Practical-1

Aim: Implement Breadth first search and Depth first search.

Breadth First Search:

```
from collections import defaultdict
class Graph:
        def __init_(self):
                self.graph = defaultdict(list)
        def addEdge(self,u,v):
                self.graph[u].append(v)
        def BFS(self, s):
                visited = [False] * (len(self.graph))
                queue = []
                queue.append(s)
                visited[s]  =  True
                while queue:
                        s = queue.pop(0)
                        print (s, end = " ")
                        for i in self.graph[s]:
                                if visited[i] == False:
                                        queue.append(i)
                                        visited[i] = True
g = Graph()
g.addEdge(0,  1)
g.addEdge(0,  2)
g.addEdge(1,  2)
g.addEdge(2,  0)
g.addEdge(2,  3)
g.addEdge(3,  3)
print ("Following is Breadth First Traversal"
                                " (starting from vertex 2)")
g.BFS(2)
```

Output:-

```
Following is Breadth First Traversal (starting from vertex 2)
2 0 3 1
```

Depth First Search:

```
from collections import defaultdict
class Graph:
  def __init_(self):
    self.graph = defaultdict(list)
```

```
def addEdge(self, u, v):
self.graph[u].append(v)
def DFSUtil(self, v, visited):
```

```
    visited.add(v)
    print(v, end=' ')
    for neighbour in self.graph[v]:
      if neighbour not in visited:
        self.DFSUtil(neighbour, visited)
  def DFS(self, v):
    visited = set()
    self.DFSUtil(v,  visited)
if __name___== "_main_":
  g = Graph()
  g.addEdge(0,  1)
  g.addEdge(0,  2)
  g.addEdge(1,  2)
  g.addEdge(2,  0)
  g.addEdge(2,  3)
  g.addEdge(3,  3)
  print("Following is DFS from (starting from vertex 2)")
  g.DFS(2)
```

Output:

```
Following is DFS from (starting from vertex 2)
2 0 1 3
```

Practical-2

Aim: Implement solution of 8-puzzle problem using A*.

```
import copy
from heapq import heappush,
heappopn = 3
rows = [ 1, 0, -1, 0 ]
cols = [ 0, -1, 0, 1 ]
class priorityQueue:
    def __init_(self):
        self.heap = []
    def push(self, key):
        heappush(self.heap, key)
    def pop(self):
        return heappop(self.heap)
    def empty(self):
        if not self.heap:
            return True
        else:
            return False
class nodes:
    def __init_(self, parent, mats,
                empty_tile_posi,costs, levels):
        self.parent = parent
        self.mats = mats
        self.empty_tile_posi = empty_tile_posi
        self.costs = costs
        self.levels = levels
    def __lt_(self, nxt):
        return self.costs < nxt.costs
def calculateCosts(mats, final) -> int:
    count = 0
    for i in range(n):
        for j in range(n):
            if ((mats[i][j]) and (mats[i][j]
                != final[i][j])):
                count += 1
    return count
def newNodes(mats, empty_tile_posi, new_empty_tile_posi,
            levels, parent, final) -> nodes:
    new_mats = copy.deepcopy(mats)
    x1 = empty_tile_posi[0]
    y1 = empty_tile_posi[1]
    x2 = new_empty_tile_posi[0]
    y2 = new_empty_tile_posi[1]
```

```
    new_mats[x1][y1], new_mats[x2][y2] = new_mats[x2][y2], new_mats[x1][y1]costs
    = calculateCosts(new_mats, final)
    new_nodes = nodes(parent, new_mats, new_empty_tile_posi,
                costs, levels)
    return new_nodes
def printMatsrix(mats):
    for i in range(n):
        for j in range(n):
            print("%d " % (mats[i][j]), end = " ")
        print()
def isSafe(x, y):
    return x >= 0 and x < n and y >= 0 and y < ndef
printPath(root):
    if root == None:
        return
    printPath(root.parent)
    printMatsrix(root.mats)
    print()
def solve(initial, empty_tile_posi, final):pq
    = priorityQueue()
    costs = calculateCosts(initial, final)
    root = nodes(None, initial,
            empty_tile_posi, costs,
    0)pq.push(root)
    while not pq.empty():
        minimum = pq.pop()
        if minimum.costs == 0:
            printPath(minimum)
            return
```

```
    for i in range(n):
            new_tile_posi = [
                minimum.empty_tile_posi[0] + rows[i],
                minimum.empty_tile_posi[1]  + cols[i],  ]
            if isSafe(new_tile_posi[0], new_tile_posi[1]):
                child = newNodes(minimum.mats,
                            minimum.empty_tile_posi,
                            new_tile_posi,
                            minimum.levels + 1,
                            minimum, final,)
                pq.push(child)
initial = [ [ 1, 2, 3 ],
            [ 5, 6, 0 ],
            [ 7, 8, 4 ] ]
    final = [ [ 1, 2, 3 ],
            [ 5, 8, 6 ],
            [ 0, 7, 4 ] ]
empty_tile_posi = [ 1, 2 ] solve(initial,
empty_tile_posi,  final)
```

Output:

```
1  2  3
5  6  0
7  8  4

1  2  3
5  0  6
7  8  4

1  2  3
5  8  6
7  0  4

1  2  3
5  8  6
0  7  4
```

Practical-3

Aim: Write a program to solve a given cryptarithmetic problem.

```
import re
import itertools
def solve(puzzle):
    words = re.findall('[A-Z]+', puzzle.upper())
    unique_characters = set(''.join(words))
    assert len(unique_characters) <= 10, 'Too many letters'
    first_letters = {word[0] for word in words}
    n = len(first_letters)
    sorted_characters = ''.join(first_letters) + ''.join(unique_characters - first_letters) characters
    = tuple(ord(c) for c in sorted_characters)
    digits = tuple(ord(c) for c in '0123456789')
    zero = digits[0]
    for guess in itertools.permutations(digits, len(characters)):if
        zero not in guess[:n]:
            equation = puzzle.translate(dict(zip(characters, guess)))
            lhs, rhs = equation.split('=')
            lhs = lhs.strip()rhs
            = rhs.strip()
            if eval(lhs) == int(rhs):
                return equation
if __name___== '__main__':
    puzzle = 'SEND + MORE = MONEY'
    solution = solve(puzzle)if
    solution:
        print('SEND + MORE = MONEY =>'+ solution)else:
        print('No solution found')
```

Output:

```
SEND + MORE = MONEY =>9567 + 1085 = 10652
```

Practical-4

Aim: Write a program to perform following operation

- Load the data from file
- Find out null and missing value
- Handle missing Value using different approach
- Plot the data using scatter plot

```
import pandas as pd
import matplotlib.pyplot as plt
df = pd.read_csv('/content/drive/MyDrive/datat1.csv')
df.head()
df.notnull()
df.fillna(0)
df.fillna(method ='pad')
DriverID = df['Driver id']
Status = df['Request id']
plt.scatter(df['Request id'], df['Driver id'], c=df['Status'].apply(lambda x: 'red' if x == 'Trip Completed' else 'blue'))
plt.xlabel('Request id')
plt.ylabel('Driver id')
plt.show()
```

Output:

Load data

	Request id	Pickup point	Driver id	Status	Request timestamp	Drop timestamp
0	619	Airport	1.0	Trip Completed	11-07-2016 11:51	11-07-2016 13:00
1	867	Airport	NaN	Trip Completed	11-07-2016 17:57	11-07-2016 18:47
2	1807	NaN	1.0	NaN	12-07-2016 09:17	12-07-2016 09:58
3	2532	Airport	1.0	Trip Completed	12-07-2016 21:08	12-07-2016 22:03
4	3112	City	1.0	Trip Completed	13-07-2016 08:33	13-07-2016 09:25

Null or missing data

	Request id	Pickup point	Driver id	Status	Request timestamp	Drop timestamp
0	True	True	True	True	True	True
1	True	True	False	True	True	True
2	True	False	True	False	True	True
3	True	True	True	True	True	True
4	True	True	True	True	True	True

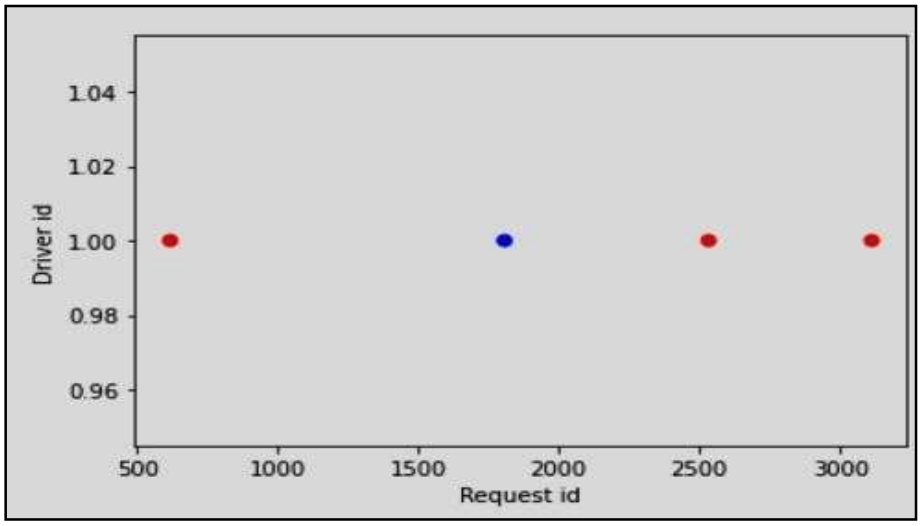
Handling missing or null data by filling it with 0

	Request id	Pickup point	Driver id	Status	Request timestamp	Drop timestamp
0	619	Airport	1.0	Trip Completed	11-07-2016 11:51	11-07-2016 13:00
1	867	Airport	0.0	Trip Completed	11-07-2016 17:57	11-07-2016 18:47
2	1807	0	1.0	0	12-07-2016 09:17	12-07-2016 09:58
3	2532	Airport	1.0	Trip Completed	12-07-2016 21:08	12-07-2016 22:03
4	3112	City	1.0	Trip Completed	13-07-2016 08:33	13-07-2016 09:25

Handling missing or null data by filling it with previous value

	Request id	Pickup point	Driver id	Status	Request timestamp	Drop timestamp
0	619	Airport	1.0	Trip Completed	11-07-2016 11:51	11-07-2016 13:00
1	867	Airport	1.0	Trip Completed	11-07-2016 17:57	11-07-2016 18:47
2	1807	Airport	1.0	Trip Completed	12-07-2016 09:17	12-07-2016 09:58
3	2532	Airport	1.0	Trip Completed	12-07-2016 21:08	12-07-2016 22:03
4	3112	City	1.0	Trip Completed	13-07-2016 08:33	13-07-2016 09:25

Scatter plot

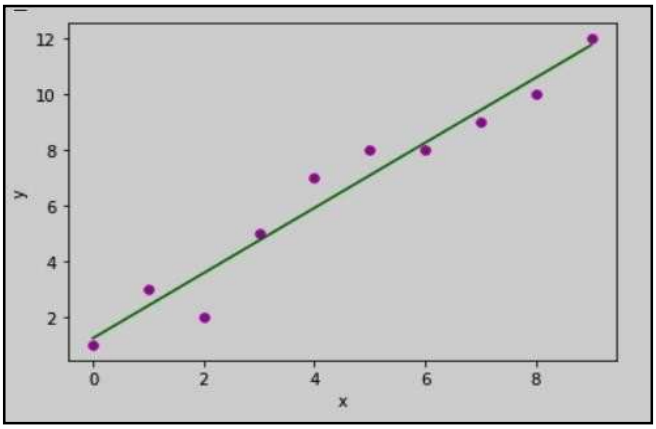


Practical-5

Aim: Write a program to implement Linear Regression.

```
import numpy as np
import matplotlib.pyplot as
pltdef estimate_coef(x, y):
  n =  np.size(x)
  m_x = np.mean(x)
  m_y =  np.mean(y)
  SS_xy =  np.sum(y*x)  -  n*m_y*m_x
  SS_xx =  np.sum(x*x)  -  n*m_x*m_x
  b_1 = SS_xy / SS_xx
  b_0 = m_y - b_1*m_x
  return (b_0, b_1)
def plot_regression_line(x, y, b):
  plt.scatter(x, y, color = "m",
      marker = "o", s = 30)
  y_pred = b[0] + b[1]*x
  plt.plot(x, y_pred, color = "g")
  plt.xlabel('x')
  plt.ylabel('y')
  plt.show()
def main():
  x = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
  y = np.array([1, 3, 2, 5, 7, 8, 8, 9, 10, 12])
  b = estimate_coef(x, y)
  print("Estimated coefficients:\nb_0 = {} \
    \nb_1 = {}".format(b[0], b[1]))
  plot_regression_line(x, y, b)
if _name_ == "_main_":
  main()
```

OUTPUT:-



Practical-6

Aim: Write a program to implement k-Nearest Neighbour algorithm to classify the iris data set.Print both correct and wrong predictions

```
import numpy as np
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier iris
= load_iris()
X_train, X_test, y_train, y_test = train_test_split(iris.data, iris.target, test_size=0.3, random_
state=42)
knn = KNeighborsClassifier(n_neighbors=3)
knn.fit(X_train, y_train)
y_pred = knn.predict(X_test)
correct = 0
wrong = 0
for i in range(len(y_pred)):if
    y_pred[i] == y_test[i]:
        print(f"Correct prediction: {i+1}")
        correct += 1
    else:
        print(f"Wrong prediction: {i+1}")
        wrong += 1
print(f"\nTotal  predictions: {len(y_pred)}")
print(f"Correct predictions: {correct}")
print(f"Wrong predictions: {wrong}")
```

Output:

```
Correct prediction: 1
Correct prediction: 2
Correct prediction: 3
Correct prediction: 4
Correct prediction: 5
Correct prediction: 6
Correct prediction: 7
Correct prediction: 8
Correct prediction: 9
Correct prediction: 10
Correct prediction: 11
Correct prediction: 12
Correct prediction: 13
Correct prediction: 14
Correct prediction: 15
Correct prediction: 16
Correct prediction: 17
Correct prediction: 18
Correct prediction: 19
Correct prediction: 20
Correct prediction: 21
Correct prediction: 22
Correct prediction: 23
Correct prediction: 24
Correct prediction: 25
Correct prediction: 26
Correct prediction: 23
Correct prediction: 24
Correct prediction: 25
Correct prediction: 26
Correct prediction: 27
Correct prediction: 28
Correct prediction: 29
Correct prediction: 30
Correct prediction: 31
Correct prediction: 32
Correct prediction: 33
Correct prediction: 34
Correct prediction: 35
Correct prediction: 36
Correct prediction: 37
Correct prediction: 38
Correct prediction: 39
Correct prediction: 40
Correct prediction: 41
Correct prediction: 42
Correct prediction: 43
Correct prediction: 44
Correct prediction: 45

Total predictions: 45
Correct predictions: 45
Wrong predictions: 0
```

Practical-7

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

```
import pandas as pd
import math
iris = pd.read_csv("/content/drive/MyDrive/iris.csv")
train_data = iris.sample(frac=0.7, random_state=1)
test_data = iris.drop(train_data.index)
def entropy(data):
    classes = data['Species'].unique()
    entropy = 0
    for c in classes:
        p = len(data[data['Species'] == c]) / len(data)
        entropy -= p * math.log2(p)
    return entropy
def info_gain(data, feature):
    values = data[feature].unique()
    gain = entropy(data)
    for v in values:
        subset = data[data[feature] == v]
        gain -= len(subset) / len(data) * entropy(subset)
    return gain
def build_tree(data, features):
    if len(data['Species'].unique()) == 1:
        return data['Species'].iloc[0]
    if len(features) == 0:
        return data['Species'].value_counts().idxmax()
    best_feature = max(features, key=lambda f: info_gain(data, f))
    tree = {best_feature: {}}
    for value in data[best_feature].unique():
        subset = data[data[best_feature] == value]if
        len(subset) == 0:
            tree[best_feature][value] = data['Species'].value_counts().idxmax()else:
            remaining_features = [f for f in features if f != best_feature]
            tree[best_feature][value] = build_tree(subset, remaining_features)
    return tree
def predict(tree, sample):
    while isinstance(tree, dict):
        feature = list(tree.keys())[0]
        if sample[feature] in tree[feature]:
            tree = tree[feature][sample[feature]]
        else:
            subtree_classes = [subtree for subtree in tree[feature].values() if isinstance(subtree, s
```

```
tr)]
        tree = max(set(subtree_classes), key=subtree_classes.count)
    return tree

features = iris.columns[:-1]
tree = build_tree(train_data, features)
correct_predictions = 0
for i, row in test_data.iterrows():
    prediction = predict(tree, row) if
    prediction == row['Species']:
        correct_predictions += 1
    else:
        print(f"Wrong prediction for sample {i}: expected {row['Species']}, but got {prediction}
")

accuracy = correct_predictions / len(test_data)
print(f"Accuracy: {accuracy:.2f}")
```

Output:

```
Wrong prediction for sample 43: expected Iris-setosa, but got Iris-virginica
Wrong prediction for sample 68: expected Iris-versicolor, but got Iris-virginica
Wrong prediction for sample 70: expected Iris-versicolor, but got Iris-virginica
Wrong prediction for sample 106: expected Iris-virginica, but got Iris-versicolor
Wrong prediction for sample 129: expected Iris-virginica, but got Iris-versicolor
Wrong prediction for sample 133: expected Iris-virginica, but got Iris-versicolor
Wrong prediction for sample 134: expected Iris-virginica, but got Iris-versicolor
Accuracy: 0.84
```

Practical-8

Aim: Write a program to classify IRIS data using Random Forest classifier.

```
import pandas as pd
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score
iris = load_iris()
X = pd.DataFrame(iris.data, columns=iris.feature_names) y
= pd.Series(iris.target)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
rf_classifier = RandomForestClassifier(n_estimators=100, max_depth=2)
rf_classifier.fit(X_train, y_train)
y_pred = rf_classifier.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy of Random Forest classifier on IRIS dataset: {accuracy}")
```

Output:

```
Accuracy of Random Forest classifier on IRIS dataset: 1.0
```

Practical-9

Aim: Write a program to classify iris dataset using SVM. Experiment with different kernelfunctions.

```
import pandas as
pdimport numpy as
np
from sklearn.model_selection import
train_test_splitfrom sklearn.svm import SVC
from sklearn.metrics import accuracy_score
iris_data = pd.read_csv("/content/drive/MyDrive/iris.csv")
X_train, X_test, y_train, y_test = train_test_split(iris_data.iloc[:, :-1], iris_data.iloc[:, - 1],
test_size=0.2, random_state=42)
svm_linear = SVC(kernel='linear') svm_poly
= SVC(kernel='poly', degree=3)svm_rbf =
SVC(kernel='rbf') svm_linear.fit(X_train,
y_train) svm_poly.fit(X_train, y_train)
svm_rbf.fit(X_train, y_train)
y_pred_linear = svm_linear.predict(X_test)
y_pred_poly = svm_poly.predict(X_test)
y_pred_rbf = svm_rbf.predict(X_test)
acc_linear = accuracy_score(y_test, y_pred_linear)
acc_poly = accuracy_score(y_test, y_pred_poly)
acc_rbf = accuracy_score(y_test, y_pred_rbf)
print("Accuracy scores:")
print(f"Linear kernel: {acc_linear}")
print(f"Polynomial kernel: {acc_poly}")
print(f"RBF kernel: {acc_rbf}")
```

Output:

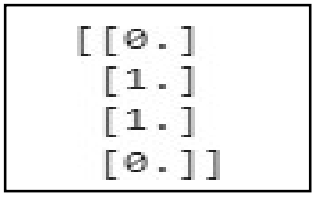
```
Accuracy scores:
Linear kernel: 1.0
Polynomial kernel: 1.0
RBF kernel: 1.0
```

Practical-10

Aim: Build an Artificial Neural Network by implementing the Backpropagation algorithm and testthe same using appropriate data sets.

```
import numpy as np
class  NeuralNetwork:
    def __init_(self, input_dim, hidden_dim, output_dim):
        self.input_weights = np.random.randn(input_dim, hidden_dim)
        self.output_weights = np.random.randn(hidden_dim, output_dim)
    def sigmoid(self, x):
        return 1 / (1 + np.exp(-x))
    def  sigmoid_derivative(self,  x):
        return x * (1 - x)
    def feedforward(self, X):
        self.hidden_output = self.sigmoid(np.dot(X, self.input_weights))
        self.output = self.sigmoid(np.dot(self.hidden_output, self.output_weights))
        return self.output
    def  backpropagate(self,  X,  y,  learning_rate):
        output_error = y - self.output
        output_delta = output_error * self.sigmoid_derivative(self.output)
        hidden_error = np.dot(output_delta, self.output_weights.T)
        hidden_delta = hidden_error * self.sigmoid_derivative(self.hidden_output)
        self.output_weights += learning_rate * np.dot(self.hidden_output.T, output_delta)
        self.input_weights += learning_rate * np.dot(X.T, hidden_delta)
    def train(self, X, y, epochs, learning_rate):
        for i in range(epochs):
            output = self.feedforward(X)
            self.backpropagate(X,  y,  learning_rate)
    def predict(self, X):
        output = self.feedforward(X)
        predictions = np.round(output)
        return predictions
X = np.array([[0,0], [0,1], [1,0], [1,1]])
y = np.array([[0], [1], [1], [0]])
nn = NeuralNetwork(2, 4, 1)
nn.train(X, y, epochs=10000, learning_rate=0.1)
new_data  =  np.array([[0,0],  [0,1],  [1,0],  [1,1]])
predictions = nn.predict(new_data)
print(predictions)
```

Output:


[[0.]
[1.]
[1.]
[0.]]

Practical-11

Aim: Write a Program to implement K-Means clustering Algorithm.

```
import numpy as
npimport random
class KMeans:
    def __init_(self, k=2, max_iters=100, random_state=None):
        self.k = k
        self.max_iters = max_iters
        self.random_state = random_state
    def fit(self, X):
        n_samples, n_features = X.shape
        if self.random_state is not None:
            np.random.seed(self.random_state)
        self.centroids = X[np.random.choice(range(n_samples), self.k), :]for
        i in range(self.max_iters):
            distances = np.sqrt(((X - self.centroids[:, np.newaxis])**2).sum(axis=2))self.labels
            = np.argmin(distances, axis=0)
            for j in range(self.k):
                self.centroids[j, :] = X[self.labels == j, :].mean(axis=0)def
    predict(self, X):
        distances = np.sqrt(((X - self.centroids[:, np.newaxis])**2).sum(axis=2))
        return np.argmin(distances, axis=0)
import pandas as pd
from sklearn.datasets import load_iris
iris = load_iris()
X = iris.data
kmeans = KMeans(k=3, random_state=42)
kmeans.fit(X)
print("Predicted Labels:\n", kmeans.labels)
print("Centroids:\n", kmeans.centroids)
```

Output:

```
Predicted Labels:
 [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 0 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 0 0 0 0 1 0 0 0 0
 0 0 1 1 0 0 0 0 1 0 1 0 1 0 0 1 1 0 0 0 0 0 1 0 0 0 0 1 0 0 0 1 0 0 0 1 0
 0 1]
Centroids:
 [[6.85384615 3.07692308 5.71538462 2.05384615]
 [5.88360656 2.74098361 4.38852459 1.43442623]
 [5.006      3.428      1.462      0.246     ]]
```