LINEAR ALGEBRA ASSIGNMENT

GAURAV MAHAJAN

SEM 4 SEC C

PES1UG20CS150

UNIT 1:

```
import numpy as np
def encryption(matrix):
    matrix= matrix.replace(" ", "")
    C = make_key()
    length_check = len(matrix) % 2 == 0
    if not length_check:
        matrix += "0"
    P = stringmatrix(matrix)
    message_length = int(len(matrix) / 2)
    encryptionmatrix = ""
    for i in range(message_length):
        row_0 = P[0][i] * C[0][0] + P[1][i] * C[0][1]
        integer = int(row_0 % 26 + 65)
        encryptionmatrix += chr(integer)
        row_1 = P[0][i] * C[1][0] + P[1][i] * C[1][1]
        integer = int(row_1 % 26 + 65)
        encryptionmatrix += chr(integer)
    return encryptionmatrix

def decryption(encryptionmatrix):
    C = make_key()
    determinant = C[0][0] * C[1][1] - C[0][1] * C[1][0]
    determinant = determinant % 26
    multiplicative_inverse = inverse(determinant)
    C_inverse = C
    C_inverse[0][0], C_inverse[1][1] = C_inverse[1, 1], C_inverse[0, 0]
    C[0][1] *= -1
    C[1][0] *= -1
    for row in range(2):
        for column in range(2):
            C_inverse[row][column] *= multiplicative_inverse
            C_inverse[row][column] = C_inverse[row][column] % 26

    P = stringmatrix(encryptionmatrix)
```

```
    m_len = int(len(encryptionmatrix) / 2)
    decryption_matrix = ""
    for i in range(m_len):
        column_0 = P[0][i] * C_inverse[0][0] + P[1][i] * C_inverse[0][1]
        integer = int(column_0 % 26 + 65)
        decryption_matrix += chr(integer)
        column_1 = P[0][i] * C_inverse[1][0] + P[1][i] * C_inverse[1][1]
        integer = int(column_1 % 26 + 65)
        decryption_matrix += chr(integer)
    if decryption_matrix[-1] == "0":
        decryption_matrix = decryption_matrix[:-1]
    return decryption_matrix

def inverse(determinant):
    multiplicative_inverse = -1
    for i in range(26):
        inverse = determinant * i
        if inverse % 26 == 1:
            multiplicative_inverse = i
            break
    return multiplicative_inverse


def make_key():
    determinant = 0
    C = None
    while True:
        cipher = input("Input 4 letter cipher: ")
        C = stringmatrix(cipher)
        determinant = C[0][0] * C[1][1] - C[0][1] * C[1][0]
        determinant = determinant % 26
        inverse_element = inverse(determinant)
        if inverse_element == -1:
            print("Determinant is not relatively prime to 26, uninvertible
key")
        elif np.amax(C) > 26 and np.amin(C) < 0:
            print("Only a-z characters are accepted")
            print(np.amax(C), np.amin(C))
        else:
            break
    return C

def stringmatrix(string):
    integers = [chr_to_int(c) for c in string]
    length = len(integers)
    M = np.zeros((2, int(length / 2)), dtype=np.int32)
    iterator = 0
    for column in range(int(length / 2)):
        for row in range(2):
```

```
            M[row][column] = integers[iterator]
            iterator += 1
    return M

def chr_to_int(char):
    char = char.upper()
    integer = ord(char) - 65
    return integer

if __name__ == "__main__":
    m = input("Message: ")
    encryptionmatrix = encryption(m)
    print(encryptionmatrix)
    decryption_matrix = decryption(encryptionmatrix)
    print(decryption_matrix)
```

UNIT 2

QUESTION 1:

```
from sre_parse import State
import numpy as np
import random as rd

State=["MH","BR","KA"]
migshift=[["MM","MK","MB"],["KM","KK","KB"],["BM","BK","BB"]]
rate=[[0.85,0.10,0.05],[0.20,0.60,0.20],[0.50,0.40,0.10]]

def migration_forecast(votes):
    current_State = "BR"
    Statelist = [current_State]
    i = 0
    prob = 1
    while i != votes:
        if current_State == "MH":
            change = np.random.choice(migshift[0],replace=True,p=rate[0])
            if change == "MM":
                prob = prob * 0.85
                Statelist.append("MH")
                pass
            elif change == "MK":
                prob = prob * 0.10
                current_State = "KA"
                Statelist.append("KA")
            else:
                prob = prob * 0.05
                current_State = "BR"
                Statelist.append("BR")
```

```
        elif current_State == "KA":
            change = np.random.choice(migshift[1],replace=True,p=rate[1])
            if change == "KK":
                prob = prob * 0.60
                Statelist.append("KA")
                pass
            elif change == "KM":
                prob = prob * 0.20
                current_State = "MH"
                Statelist.append("MH")
            else:
                prob = prob * 0.20
                current_State = "BR"
                Statelist.append("BR")
        elif current_State == "BR":
            change = np.random.choice(migshift[2],replace=True,p=rate[2])
            if change == "BB":
                prob = prob * 0.10
                Statelist.append("BB")
                pass
            elif change == "BM":
                prob = prob * 0.50
                current_State = "MH"
                Statelist.append("MH")
            else:
                prob = prob * 0.40
                current_State = "KA"
                Statelist.append("KA")
        i += 1
    return Statelist

list_state = []
count = 0

for iterations in range(1,10000):
        list_state.append(migration_forecast(2))

for smaller_list in list_state:
    if(smaller_list[2] == "MH"):
        count += 1

percentage = (count/10000) * 100
print("The probability of people migrating from 'Bihar' to 'Maharashtra'= " +
str(percentage) + "%")
```

QUESTION2

```
import numpy as np
import random as rd

party=["BJP","INC","AAP"]
voteshift=[["BB","BA","BI"],["AB","AA","AI"],["IB","IA","II"]]
voteshare=[[0.70,0.10,0.20],[0.10,0.60,0.30],[0.50,0.10,0.40]]

def voting_forecast(votes):

    current_party = "BJP"
    partylist = [current_party]
    i = 0
    prob = 1
    while i != votes:
        if current_party == "BJP":
            change =
np.random.choice(voteshift[0],replace=True,p=voteshare[0])
            if change == "BB":
                prob = prob * 0.70
                partylist.append("BJP")
                pass
            elif change == "BA":
                prob = prob * 0.10
                current_party = "AAP"
                partylist.append("AAP")
            else:
                prob = prob * 0.20
                current_party = "INC"
                partylist.append("INC")
        elif current_party == "AAP":
            change =
np.random.choice(voteshift[1],replace=True,p=voteshare[1])
            if change == "AA":
                prob = prob * 0.60
                partylist.append("AAP")
                pass
            elif change == "AB":
```

```
                prob = prob * 0.10
                current_party = "BJP"
                partylist.append("BJP")
            else:
                prob = prob * 0.30
                current_party = "INC"
                partylist.append("INC")
        elif current_party == "INC":
            change =
np.random.choice(voteshift[2],replace=True,p=voteshare[2])
            if change == "II":
                prob = prob * 0.40
                partylist.append("INC")
                pass
            elif change == "IB":
                prob = prob * 0.50
                current_party = "BJP"
                partylist.append("BJP")
            else:
                prob = prob * 0.10
                current_party = "AAP"
                partylist.append("AAP")
        i += 1
    return partylist

list_vote = []
count = 0

for iterations in range(1,10000):
        list_vote.append(voting_forecast(2))

for smaller_list in list_vote:
    if(smaller_list[2] == "AAP"):
        count += 1
percentage = (count/10000) * 100
print("The probability of votes transferring from 'BJP' to 'AAP'= " +
str(percentage) + "%")
```