Group A

1.Write a program to calculate Fibonacci numbers and find its step count.

```
n_terms = int(input ("How many terms the user wants to print? "))

# First two terms
n_1 = 0
n_2 = 1
count = 0

# Now, we will check if the number of terms is valid or not
if n_terms <= 0:
    print ("Please enter a positive integer, the given number is not valid")
# if there is only one term, it will return n_1
elif n_terms == 1:
    print ("The Fibonacci sequence of the numbers up to", n_terms, ": ")
    print(n_1)
# Then we will generate Fibonacci sequence of number
else:
    print ("The fibonacci sequence of the numbers is:")
    while count < n_terms:
        print(n_1)
        nth = n_1 + n_2
        # At last, we will update values
        n_1 = n_2
        n_2 = nth
        count += 1
```

Implement job sequencing with deadlines using a greedy method.

```
def printjobschedule(array, t):
m = len(array)
# Sort all jobs accordingly
for j in range(m):
for q in range(m - 1 - j):
if array[q][2] < array[q + 1][2]:
array[q], array[q + 1] = array[q + 1], array[q]
res = [False] * t
# To store result
job = ['-1'] * t
for q in range(len(array)):
# Find a free slot
for q in range(min(t - 1, array[q][1] - 1), -1, -1):
if res[q] is False:
res[q] = True
job[q] = array[q][0]
break
# print
print(job)
# Driver
array = [['a', 7, 202],
['b', 5, 29],
['c', 6, 84],
['d', 1, 75],
['e', 2, 43]]
print("Maximum profit sequence of jobs is- ")
printjobschedule(array, 3)
```

3. **Write a program to solve a fractional Knapsack problem using a greedy method.**

```
class Solution:
  def solve(self, weights, values, capacity):
    res = 0
    for pair in sorted(zip(weights, values), key=lambda x: - x[1]/x[0]):
      if not bool(capacity):
        break
      if pair[0] > capacity:
        res += int(pair[1] / (pair[0] / capacity))
        capacity = 0
      elif pair[0] <= capacity:
        res += pair[1]
        capacity -= pair[0]
    return int(res)

ob = Solution()
weights = [6, 7, 3]
values = [110, 120, 2]
capacity = 10
print(ob.solve(weights, values, capacity))
```

i/p : [6, 7, 3],[110, 120, 2],10

o/p : 230

4. **Write a program to solve a 0-1 Knapsack problem using dynamic programming or branch and bound strategy.**

```
def knapSack(W, wt, val, n):

    Base Case
    if n == 0 or W == 0:
        return 0



    if (wt[n-1] > W):
        return knapSack(W, wt, val, n-1)


    else:
        return max(
            val[n-1] + knapSack(
                W-wt[n-1], wt, val, n-1),
            knapSack(W, wt, val, n-1))


val = [60, 100, 120]
wt = [10, 20, 30]
W = 50
n = len(val)
print knapSack(W, wt, val, n)
```

```
O/p: 220
```

5. Python3 program to solve N Queen Problem using

backtracking '''

```
result = []

# A utility function to print solution
```

```
''' A utility function to check if a queen can
be placed on board[row][col]. Note that this
function is called when "col" queens are
already placed in columns from 0 to col -1.
So we need to check only left side for
attacking queens '''


def isSafe(board, row, col):

    # Check this row on left side
    for i in range(col):
        if (board[row][i]):
            return False

    # Check upper diagonal on left side
    i = row
    j = col
    while i >= 0 and j >= 0:
        if(board[i][j]):
            return False
        i -= 1
        j -= 1

    # Check lower diagonal on left side
    i = row
    j = col
    while j >= 0 and i < 4:
        if(board[i][j]):
            return False
        i = i + 1
        j = j - 1

    return True


''' A recursive utility function to solve N
Queen problem '''


def solveNQUtil(board, col):
    ''' base case: If all queens are placed
    then return true '''
```

```
    if (col == 4):
        v = []
        for i in board:
          for j in range(len(i)):
            if i[j] == 1:
              v.append(j+1)
        result.append(v)
        return True

    ''' Consider this column and try placing
    this queen in all rows one by one '''
    res = False
    for i in range(4):

        ''' Check if queen can be placed on
        board[i][col] '''
        if (isSafe(board, i, col)):

            # Place this queen in board[i][col]
            board[i][col] = 1

            # Make result true if any placement
            # is possible
            res = solveNQUtil(board, col + 1) or res

            ''' If placing queen in board[i][col]
            doesn't lead to a solution, then
            remove queen from board[i][col] '''
            board[i][col] = 0  # BACKTRACK

    ''' If queen can not be place in any row in
        this column col then return false '''
    return res


''' This function solves the N Queen problem using
Backtracking. It mainly uses solveNQUtil() to
solve the problem. It returns false if queens
cannot be placed, otherwise return true and
prints placement of queens in the form of 1s.
Please note that there may be more than one
solutions, this function prints one of the
feasible solutions.'''
```

```
def solveNQ(n):
    result.clear()
    board = [[0 for j in range(n)]
             for i in range(n)]
    solveNQUtil(board, 0)
    result.sort()
    return result


# Driver Code
n = 4
res = solveNQ(n)
print(res)

# This code is contributed by YatinGupta
```

Machine learning

1. 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.

```
#Importing the dataset

df = pd.read_csv('../input/uber-fares-dataset/uber.csv')

df.drop(['Unnamed: 0','key'], axis=1, inplace=True)
display(df.head())

target = 'fare_amount'
features = [i for i in df.columns if i not in [target]]

print('\n\033[1mInference:\033[0m The Datset consists of {} features & {} samples.'.format(df.shape[1], df.shape[0]))
```

	fare_amount	pickup_datetime	pickup_longitude	pickup_latitude	dropoff_longitude	dropoff
0	7.5	2015-05-07 19:52:06 UTC	-73.999817	40.738354	-73.999512	40.723
1	7.7	2009-07-17 20:04:56 UTC	-73.994355	40.728225	-73.994710	40.750
2	12.9	2009-08-24 21:45:00 UTC	-74.005043	40.740770	-73.962565	40.772
3	5.3	2009-06-26 08:22:21 UTC	-73.976124	40.790844	-73.965316	40.803
4	16.0	2014-08-28 17:47:00 UTC	-73.925023	40.744085	-73.973082	40.761

```
#Check for empty elements

nvc = pd.DataFrame(df.isnull().sum().sort_values(), columns=['Total Null Values'])

nvc['Percentage'] = round(nvc['Total Null Values']/df.shape[0],3)*100

print(nvc)

df.dropna(inplace=True)
```

```
                   Total Null Values  Percentage
fare_amount                        0         0.0
pickup_datetime                    0         0.0
pickup_longitude                   0         0.0
pickup_latitude                    0         0.0
passenger_count                    0         0.0
dropoff_longitude                  1         0.0
dropoff_latitude                   1         0.0
```

```
# Reframing the columns

df = df[(df.pickup_latitude<90) & (df.dropoff_latitude<90) &
        (df.pickup_latitude>-90) & (df.dropoff_latitude>-90) &
        (df.pickup_longitude<180) & (df.dropoff_longitude<180) &
        (df.pickup_longitude>-180) & (df.dropoff_longitude>-180)]

df.pickup_datetime=pd.to_datetime(df.pickup_datetime)

df['year'] = df.pickup_datetime.dt.year
df['month'] = df.pickup_datetime.dt.month
df['weekday'] = df.pickup_datetime.dt.weekday
df['hour'] = df.pickup_datetime.dt.hour

df['Monthly_Quarter'] = df.month.map({1:'Q1',2:'Q1',3:'Q1',4:'Q2',5:'Q2',6:'Q2',7:'Q3',
                                      8:'Q3',9:'Q3',10:'Q4',11:'Q4',12:'Q4'})
df['Hourly_Segments'] = df.hour.map({0:'H1',1:'H1',2:'H1',3:'H1',4:'H2',5:'H2',6:'H2',7:'H2',8:'
H3',
                                     9:'H3',10:'H3',11:'H3',12:'H4',13:'H4',14:'H4',15:'H4',16:'H5',
                                     17:'H5',18:'H5',19:'H5',20:'H6',21:'H6',22:'H6',23:'H6'})

df['Distance']=[round(geopy.distance.distance((df.pickup_latitude[i], df.pickup_longitude[i]),
(df.dropoff_latitude[i], df.dropoff_longitude[i])).m,2) for i in df.index]

df.drop(['pickup_datetime','month', 'hour',], axis=1, inplace=True)

original_df = df.copy(deep=True)

df.head()
```

	fare_amount	pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_latitude	passeng
0	7.5	-73.999817	40.738354	-73.999512	40.723217	1
1	7.7	-73.994355	40.728225	-73.994710	40.750325	1
2	12.9	-74.005043	40.740770	-73.962565	40.772647	1
3	5.3	-73.976124	40.790844	-73.965316	40.803349	3
4	16.0	-73.925023	40.744085	-73.973082	40.761247	5

```
#Checking the dtypes of all the columns

df.info()
```

```
Data columns (total 11 columns):
```

```
 #   Column             Non-Null Count   Dtype  
---  ------             --------------   -----  
 0   fare_amount        199987 non-null  float64
 1   pickup_longitude   199987 non-null  float64
 2   pickup_latitude    199987 non-null  float64
 3   dropoff_longitude  199987 non-null  float64
 4   dropoff_latitude   199987 non-null  float64
 5   passenger_count    199987 non-null  int64  
 6   year               199987 non-null  int64  
 7   weekday            199987 non-null  int64  
 8   Monthly_Quarter    199987 non-null  object 
 9   Hourly_Segments    199987 non-null  object 
 10  Distance           199987 non-null  float64
```

```python
#Checking number of unique rows in each feature

df.nunique().sort_values()
```

```
Monthly_Quarter           4
Hourly_Segments           6
year                      7
weekday                   7
passenger_count           8
fare_amount            1244
pickup_longitude      71055
dropoff_longitude     76890
pickup_latitude       83831
dropoff_latitude      90582
Distance             164542
dtype: int64
```

```python
#Checking the stats of all the columns

display(df.describe())
```

	fare_amount	pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_latitude
count	199987.000000	199987.000000	199987.000000	199987.000000	199987.00000
mean	11.359849	-72.501786	39.917937	-72.511608	39.922031
std	9.901868	10.449955	6.130412	10.412192	6.117669
min	-52.000000	-93.824668	-74.015515	-75.458979	-74.015750
25%	6.000000	-73.992064	40.734793	-73.991407	40.733823
50%	8.500000	-73.981822	40.752592	-73.980092	40.753042
75%	12.500000	-73.967154	40.767157	-73.963658	40.768000
max	499.000000	40.808425	48.018760	40.831932	45.031598

Exploratory Data Analysis (EDA)

plt.figure(figsize=[15,10])

a=plt.imread('https://raw.githubusercontent.com/Masterx-AI/Project_Uber_Fare_Prediction /main/wm.png')

plt.imshow(a, alpha=0.2)

plt.scatter((df.pickup_longitude+180)*3,(df.pickup_latitude+215)*1.45555555,alpha=0.3, color='red')

#mdf.plot(kind='scatter',x='pickup_latitude',y='pickup_longitude',alpha=0.1)

plt.show()

Data Preprocessing

#Removal of any Duplicate rows (if any)

counter = 0
rs,cs = original_df.shape

```
df.drop_duplicates(inplace=True)
df.drop(['pickup_latitude','pickup_longitude',
        'dropoff_latitude','dropoff_longitude'],axis=1)

if df.shape==(rs,cs):
    print('\n\033[1mInference:\033[0m The dataset doesn\'t have any duplicates')
else:
    print(f'\n\033[1mInference:\033[0m Number of duplicates dropped/fixed ---> {rs-df.shape[0]}')
```

```
#Let us first analyze the distribution of the target variable

plt.figure(figsize=[8,4])

sns.distplot(df[target], color='g',hist_kws=dict(edgecolor="black", linewidth=2), bins=30)

plt.title('Target Variable Distribution - Median Value of Homes ($1Ms)')

plt.show()
```

```
#  Since only one pair of values are missing in the dataset, we can just drop them

# df.dropna(inplace=True)
```

Data Manipulation

```
#Splitting the data intro training & testing sets

m=[]
for i in df.columns.values:
    m.append(i.replace(' ','_'))

df.columns = m
X = df.drop([target],axis=1)
Y = df[target]
```

```
Train_X, Test_X, Train_Y, Test_Y = train_test_split(X, Y, train_size=0.8, test_size=
0.2, random_state=100)
Train_X.reset_index(drop=True,inplace=True)

print('Original set  ---> ',X.shape,Y.shape,'\nTraining set  ---> ',Train_X.shape,Train
_Y.shape,'\nTesting set   ---> ', Test_X.shape,'', Test_Y.shape)
```

```
Original set  --->  (163203, 32) (163203,)
Training set  --->  (130562, 32) (130562,)
Testing set   --->  (32641, 32)  (32641,)
```

Feature Selection/Extraction

```
#Checking the correlation

print('\033[1mCorrelation Matrix'.center(100))
plt.figure(figsize=[24,20])
sns.heatmap(df.corr(), annot=True, vmin=-1, vmax=1, center=0) #cmap='BuGn'
plt.show()
```

```
#Testing a Linear Regression model with statsmodels

Train_xy = pd.concat([Train_X,Train_Y.reset_index(drop=True)],axis=1)
a = Train_xy.columns.values

API = api.ols(formula='{} ~ {}'.format(target,' + '.join(i for i in Train_X.columns)), dat
a=Train_xy).fit()
#print(API.conf_int())
#print(API.pvalues)
API.summary()
```

2 .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.

Libraries Used

```
import os

import string from nltk.corpus

import stopwords from sklearn.model_selection

import train_test_split from sklearn.metrics

import accuracy_score

import numpy as np
```

Loading the Data

```
def load_data():

print("Loading data...")

ham_files_location = os.listdir("dataset/ham")

spam_files_location = os.listdir("dataset/spam")

data = []


# Load ham email

for file_path in ham_files_location:

f = open("dataset/ham/" + file_path, "r")

text = str(f.read())

data.append([text, "ham"])



data = np.array(data)

print("flag 1: loaded data")
```

```
return data
```

Data Pre-processing

Preprocessing data: noise removaldef preprocess_data(data):

print("Preprocessing data...")

punc = string.punctuation

sw = stopwords.words('english')

```
for record in data:
# Remove common punctuation and symbols
for item in punc:
record[0] = record[0].replace(item, "")
```

Splitting the Data into Training and Testing Sets

Splitting original dataset into training dataset and test datasetdef split_data(data):

datasetdef split_data(data):
print("Splitting data...")

features = data[:, 0]   # array containing all email text bodies
   labels = data[:, 1]   # array containing corresponding labels
   print(labels)
training_data, test_data, training_labels, test_labels =\

train_test_split(features, labels, test_size = 0.27, random_state = 42)

print("flag 3: splitted data")
   return training_data, test_data, training_labels, test_labels

The KNN Algorithm

get_count() function

```
def get_count(text):
    wordCounts = dict()

for word in text.split():
        if word in wordCounts:
            wordCounts[word] += 1
        else:
            wordCounts[word] = 1

return wordCounts
```

euclidean_difference() function

```
def euclidean_difference(test_WordCounts, training_WordCounts):
total = 0

for word in test_WordCounts:
if word in test_WordCounts and word in training_WordCounts:
total += (test_WordCounts[word] - training_WordCounts[word])**2
```

```
del training_WordCounts[word]
```

```
else:
```

```
total += test_WordCounts[word]**2
```

```
for word in training_WordCounts:
```

```
total += training_WordCounts[word]**2
```

```
return total**0.5
```

get_class() function

```
def get_class(selected_Kvalues):
    spam_count = 0
    ham_count = 0
```

```
or value in selected_Kvalues:
```

```
if value[0] == "spam":
```

```
spam_count += 1
```

```
else:
```

```
ham_count += 1
```

```
if spam_count > ham_count:
```

```
return "spam"
```

```
else:
```

```
return "ham"
```

knn_classifier() function

```
def knn_classifier(training_data, training_labels, test_data, K,
tsize):
    print("Running KNN Classifier...")

    result = []
    counter = 1
```

```
# word counts for training email

training_WordCounts = []

for training_text in training_data:

training_WordCounts.append(get_count(training_text))
```

```
for test_text in test_data:

similarity = [] # List of euclidean distances

test_WordCounts = get_count(test_text) # word counts for test email
```

```
# Getting euclidean difference

for index in range(len(training_data)):

euclidean_diff =\

euclidean_difference(test_WordCounts, training_WordCounts[index])

similarity.append([training_labels[index], euclidean_diff])
```

```
# Sort list in ascending order based on euclidean difference

similarity = sorted(similarity, key = lambda i:i[1])



# Select K nearest neighbours

selected_Kvalues = []

for i in range(K):

selected_Kvalues.append(similarity[i])



# Predicting the class of email

result.append(get_class(selected_Kvalues))



return result
```

main() function

```
def main(K):
    data = load_data()
    data = preprocess_data(data)
    training_data, test_data, training_labels, test_labels = 
split_data(data)

tsize = len(test_data)

result = knn_classifier(training_data, training_labels, test_data[:tsize], K, 
tsize)
accuracy = accuracy_score(test_labels[:tsize], result)
```

```
print("training data size\t: " + str(len(training_data)))
print("test data size\t\t: " + str(len(test_data)))
print("K value\t\t\t\t: " + str(K))
print("Samples tested\t\t: " + str(tsize))
print("% accuracy\t\t\t: " + str(accuracy * 100))
print("Number correct\t\t: " + str(int(accuracy * tsize)))
print("Number wrong\t\t: " + str(int((1 - accuracy) * tsize)))
```

3. 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. Link to the Kaggle project: https://www.kaggle.com/barelydedicated/bank-customer-churn-modeling 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)

Load the dataset

In [ ]:

linkcode

```
# load the file using read_csv. RowNumber column in the data can be used as index_col
df = pd.read_csv("bank.csv", index_col="RowNumber")

# back up data to preserve the initial version for reference
df_back = df.copy()
```

Shape of data

```
# print the data set information as number of rows and columns
print(f"There are {df.shape[0]} rows and {df.shape[1]} columns.")  # f-string
```

Dataset information

```
# check the dataset information
df.info()
```

Check duplicates

```
# check duplicate records
df.duplicated().sum()
```

Check NULL values

```
# precentage of missing values in columns
round(df.isna().sum() / df.isna().count() * 100, 2)
```

```
# drop unused features
df.drop(['CustomerId', 'Surname'], axis=1, inplace=True)
```

Splitting the dataset into the Training set and Test set

```
from sklearn.model_selection import train_test_splitX_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state = 0)# Feature Scaling (very important)
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)
```

2. Creating our Artificial Neural Network!

```
#Import Keras library and packagesimport keras
import sys
from keras.models import Sequential #to initialize NN
from keras.layers import Dense #used to create layers in NN
```

```
#Initialising the ANN - Defining as a sequence of layers or a Graph
classifier = Sequential()
```

Adding the input layer

```
#Input Layer
classifier.add(Dense(units = 6, kernel_initializer = 'uniform',
activation = 'relu', input_dim = 11 ))
```

Adding Second hidden layer

```
classifier.add(Dense(units = 6, kernel_initializer = ‘uniform’,
activation = ‘relu’))
```

Adding Output layer

```
classifier.add(Dense(units = 1, kernel_initializer = ‘uniform’,
activation = ‘sigmoid’))
```

```
classifier.compile(optimizer = ‘adam’,loss=
“binary_crossentropy”,metrics=[“accuracy”])
```

Fitting the ANN to the Training Set

```
classifier.fit(X_train, y_train, batch_size = 10, epochs = 100 )
```

Making Predictions

```
#Making the COnfusion Matrix
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_test, y_pred)cm
```

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

```
		# Importing Libraries
import numpy as np
import matplotlib.pyplot as plt

def mean_squared_error(y_true, y_predicted):

    # Calculating the loss or cost
    cost = np.sum((y_true-y_predicted)**2) / len(y_true)
    return cost

# Gradient Descent Function
# Here iterations, learning_rate, stopping_threshold
# are hyperparameters that can be tuned
def gradient_descent(x, y, iterations = 1000, learning_rate = 0.0001,
                     stopping_threshold = 1e-6):

    # Initializing weight, bias, learning rate and iterations
    current_weight = 0.1
```

```
    current_bias = 0.01

    iterations = iterations

    learning_rate = learning_rate

    n = float(len(x))


    costs = []

    weights = []

    previous_cost = None


    # Estimation of optimal parameters

    for i in range(iterations):


        # Making predictions

y_predicted = (current_weight * x) + current_bias


        # Calculationg the current cost

        current_cost = mean_squared_error(y, y_predicted)


        # If the change in cost is less than or equal to

        # stopping_threshold we stop the gradient descent
```

```
if previous_cost and abs(previous_cost-current_cost)<=stopping_threshold:
    break

previous_cost = current_cost

costs.append(current_cost)
weights.append(current_weight)

# Calculating the gradients
weight_derivative = -(2/n) * sum(x * (y-y_predicted))
bias_derivative = -(2/n) * sum(y-y_predicted)
# Updating weights and bias
current_weight = current_weight - (learning_rate * weight_derivative)
current_bias = current_bias - (learning_rate * bias_derivative)

# Printing the parameters for each 1000th iteration
print(f"Iteration {i+1}: Cost {current_cost}, Weight \
{current_weight}, Bias {current_bias}")
```

```
    # Visualizing the weights and cost at for all iterations

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

    plt.plot(weights, costs)

    plt.scatter(weights, costs, marker='o', color='red')

    plt.title("Cost vs Weights")

    plt.ylabel("Cost")

    plt.xlabel("Weight")

    plt.show()


    return current_weight, current_bias




def main():


    # Data

    X = np.array([32.50234527, 53.42680403, 61.53035803, 47.47563963,
59.81320787,

            55.14218841, 52.21179669, 39.29956669, 48.10504169, 52.55001444,

            45.41973014, 54.35163488, 44.1640495 , 58.16847072, 56.72720806,

            48.95588857, 44.68719623, 60.29732685, 45.61864377, 38.81681754])
```

```
Y = np.array([31.70700585, 68.77759598, 62.5623823 , 71.54663223,
87.23092513,
        78.21151827, 79.64197305, 59.17148932, 75.3312423 , 71.30087989,
        55.16567715, 82.47884676, 62.00892325, 75.39287043, 81.43619216,
        60.72360244, 82.89250373, 97.37989686, 48.84715332, 56.87721319])


# Estimating weight and bias using gradient descent
estimated_weight, eatimated_bias = gradient_descent(X, Y, iterations=2000)
print(f"Estimated Weight: {estimated_weight}\nEstimated Bias:
{eatimated_bias}")


# Making predictions using estimated parameters
Y_pred = estimated_weight*X + eatimated_bias


# Plotting the regression line
plt.figure(figsize = (8,6))
plt.scatter(X, Y, marker='o', color='red')
plt.plot([min(X), max(X)], [min(Y_pred), max(Y_pred)],
color='blue',markerfacecolor='red',
         markersize=10,linestyle='dashed')
plt.xlabel("X")
```

```
    plt.ylabel("Y")

    plt.show()



if __name__=="__main__":

    main()
```

Output:

Iteration 1: Cost 4352.088931274409, Weight 0.7593291142562117, Bias 0.02288558130709

Iteration 2: Cost 1114.8561474350017, Weight 1.081602958862324, Bias 0.02918014748569513

Iteration 3: Cost 341.42912086804455, Weight 1.2391274084945083, Bias 0.03225308846928192

Iteration 4: Cost 156.64495290904443, Weight 1.3161239281746984, Bias 0.03375132986012604

Iteration 5: Cost 112.49704004742098, Weight 1.3537591652024805, Bias 0.034479873154934775

Iteration 6: Cost 101.9493925395456, Weight 1.3721549833978113, Bias 0.034832195392868505

Iteration 7: Cost 99.4293893333546, Weight 1.3811467575154601, Bias 0.03500062439068245

Iteration 8: Cost 98.82731958262897, Weight 1.3855419247507244, Bias 0.03507916814736111

Iteration 9: Cost 98.68347500997261, Weight 1.3876903144657764, Bias 0.035113776874486774

Iteration 10: Cost 98.64910780902792, Weight 1.3887405007983562, Bias 0.035126910596389935

Iteration 11: Cost 98.64089651459352, Weight 1.389253895811451, Bias 0.03512954755833985

Iteration 12: Cost 98.63893428729509, Weight 1.38950491235671, Bias 0.035127053821718185

Iteration 13: Cost 98.63846506273883, Weight 1.3896276808137857, Bias 0.035122052266051224

Iteration 14: Cost 98.63835254057648, Weight 1.38968776283053, Bias 0.03511582492978764

Iteration 15: Cost 98.63832524036214, Weight 1.3897172043139192, Bias 0.03510899846107016

Iteration 16: Cost 98.63831830104695, Weight 1.389731668997059, Bias 0.035101879159522745

Iteration 17: Cost 98.63831622628217, Weight 1.389738813163012, Bias 0.03509461674147458

Estimated Weight: 1.389738813163012

Estimated Bias: 0.03509461674147458

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

the elbow method. Dataset link : https://www.kaggle.com/datasets/kyanyoga/sample-sales-data

```
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
import matplotlib.pyplot as plt # for data visualization
import seaborn as sns # for statistical data visualization
%matplotlib inline


# Any results you write to the current directory are saved as output.


import os
for dirname, _, filenames in os.walk('/kaggle/input'):
    for filename in filenames:
        print(os.path.join(dirname, filename))
```

Import dataset

```
data = '/kaggle/input/facebook-live-sellers-in-thailand-uci-ml-repo/Live.csv'

df = pd.read_csv(data)
```

Exploratory data analysis

Check shape of the dataset

```
df.shape
```

```
df.head()
```

View summary of dataset

```
df.info()
```

Check for missing values in dataset

```
df.isnull().sum()
```

Group C :

3) https://coinsbench.com/create-a-simple-bank-contract-with-solidity-3a47b00b929a

```
pragma solidity ^0.4.19;

contract TipJar {

    address owner;    // current owner of the contract

    function TipJar() public {
        owner = msg.sender;
    }

    function withdraw() public {
        require(owner == msg.sender);
        msg.sender.transfer(address(this).balance);
    }

    function deposit(uint256 amount) public payable {
        require(msg.value == amount);
    }

    function getBalance() public view returns (uint256) {
        return address(this).balance;
    }
}
```

4. https://www.analyticsvidhya.com/blog/2022/07/create-smart-contract-using-solidity-programming/

https://www.tutorialspoint.com/solidity/solidity_quick_guide.htm