Expt. No. | Page. No.

```
def iddfs(graph, start, goal):
    def dls(node, goal, depth, path):
        if node == goal:
            return path + [node]
        if depth == 0:
            return None
        for neighbor in graph.get(node, []):
            result = dls(neighbor, goal,
                         depth - 1, path + [node])
            if result:
                return result
        return None

    for depth in range(len(graph)):
        result = dls(start, goal, depth, [])
        if result:
            return result
    return None

graph = { BFS }

path = iddfs(graph 'A', 'G')

if path:
    print(f'Path found : {path}')
else:
    print("No path found")
```

OUTPUT

```
graph = {'A': [('B',1), ('C',3), ('D',7)],
         'B': [('E',5), ('F',12)]
         'C': [('G',2)],
         'D': [],
         'E': [],
         'F': [('G',3)],
         'G': []}
def heuristic(n):
    H = {'A':10, 'B':6, 'C':4, 'D':7,
         'E':5, 'F':3, 'G':0}
    return H(n)
print(a_star(graph, 'A', 'G', heuristic))
```

O/P:

```
['A', 'C', 'G']
```

4) Memory Bounded A*

```
import heapq
def mb_a_star(graph, start, goal, h,
              memory_limit):
    open_list = []
    heapq.heappush(open_list, (0+h(start), 0,
                               start))
```

```
import heapq

def dijikstra(graph, start, goal):
    queue = [(0, start)]
    distance = {vertex: float('infinity')
                for vertex in graph}
    distance[start] = 0
    came_from = {start: None}

    while queue:
        current_distance, current_vertex =
                heapq.heappop(queue)
        if current_distance > distance[current
                vertex]:
            continue

        for neighbor, weight in graph[current
                vertex].items():
            distance = current_distance + weight

            if distance < distances[neighbour]:
                distances[neighbour] = distance
                came_from[neighbour] =
                        current_vertex
                heapq.heappush(queue,
                        (distance, neighbour))
    path=[]
    current = goal
```

```
    while current is not None:
        path.append(current)
        current = came_from[current]
    path.reverse()
    return path, distances[goal]
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}
        }
path, cost = dijkstra(graph, 'A', 'D')
print(f'Path: {path}, Cost = {cost}')
```

O/P

Path: ['A', 'B', 'C', 'D'], cost = 4

```
import tensorflow as tf
from tensorflow.keras.models import
                                Sequential
from tensorflow.keras.layers import
                                Dense

import numpy as np
x = np.random.randn(1000,10)
y = np.dot(x, np.array([1.5,-2.0, 3.0,
    -0.5, 2.5,-1.5, 2.0,-2.5, 1.0,
    0.5])) + 0.1 * np.random.
                        randn(1000)

model = Sequential([
    Dense(64, activation='relu',
                    input_shape(10,)),
    Dense(64, activation='relu'),
    Dense(1)])

model.compile(optimizer='adam',
                        loss='mse')

model.fit(x,y, epochs=10, batch_size=3

print(model.predict(x[:5]))
```

10(P)

[[2.60 :]
[ ]
[ ]
[ ]
[ ]]

```
print(f'Decision Tree Accuracy : {accuracy_tree*
                    100:.2f}%')
printf(f'Random Forest Accuracy : {accuracy_forest*
                    100:.2f}%')
print("\nDecision Tree Classifier . Report")
print(classification_report(y_test, y_pred_tree,
                    target_name = iris.target_names))
print("Random forest Classifier . Report")
print(classification_report(y_test, y_pred_forest,
                    target_name = iris.target_names))
print('Dec Tree Confusion matrix")
print(confusion_matrix(y_test, y_pred_tree)
print('Rand for confusion matrix")
print(confusion_matrix(y_test, y_pred_for)
print("Decision Tree prediction for 1st 10 test
                    samples")
print("Predicted :", y_pred_tree[:10])
print("Actual :", y_test[:10])
print("Random Forest prediction for 1st 10 test
                    samples")
print("Predicted :", y_pred_for[:10])
print("Actual :", y_test[:10])
```

) Random forest by compare decision tree & random fore

t. No. | Page. No.

```
import numpy as np
from sklearn.datasets import load_iris
from sklearn.model_selection import
                        train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score,
    classification_report, confusion_matrix

iris = load_iris()
x = iris.data
y = iris.target

x_train, x_test, y_train, y_test = train_test_
    split(x, y, test_size=0.3, random_
                        state = 42)
decision_tree = DecisionTreeClassifier()
decision_tree.fit(x_train, y_train)

random_forest = RandomForestClassifier(n_estimators
                = 100, random_state = 42)
random_forest.fit(x_train, y_train)

y_pred_tree = decision_tree.predict(x_test)
y_pred_for = random_forest.predict(x_test)
```

(10) Decision Tree

Expt. No. | Page. No

```
import matplotlib.pyplot as plt
from sklearn.datasets import
                    load_iris
from sklearn.tree import DecisionTree
                    Classifier
from sklearn.import tree

iris = load_iris()
x, y = iris.data, iris.target

df = DecisionTreeClassifier()
df.fit(x, y)

plt.figure(figsize=(20, 10))

tree.plot_tree(df, filled=True,
        feature_names=iris,
    feature_names, class_names=
                iris.target_names)

plt.show()
```

OUTPUT

```
d1 = DiscreteDistribution({'T': 0.6,
                           'F': 0.4})
d2 = ConditionalProbabilityTable(
        [['T', 'T', 0.8], ['T', 'F', 0.2],
         ['F', 'T', 0.4], ['F', 'F', 0.6]], [d1])

s1 = State(d1, name="A")
s2 = State(d2, name="B")

net = BayesianNetwork("Simple Network")
net.add_states(s1, s2)
net.add_edge(s1, s2)
net.bake()

data = np.array([['T', 'T'], ['T', 'F'],
                 ['F', 'T'], ['F', 'F']])
net.fit(data, algorithm='em')

print(net.probability([['T', 'T']]))
```

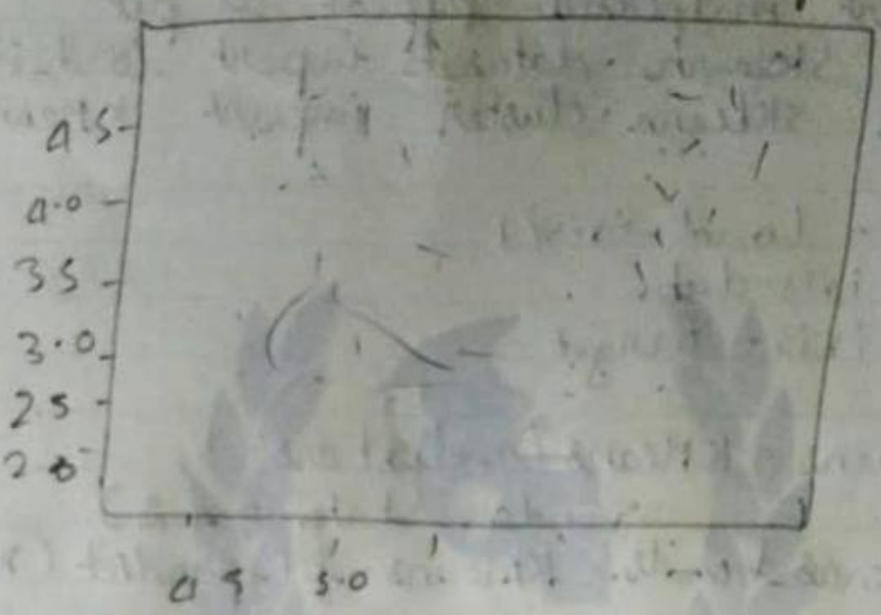
O/P

0.64

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.cluster import KMeans

iris = load_iris()
x = iris.data
y = iris.target

Kmeans = KMeans(n_clusters = 3,
                random_state = 42)
Kmeans_labels = Kmeans.fit_predict(x)

plt.figure(figsize = (8,6))

plt.scatter(x[:,0], x[:,1], c = Kmeans_
        labels, cmap = 'viridis', marker = 'o',
                edgecolor = 'K')

plt.title('K-means clustering')
plt.xlabel('x')
plt.ylabel('y')
plt.show()
```

Q(a)

K-means clustering



```
print("In Predictions for 1st 10 test samples:")
print("Predicted:", y_pred[:10])
print("Actual:", y_test[:10])
```

12) Support Vector

```
from sklearn.svm import SVC

model= SVC(kernal='linear')
```

same as naive Bayes model

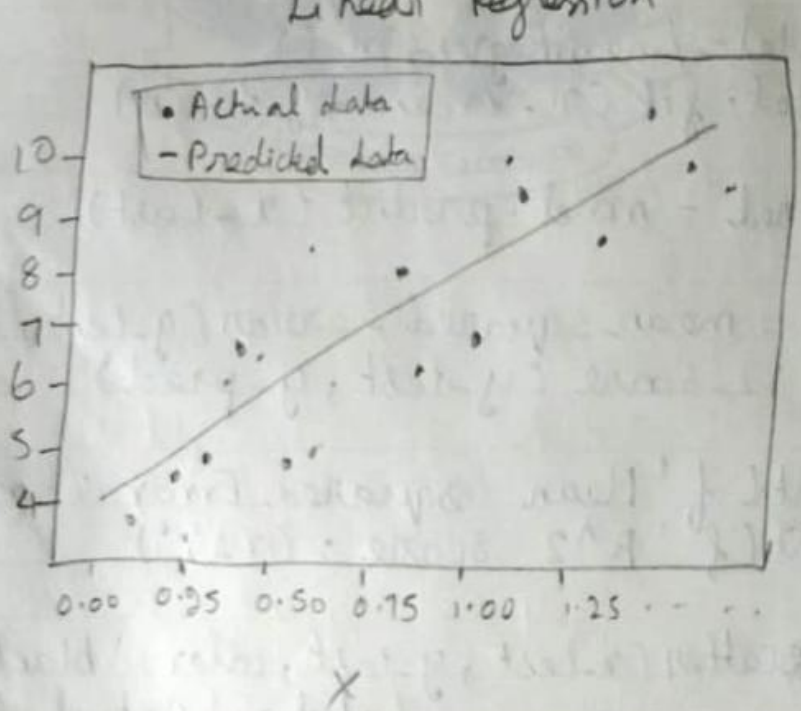
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import
                    train_test_split
from sklearn.linear_model import
                    LinearRegression
from sklearn.metrics import mean_squared_
                    error, r2_score
np.random.seed(0)
x = 2 * np.random.rand(100,1)
y = 4+3 * x + np.random.randn(100,1)

x_train, x_test, y_train, y_test =
        train_test_split(x,y, test_size = 0.2,
                    random_state =0)

model = LinearRegression()
model.fit(x_train, y_train)

y_pred = model.predict(x_test)

mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)

print(f'Mean squared Error : {mse}')
print(f'R^2 score : {r2}')

plt.scatter(x_test, y_test, color='black',
                    label='Actual data')
```

```
plt.plot(x_test, y_pred, color='blue',
         linewidth=2, label='Predicted line')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Linear Regression')
plt.legend()
plt.show()
```

O/P:

Mean Squared error = 1.0494 3381

R^2 score = 0.742 44523...



Expt. No.

Page No.

```
1) import numpy as np
2) from sklearn.datasets import load_iris
3. from sklearn.model_selection import
                                train_test_split
4. from sklearn.naive_bayes import GaussianNB
5. from sklearn.metrics import accuracy_score,
                        classification_report,
                        confusion_matrix
6. iris = load_iris()
7. x = iris.data
8. y = iris.target

9. x_train, x_test, y_train, y_test = train_test_
                split(x, y, test_size = 0.3,
                        random_state = 42)
10. model = GaussianNB()
11. model.fit(x_train, y_train)
12 y_pred = model.predict(x_test)

13. accuracy = accuracy_score(y_test, y_pred)
14. print(f"Accuracy: {accuracy * 100:.2f}%")

15. print("Classification report")
16. print(Classification_report(y_test,
                y_pred, target_names =
                        iris.target_names))

17. print("Confusion matrix")
18. print(confusion_matrix(y_test, y_pred))
```

```
graph = { 'A': [('B', 1), ('C', 3), ('D', 7)],
          'B': [('E', 5), ('F', 12)],
          'C': [('G', 2)],
          'D': [],
          'E': [],
          'F': [('G', 3)],
          'G': [] }

def heuristic(n):
    H = { 'A': 10, 'B': 6, 'C': 9, 'D': 7,
          'E': 5, 'F': 3, 'G': 0 }
    return H[n]

print(mb_a_star(graph, 'A', 'G',
                heuristic, memory_limit=5))
```

[O/P]

```
['A', 'C', 'G']
```

```
came_from = {start : None}
g_score = {start : 0}
closed_list = set()

while open_list :
    if len(closed_list) > memory_limit:
        closed_list.pop()

    _, current_g, current = heapq.heappop
                            (open_list)
    closed_list.add(current)

    if current == goal:
        path=[]
        while current:
            path.append(current)
            current = came_from[current]
        return path[::-1]
    for neighbor, cost in graph[current]
        tentative_g = current_g + cost
        if neighbor not in g_score or
            tentative_g < g_score[neighbor]
        came_from[neighbor] = current
        g_score[neighbor] = tentative_g
        f_score = tentative_g + h(neighbor)
        heapq.heappush(open_list,
                (f_score, tentative_g,
                        neighbor))
return None
```

```
import heapq
def a_star(graph, start, goal, h):
    open_list = []
    heapq.heappush(open_list, (0 + h(start),
                               0, start))
    came_from = {start: None}
    g_score = {start: 0}

    while open_list:
        _, current_g, current = heapq.heappop
                                (open_list)
        if current == goal:
            path = []
            while current:
                path.append(current)
                current = came_from[current]
            return path[::-1]

        for neighbor, cost in graph[current]:
            tentative_g = current_g + cost
            if neighbor not in g_score or
                tentative_g < g_score[neighbor]:

                came_from[neighbor] = current
                g_score[neighbor] = tentative_g
                f_score = tentative_g + h(neighbor)
                heapq.heappush(open_list, (f_score
                                tentative_g, neighbor))
    return None
```