Table of Contents:

10		Given a bank customer, build a neural network-based classifier that can determine whether they will leave or not in the next 6 months. Dataset Description: The case study is from an open-source dataset from Kaggle. The dataset contains 10,000 sample points with 14 distinct features such as CustomerId, CreditScore, Geography, Gender, Age, Tenure, Balance, etc. Perform following steps: 1. Read the dataset. 2. Distinguish the feature and target set and divide the data set into training and test sets. 3. Normalize the train and test data. 4. Initialize and build the model. Identify the points of improvement and implement the same. 5. Print the accuracy score and confusion matrix (5 points)		
11		Implement Gradient Descent Algorithm to find the local minima of a function. For example, find the local minima of the function $y=(x+3)^2$ starting from the point x=2.		
12		Implement K-Nearest Neighbors algorithm on diabetes.csv dataset. Compute confusion matrix, accuracy, error rate, precision and recall on the given dataset.		
13		Implement K-Means clustering/ hierarchical clustering on sales_data_sample.csv dataset. Determine the number of clusters using the elbow method.		

1. Write a Program to Implement Tic-Tac-Toe game using Python.

Program Code:

```
def print_board(board):
    for row in board:
        print("|".join(row))
        print("."*5)
def check_winner(board,player):
    #check rows
    for row in board:
        if all(cell==player for cell in row):
            return True
    #check columns
    for col in range(3):
        if all(board[row][col]==player for row in range(3)):
            return True
    #check diagonals
    if all (board[i][i]==player for i in range(3) or all (board[i][2-i]==player
for i in range(3))):
        return True
    return False
def is_board_full(board):
    for row in board:
        for cell in row:
            if cell==" ":
                return False
    return True
def main():
    board=[[" " for _ in range(3)]for _ in range(3)]
    players=["X","O"]
    turn=0
    print("welcome to Tic-Tac-Toe")
    print_board(board)
    while True:
        player=players[turn%2]
        print(f"player{player}'s turn")
        row=int(input("enter row(0,1,or 2):"))
        col=int(input("enter column(0,1,or 2):"))

        if board[row][col]==" ":
            board[row][col]=player
            print_board(board)
            if check_winner(board,player):
                print(f"player{player}wins!")
                break
            elif is_board_full(board):
                print("it's draw!")
                break
            turn+=1
        else:
           print("That cell is already ocucupied.Try again")

if __name__=="__main__":
   main()
```

Output:

```
welcome to Tic-Tac-Toe
 ||
.....
 ||
.....
 ||
.....
playerX's turn
enter row(0,1,or 2):1
enter column(0,1,or 2):1
 ||
.....
 |X|
.....
 ||
.....
playerO's turn
enter row(0,1,or 2):2
enter column(0,1,or 2):2
 ||
.....
 |X|
.....
 | |O
.....
playerX's turn
enter row(0,1,or 2):1
enter column(0,1,or 2):2
 | |
.....
 |X|X
.....
 | |O
.....
playerO's turn
enter row(0,1,or 2):1
enter column(0,1,or 2):0
 | |
.....
O|X|X
.....
 | |O
.....
playerX's turn
enter row(0,1,or 2):0
enter column(0,1,or 2):1
 |X|
.....
O|X|X
.....
 | |O
.....
playerO's turn
enter row(0,1,or 2):2
enter column(0,1,or 2):1
 |X|
.....
O|X|X
.....
 |O|O
.....
playerX's turn
```

```
enter row(0,1,or 2):2
enter column(0,1,or 2):1
That cell is already ocucupied.Try again
playerX's turn
enter row(0,1,or 2):2
enter column(0,1,or 2):0
 |X|
.....
O|X|X
.....
X|O|O
.....
playerO's turn
enter row(0,1,or 2):0
enter column(0,1,or 2):2
 |X|O
.....
O|X|X
.....
X|O|O
.....
playerX's turn
enter row(0,1,or 2):0
enter column(0,1,or 2):0
X|X|O
.....
O|X|X
.....
X|O|O
.....
it's draw!
```

2. Write a Program to Implement Water-Jug problem using Python.

Program Code:

```
x=0
y=0
m=4
n=3
print("initial state =(0,0)")
print("capacitioes =(4,3)")
print("goal state =(2,y)")
while(x!=2):
    r=int(input("enter the rule:"))
    if(r==1):
        x=m
    elif(r==2):
        y=n
    elif(r==3):
        x=0
    elif(r==4):
        y=0
    elif(r==5):
        t=n-y
        y=n
        x-=t
    elif(r==6):
        t=m-x
        x=m
        y-=t
    elif(r==7):
        y+=x
        x=0
    elif(r==8):
        x+=y
        y=0
    else:
        print("invalid rule")
print(x,y)
```

Output:

```
initial state =(0,0)
capacitioes =(4,3)
goal state =(2,y)
enter the rule:2
0 3
enter the rule:8
3 0
enter the rule:2
3 3
enter the rule:6
4 2
enter the rule:3
0 2
enter the rule:8
2 0
```

3. Write a Program to implement 8-Puzzle problem using Python.

Program Code:

```
import numpy as np
import pandas as pd
import os
def bfs(src,target):
    queue=[]
    queue.append(src)
    exp=[]
    while len(queue)>0:
        source=queue.pop(0)
        exp.append(source)
        print(source)
        if source==target:
            print("success")
            return
        poss_moves_to_do=[]
        poss_moves_to_do=possible_moves(source,exp)
        for move in poss_moves_to_do:
            if move not in exp and move not in queue:
                queue.append(move)

def possible_moves(state,visited_states):
    b=state.index(0)
    d=[]
    if b not in [0,1,2]:
        d.append('u')
    if b not in[6,7,8]:
        d.append('d')
    if b not in[0,3,6]:
        d.append('l')
    if b not in[2,5,8]:
        d.append('r')
    poss_moves_it_can=[]
    for i in d:
        poss_moves_it_can.append(gen(state,i,b))
    return[move_it_can for move_it_can in poss_moves_it_can if move_it_can not in visited_states]

def gen(state,m,b):
    temp=state.copy()
    if m=='d':
        temp[b+3],temp[b]=temp[b],temp[b+3]
    if m=='u':
        temp[b-3],temp[b]=temp[b],temp[b-3]
    if m=='l':
        temp[b-1],temp[b]=temp[b],temp[b-1]
    if m=='r':
        temp[b+1],temp[b]=temp[b],temp[b+1]
    return temp
src=[1,2,3,4,5,6,0,7,8]
target=[1,2,3,4,5,6,7,8,0]
bfs(src,target)
```

Output:

```
[1, 2, 3, 4, 5, 6, 0, 7, 8]
[1, 2, 3, 0, 5, 6, 4, 7, 8]
[1, 2, 3, 4, 5, 6, 7, 0, 8]
[0, 2, 3, 1, 5, 6, 4, 7, 8]
[1, 2, 3, 5, 0, 6, 4, 7, 8]
[1, 2, 3, 4, 0, 6, 7, 5, 8]
[1, 2, 3, 4, 5, 6, 7, 8, 0]
success
```

4. Write a Program to Implement AO* Algorithm using Python.

Program Code:

```
class Node:
    def __init__(self, name, heuristic, is_goal=False):
        self.name = name
        self.heuristic = heuristic
        self.is_goal = is_goal
        self.children = []
        self.marked = False

    def add_children(self, *child_nodes):
        self.children.append(child_nodes)



def ao_star(node, path_cost):
    if node.is_goal:
        node.marked = True
        return node.heuristic

    if not node.children:
        return node.heuristic

    min_cost = float("inf")
    best_child_set = None

    for child_set in node.children:
        cost = 0
        for child in child_set:
            cost += child.heuristic + path_cost

            if not child.marked:
                cost += ao_star(child, path_cost)

        if cost < min_cost:
            min_cost = cost
            best_child_set = child_set

    if best_child_set:
        node.marked = True
        for child in best_child_set:
            child.marked = True

    return min_cost



A = Node('A', 6)
B = Node('B', 4)
C = Node('C', 2)
D = Node('D', 0, is_goal=True)
E = Node('E', 0, is_goal=True)
F = Node('F', 0, is_goal=True)
G = Node('G', 0, is_goal=True)

A.add_children([B, C])  # Use lists for child sets
```

```
B.add_children([D], [E])  # Use lists for child sets
C.add_children([F, G])  # Use lists for child sets

print("Running AO* algorithm...")
total_cost = ao_star(A, path_cost=1)
print("Optimal solution path cost:", total_cost)



def display_solution_path(node):
    if node.marked:
        print(node.name, end=" ")
        for child_set in node.children:
            for child in child_set:
                display_solution_path(child)



print("\nOptimal solution path:")
display_solution_path(A)
```

Output:

```
Expanding Node: A
Expanding Node: B
Nodes which give optimal cost are
F ->9
 optimal cost is :: 9
```

5. Predict the price of the Uber ride from a given pickup point to the agreed drop-off location.

Perform following tasks:

1. Pre-process the dataset.
2. Identify outliers.
3. Check the correlation.
4. Implement linear regression and random forest regression models.
5. Evaluate the models and compare their respective scores like R2, RMSE, etc.

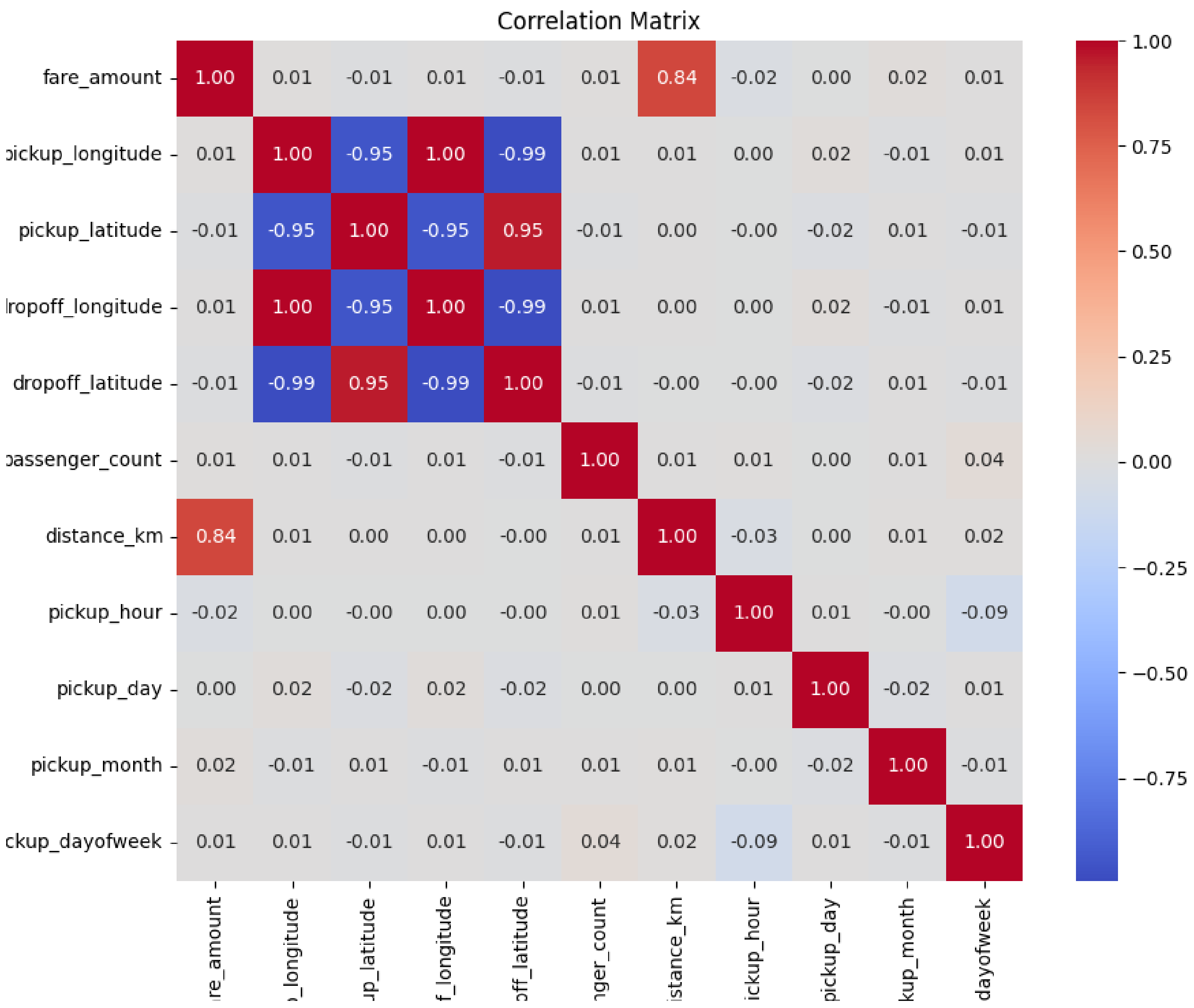
Program Code:

```
import pandas as pd

# Load the dataset to examine its structure

file_path = 'uber.csv'

uber_data = pd.read_csv(file_path)

# Display the first few rows and a summary of the data

uber_data.head(), uber_data.info(), uber_data.describe(include='all')


import numpy as np

from math import radians, cos, sin, sqrt, atan2

# Drop unnecessary columns

uber_data = uber_data.drop(columns=['Unnamed: 0', 'key'])

# Handle missing values by dropping rows with any missing coordinate

uber_data = uber_data.dropna(subset=['dropoff_longitude',
'dropoff_latitude'])

# Define a function to calculate haversine distance

def haversine(lat1, lon1, lat2, lon2):

    R = 6371  # Earth's radius in kilometers

    lat1, lon1, lat2, lon2 = map(radians, [lat1, lon1, lat2, lon2])
    dlat = lat2 - lat1
    dlon = lon2 - lon1
    a = sin(dlat / 2)**2 + cos(lat1) * cos(lat2) * sin(dlon / 2)**2
    c = 2 * atan2(sqrt(a), sqrt(1 - a))
    return R * c
# Calculate distance and add it as a feature
uber_data['distance_km'] = uber_data.apply(
    lambda row: haversine(row['pickup_latitude'], row['pickup_longitude'],
                          row['dropoff_latitude'], row['dropoff_longitude']), axis=1)
# Convert pickup_datetime to datetime and extract features
uber_data['pickup_datetime'] =
pd.to_datetime(uber_data['pickup_datetime'])
uber_data['pickup_hour'] = uber_data['pickup_datetime'].dt.hour
uber_data['pickup_day'] = uber_data['pickup_datetime'].dt.day
uber_data['pickup_month'] = uber_data['pickup_datetime'].dt.month
uber_data['pickup_dayofweek'] =
uber_data['pickup_datetime'].dt.dayofweek
# Drop original datetime column
uber_data = uber_data.drop(columns=['pickup_datetime'])
# Remove obvious outliers
uber_data = uber_data[
    (uber_data['fare_amount'] > 0) & (uber_data['fare_amount'] < 500) &
    (uber_data['distance_km'] > 0) & (uber_data['distance_km'] < 100) &
    (uber_data['passenger_count'] > 0) & (uber_data['passenger_count'] <=
6)
]
# Display the cleaned data summary
uber_data.info(), uber_data.describe()
```

```
# %%
import seaborn as sns
import matplotlib.pyplot as plt
# Calculate the correlation matrix
correlation_matrix = uber_data.corr()
# Plot the heatmap for correlations
plt.figure(figsize=(10, 8))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm',
fmt=".2f")
plt.title('Correlation Matrix')
plt.show()
# %%
# Import necessary libraries
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import r2_score, mean_squared_error
import numpy as np
# Select features and target
features = ['distance_km', 'pickup_hour', 'passenger_count']
target = 'fare_amount'
X = uber_data[features]
y = uber_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)
# Initialize models
linear_model = LinearRegression()
```

```
rf_model = RandomForestRegressor(random_state=42, n_estimators=100)
# Train models
linear_model.fit(X_train, y_train)
rf_model.fit(X_train, y_train)
# Make predictions
linear_predictions = linear_model.predict(X_test)
rf_predictions = rf_model.predict(X_test)
# Evaluate models
linear_r2 = r2_score(y_test, linear_predictions)
linear_rmse = np.sqrt(mean_squared_error(y_test, linear_predictions))
rf_r2 = r2_score(y_test, rf_predictions)
rf_rmse = np.sqrt(mean_squared_error(y_test, rf_predictions))
# Print results
print("Linear Regression:")
print(f"R² Score: {linear_r2:.2f}")
print(f"RMSE: {linear_rmse:.2f}")
print("\nRandom Forest Regression:")
print(f"R² Score: {rf_r2:.2f}")
print(f"RMSE: {rf_rmse:.2f}")
```

Output:



	re_amount	_longitude	up_latitude	f_longitude	off_latitude	nger_count	istance_km	ickup_hour	pickup_day	kup_month	dayofweek
fare_amount	1.00	0.01	-0.01	0.01	-0.01	0.01	0.84	-0.02	0.00	0.02	0.01
ickup_longitude	0.01	1.00	-0.95	1.00	-0.99	0.01	0.01	0.00	0.02	-0.01	0.01
pickup_latitude	-0.01	-0.95	1.00	-0.95	0.95	-0.01	0.00	-0.00	-0.02	0.01	-0.01
ropoff_longitude	0.01	1.00	-0.95	1.00	-0.99	0.01	0.00	0.00	0.02	-0.01	0.01
dropoff_latitude	-0.01	-0.99	0.95	-0.99	1.00	-0.01	-0.00	-0.00	-0.02	0.01	-0.01
assenger_count	0.01	0.01	-0.01	0.01	-0.01	1.00	0.01	0.01	0.00	0.01	0.04
distance_km	0.84	0.01	0.00	0.00	-0.00	0.01	1.00	-0.03	0.00	0.01	0.02
pickup_hour	-0.02	0.00	-0.00	0.00	-0.00	0.01	-0.03	1.00	0.01	-0.00	-0.09
pickup_day	0.00	0.02	-0.02	0.02	-0.02	0.00	0.00	0.01	1.00	-0.02	0.01
pickup_month	0.02	-0.01	0.01	-0.01	0.01	0.01	0.01	-0.00	-0.02	1.00	-0.01
ckup_dayofweek	0.01	0.01	-0.01	0.01	-0.01	0.04	0.02	-0.09	0.01	-0.01	1.00

6. Implement and demonstrate the FIND-S algorithm for finding the most specific hypothesis based on a given set of training data samples. Read the training data from a .CSV file.

Data Set:

```
sky ,    airTemp ,   humidity ,   wind ,     water ,   forecast ,   enjoySport
sunny,  warm    ,    normal    ,  strong   , warm ,    same        ,no
sunny , warm    ,    high       ,  strong   , warm,    same     ,    yes
Rainy  , cold ,          high    ,     strong  ,  warm,   change,        no
sunny , warm ,       high         ,    strong  ,  cool ,  same ,           yes
```

Program Code:

```
import csv
num_attribute=6
a=[]
with open('enjoysport.csv','r') as file:
   reader=csv.reader(file)
   a=list(reader)
hypothesis=a[1][:-1]
for i in a:
   if i[-1]=='yes':
      for j in range(num_attribute):
         if i[j]!=hypothesis[j]:
            hypothesis[j]='?'
print(hypothesis)
print("\n the maximally specific hyothesis for a given training
examples\n")
print(hypothesis)
```

Output:

```
['1', 'Sunny', 'Warm', 'Normal', 'Strong', 'Warm', 'Same']

 the maximally specific hyothesis for a given training examples

['1', 'Sunny', 'Warm', 'Normal', 'Strong', 'Warm', 'Same']
```

7. For a given set of training data examples stored in a .CSV file, implement and demonstrate the Candidate-Elimination algorithm to output a description of the set of all hypotheses consistent with the training examples.

Data Set:

```
sky ,    airTemp ,   humidity ,   wind ,     water ,   forecast ,   enjoySport
sunny,  warm    ,    normal    ,  strong   , warm ,   same       ,no
sunny , warm    ,    high      ,  strong   , warm,    same    ,   yes
Rainy  , cold ,        high    ,   strong  ,  warm,   change,        no
sunny , warm ,       high        ,   strong  ,  cool ,  same ,          yes
```

Program Code:

```
import csv
with open("trainingdata.csv") as f:
    csv_file =csv.reader(f)
    data=list(csv_file)
    s=data[1][:-1]
    print(s)
    g=[['?' for i in range(len(s))] for j in range(len(s))]
    print(g)
    for i in data:
        if i[-1]=="yes":
            for j in range(len(s)):
                if i[j]!=s[j]:
                    s[j]='?'
                    g[j][j]='?'
        elif i[-1]=="no":
            for j in range(len(s)):
                if i[j]!=s[j]:
                    g[j][j]=s[j]
                else:
                    g[j][j]="?"
                    print("\n step" + str(data.index(i)+1)+ "of candidate
elimination algorithm")
                    print(s)
        print(g)
        gh=[]
        for i in g:
            for j in i:
                if j != '?':
                    gh.append(i)
                break
        print("\n Final Specific Hypothesis:\n",s)
        print("\n Final General Hypothesis:\n",gh)
```

Output:

```
['Sunny', 'Warm', 'High', 'Strong', 'Warm', 'Same']
[['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?',
'?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?']]
[['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?',
'?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?']]

 Final Specific Hypothesis:
 ['Sunny', 'Warm', 'High', 'Strong', 'Warm', 'Same']

 Final General Hypothesis:
 []
[['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?',
'?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?']]

 Final Specific Hypothesis:
 ['Sunny', 'Warm', 'High', 'Strong', 'Warm', 'Same']

 Final General Hypothesis:
 []
```

```
[['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?',
'?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?']]

 Final Specific Hypothesis:
 ['Sunny', 'Warm', 'High', 'Strong', 'Warm', 'Same']

 Final General Hypothesis:
 []
[['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?',
'?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?']]

 Final Specific Hypothesis:
 ['Sunny', 'Warm', 'High', 'Strong', 'Warm', 'Same']

 Final General Hypothesis:
 [[sunny,’?’,’?’,’?’,’?’,’?’],[‘?’,warm,’?’,’?’,’?’,’?’]]
```

8. Write a program to demonstrate the working of the decision tree basedID3 algorithm.

Data Set:

Outlook ,Temperature,	Humidity,	Wind,	PlayTennis	
sunny,	hot,	high,	weak,	no
sunny,	hot,	high,	weak,	no
overcast,	hot,	high,	weak,	yes
rain,	mild,	high,	weak,	yes
rain,	cool,	normal,	weak,	yes
rain,	mild,	normal,	strong,	yes

Program Code:

```
import pandas as pd
from sklearn.tree import DecisionTreeClassifier
from sklearn.preprocessing import LabelEncoder
pd.options.mode.copy_on_write=True
data=pd.read_csv('m.csv')
print('first 5 values of data are:\n',data.head())
x=data.iloc[:,:-1]
y=data.iloc[:,-1]
print("\n values of x:\n",x.head())
print('\n first 5 values of train output:\n',y.head())
le_Outlook=LabelEncoder()
le_Temperature=LabelEncoder()
le_Humidity=LabelEncoder()
le_Windy=LabelEncoder()
le_PlayTennis=LabelEncoder()
x['Outlook']=le_Outlook.fit_transform(x['Outlook'])
x['Temperature']=le_Temperature.fit_transform(x['Temperature'])
x['Humidity']=le_Humidity.fit_transform(x['Humidity'])
x['Windy']=le_Windy.fit_transform(x['Windy'])
x.columns=['Outlook','Temperature','Humidity','Windy']
y=le_PlayTennis.fit_transform(y)
print('\nnow the rain data is:\n',x.head())
print('\n now the train output is:\n',y)
```

```
classifier=DecisionTreeClassifier()
classifier.fit(x,y)
inp=["Overcast","Cool","Normal","Strong"]
inp_df=pd.DataFrame([inp],columns=['Outlook','Temperature','Humdity','
Windy'])
inp_df['Outlook']=le_Outlook.transform(inp_df['Outlook'])
inp_df['Temperature']=le_Temperature.transform(inp_df['Temperature'])
inp_df['Humidity']=le_Humidity.transform(inp_df['Humidity'])
inp_df['Windy']=le_Windy.transform(inp_df['Windy'])
y_pred=classifier.predict(inp_df)
predicted_label=le_PlayTennis.inverse_transform(y_pred)[0]
print("\n for input {0} ,we obtain {1} ".format(inp,predicted_label))
```

Output:

```
first 5 values of data are:
    Outlook Temperature Humidity   Windy PlayTennis
0     Sunny         Hot     High    Weak         No
1     Sunny         Hot     High  Strong         No
2  Overcast         Hot     High    Weak        Yes
3      Rain        Mild     High    Weak        Yes
4      Rain        Cool   Normal    Weak        Yes
 values of x:
    Outlook Temperature Humidity   Windy
0     Sunny         Hot     High    Weak
1     Sunny         Hot     High  Strong
2  Overcast         Hot     High    Weak
3      Rain        Mild     High    Weak
4      Rain        Cool   Normal    Weak

 first 5 values of train output:
 0     No
1     No
2    Yes
3    Yes
4    Yes
Name: PlayTennis, dtype: object
```

```
now the rain data is:
   Outlook  Temperature  Humidity  Windy
0        2            1         0      1
1        2            1         0      0
2        0            1         0      1
3        1            2         0      1
4        1            0         1      1
 now the train output is:
 [0 0 1 1 1 0 1 0 1 1 1 1 1 0]

 for input ['Overcast', 'Cool', 'Normal', 'Strong'] ,we obtain Yes
```

9. Classify the email using the binary classification method. Email Spam detection has two states: a) Normal State – Not Spam, b) Abnormal State – Spam. Use K-Nearest Neighbors and Support Vector Machine for classification. Analyze their performance.

Program Code:

```
# %%
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder, StandardScaler
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score, precision_score, recall_score,
f1_score, classification_report

# Load the dataset
data = pd.read_csv('emails.csv')

# Inspect the data (optional)
print("Dataset head:\n", data.head())
print("Dataset description:\n", data.describe())

# Preprocessing
# Initialize the label encoder
label_encoder = LabelEncoder()

# Encode the target labels: 1 for spam, 0 for not spam
data['Prediction'] = label_encoder.fit_transform(data['Prediction'])

# Drop non-numeric columns that are not features
data = data.drop(columns=['Email No.'])

# Features and target variable
X = data.drop(columns=['Prediction'])  # Features
y = data['Prediction']  # 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 the features for optimal performance of KNN and SVM
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# K-Nearest Neighbors (KNN) Model
knn_model = KNeighborsClassifier(n_neighbors=5)
knn_model.fit(X_train, y_train)
y_pred_knn = knn_model.predict(X_test)

# Support Vector Machine (SVM) Model
svm_model = SVC(kernel='linear', random_state=42)
svm_model.fit(X_train, y_train)
y_pred_svm = svm_model.predict(X_test)

# Evaluate models
def evaluate_model(y_test, y_pred, model_name):
    accuracy = accuracy_score(y_test, y_pred)
    precision = precision_score(y_test, y_pred)
    recall = recall_score(y_test, y_pred)
    f1 = f1_score(y_test, y_pred)
    print(f"\n{model_name} Performance:")
    print(f"Accuracy: {accuracy:.2f}")
    print(f"Precision: {precision:.2f}")
    print(f"Recall: {recall:.2f}")
    print(f"F1 Score: {f1:.2f}")
    print("\nClassification Report:\n", classification_report(y_test, y_pred))

# Print evaluation results for KNN and SVM
evaluate_model(y_test, y_pred_knn, "K-Nearest Neighbors (KNN)")
evaluate_model(y_test, y_pred_svm, "Support Vector Machine (SVM)")
```

Output:

```
K-Nearest Neighbours (KNN) Performance:
Accuracy: 0.85
Precision: 0.66
Recall: 0.95
F1 Score: 0.78

Classification Report:
              precision    recall  f1-score   support

           0       0.98      0.80      0.88       739
           1       0.66      0.95      0.78       296

    accuracy                           0.85      1035
   macro avg       0.82      0.88      0.83      1035
weighted avg       0.89      0.85      0.85      1035


Support Vector Machine (SVM) Performance:
Accuracy: 0.95
Precision: 0.90
Recall: 0.92
F1 Score: 0.91

Classification Report:
              precision    recall  f1-score   support

           0       0.97      0.96      0.96       739
           1       0.90      0.92      0.91       296

    accuracy                       0.95      1035
   macro avg       0.93      0.94      0.94      1035
weighted avg       0.95      0.95      0.95      1035
```

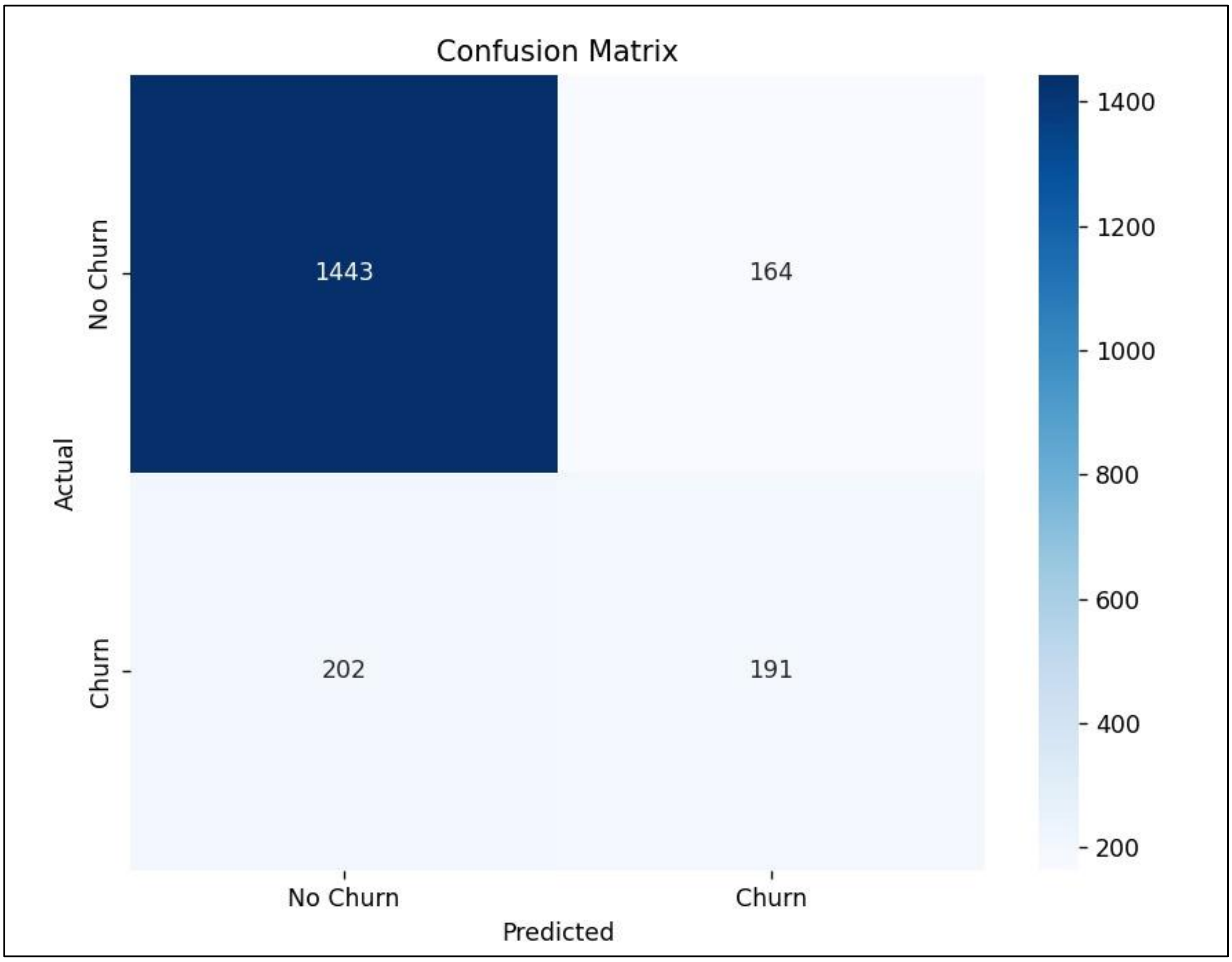
10. Given a bank customer, build a neural network-based classifier that can determine whether they will leave or not in the next 6 months. Dataset Description: The case study is from an open-source dataset from Kaggle. The dataset contains 10,000 sample points with 14 distinct features such as CustomerId, CreditScore, Geography, Gender, Age, Tenure, Balance, etc. Perform following steps:

1. **Read the dataset.**
2. **Distinguish the feature and target set and divide the data set into training and test sets.**
3. **Normalize the train and test data.**
4. **Initialize and build the model. Identify the points of improvement and implement the same. 5. Print the accuracy score and confusion matrix (5 points)**

Program code:

```
# Import required libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score, confusion_matrix
from sklearn.neural_network import MLPClassifier

# 1. Read the dataset
url = "C:/Users/palla/OneDrive/Desktop/Bank/Churn_Modelling.csv"  #
Local path to your dataset
dataset = pd.read_csv(url)

# 2. Check for missing values (optional)
print(dataset.isnull().sum())  # To check if there are any missing values

# 3. Handle categorical variables (Geography, Gender)
# Use one-hot encoding to convert categorical variables to numeric values
dataset = pd.get_dummies(dataset, columns=['Geography', 'Gender'],
drop_first=True)

# 4. Distinguish the feature and target set
X = dataset.drop(columns=['Exited', 'CustomerId', 'Surname'])  # Dropping
non-useful columns like 'CustomerId' and 'Surname'
y = dataset['Exited']  # 'Exited' is the target variable (1 = Churn, 0 = No
Churn)

# 5. Split the data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# 6. Normalize the train and test data
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# 7. Initialize and build the model using MLPClassifier
model = MLPClassifier(hidden_layer_sizes=(128, 64, 32), max_iter=1000,
activation='relu', solver='adam', random_state=42)

# 8. Train the model
model.fit(X_train_scaled, y_train)

# 9. Predict on the test set
y_pred = model.predict(X_test_scaled)

# 10. Print the accuracy score
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy * 100:.2f}%")

# 11. Print the confusion matrix
cm = confusion_matrix(y_test, y_pred)
print("Confusion Matrix:")
print(cm)

# 12. Visualization of the confusion matrix
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=['No
Churn', 'Churn'], yticklabels=['No Churn', 'Churn'])
plt.title('Confusion Matrix')
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.show()
```

Output:



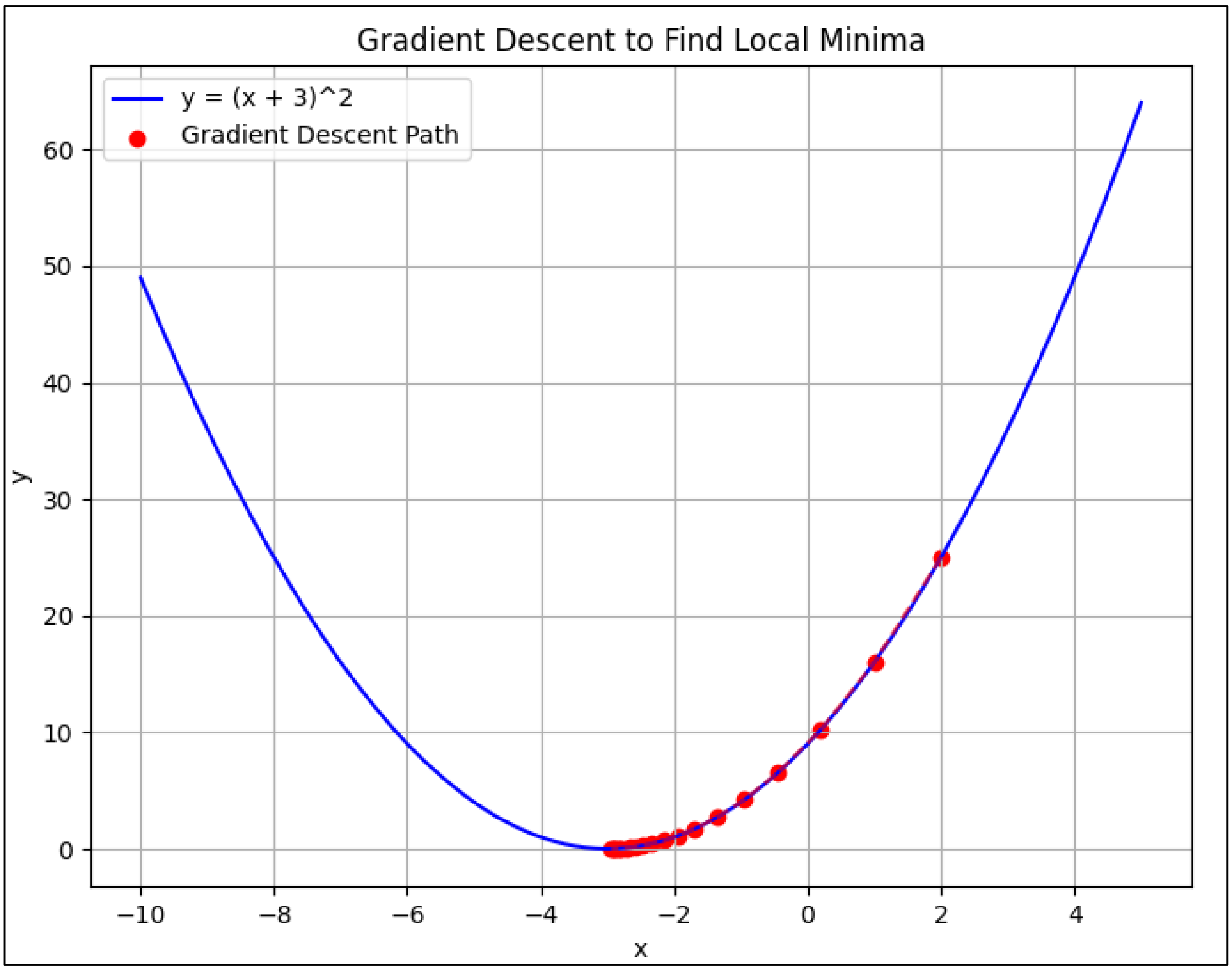
11. Implement Gradient Descent Algorithm to find the local minima of a function. For example, find the local minima of the function y=(x+3)² starting from the point x=2.

Program Code:

```
# %%
import numpy as np
import matplotlib.pyplot as plt
# %%
def func(x):
    return (x + 3) ** 2
# %%
def grad_func(x):
    return 2 * (x + 3)
# %%
# Gradient Descent Algorithm
def gradient_descent(starting_point, learning_rate, num_iterations):
    x = starting_point
    x_history = [x]  # Keep track of x values for plotting
    for _ in range(num_iterations):
        # Calculate gradient
        gradient = grad_func(x)
        # Update x by moving against the gradient
        x = x - learning_rate * gradient
        # Append the new x value to history
        x_history.append(x)
   return x, x_history
# %%
# Parameters for gradient descent
```

```
starting_point = 2  # Starting point x = 2
learning_rate = 0.1  # Learning rate
num_iterations = 20  # Number of iterations
# %%
final_x, x_history = gradient_descent(starting_point, learning_rate,
num_iterations)
# %%
print(f"Final x value after {num_iterations} iterations: {final_x}")
print(f"Minimum value of the function: {func(final_x)}")
# %%
x_vals = np.linspace(-10, 5, 100)
y_vals = func(x_vals)
# %%
plt.figure(figsize=(8, 6))
plt.plot(x_vals, y_vals, label='y = (x + 3)^2', color='blue')
plt.scatter(x_history, [func(x) for x in x_history], color='red',
label='Gradient Descent Path')
plt.plot(x_history, [func(x) for x in x_history], color='red',
linestyle='dashed', alpha=0.6)
plt.title('Gradient Descent to Find Local Minima')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.grid(True)
plt.show()
```

Output:


Gradient Descent to Find Local Minima
y = (x + 3)^2
Gradient Descent Path
y
x
60
50
40
30
20
10
0
−10
−8
−6
−4
−2
0
2
4

12. Implement K-Nearest Neighbors algorithm on diabetes.csv dataset. Compute confusion matrix, accuracy, error rate, precision and recall on the given dataset.

Program Code:

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import confusion_matrix, accuracy_score,
precision_score, recall_score

# Step 1: Load the dataset
dataset = pd.read_csv("diabetes.csv")

# Step 2: Preprocess the data (fill missing values if any)
dataset.fillna(dataset.median(), inplace=True)

# Step 3: Define the feature set X and target variable y
X = dataset.drop("Outcome", axis=1)  # Features
y = dataset["Outcome"]  # Target variable

# Step 4: Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=42)

# Step 5: Initialize and train the KNN model
knn = KNeighborsClassifier(n_neighbors=5)  # You can experiment with
different k-values
knn.fit(X_train, y_train)

# Step 6: Make predictions
y_pred = knn.predict(X_test)

# Step 7: Compute evaluation metrics
cm = confusion_matrix(y_test, y_pred)
accuracy = accuracy_score(y_test, y_pred)
error_rate = 1 - accuracy
```

```
precision = precision_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)

# Output results
print("Confusion Matrix:\n", cm)
print("Accuracy:", accuracy)
print("Error Rate:", error_rate)
print("Precision:", precision)
print("Recall:", recall)
```

Output:

```
Confusion Matrix:
        [[114     37]
         [35       45]]
Accuracy: 0.6883116883110883
Error Rate: 0.3116883116883117
Precision: 0.5487804878048781
Recall: 0.5625
```

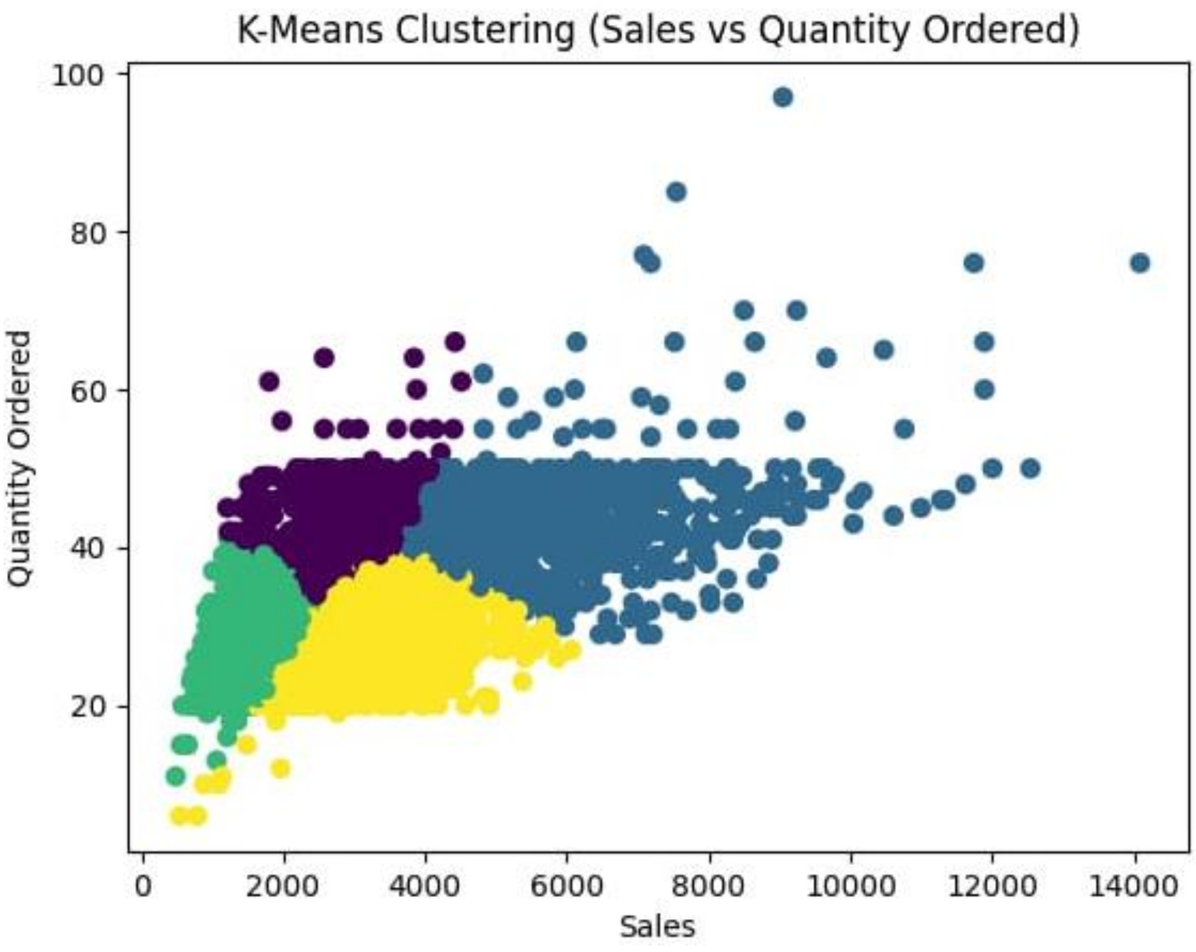
13. Implement K-Means clustering/ hierarchical clustering on sales_data_sample.csv dataset. Determine the number of clusters using the elbow method.

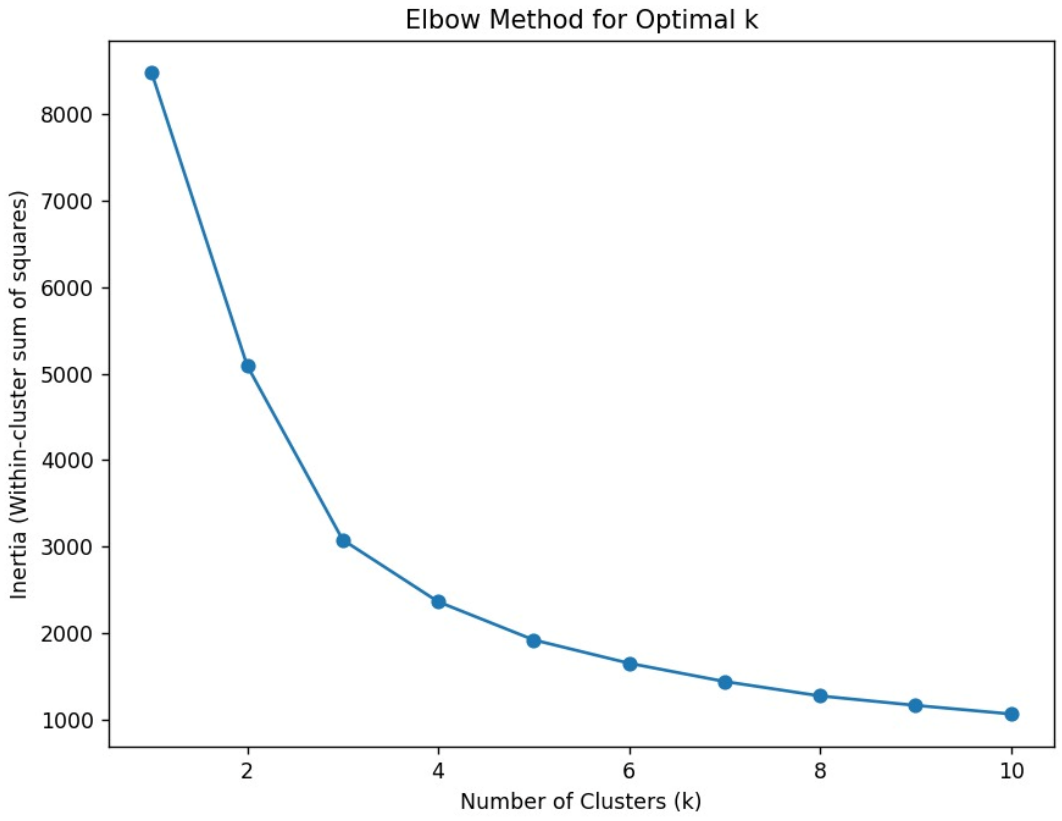
Program code:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler

# Step 1: Load and Preprocess the Data
url = 'C:/Users/palla/OneDrive/Desktop/pro/k
means/sales_data_sample.csv'

# Read the file with proper encoding
data = pd.read_csv(url,encoding='latin1')    # Removed 'errors'
argument

# Display column names to verify
print("Columns in Dataset:")
print(data.columns)

# Selecting relevant columns for clustering
# Update these column names based on your dataset structure
data_selected = data[['QUANTITYORDERED', 'PRICEEACH',
'SALES']]  # Replace with actual column names

# Handle missing values by filling with the mean
data_selected = data_selected.fillna(data_selected.mean())

# Normalize the data (important for K-Means)
scaler = StandardScaler()
data_normalized = scaler.fit_transform(data_selected)

# Step 2: Determine the Optimal Number of Clusters using the Elbow
Method (for K-Means)
```

```
inertia = []
for k in range(1, 11):  # Trying k values from 1 to 10
    kmeans = KMeans(n_clusters=k, random_state=42)
    kmeans.fit(data_normalized)
    inertia.append(kmeans.inertia_)

# Plot the inertia to visualize the Elbow point
plt.figure(figsize=(8, 6))
plt.plot(range(1, 11), inertia, marker='o')
plt.title('Elbow Method for Optimal k')
plt.xlabel('Number of Clusters (k)')
plt.ylabel('Inertia (Within-cluster sum of squares)')
plt.show()

# From the plot, look for the "elbow" point where inertia starts
decreasing slowly
optimal_k = 4  # Replace with the value determined from the plot

# Step 3: K-Means Clustering
kmeans = KMeans(n_clusters=optimal_k, random_state=42)
kmeans_labels = kmeans.fit_predict(data_normalized)

# Add the K-Means cluster labels to the dataset
data['KMeans_Cluster'] = kmeans_labels

# Step 4: Visualize the Clusters
plt.scatter(data['SALES'],                data['QUANTITYORDERED'],
c=data['KMeans_Cluster'], cmap='viridis')
plt.title('K-Means Clustering (Sales vs Quantity Ordered)')
plt.xlabel('Sales')
plt.ylabel('Quantity Ordered')
plt.show()
```

Output:


K-Means Clustering (Sales vs Quantity Ordered)
Quantity Ordered
Sales
100
80
60
40
20
0
2000
4000
6000
8000
10000
12000
14000


Elbow Method for Optimal k
Inertia (Within-cluster sum of squares)
Number of Clusters (k)
8000
7000
6000
5000
4000
3000
2000
1000
2
4
6
8
10