Linear Algebra Assignment 1 (I & II included)

Sriram Radhakrishna PES1UG20CS435 Section : 'H'

Python code :

1. Hill cipher encryption using matrix inversion :

```
# implementation of basic cryptographic techniques to appreciate matrix
inversion

keyMatrix = [[0] * 3 for i in range(3)]
messageVector = [[0] for i in range(3)]
Cipher = [[0] for i in range(3)]
cipherMatrix = [[0] for i in range(3)]
DTextMatrix = [[0] for i in range(3)]


def transposeMatrix(m):
    return list(map(list,zip(*m)))

def getMatrixMinor(m,i,j):
    return [row[:j] + row[j+1:] for row in (m[:i]+m[i+1:])]

def getMatrixDeternminant(m):

    # base case for 3x3 matrix
    if len(m) == 2:
        return m[0][0]*m[1][1]-m[0][1]*m[1][0]

    determinant = 0

    for c in range(len(m)):
        determinant += ((-
1)**c)*m[0][c]*getMatrixDeternminant(getMatrixMinor(m,0,c))
        return abs(determinant)

def getmodInverse(a, m):
    for x in range(1, m):
        if (((a%m) * (x%m)) % m == 1):
            return x
    return -1

def getMatrixInverse(m):
    determinant = getMatrixDeternminant(m)

    if(getmodInverse(determinant,26) != -1):
        det = getmodInverse(determinant,26)

        # special case for 2x2 matrix:
        if len(m) == 2:
            return [[m[1][1]/determinant, -1*m[0][1]/determinant],[-
1*m[1][0]/determinant, m[0][0]/determinant]]
```

```
        # find matrix of cofactors
        cofactors = []
        for r in range(len(m)):
            cofactorRow = []

            for c in range(len(m)):
                minor = getMatrixMinor(m,r,c)
                cofactorRow.append(((-1)**(r+c)) *
                getMatrixDeternminant(minor))
                cofactors.append(cofactorRow)

        cofactors = transposeMatrix(cofactors)
        for r in range(len(cofactors)):
            for c in range(len(cofactors)):
                if (cofactors[r][c] < 0):
                    cofactors[r][c] = (cofactors[r][c] + 26) % 26
                cofactors[r][c] = (cofactors[r][c] * det) % 26

        return cofactors

    else:
        print('Non-invertible matrix. Decryption not possible')
        exit

def getKeyMatrix(key):
    k = 0
    for i in range(3):
        for j in range(3):
            keyMatrix[i][j] = ord(key[k]) % 65
            k += 1
    return keyMatrix

def encrypt(messageVector):
    for i in range(3):
        for j in range(1):
            cipherMatrix[i][j] = 0
            for x in range(3):
                cipherMatrix[i][j] += (keyMatrix[i][x] *
messageVector[x][j])
            cipherMatrix[i][j] = cipherMatrix[i][j] % 26

def decrypt(inverse,Cipher):
    for k in range(3):
        for l in range(1):
            DTextMatrix[k][l] = 0
            for y in range(3):
                DTextMatrix[k][l] += (inverse[k][y] * Cipher[y][l])
            DTextMatrix[k][l] = DTextMatrix[k][l] % 26

def HillCipher(message, key):

    # encryption

    keyMatrix = getKeyMatrix(key)

    for i in range(3):
        messageVector[i][0] = ord(message[i]) % 65

    encrypt(messageVector)

    CipherText = []
```

```
    for i in range(3):
        CipherText.append(chr(cipherMatrix[i][0] + 65))

    print('Cipher text : ', ''.join(CipherText))
    Ciphertext = ''.join([str(elem) for elem in CipherText])

    # decryption
    inverse = getMatrixInverse(keyMatrix)

    for i in range(3):
        Cipher[i][0] = ord(Ciphertext[i]) % 65

    if(inverse):
        decrypt(inverse,Cipher)
        Text = []
        for i in range(3):
            Text.append(chr(int(DTextMatrix[i][0]) + 65))

        print('Message vector : ', ''.join(Text))

# Driver Code
def main():
    message = input('Input a 3 Letter message(All in capital Letters): ')
    key = input('Input a 9 Letter key(All in Capital Letters): ')
    HillCipher(message, key)

# main
if __name__ == '__main__':
    main()
```

2. Implementation of Markov Chains for :

   - Population migration distribution between two Indian states :

```
import numpy as np
import random as rm

state = ["S", "T"]
transitionName = [["SS", "ST"], ["TS", "TT"]]
transitionMatrix = [[0, 100], [250, 0]]

if len(transitionMatrix) == 2:
    print("Move forward.")
else:
    print("Transition matrix error")


def pop_mig(transition):

    activityToday = "S"
    print("Start state: " + activityToday)
    activityList = [activityToday]
    i = 0
    prob = 0

    while i != transition:
```

```
        if activityToday == "S":
            change = np.random.choice(transitionName[0], replace=True)
            if change == "SS":
                prob = prob + 0
                activityList.append("S")
                pass
            else:
                prob = prob + 400
                activityToday = "ST"
                activityList.append("T")

        elif activityToday == "T":
            change = np.random.choice(transitionName[1], replace=True)
            if change == "TS":
                prob = prob + 500
                activityList.append("S")
                pass
            else:
                prob = prob + 0
                activityToday = "TT"
                activityList.append("T")

        else:
            return -1;

        i += 1

    print("Possible states : " + str(activityList))
    print("End state after " + str(transition) + " transition: " +
activityToday + ", current population of " + str(transition) + " : " +
str(prob))

pop_mig(1)
```

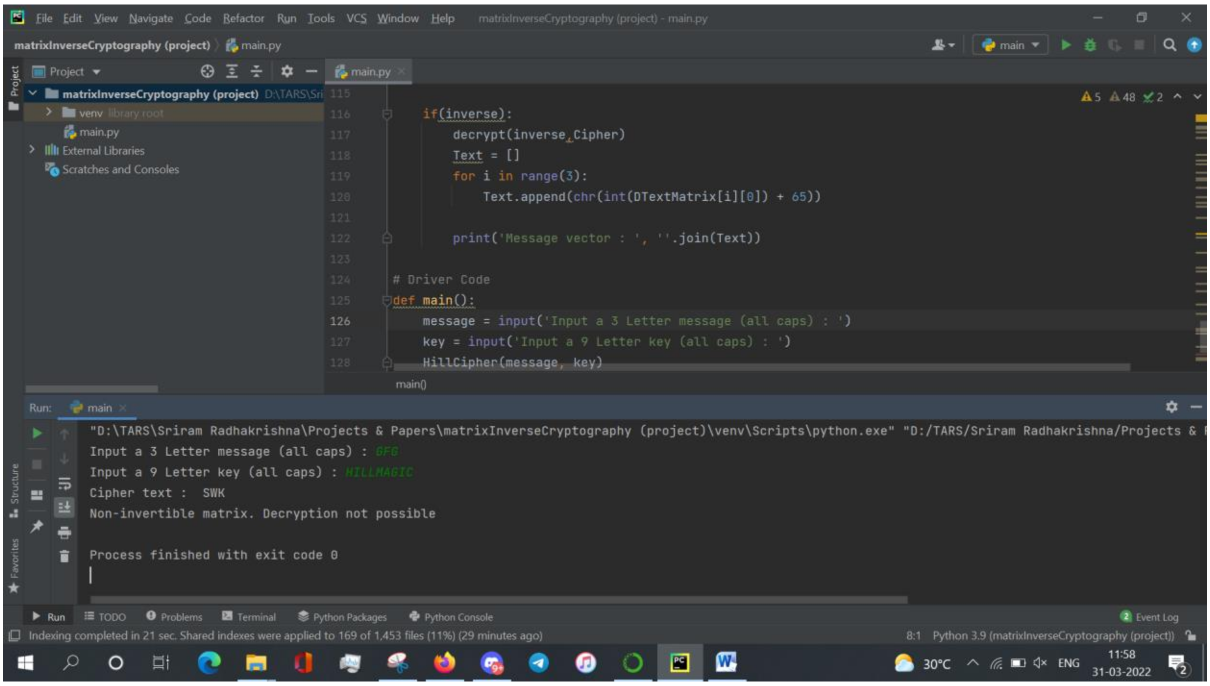
- Vote changing pattern of three political parties from one election to the next :

```
import numpy as np

states = ["A", "B", "C"]
transitionName = [["AA", "AB", "AC"], ["BA", "BB", "BC"], ["CA", "CB",
"CC"]]

if len(transitionName) == 3:
    print("move forward")
else:
    print("N/A")


def vote_change(elections):

    activityToday = "A"
    print("Start state: " + activityToday)

    activityList = [activityToday]
```

```
    i = 0

    while i != elections:

        if activityToday == "A":

            change = np.random.choice(transitionName[0], replace=True)

            if change == "AA":
                activityList.append("A")
                pass
            elif change == "AB":
                activityToday = "B"
                activityList.append("B")
            else:
                activityToday = "C"
                activityList.append("C")

        elif activityToday == "B":

            change = np.random.choice(transitionName[1], replace=True)
            if change == "BB":
                activityList.append("B")
                pass
            elif change == "BC":
                activityToday = "C"
                activityList.append("C")
            else:
                activityToday = "A"
                activityList.append("A")

        elif activityToday == "C":

            change = np.random.choice(transitionName[2], replace=True)

            if change == "CC":
                activityList.append("C")
                pass
            elif change == "CA":
                activityToday = "A"
                activityList.append("A")
            else:
                activityToday = "B"
                activityList.append("B")

        i += 1

    print("Possible states: " + str(activityList))
    print("After " + str(elections) + " elections the votes shifted to
party " + activityToday)

vote_change(4)
```

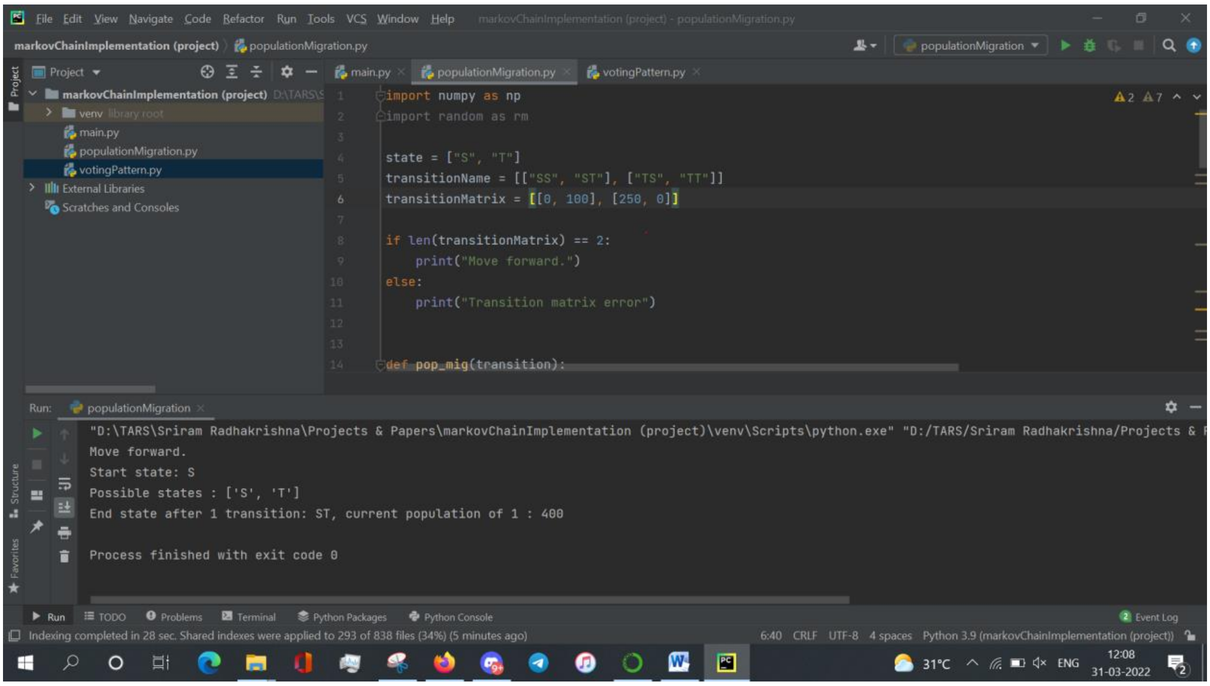
Output screenshots :

1. Hill cipher encryption using matrix inversion :



1. Implementation of Markov Chains for :

- Population migration distribution between two Indian states :



- Vote changing pattern of three political parties from one election to the next :