# Group A: Design and Analysis of Algorithms

**A1. Write a program non-recursive and recursive program to calculate Fibonacci numbers and analyze their time and space complexity.**

**Iterative Program**

# Program to display the Fibonacci sequence up to n-th term nterms = int(input("Enter number of terms "))

# first two terms n1, n2 = 0, 1

count = 0

# check if the number of terms is valid if nterms <= 0:

print("Please enter a positive integer") # if there is only one term, return n1

elif nterms == 1:

print("Fibonacci sequence upto", nterms,":") print(n1)

# generate fibonacci sequence else:

print("Fibonacci sequence:") while count < nterms:

print(n1)

nth = n1 + n2

# update values n1 = n2

n2 = nth count += 1

#### Output

Enter number of terms 4 Fibonacci sequence: 0

1

1

2

#### Recursive Program

def fibonacci(n): if(n <= 1):

return n else:

return(fibonacci(n-1) + fibonacci(n-2)) n = int(input("Enter number of terms:")) print("Fibonacci sequence:")

for i in range(n): print(fibonacci(i))

#### Output

Enter number of terms:4 Fibonacci sequence: 0

1

1

2

**A2. Write a program to implement Huffman Encoding using a greedy strategy.**

### Implementation

def printNodes(node, val=''):

newVal = val + str(node.huff) if(node.left):

printNodes(node.left, newVal) if(node.right):

printNodes(node.right, newVal) if(not node.left and not node.right):

print(f"{node.symbol} -> {newVal}")

* # characters for huffman tree
* chars = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
* # frequency of characters
* freq = [ 4, 7, 12, 14, 17, 43, 54]
* # list containing unused nodes
* nodes = []
* # converting characters and frequencies into huffman tree nodes
* for x in range(len(chars)):
* nodes.append(node(freq[x], chars[x]))
* while len(nodes) > 1:
* # sort all the nodes in ascending order based on their frequency
* nodes = sorted(nodes, key=lambda x: x.freq)
* # pick 2 smallest nodes
* left = nodes[0]
* right = nodes[1]
* # assign directional value to these nodes
* left.huff = 0
* right.huff = 1
* # combine the 2 smallest nodes to create new node as their parent
* newNode = node(left.freq+right.freq, left.symbol+right.symbol, left, right)
* # remove the 2 nodes and add their parent as new node among others
* nodes.remove(left)
* nodes.remove(right) nodes.append(newNode)
* # Huffman Tree is ready!
* printNodes(nodes[0])

#### Output

a -> 0000 b -> 0001 c -> 001

d -> 010

e -> 011

f -> 10

g -> 11

## A3. Write a program to solve a fractional Knapsack problem using a greedy method.

**Implementation**

def fractional\_knapsack(value, weight, capacity): # index = [0, 1, 2, ..., n - 1] for n items

index = list(range(len(value)))

# contains ratios of values to weight

ratio = [v/w for v, w in zip(value, weight)]

# index is sorted according to value-to-weight ratio in decreasing order index.sort(key=lambda i: ratio[i], reverse=True)

max\_value = 0

fractions = [0]\*len(value) for i in index:

if weight[i] <= capacity: fractions[i] = 1 max\_value += value[i] capacity -= weight[i]

else:

fractions[i] = capacity/weight[i] max\_value += value[i]\*capacity/weight[i] break

return max\_value, fractions

n = int(input('Enter number of items: '))

value = input('Enter the values of the {} item(s) in order: '.format(n)).split() value = [int(v) for v in value]

weight = input('Enter the positive weights of the {} item(s) in order: '.format(n)).split() weight = [int(w) for w in weight]

capacity = int(input('Enter maximum weight: '))

max\_value, fractions = fractional\_knapsack(value, weight, capacity) print('The maximum value of items that can be carried:', max\_value) print('The fractions in which the items should be taken:', fractions)

#### Output

Enter number of items: 3

Enter the values of the 3 item(s) in order: 24 15 25

Enter the positive weights of the 3 item(s) in order: 15 10 18 Enter maximum weight: 20 The maximum value of items that can be carried: 31.5

The fractions in which the items should be taken: [1, 0.5, 0]

## A4. Write a Program for Queens matrix having first Queen placed. Use backtracking to place remaining Queens to generate the final 8-queen’s matrix.

/\* C++ program to solve N Queen Problem using backtracking \*/

#include <bits/stdc++.h> #define N 4

using namespace std;

/\* A utility function to print solution \*/ void printSolution(int board[N][N])

{

for (int i = 0; i < N; i++) {

for (int j = 0; j < N; j++)

cout << " " << board[i][j] << " "; printf("\n");

}

}

/\* 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 \*/

bool isSafe(int board[N][N], int row, int col)

{

int i, j;

/\* Check this row on left side \*/ for (i = 0; i < col; i++)

if (board[row][i])

return false;

/\* Check upper diagonal on left side \*/

for (i = row, j = col; i >= 0 && j >= 0; i--, j--) if (board[i][j])

return false;

/\* Check lower diagonal on left side \*/

for (i = row, j = col; j >= 0 && i < N; i++, j--) if (board[i][j])

return false;

return true;

}

/\* A recursive utility function to solve N Queen problem \*/

bool solveNQUtil(int board[N][N], int col)

{

/\* base case: If all queens are placed then return true \*/

if (col >= N)

return true;

/\* Consider this column and try placing this queen in all rows one by one \*/

for (int i = 0; i < N; i++) {

/\* Check if the queen can be placed on board[i][col] \*/

if (isSafe(board, i, col)) {

/\* Place this queen in board[i][col] \*/ board[i][col] = 1;

/\* recur to place rest of the queens \*/ if (solveNQUtil(board, col + 1))

return true;

/\* 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 the queen cannot be placed in any row in this column col then return false \*/

return false;

}

/\* 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.\*/

bool solveNQ()

{

int board[N][N] = { { 0, 0, 0, 0 },

{ 0, 0, 0, 0 },

{ 0, 0, 0, 0 },

{ 0, 0, 0, 0 } };

if (solveNQUtil(board, 0) == false) {

cout << "Solution does not exist"; return false;

}

printSolution(board); return true;

}

// driver program to test above function int main()

{

}

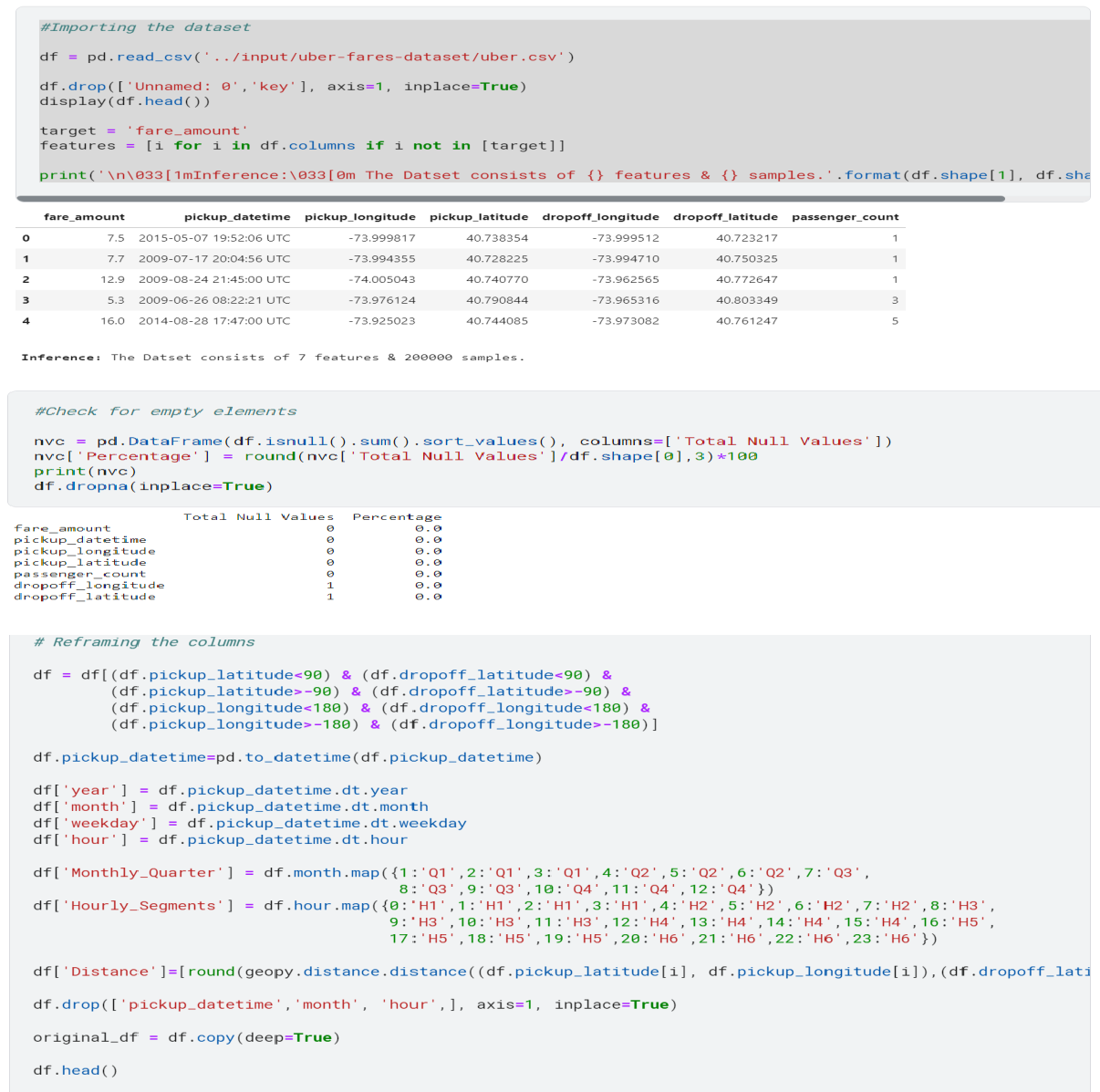
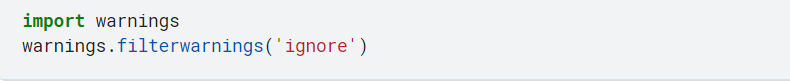
Output

solveNQ(); return 0;

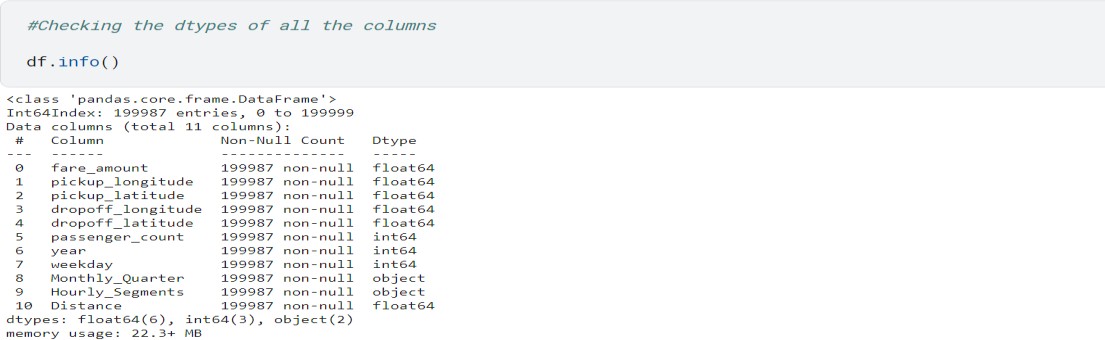
* 0 0 1 0
* 1 0 0 0
* 0 0 0 1
* 0 1 0 0

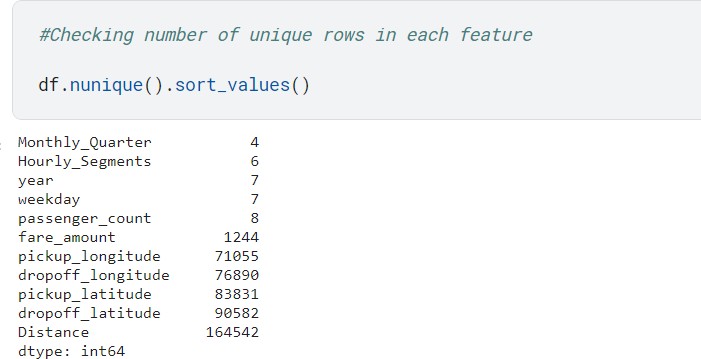
**Group B: Machine Learning:**

**Assignment No. 1**



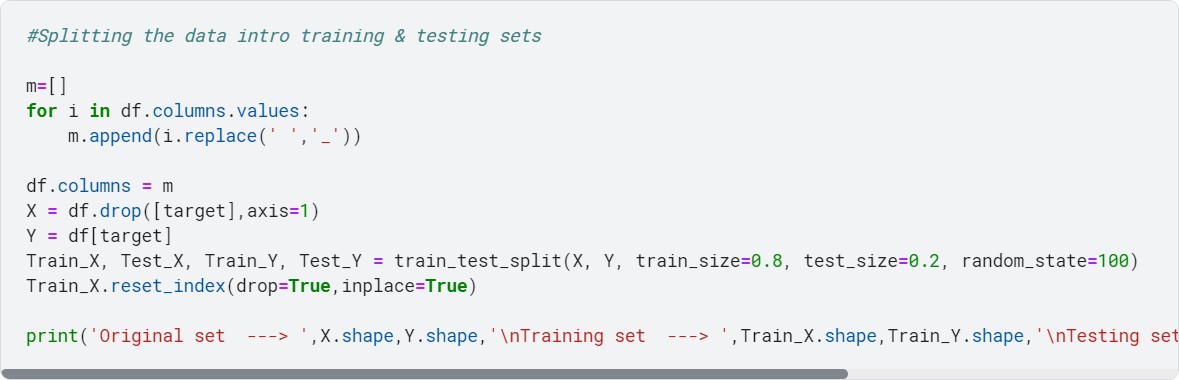


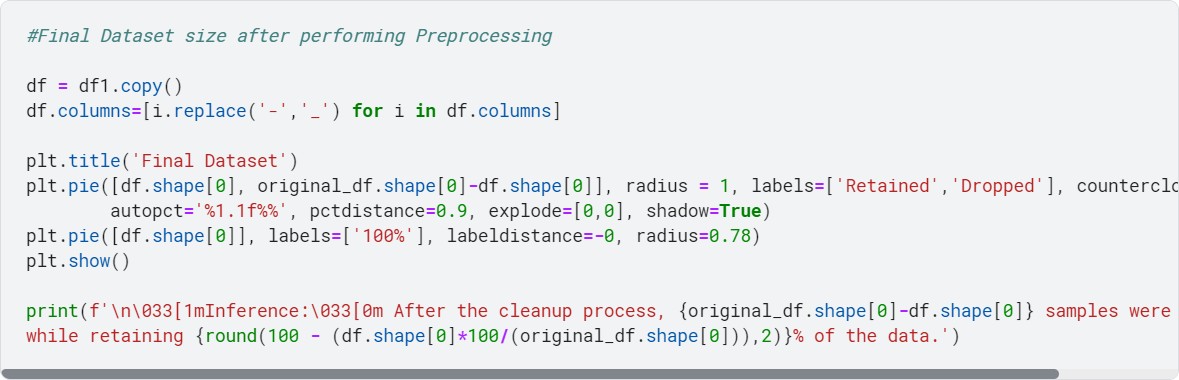
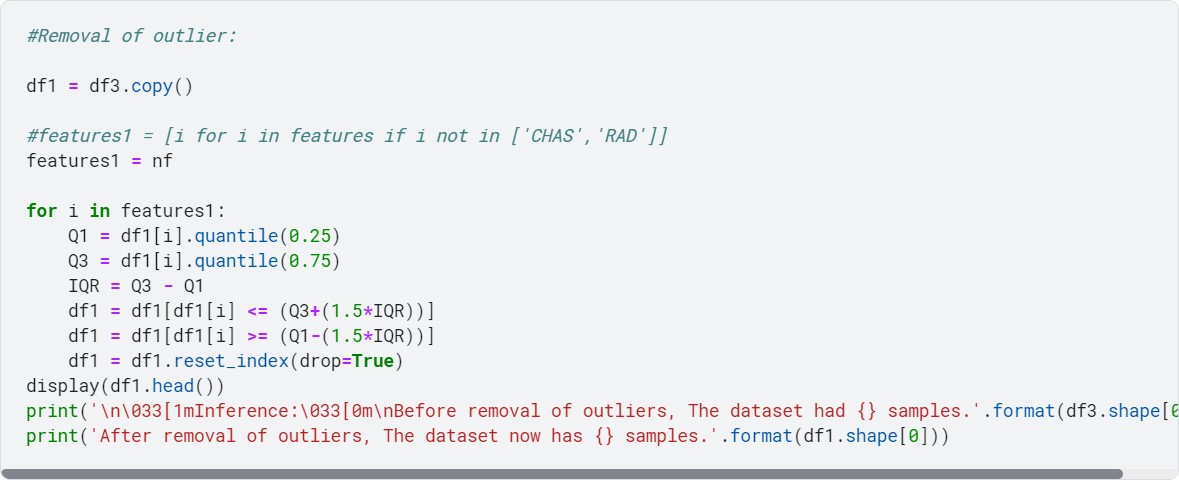








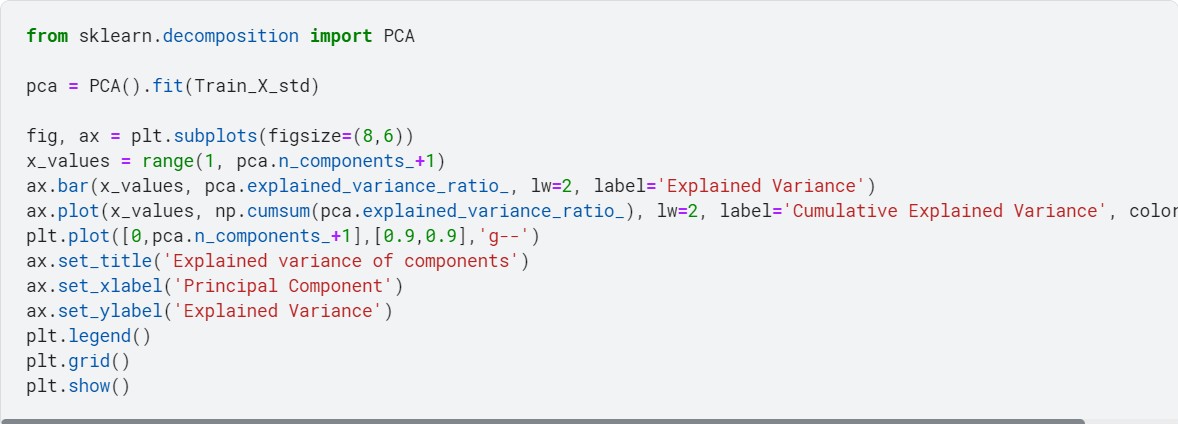
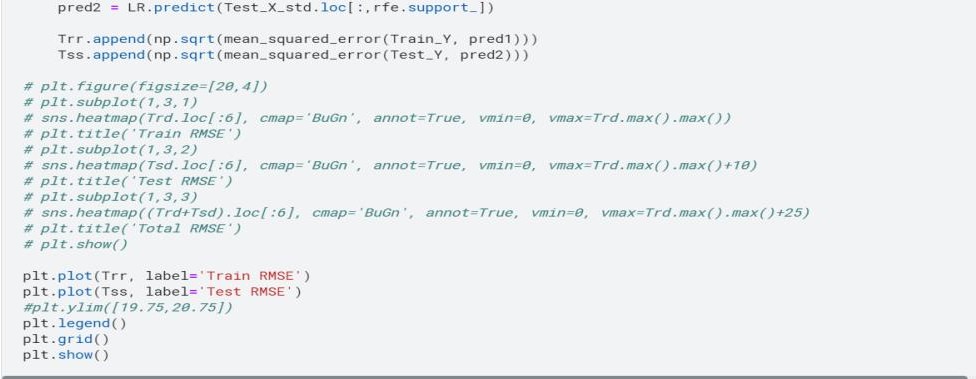




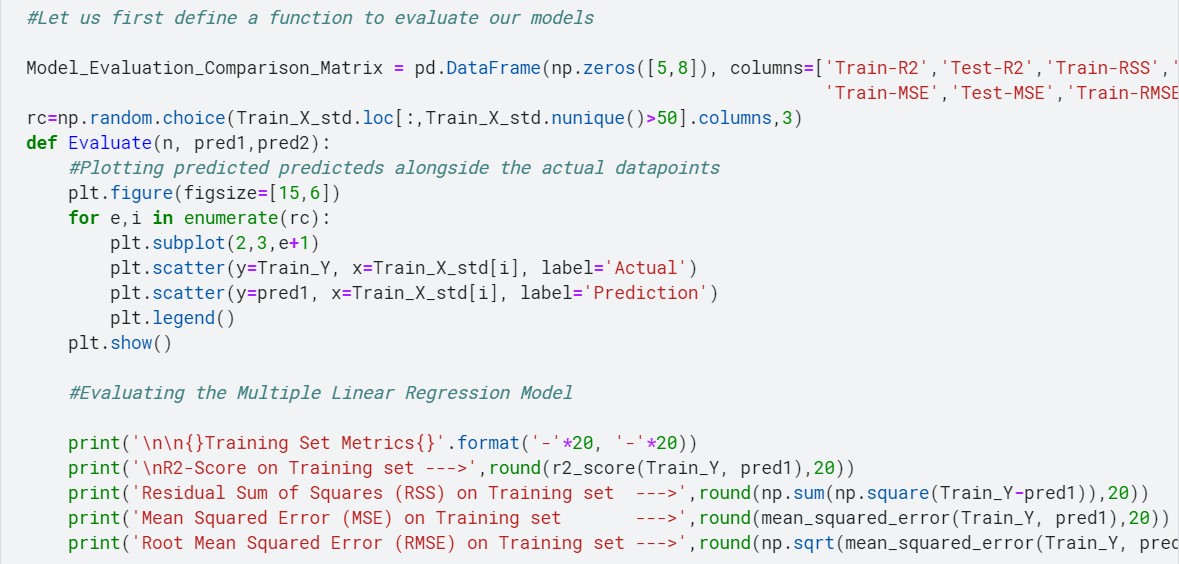
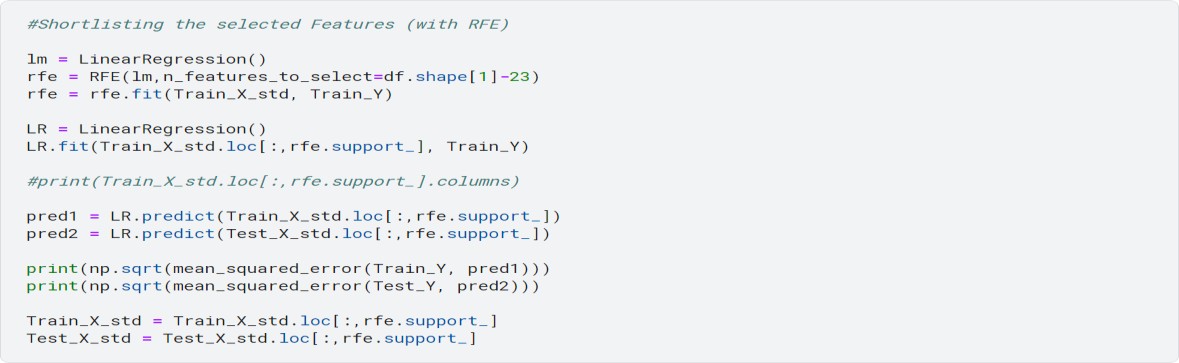
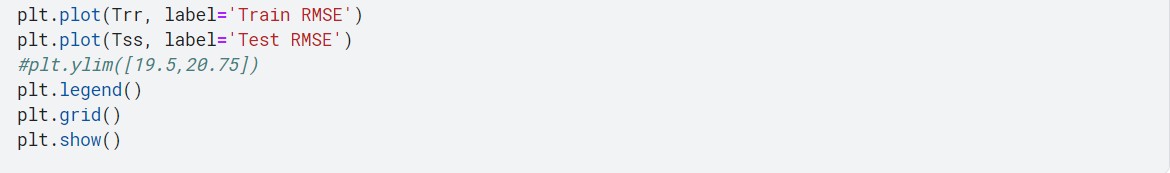


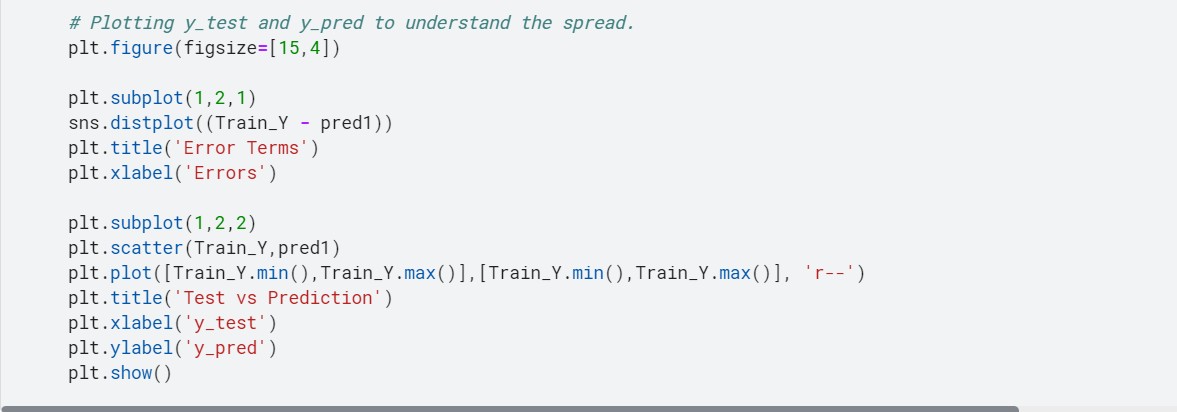


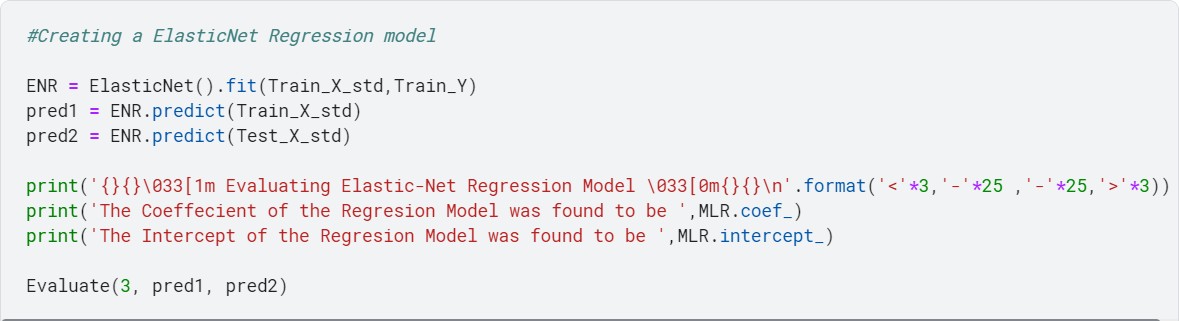
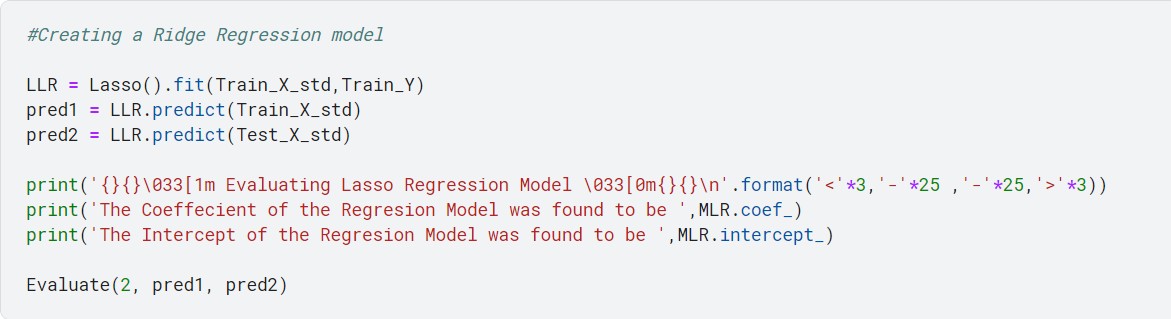
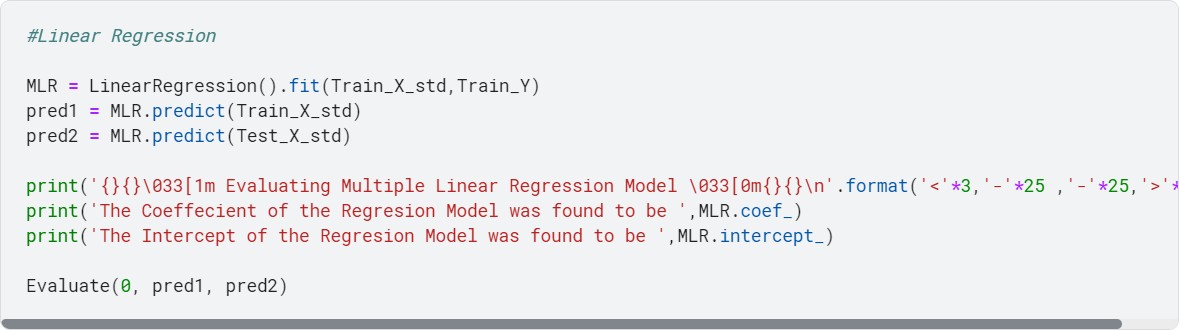




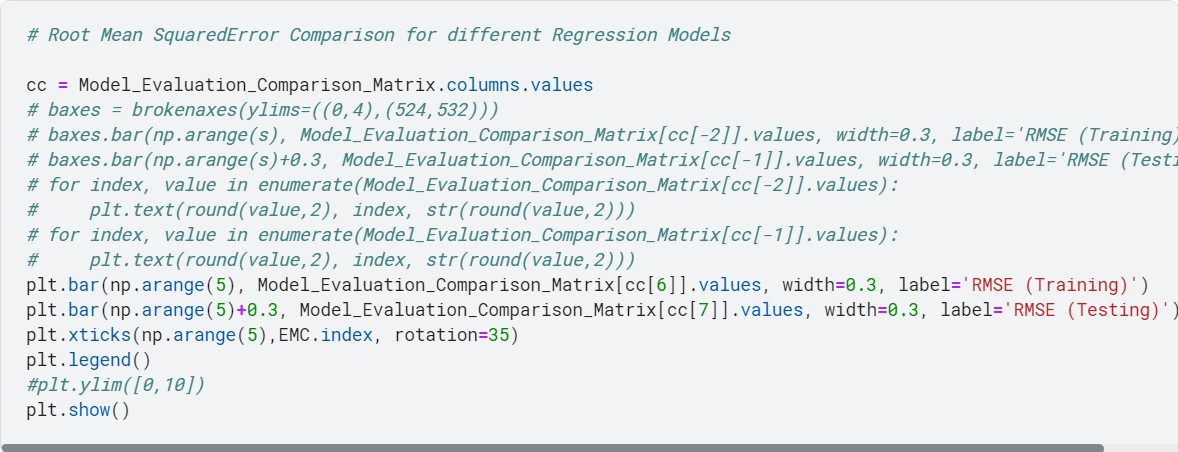
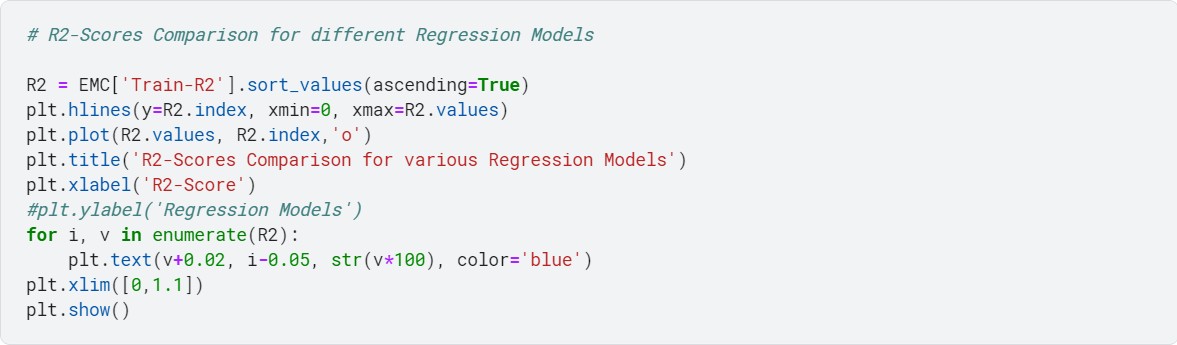
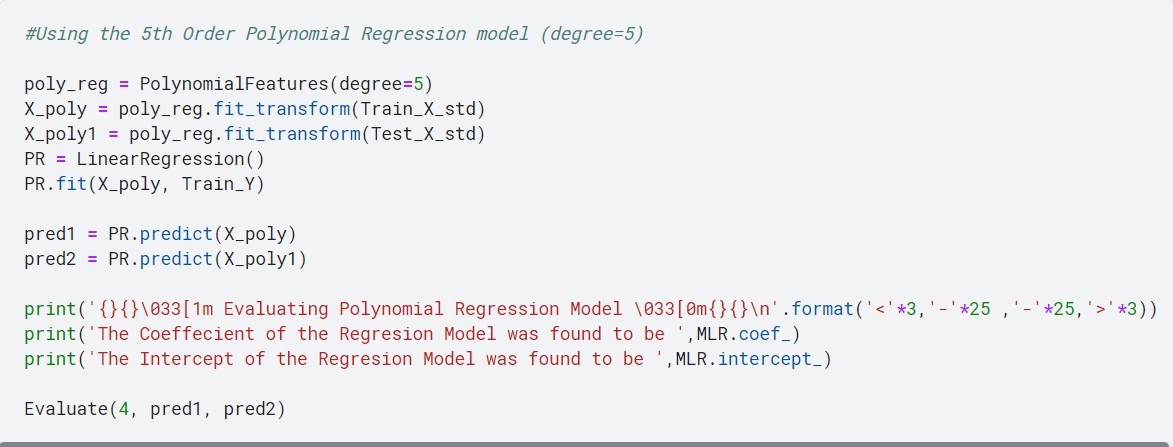












**Assignment No. 2**

cur\_x = 3

# The algorithm starts at x=3 rate = 0.01 # Learning rate

precision = 0.000001 #This tells us when to stop the algorithm previous\_step\_size = 1 max\_iters = 10000

# maximum number of iterations iters = 0 #iteration counter

df = lambda x: 2\*(x+5) #Gradient of our function

while previous\_step\_size > precision and iters < max\_iters: prev\_x = cur\_x #Store current x value in prev\_x

cur\_x = cur\_x - rate \* df(prev\_x) #Grad descent

previous\_step\_size = abs(cur\_x - prev\_x) #Change in x iters = iters+1 #iteration count print("Iteration",iters,"\nX value is",cur\_x)

#Print iterations print("The local minimum occurs at", cur\_x)

X value is -4.998919090416489

Iteration 442

X value is -4.99894070860816

Iteration 443

X value is -4.998961894435997

Iteration 444

X value is -4.998982656547277

Iteration 445

X value is -4.999003003416331

Iteration 446

X value is -4.999022943348004

Iteration 447

X value is -4.999042484481044

Iteration 448

X value is -4.999061634791423

Iteration 449

X value is -4.999080402095594

Iteration 450

X value is -4.999098794053682

Iteration 451

X value is -4.999116818172609

Iteration 452

X value is -4.999134481809157

Iteration 453

X value is -4.999151792172974

Iteration 454

X value is -4.999168756329515

Iteration 455

X value is -4.999185381202924

Iteration 456

X value is -4.999201673578866

Iteration 457

X value is -4.999217640107289

Iteration 458

X value is -4.999233287305143

Iteration 459

X value is -4.9992486215590395

Iteration 460

X value is -4.999263649127859

Iteration 461

X value is -4.999278376145302

Iteration 462

X value is -4.999292808622396

Iteration 463

X value is -4.999306952449948

Iteration 464

X value is -4.999320813400949

Iteration 465

X value is -4.99933439713293

Iteration 466

X value is -4.999347709190272

Iteration 467

X value is -4.9993607550064665

Iteration 468

X value is -4.999373539906337

Iteration 469

X value is -4.99938606910821

Iteration 470

### Assignment 3

**KNN algorithm on diabetes dataset**

import pandas as pd import numpy as np import seaborn as sns

import matplotlib.pyplot as plt

%matplotlib inline import warnings warnings.filterwarnings('ignore')

from sklearn.model\_selection import train\_test\_split from sklearn.svm import SVC from sklearn import metrics

df=pd.read\_csv('diabetes.csv') df.columns

### Index(['Pregnancies', 'Glucose', 'BloodPressure', 'SkinThickness', 'Insulin', 'BMI', 'Pedigree', 'Age', 'Outcome'], dtype='object')

Check for null values. If present remove null values from the dataset

|  |  |  |
| --- | --- | --- |
| df.isnull().sum() |  |  |
| **Pregnancies** | **0** |
| **Glucose** | **0** |
| **BloodPressure** | **0** |
| **SkinThickness** | **0** |
| **Insulin 0** |  |
| **BMI** |  | **0** |
| **Pedigree** |  | **0** |
| **Age** |  | **0** |
| **Outcome**  **dtype: int64** |  | **0** |

### Outcome is the label/target, other columns are features

X = df.drop('Outcome',axis = 1) y = df['Outcome'] from sklearn.preprocessing import scale X = scale(X)

### # split into train and test

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size = 0.3, random\_state = 42)

from sklearn.neighbors import KNeighborsClassifier knn = KNeighborsClassifier(n\_neighbors=7) knn.fit(X\_train, y\_train)

y\_pred = knn.predict(X\_test) print("Confusion matrix: ")

cs = metrics.confusion\_matrix(y\_test,y\_pred) print(cs)

### Confusion matrix: [[123 28]

**[ 37 43]]**

print("Acccuracy ",metrics.accuracy\_score(y\_test,y\_pred))

### Acccuracy 0.7186147186147186

Classification error rate: proportion of instances misclassified over the whole set of instances.

Error rate is calculated as the total number of two incorrect predictions (FN + FP) divided by the total number of a dataset (examples in the dataset.

Also error\_rate = 1- accuracy

total\_misclassified = cs[0,1] + cs[1,0] print(total\_misclassified) total\_examples = cs[0,0]+cs[0,1]+cs[1,0]+cs[1,1] print(total\_examples) print("Error rate",total\_misclassified/total\_examples)

print("Error rate ",1-metrics.accuracy\_score(y\_test,y\_pred))

### 65

**231**

### Error rate 0.2813852813852814

**Error rate 0.2813852813852814**

print("Precision score",metrics.precision\_score(y\_test,y\_pred))

### Precision score 0.6056338028169014

print("Recall score ",metrics.recall\_score(y\_test,y\_pred))

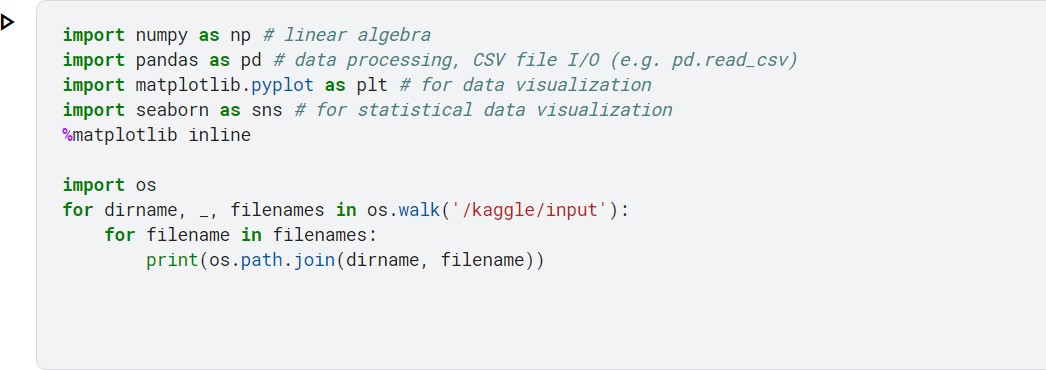
### Recall score 0.5375

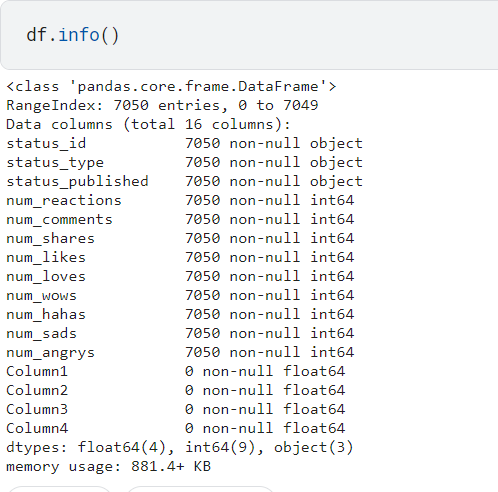
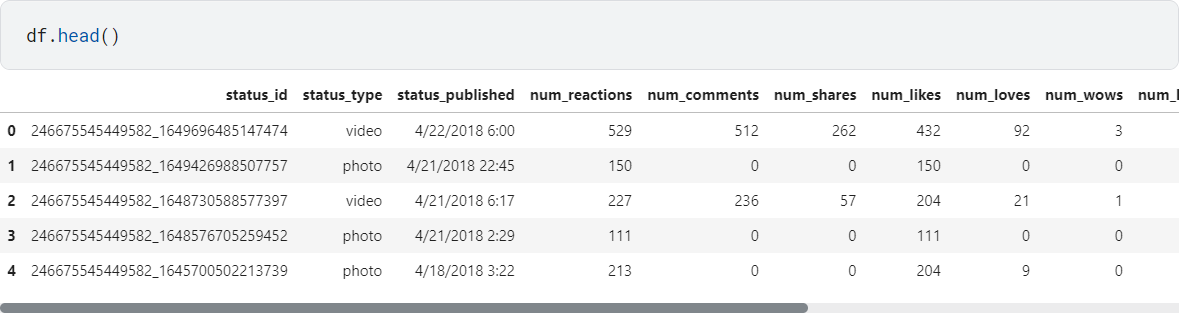
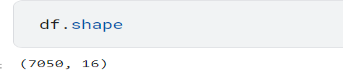
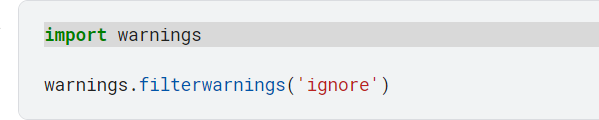
print("Classification report ",metrics.classification\_report(y\_test,y\_pred))

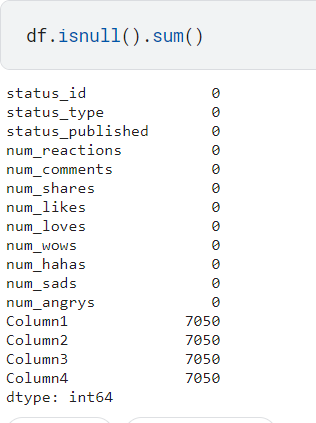
**Classification report precision recall f1-score support**

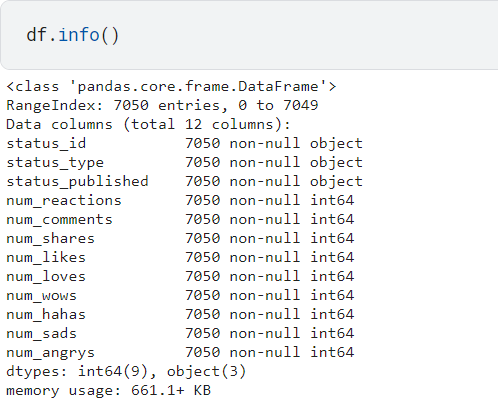
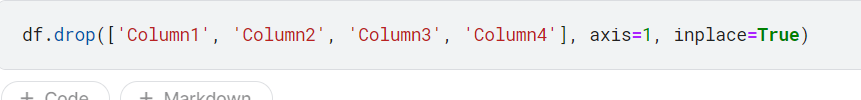
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **0** | **0.77** | **0.81** | **0.79** | **151** |
| **1** | **0.61** | **0.54** | **0.57** | **80** |
| **accuracy** |  |  | **0.72** | **231** |
| **macro avg** | **0.69** | **0.68** | **0.68** | **231** |
| **weighted avg** | **0.71** | **0.72** | **0.71** | **231** |

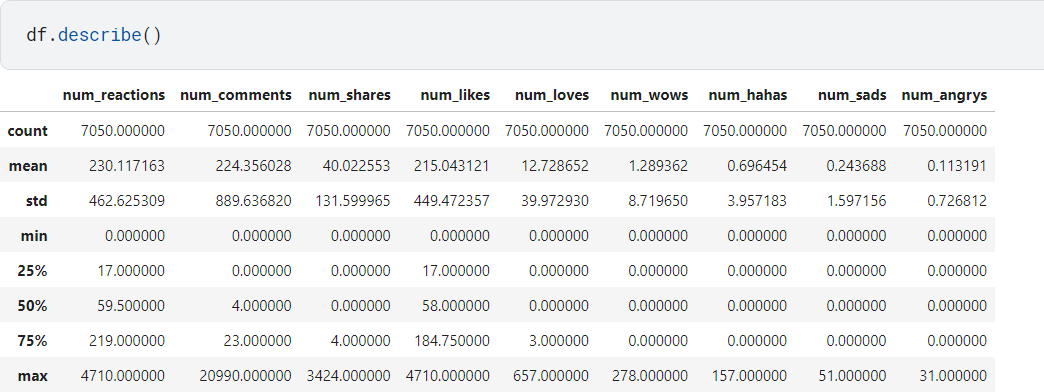
# Assignment No : 4

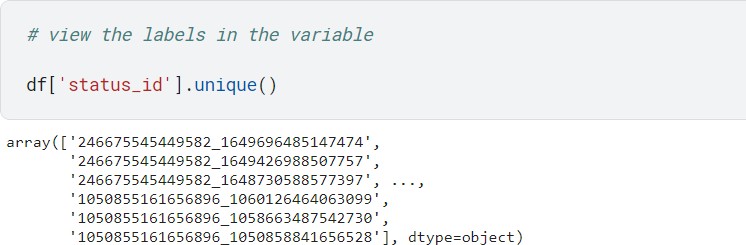


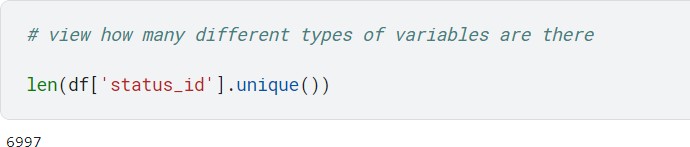


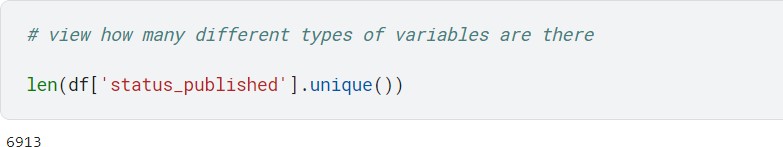
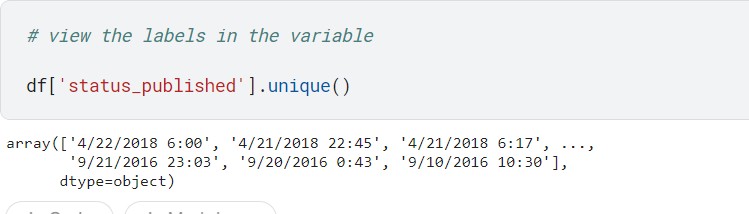


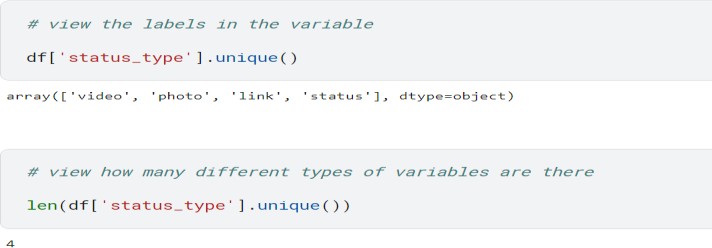




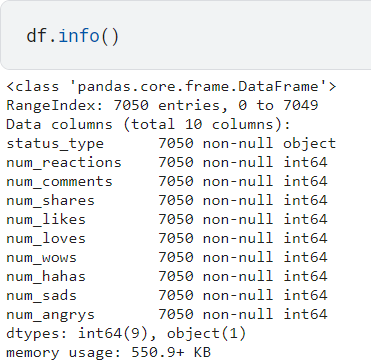


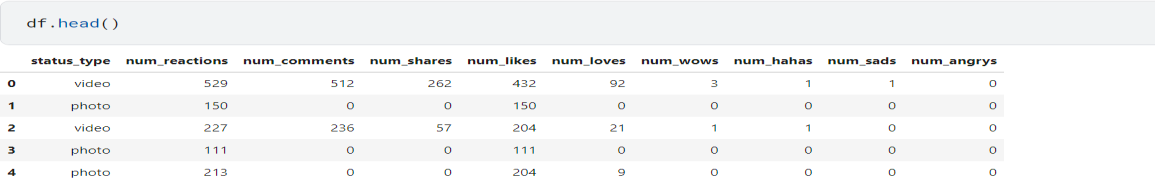


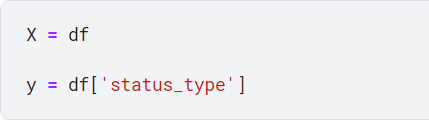




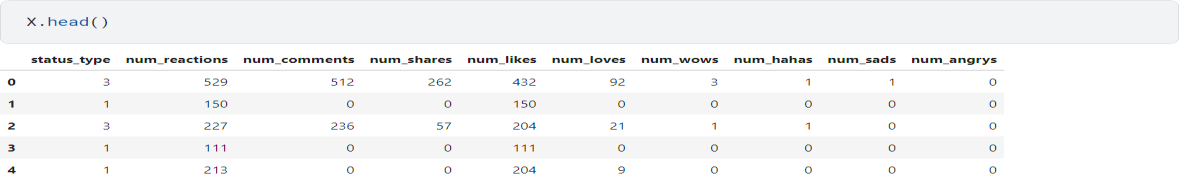
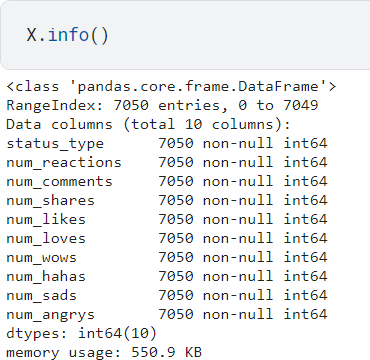


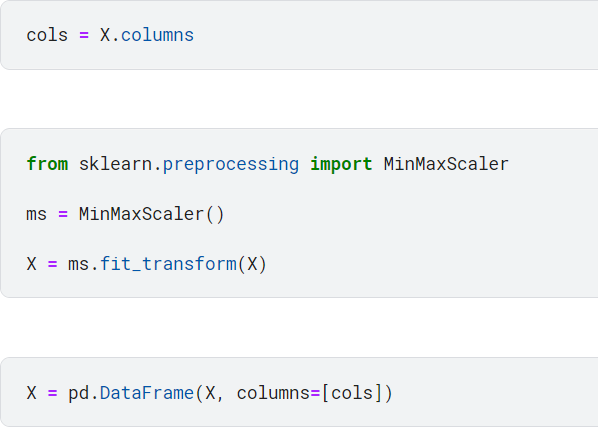


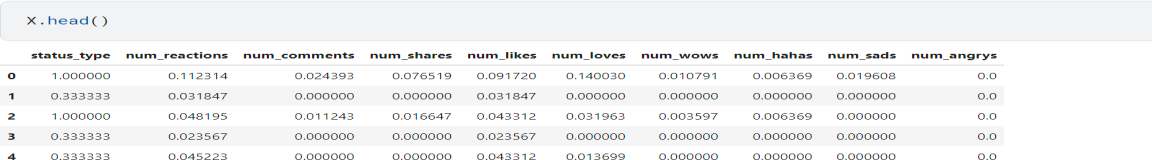


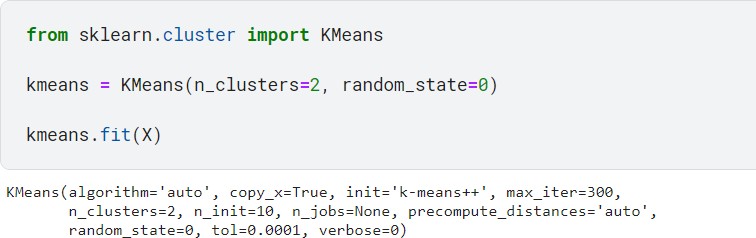


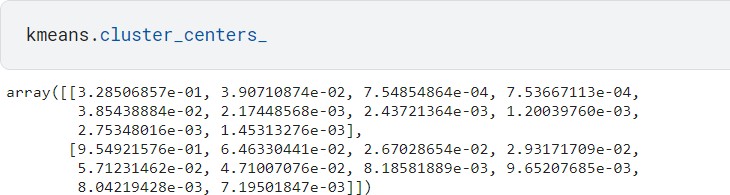


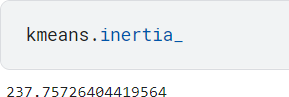




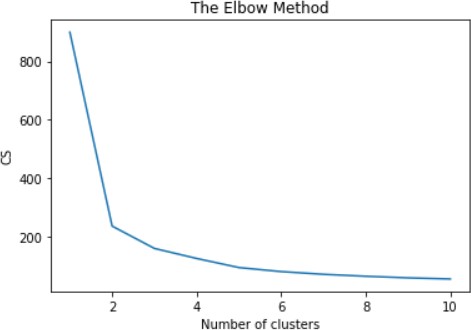


















**GROUP C**

## Assignment: 3

#### Loops in Solidity:

pragma solidity >= 0.5.0 < 0.9.0; contract Loops { uint [3] public arr; uint public count;

function Whileloop() public { while(count < arr.length) { arr[count] = count; count++;

}

}

function Forloop() public {

for(uint i=count; i<arr.length; i++) { arr[count] = count; count++;

}

}

**If-Else in Solodity :**

pragma solidity >= 0.5.0 < 0.9.0; contract Array {

function check(int a) public pure returns(string memory) { string memory value; if(a > 0) {

value = "Greater Than zero";

}

else if(a == 0) {

value = "Equal to zero";

}

else {

value = "Less than zero";

}

return value;

}

}

#### Create a Smart Contract with CRUD Functionality

pragma solidity ^0.5.0; contract Crud {

struct User {

uint id; string name;

}

User[] public users; uint public nextId = 0;

function Create(string memory name) public { users.push(User(nextId, name)); nextId++;

}

function Read(uint id) view public returns(uint, string memory) {for(uint i=0; i<users.length; i++) {

if(users[i].id == id) { return(users[i].id, users[i].name);

}

}

function Update(uint id, string memory name) public {for(uint i=0; i<users.length; i++) {

if(users[i].id == id) { users[i].nam e =name;

}

}

}

function Delete(uint id) public { delete users[id];

}

function find(uint id) view internal returns(uint) {for(uint i=0; i< users.length; i++) {

if(users[i].id

== id) { return i;

}

}

// if user does not exist then revert back revert("User does not exist");

}

}