4 8
Process finished with exit code 0
```

1)ii) To write a Python program to implement Depth First Search (DFS)

```
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')
# Depth-First Search
```

Output :

```
"C:\Program Files\Python311\python.exe" "C:\Users\
Following is the Depth-First Search
5
3
2
4
8
7
```

2)i) To write a Python program to implement A* search algorithm.

```
import heapq
class Node:
    def __init__(self, state, parent, cost, heuristic):
        self.state = state
        self.parent = parent
        self.cost = cost
        self.heuristic = heuristic
    def __lt__(self, other):
        return (self.cost + self.heuristic) < (other.cost + other.heuristic)
def astar(start, goal, graph):
    heap = []
    heapq.heappush(heap, (0, Node(start, None, 0, 0)))
    visited = set()
    while heap:
        cost, current = heapq.heappop(heap)
        if current.state == goal:
            path = []
            while current is not None:
                path.append(current.state)
```

```
                current = current.parent
            # Return reversed path
            return path[::-1]
        if current.state in visited:
            continue
        visited.add(current.state)
        for neighbor, cost in graph[current.state].items():
            if neighbor not in visited:
                heuristic = 0  # replace with your heuristic function
                heapq.heappush(heap, (cost, Node(neighbor, current, current.cost + cost, heuristic)))
    return None  # No path found
graph = {
    'A': {'B': 1, 'D': 3},
    'B': {'A': 1, 'C': 2, 'D': 4},
    'C': {'B': 2, 'D': 5, 'E': 2},
    'D': {'A': 3, 'B': 4, 'C': 5, 'E': 3},
    'E': {'C': 2, 'D': 3}
}
start = 'A'
goal = 'E'
result = astar(start, goal, graph)
print(result)
# A* Search
```

Output:

```
"C:\Program Files\Python311\python.exe" "C:\Use
['A', 'B', 'C', 'E']

Process finished with exit code 0
```

2)ii) To write a Python program to implement memory- bounded A* search algorithm.

```
import heapq
class Node:
    def __init__(self, state, parent, cost, heuristic):
        self.state = state
        self.parent = parent
        self.cost = cost
        self.heuristic = heuristic
    def __lt__(self, other):
        return (self.cost + self.heuristic) < (other.cost + other.heuristic)
def astar(start, goal, graph, max_nodes):
    heap = []
    heapq.heappush(heap, (0, Node(start, None, 0, 0)))
    visited = set()
    node_counter = 0
    while heap and node_counter < max_nodes:
        cost, current = heapq.heappop(heap)
        if current.state == goal:
            path = []
            while current is not None:
                path.append(current.state)
                current = current.parent
            return path[::-1]
        if current.state in visited:
            continue
        visited.add(current.state)
        node_counter += 1
```

```
        for neighbor, cost in graph[current.state].items():
            if neighbor not in visited:
                heuristic = 0  # Replace with your heuristic function
                heapq.heappush(heap, (cost, Node(neighbor, current, current.cost + cost, heuristic)))
    return None
# Example usage
graph = {'A': {'B': 1, 'C': 4},
         'B': {'A': 1, 'C': 2, 'D': 5},
         'C': {'A': 4, 'B': 2, 'D': 1},
         'D': {'B': 5, 'C': 1}}
start = 'A'
goal = 'D'
max_nodes = 10
result = astar(start, goal, graph, max_nodes)
print(result)
# Memory Bounded A* Search
```

Output:

```
"C:\Program Files\Python311\python.exe" "C:
['A', 'B', 'C', 'D']

Process finished with exit code 0
```

3)To write a python program to implement Naïve Bayes model

```
import pandas as pd
import numpy as np
from sklearn.naive_bayes import GaussianNB
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
```

```
df = pd.read_csv('data.csv')


X = df.drop('buy_computer', axis=1).values

y = df['buy_computer'].values

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=0)


model = GaussianNB()

model.fit(X_train, y_train)


y_pred = model.predict(X_test)

accuracy = accuracy_score(y_test, y_pred)

print("Accuracy:", accuracy)


new_data = np.array([[35, 60000, 1, 100]])

prediction = model.predict(new_data)

print("Prediction:", prediction)


# Naive Bayes Model
```

Output:

```
"C:\Program Files\Python311\python.exe" "C:\Users\A
Accuracy: 0.0
Prediction: ['No']

Process finished with exit code 0
```

4) To write a python program to implement a Bayesian network for the Monty Hall problem.

```
from pgmpy.models import BayesianNetwork
from pgmpy.factors.discrete import TabularCPD
from pgmpy.inference import VariableElimination
model = BayesianNetwork([('A', 'C'), ('B', 'C'), ('C', 'D')])
cpd_a = TabularCPD(variable='A', variable_card=2, values=[[0.3], [0.7]])
cpd_b = TabularCPD(variable='B', variable_card=2, values=[[0.6], [0.4]])
cpd_c = TabularCPD(variable='C', variable_card=2,
        values=[[0.8, 0.9, 0.4, 0.6],
            [0.2, 0.1, 0.6, 0.4]],
        evidence=['A', 'B'], evidence_card=[2, 2])
cpd_d = TabularCPD(variable='D', variable_card=2,
        values=[[0.9, 0.6],
            [0.1, 0.4]],
        evidence=['C'], evidence_card=[2])
model.add_cpds(cpd_a, cpd_b, cpd_c, cpd_d)
print("Model is valid:", model.check_model())
print("CPD A:")
print(cpd_a)
print("CPD B:")
print(cpd_b)
print("CPD C:")
print(cpd_c)
print("CPD D:")
print(cpd_d)
infer = VariableElimination(model)
posterior_d = infer.query(variables=['D'], evidence={'A': 0, 'B': 1})
```

```
print("Posterior probability of D given evidence A=0, B=1:")

print(posterior_d)

# Bayesian Networks
```

Output:

```
"C:\Program Files\Python311\python.exe" "C:\Users\Abishek\Py
Model is valid: True
CPD A:
+------+-----+
| A(0) | 0.3 |
+------+-----+
| A(1) | 0.7 |
+------+-----+
CPD B:
+------+-----+
| B(0) | 0.6 |
+------+-----+
| B(1) | 0.4 |
+------+-----+
CPD C:
+------+------+------+------+------+
| A    | A(0) | A(0) | A(1) | A(1) |
+------+------+------+------+------+
| B    | B(0) | B(1) | B(0) | B(1) |
+------+------+------+------+------+
| C(0) | 0.8  | 0.9  | 0.4  | 0.6  |
+------+------+------+------+------+
| C(1) | 0.2  | 0.1  | 0.6  | 0.4  |
+------+------+------+------+------+
```

5) To write a Python program to build Regression models

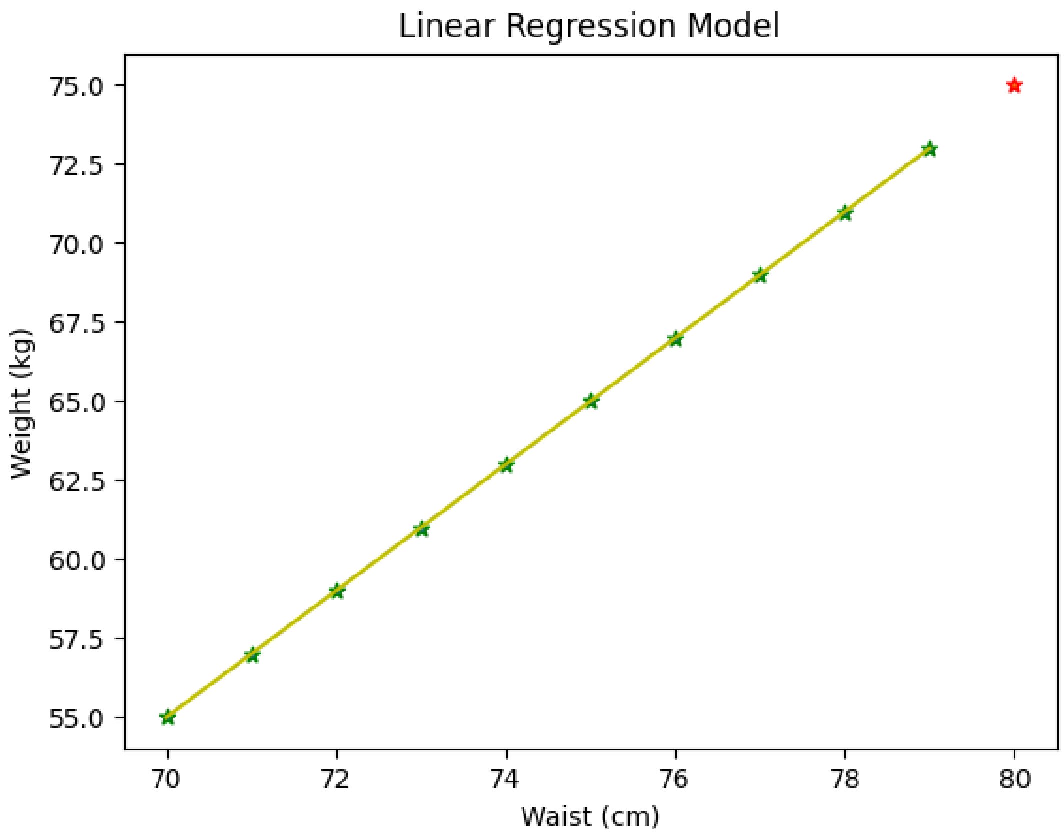
```
import numpy as np

import pandas as pd
```

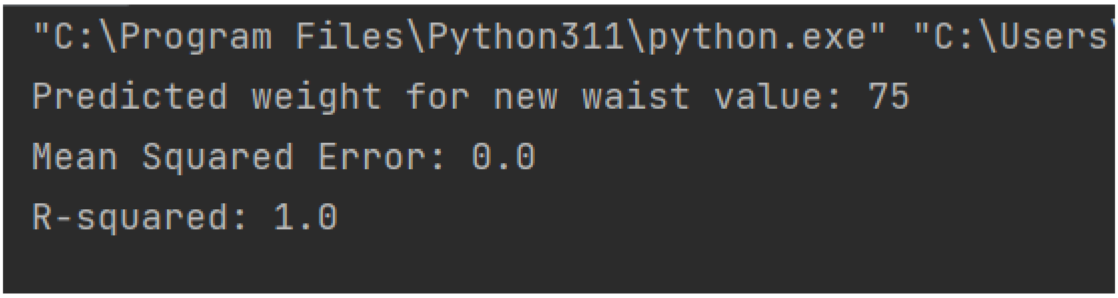
```
import matplotlib.pyplot as plt

from sklearn.linear_model import LinearRegression

from sklearn.metrics import mean_squared_error, r2_score

waist = np.array([70, 71, 72, 73, 74, 75, 76, 77, 78, 79])

weight = np.array([55, 57, 59, 61, 63, 65, 67, 69, 71, 73])

data = pd.DataFrame({'waist': waist, 'weight': weight})

# extract input and output variables

X = data[['waist']]

y = data['weight']

# fit a linear regression model

model = LinearRegression()

model.fit(X, y)

# make predictions on new data

new_data = pd.DataFrame({'waist': [80]})

predicted_weight = model.predict(new_data[['waist']])

print("Predicted weight for new waist value:", int(predicted_weight))

#calculate MSE and R-squared

y_pred = model.predict(X)

mse = mean_squared_error(y, y_pred)

print('Mean Squared Error:', mse)

r2 = r2_score(y, y_pred)

print('R-squared:', r2)

# plot the actual and predicted values

plt.scatter(X, y, marker='*', edgecolors='g')

plt.scatter(new_data, predicted_weight, marker='*', edgecolors='r')

plt.plot(X, y_pred, color='y')

plt.xlabel('Waist (cm)')

plt.ylabel('Weight (kg)')

plt.title('Linear Regression Model')
```

plt.show()

Regression Model

Output:





6) To write a Python program to build SVM model.

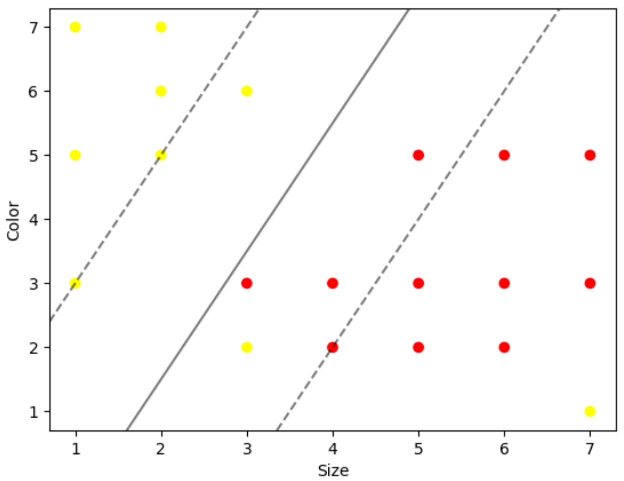
import matplotlib.pyplot as plt

import numpy as np

from sklearn import svm

```
X = np.array([[5, 2], [4, 3], [1, 7], [2, 6], [5, 5], [7, 1], [6, 2], [5, 3], [3, 6], [2, 7], [6, 3], [3, 3],
[1, 5], [7, 3], [6, 5], [2, 5], [3, 2], [7, 5], [1, 3], [4, 2]])
y = np.array([0, 0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 0, 1, 1, 0, 1, 0])
clf = svm.SVC(kernel='linear')
clf.fit(X, y)
colors = ['red' if label == 0 else 'yellow' for label in y]
plt.scatter(X[:, 0], X[:, 1], c=colors)
ax = plt.gca()
ax.set_xlabel('Size')
ax.set_ylabel('Color')
xlim = ax.get_xlim()
ylim = ax.get_ylim()
xx, yy = np.meshgrid(np.linspace(xlim[0], xlim[1], 100), np.linspace(ylim[0], ylim[1], 100))
Z = clf.decision_function(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)
ax.contour(xx, yy, Z, colors='k', levels=[-1, 0, 1], alpha=0.5, linestyles=['--', '-', '--'])
plt.show()
# SVM Models
```

Output:

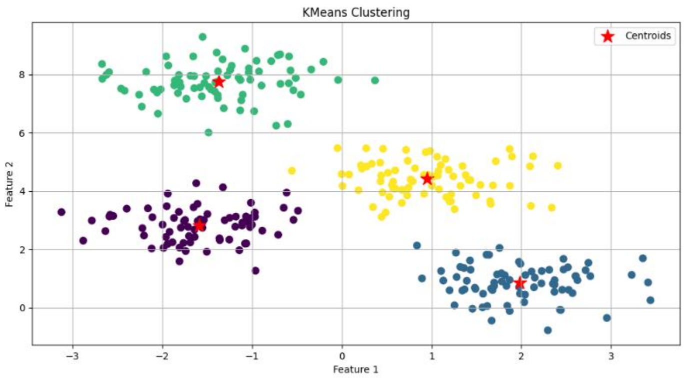


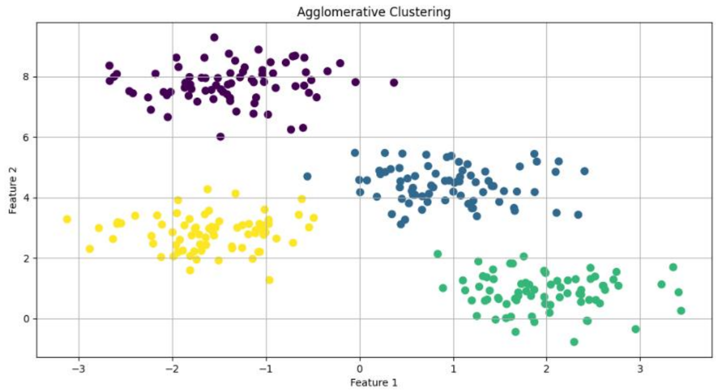
7) Implement clustering algorithms

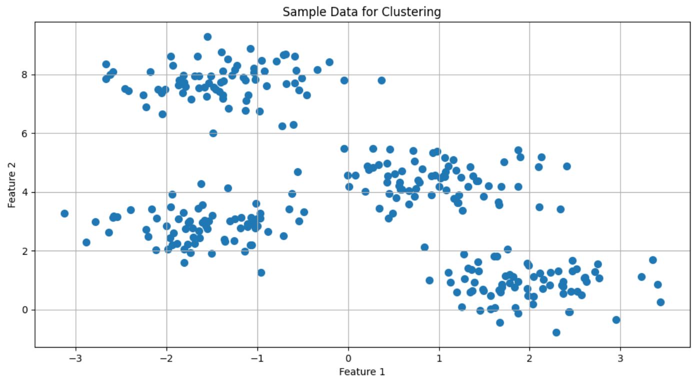
```
import numpy as np

import matplotlib.pyplot as plt

from sklearn.datasets import make_blobs

from sklearn.cluster import KMeans, AgglomerativeClustering

from sklearn.metrics import silhouette_score

# Generating sample data

X, _ = make_blobs(n_samples=300, centers=4, cluster_std=0.60, random_state=0)

# Plotting the sample data

plt.figure(figsize=(12, 6))

plt.scatter(X[:, 0], X[:, 1], s=50)

plt.title('Sample Data for Clustering')

plt.xlabel('Feature 1')

plt.ylabel('Feature 2')

plt.grid(True)

plt.show()

# Implementing KMeans clustering

kmeans = KMeans(n_clusters=4)

kmeans.fit(X)

kmeans_labels = kmeans.labels_

# Plotting the results of KMeans clustering

plt.figure(figsize=(12, 6))

plt.scatter(X[:, 0], X[:, 1], c=kmeans_labels, s=50, cmap='viridis')

plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1], s=200, c='red',

marker='*', label='Centroids')

plt.title('KMeans Clustering')

plt.xlabel('Feature 1')

plt.ylabel('Feature 2')
```

```
plt.legend()
plt.grid(True)
plt.show()
# Implementing Agglomerative Clustering
agglomerative = AgglomerativeClustering(n_clusters=4)
agglomerative.fit(X)
agglomerative_labels = agglomerative.labels_
# Plotting the results of Agglomerative Clustering
plt.figure(figsize=(12, 6))
plt.scatter(X[:, 0], X[:, 1], c=agglomerative_labels, s=50, cmap='viridis')
plt.title('Agglomerative Clustering')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.grid(True)
plt.show()
# Evaluating clustering performance using silhouette score
kmeans_score = silhouette_score(X, kmeans_labels)
agglomerative_score = silhouette_score(X, agglomerative_labels)
print("Silhouette Score for KMeans: {:.2f}".format(kmeans_score))
print("Silhouette Score for Agglomerative Clustering: {:.2f}".format(agglomerative_score))
# Clustering
```

Output :




Agglomerative Clustering
Feature 1
Feature 2


Sample Data for Clustering
Feature 1
Feature 2