**CE/CZ 1104 Linear Algebra for Computing**

Lab 1

**Instructions:** There are 3 exercises in this lab with questions for each exercise.

Exercise 1: Computer Security - Simple authentication scheme [1]

An authentication scheme allows a human to log onto a computer over an insecure network. The most familiar scheme is based on *passwords* where Harry, the human, sends his password to Carole, the computer, and the computer verifies that it is the correct password. However, if Eve, the eavesdropper, can read the bits going over the network, she can learn the password by observing just one log-in.

A more secure scheme is a *challenge-response* scheme. In a series of trials, Carole repeatedly asks Harry questions that someone not possessing the password would be unlikely to answer correctly. If Harry answers the questions correctly, Carole concludes that Harry knows the password. Let us look at an example.

Suppose the password is an *n*-bit string, i.e., a vectorwith *n* entries, where the entries are binary (zeros and ones), and chosen uniformly at random. More formally, the vector is said to be defined over *GF(2)*, which is short for *Galois Field 2*. The field *GF(2)* has only two elements, 0 and 1. Addition in *GF(2)* is modulo 2, i.e., equivalent to exclusive-OR. Multiplication in *GF(2)* is just like ordinary multiplication. In this exercise, all data is assumed to be in *GF(2)*.

In the *ith* trial, Carole selects a nonzero vector **c***i*, a *challenge vector*, and sends it to Harry. Harry is required to send back a single bit βi, which is supposed to be the dot product of **c***i* and the password . Carole then checks whether . If Harry passes enough trials, Carole concludes that Harry knows the password, and allows him to log in.

Question 1:

Suppose the password **x** is 10111. Harry initiates log-in. What is Harry’s response to Carole’s challenge vectors **c***1* = 01011 and **c***2* = 11110.

[NOTE: In Python, the dot product of two vectors **v***1* = 1101 and **v***2* = 1111 will return 3. However *GF(2)* has only the elements 0 and 1. To ensure that the answer is in *GF(2)*, you should compute 3 mod 2.]

**Sample code (copy and paste into Jupyter notebook)**

**import** **numpy** **as** **np**

*# Create vectors to represent c1,c2 and x*

c1 = np.array([0,1,0,1,1])

c2 = np.array([1,1,1,1,0])

x = np.array([1,0,1,1,1])

*#calculate the 1st response beta1 equals to the dot product of c1 and x*

beta1 = np.dot(c1,x)

beta1 = beta1 % 2 *#mod 2 as in GF(2)*

*#calculate the 2nd response*

beta2 = np.dot(c2,x)

beta2 = beta2 % 2

*#display the result*

print("c1 = ", c1)

print("c2 = ", c2)

print("x = ", x)

print("beta1 = ", beta1)

print("beta2 = ", beta2)

**Expected Output**

c1 = [0 1 0 1 1]

c2 = [1 1 1 1 0]

x = [1 0 1 1 1]

beta1 = 0

beta2 = 1

Question 2:

Enter Eve! Suppose Eve had observed Harry’s response ( and the first two challenge vectors (. Subsequently, she tries to login as Harry and Carole happens to send her as a challenge vector the sum of **c***1* = 01011 and **c***2* = 11110. Even though Eve does not know the password, she can use the distributive property to compute the dot product of this sum with the password **x**:

Find the response to this challenge vector without using **x**. Next, since you know the password, verify that this is indeed the correct response to the challenge vector by adding the terms in the bracket and taking the dot product with **x**.

**Sample code**

beta3 = (beta1 + beta2) % 2

*#double check*

c3 = (c1 + c2) % 2

beta3\_check = (np.dot(c3,x)) % 2

*#display the result*

print("beta3 = ", beta3)

print("beta3\_check = ", beta3\_check)

print("beta1 = ", beta1)

print("beta2 = ", beta2)

**Expected Output**

beta3 = 1

beta3\_check = 1

beta1 = 0

beta2 = 1

Question 3:

Extending the above idea, Eve can compute the right response to the sum of any number of previous challenges for which she has the right response. Mathematically,

Assume Eve knows the following challenges and responses:

|  |  |
| --- | --- |
| Challenge | Response |
| 110011 | 0 |
| 101010 | 0 |
| 111011 | 1 |
| 001100 | 1 |

Show how she can derive the right response to two new challenges ca = 011001 and cb = 110111. You can consider your Python script to be a function whose inputs are Ch and c, where Ch is the matrix whose rows are the challenges that she already knows (from the table) and c is either ca or cb. The output of the function will be the response (to ca or cb).

**Sample code**

*# Define challenges matrix*

**import** **numpy** **as** **np**

*# Define function to pick up two from known challenges set that*

*# summing them results the new challenge, then return the response for the new challenge.*

**def** getResponse(Ch,beta,c\_new):

C\_size = np.size(Ch,0)

**for** c1 **in** range(C\_size-1):

**for** c2 **in** range(c1+1,C\_size):

**if** np.array\_equal(np.logical\_xor(Ch[c1],Ch[c2]).astype(int),c\_new):

print("The new challenge ",c\_new)

print("match with XOR of the known challenges:")

print(Ch[c1], "with response:", beta[c1])

print(Ch[c2], "with response:", beta[c2])

print("The response for the new challenge is: ", np.logical\_xor(beta[c1],beta[c2]).astype(int))

**return**

print("Oops, there is not match")

Ch = np.array([

[1,1,0,0,1,1],

[1,0,1,0,1,0],

[1,1,1,0,1,1],

[0,0,1,1,0,0]])

*# Define response for challenges c1 to c4*

beta = np.array([0,0,1,1])

*# Define new challenge ca and cb*

ca = np.array([0,1,1,0,0,1])

cb = np.array([1,1,0,1,1,1])

*# Get response for new challenges*

getResponse(Ch,beta,ca)

print("")

getResponse(Ch,beta,cb)

**Expected Output**

The new challenge [0 1 1 0 0 1]

match with XOR of the known challenges:

[1 1 0 0 1 1] with response: 0

[1 0 1 0 1 0] with response: 0

The response for the new challenge is: 0

The new challenge [1 1 0 1 1 1]

match with XOR of the known challenges:

[1 1 1 0 1 1] with response: 1

[0 0 1 1 0 0] with response: 1

The response for the new challenge is: 0

Question 4:

Suppose Eve eavesdrops on communication, and learns *m* pairs such that is the correct response to challenge . Then the password **x** is a solution to

(1)

What is the condition on the vectors in eq. (1) to have a solution? In addition to the data in the table in Question 3, she observes the following two responses also:

|  |  |
| --- | --- |
| Challenge | Response |
| 011011 | 0 |
| 110100 | 1 |

Solve eq. (1) to find the password **x**. Use scipy.linalg.solve (see [here](https://docs.scipy.org/doc/scipy/reference/generated/scipy.linalg.solve.html)).

**Sample code**

*# Define challenges matrix*

C = np.array([

[1,1,0,0,1,1],

[1,0,1,0,1,0],

[1,1,1,0,1,1],

[0,0,1,1,0,0],

[0,1,1,0,1,1],

[1,1,0,1,0,0]])

*# Define response vector*

beta = np.array([0,0,1,1,0,1])

*# Find solution of Cx = beta*

**from** **scipy** **import** linalg

x = linalg.solve(C,beta) % 2 *#return x is a matrix with data in float type*

x = x.astype(int) *#convert data type from float to interger.*

print("x = ",x)

**Expected Output**

x = [1 0 1 0 0 1]

Exercise 2: Machine learning – Linear regression

Regression searches for relationships among variables [2].

For example, you can observe several employees of some company and try to understand how their salaries depend on the **features**, such as experience, level of education, role, city they work in, and so on.

This is a regression problem where data related to each employee represent one **observation**. The presumption is that the experience, education, role, and city are the independent features, while the salary depends on them.

Similarly, you can try to establish a mathematical dependence of the prices of houses on their areas, numbers of bedrooms, distances to the city center, and so on.

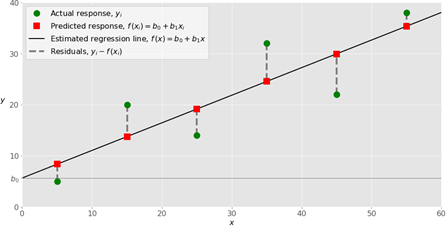
Let *xi* represent an independent feature, e.g., area of house and *yi* represent the dependent feature, e.g., price. Given a set of *n* data points {(*x*1,*y*1), (*x*2,*y*2),…,(*xn*,*yn*)}, we determine a function *ŷ=f*(*x*) that fits the data. The error in fitting data point *i* at (*xi*,*yi*) is the difference between *yi* and *f*(*xi*), i.e.

.

The error is also referred to as *residual*.

We determine the function *f*(*x*) such that the sum of the squares of the errors *ε* is minimized, where .

First, we consider a simple linear regression where a given set of *n* data points (green dots) are fitted on a straight line, i.e. *ŷ=mx+c*(see figure below). Here we determine the values of *m* and *c* such that *ε* is minimized.



Substituting  and  into *ε* , we get:



To determine the values of *m* and *c* that minimize *ε* , we take the derivatives of *ε* with respect to *m* and *c,* and equate to zero to obtain a system of linear equations, i.e.



The above set of linear equations can be written in matrix form as follows:

 (2)

The above equation in the form of can be solved for *m* and *c*. Hence we obtain the function *ŷ=mx+c.*

Question 5:

Consider the following data regarding house prices:

|  |  |  |
| --- | --- | --- |
| House | (area in 1000 sq ft) | *y* (price in 1000 dollars) |
| 1 | 0.846 | 115.00 |
| 2 | 1.324 | 234.50 |
| 3 | 1.150 | 198.00 |
| 4 | 3.037 | 528.00 |
| 5 | 3.984 | 572.50 |

* What is the matrix and the vector **b** for the above data?
* **WITHOUT** using scipy.linalg.solve, solve eq. (1) for *m* and *c*.
* Plot the data ( and the fitted line.

import matplotlib.pyplot as plt

xs = np.linspace(0,1,5)

ys = c + m\*xs

plt.plot(xs,ys,'r',linewidth=4) <- plots fitted line

plt.scatter(x,y); <- plots data(

plt.show()

**Sample code**

**import** **numpy** **as** **np**

x = np.array([0.846,1.324,1.150,3.037,3.984])

y = np.array([115.00,234.50,198.00,528.00,572.50])

c = 23.1

m = 148.2

**import** **matplotlib.pyplot** **as** **plt**

xs = np.linspace(0,5,5)

ys = c + m\*xs

fig1 = plt.figure()

plt.xlabel('Area in 1000 sq ft')

plt.ylabel('Price')

plt.plot(xs,ys,'r',linewidth=2) *# plots fitted line in red*

plt.scatter(x,y); *# plots real data(x,y) in blue dot*

plt.show()

**Expected Output**

Chart, scatter chart

Description automatically generated

Question 6:

Suppose we have additional information on the houses in the form of number of bedrooms. The above table including this information is shown below:

|  |  |  |  |
| --- | --- | --- | --- |
| House | (area in 1000 sq ft) | (number of bedrooms) | *y* (price in 1000 dollars) |
| 1 | 0.846 | 1 | 115.00 |
| 2 | 1.324 | 2 | 234.50 |
| 3 | 1.150 | 3 | 198.00 |
| 4 | 3.037 | 4 | 528.00 |
| 5 | 3.984 | 5 | 572.50 |

Now, it is no longer a line but a plane that should be fit in 3D space. The plane is of the form

.

[Note: Since *x2* represents categorical (discrete) data, fitting a plane is not totally correct because no one would be interested in fractional number of bedrooms! However, here, we ignore this discrepancy.]

If we have *n* observations, each observation *i* can be written as

.

In least squares linear regression, we want to minimize the sum of squared errors

which can be expressed as , where is the norm of a vector,

and .

It can be shown that the coefficients that minimize the sum of squared errors is the unique solution of the system

. (3)

Derivation of eq. (3) will be discussed in Part 2 of the course.

* Use scipy.linalg.solve to determine the vector .
* Plot the 3D data () and the fitted plane. [See [here](https://jakevdp.github.io/PythonDataScienceHandbook/04.12-three-dimensional-plotting.html) for help on 3D plotting]
* Using the obtained vector predict the prices of each of the 5 houses.

**Sample code**

**import** **numpy** **as** **np**

**import** **matplotlib.pyplot** **as** **plt**

**import** **scipy.linalg** **as** **la**

x1= np.array([0.846, 1.324, 1.150, 3.037, 3.984])

x2= np.array([1, 2, 3, 4, 5])

X = np.column\_stack([np.ones(len(x1)),x1, x2])

y = np.array([115.00, 234.50, 198.00, 528.00, 572.50])

print("Shape of X matrix: ",X.shape)

print(X)

a = la.solve(X.T @ X, X.T @ y)

print("parameters a0,a1,a2: ", a)

*# Use the model to predict the 5 houses*

yp=a[0]+a[1]\*x1+a[2]\*x2

**for** i **in** range(len(y)):

print("House ",i+1,": real price:", y[i], "**\t**predicted price:", "{:.1f}".format(yp[i]))

*# plot 3D figures*

**from** **mpl\_toolkits** **import** mplot3d

%matplotlib notebook

*#%matplotlib notebook will lead to interactive plots embedded within the notebook*

d1,d2 = np.meshgrid(range(5), range(5))

d3 = a[0] + a[1]\*d1 + a[2]\*d2 *##dimension 3 represent house price*

fig = plt.figure()

*# Add an axes*

ax = fig.add\_subplot(111,projection='3d')

*# plot the surface*

*# alpha is the transparancy of the plot, value between 0~1*

ax.plot\_surface(d1, d2, d3, alpha=0.2)

*# and plot the point*

ax.scatter(X[:,1], X[:,2], y, color='green')

*# add label for axes*

ax.set\_xlabel('Area in 1000 sq ft')

ax.set\_ylabel('Number of rooms')

ax.set\_zlabel('Price')

**Expected Output**

Shape of X matrix: (5, 3)

[[1. 0.846 1. ]

[1. 1.324 2. ]

[1. 1.15 3. ]

[1. 3.037 4. ]

[1. 3.984 5. ]]

parameters a0,a1,a2: [ 9.97566234 130.67172705 16.45635726]

House 1 : real price: 115.0 predicted price: 137.0

House 2 : real price: 234.5 predicted price: 215.9

House 3 : real price: 198.0 predicted price: 209.6

House 4 : real price: 528.0 predicted price: 472.7

House 5 : real price: 572.5 predicted price: 612.9

Chart

Description automatically generated

Exercise 3: Cryptography – Threshold secret sharing [1]

“Secret sharing refers to methods for distributing a secret amongst a group of participants, each of whom is allocated a share of the secret. The secret can be reconstructed only when a sufficient number, of possibly different types, of shares are combined together; individual shares are of no use on their own.”[3]. Secret sharing is used in applications such as secure multiparty computation, [Bitcoin signatures](https://bitcoinmagazine.com/articles/threshold-signatures-new-standard-wallet-security-1425937098), access control etc.

“In one type of secret sharing scheme there is one dealer and *n* players. The dealer gives a share of the secret to the players, but only when specific conditions are fulfilled will the players be able to reconstruct the secret from their shares. The dealer accomplishes this by giving each player a share in such a way that any group of *t* (for threshold) or more players can together reconstruct the secret but no group of fewer than *t* players can. Such a system is called a (*t*, *n*)-threshold scheme (sometimes it is written as an (*n*, *t*)-threshold scheme).” [3]. This scheme was invented independently by Shamir [4] and Blakely [5].

The mid-term exam is approaching! But there is a slight problem – the professor has to be away at a conference on the day of the exam. She has four teaching assistants (TAs) to help conduct the exam, but she does not want to provide each TA with the question papers because she does not trust them. What if one TA leaks the question paper ahead of time?! The question paper is password protected and the professor wants to split the password (secret) among the four TAs. She employs threshold secret sharing by which any 3 TAs could jointly recover the secret (it is risky to rely on all 4 TAs showing up for the exam) but any 2 TAs could not.

The professor uses *GF(2)* to define ten 6D vectors **a**0, **b**0, **a**1, **b**1, **a**2, **b**2, **a**3, **b**3, **a**4, **b**4, i.e., there are 6 entries in each vector and the entries are either 0 or 1. They can be considered to be forming 5 pairs: Pair 0 consists of **a**0 and **b**0, Pair 1 consists of **a**1 and **b**1, and so on. The requirement is that for any 3 pairs, the corresponding six vectors are linearly independent. These vectors are known to everyone.

Now, suppose the professor wants to share two secret bits *s* and *t*. She chooses a secret 6D vector **u** that is randomly generated such that and . She then gives TA1 the two bits and . TA2 gets two bits and and similarly for TA3 and TA4. Each TA’s share thus consists of a pair of bits.

Recoverability: How can 3 TAs get together to recover the secret bits *s* and *t* ? Suppose TA1, TA2 and TA3 come together. They can use their bits and the 6D vectors to solve the following equation (here **a**i’s and **b**i’s represent row vectors) for **u**:

.

With the knowledge of **u**, the TAs can recover the secret bits. Since the vectors **a**i and **b**i are linearly independent, the matrix is invertible and hence, there is a unique solution to the equation.

Secrecy: Can 2 TAs recover the secret bits? Suppose TA1 and TA2 go rogue and try to recover *s* and *t*. The system of equations becomes (note that **a**i’s and **b**i’s are row vectors)

,

where the first two entries on the right hand side are guessed values of *s* and *t*. Since the vectors **a**0, **b**0, **a**1, **b**1, **a**2, **b**2, are linearly independent, the matrix is invertible and there is a unique solution, no matter what bit is chosen as guess-s and as guess-t. This shows that the shares of TA1 and TA2 tell them nothing about the true values of *s* and *t*.

Question 7:

Define and **.**

* Write a function random\_vector(s, t) which takes as inputs bits *s* and *t* and outputs a 6D random binary vector **u**  (vector with 6 elements) such that and

.

* The next goal is to select vectors **a**1, **b**1, **a**2, **b**2, **a**3, **b**3, **a**4, **b**4 so that the requirement – for any three pairs, the corresponding six vectors are linearly independent – is satisfied.
* Let us say the password is “Potter”. Convert this password string to bits using function str2bits(s). Transform the generated list of bits to a matrix. Each column of this matrix represents the *s* and *t* bits. For each column of this matrix, use the previous function random\_vector(s,t) to obtain a corresponding secret vector **u**. Thus, there will be *n* secret vectors such that and .
* Next, we have to generate the secret bits for each of the TAs. This can be done by taking the dot product of **a***i*, **b***i* with **u***i*.
* Recovery: Choose any 3 TAs and their corresponding secret bits and recover the first secret vector **u**1 by solving (here we have chosen TAs 1, 2 and 3)

.

With the obtained **u**1, find the secret bits *s*1 and *t*1. Continue this process until all the **u***i*’s are obtained and from which the secret bits *s*i and *t*i are recovered. Convert the bits to a string using function bits2str(bitsArray) to check that the password “Potter” is recovered.

**Sample code**

**import** **numpy** **as** **np**

**import** **random**

*'''Function to create the random binary string'''*

**def** rand\_key(p):

key = [] *# Variable to store in an array*

**for** i **in** range(p): *# Loop to find the string of desired length*

key.append(random.randint(0, 1)) *# randint function to generate 0, 1 randomly*

**return**(key)

*'''Function to generate random vector u based on a0,b0 and s,t'''*

**def** random\_vector(a0, b0, s, t):

**for** n **in** range(200): *# Quit after trying 200 times*

u = rand\_key(6) *#generate a 6 dimension random vector*

**if** np.dot(a0,u) == s **and** np.dot(b0,u) == t:

**return** u

*'''Function to check if the generated vectors fulfil the requirements'''*

*#a and b each contains 5 vectors.*

**def** check\_dependency(a,b):

*# check if any 3 pairs of vectors from (a1,b1),(a2,b2),(a3,b3),(a4,b4) are linearly independent*

**for** v1 **in** range(1,3): *#1st vector from 1 to 2*

**for** v2 **in** range(v1+1,4):

**for** v3 **in** range(v2+1,5):

squareMatrix = np.vstack((a[v1], b[v1], a[v2], b[v2],a[v3], b[v3]))

determinant = np.linalg.det(squareMatrix)

**if** determinant == 0: *#if determinant is 0, the vectors are not linearly dependent*

**return** **False**

*# check if a0,b0 and any two random selected pairs of vectors are linearly independent*

**for** v1 **in** range(1,4): *#1st vector from 1 to 3*

**for** v2 **in** range(v1+1,5):

squareMatrix = np.vstack((a[0], b[0], a[v1], b[v1], a[v2], b[v2]))

determinant = np.linalg.det(squareMatrix)

**if** determinant == 0: *#if determinant is 0, the vectors are not linearly dependent*

**return** **False**

**return** **True**

*'''Function to converting String to binary array'''*

*'''The ord() function returns an integer representing the Unicode character.'''*

**def** str2bits(s):

res = ''.join(format(ord(i), 'b') **for** i **in** s)

bitsArray = []

**for** i **in** res:

bitsArray.append(int(i))

**return** bitsArray

*'''Function to converting binary array to String'''*

**def** bits2str(bitsArray):

NumOfChar = int(len(bitsArray)/7)

string = ''

**for** i **in** range(NumOfChar):

bitsChar = ''.join(str(j) **for** j **in** bitsArray[7\*i:7\*i+7]) *# 7 digits represents 1 char*

decimalChar = int(bitsChar,2) *#convert binary to decimal*

string = string + chr(decimalChar) *#convert decimal to string*

**return** string

*'''Function to convert a list of bits into a matrix with nrows rows'''*

**def** bits2mat(bits,nrows=2):

ncols = len(bits)//nrows *#Floor division: digits after the decimal point are removed.*

mat = []

**for** row **in** range(nrows):

idxStart = row\*ncols

idxEnd = idxStart + ncols

mat.append(bits[idxStart:idxEnd])

**return** mat

*'''PROGRAM START HERE'''*

*''' Create vectors a0, b0'''*

a0= [1,1,0,1,0,1]

b0= [1,1,0,0,1,1]

*'''Randomly generate 4 vector pairs (a1,b1),(a2,b2),(a3,b3),(a4,b4) until they fulfil below requirements'''*

*'''1. any 3 pairs from (a1,b1),(a2,b2),(a3,b3),(a4,b4) are linearly independent'''*

*'''2. a0,b0 and any two pairs of vectors are linearly independent'''*

tryTimes = 10000 *#maximum times the system will try to generate vector set*

**for** n **in** range(tryTimes):

a1 = rand\_key(6)

b1 = rand\_key(6)

a2 = rand\_key(6)

b2 = rand\_key(6)

a3 = rand\_key(6)

b3 = rand\_key(6)

a4 = rand\_key(6)

b4 = rand\_key(6)

*#put generated vectors into a list for easy handling at later process*

a = [a0,a1,a2,a3,a4]

b = [b0,b1,b2,b3,b4]

**if** check\_dependency(a,b) == **True**:

**break**

**if** n== (tryTimes-1) :

print("fail to generate vectors satisfied the conditions after 10000 tries, please rerun.")

exit()

**for** i **in** range(5):

*#print the generated vectors*

print("a",i,":",a[i], "**\t** b",i,":",b[i])

*'''initializing password string '''*

password = "Potter"

print("**\n**The password is:", password)

*'''Converting String to binary array'''*

passBits = str2bits(password)

print("**\n**The binary array of given string is:**\n**", passBits)

*'''Converting binary array to 2 x n matrix, each column of the matrix is a pair of s,t'''*

passMat = bits2mat(passBits)

print("**\n**The converted 2xn matrix is:**\n**", passMat)

*'''Generate the set of u using a0,b0,s,t'''*

s = passMat[0]

t = passMat[1]

pairNum = len(s) *#21 pairs of s&t*

u = np.empty((pairNum,6)) *#create a matrix to keep u vectors*

**for** i **in** range(pairNum):

u[i] = random\_vector(a0,b0,s[i],t[i])

print("**\n**u is a set of",pairNum,"6-dimension vectors","**\n**",u)

*'''Generate the secret bits β,γ for each TA'''*

β = np.empty((5,21))

γ = np.empty((5,21))

**for** i **in** range(5): *# for each TA (TA1 to TA4)*

**for** j **in** range(21): *# Generate 21 (number of u) pairs of β & γ*

β[i][j] = np.dot(a[i],u[j])

γ[i][j] = np.dot(b[i],u[j])

*'''Recovery'''*

t1, t2, t3 = np.sort(random.sample([1,2,3,4],3)) *# randomly choose 3 TAs*

print("**\n**Recovery with TA",t1,t2,t3)

TA3ab = np.stack((a[t1],b[t1],a[t2],b[t2],a[t3],b[t3]))

*# Sample: Get u1*

TA3βγ = np.vstack((β[t1][0],γ[t1][0],β[t2][0],γ[t2][0],β[t3][0],γ[t3][0]))

**from** **scipy** **import** linalg

u1 = linalg.solve(TA3ab,TA3βγ) % 2

print("Calculated u1 = ",u1.T)

print("Actual u1 = ",u[0])

**if** (u1.T==u[0]).all():

print("Bingo! they are the same!")

**else**:

print("OHOH, something went wrong, they are different!")

*'''Recover all u'''*

TA3βγ = np.vstack((β[t1],γ[t1],β[t2],γ[t2],β[t3],γ[t3]))

**from** **scipy** **import** linalg

u\_recovered = linalg.solve(TA3ab,TA3βγ) % 2

print("**\n**Recovered full set of u:**\n**", u\_recovered.T)

*#Recover s and t*

s\_recovered = np.dot(a[0],u\_recovered)

t\_recovered = np.dot(b[0],u\_recovered)

print("**\n**Recovered s,t:**\n**", s\_recovered,t\_recovered)

*#Convert binary back to string*

bitsArray = s+t

passwordRecover = bits2str(bitsArray)

print("password recovered is:",passwordRecover)

**Expected Output**

a 0 : [1, 1, 0, 1, 0, 1] b 0 : [1, 1, 0, 0, 1, 1]

a 1 : [0, 0, 0, 0, 1, 1] b 1 : [1, 1, 0, 1, 0, 0]

a 2 : [0, 0, 1, 0, 1, 1] b 2 : [0, 1, 1, 0, 1, 0]

a 3 : [0, 1, 0, 1, 0, 1] b 3 : [1, 0, 1, 1, 1, 1]

a 4 : [0, 0, 1, 0, 0, 1] b 4 : [1, 0, 0, 0, 1, 1]

The password is: Potter

The binary array of given string is:

[1, 0, 1, 0, 0, 0, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1, 1, 1, 0, 0, 1, 0]

The converted 2xn matrix is:

[[1, 0, 1, 0, 0, 0, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 0], [1, 1, 1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1, 1, 1, 0, 0, 1, 0]]

u is a set of 21 6-dimension vectors

[[0. 1. 0. 0. 0. 0.]

[0. 0. 1. 0. 1. 0.]

[0. 0. 0. 0. 0. 1.]

[0. 0. 0. 0. 0. 0.]

[0. 0. 0. 0. 1. 0.]

[0. 0. 0. 0. 0. 0.]

[0. 0. 1. 0. 0. 0.]

[0. 0. 0. 0. 0. 1.]

[0. 0. 0. 1. 1. 0.]

[0. 0. 0. 0. 0. 0.]

[0. 0. 0. 1. 0. 0.]

[0. 0. 0. 1. 1. 0.]

[0. 0. 0. 1. 0. 0.]

[1. 0. 0. 0. 0. 0.]

[1. 0. 0. 0. 0. 0.]

[0. 0. 1. 0. 0. 1.]

[0. 1. 1. 0. 0. 0.]

[0. 0. 1. 0. 0. 0.]

[0. 0. 0. 1. 0. 0.]

[0. 0. 1. 0. 1. 0.]

[0. 0. 0. 0. 0. 0.]]

Recovery with TA 1 2 4

Calculated u1 = [[0. 1. 0. 0. 0. 0.]]

Actual u1 = [0. 1. 0. 0. 0. 0.]

Bingo! they are the same!

Recovered full set of u:

[[0. 1. 0. 0. 0. 0.]

[0. 0. 1. 0. 1. 0.]

[0. 0. 0. 0. 0. 1.]

[0. 0. 0. 0. 0. 0.]

[0. 0. 0. 0. 1. 0.]

[0. 0. 0. 0. 0. 0.]

[0. 0. 1. 0. 0. 0.]

[0. 0. 0. 0. 0. 1.]

[0. 0. 0. 1. 1. 0.]

[0. 0. 0. 0. 0. 0.]

[0. 0. 0. 1. 0. 0.]

[0. 0. 0. 1. 1. 0.]

[0. 0. 0. 1. 0. 0.]

[1. 0. 0. 0. 0. 0.]

[1. 0. 0. 0. 0. 0.]

[0. 0. 1. 0. 0. 1.]

[0. 1. 1. 0. 0. 0.]

[0. 0. 1. 0. 0. 0.]

[0. 0. 0. 1. 0. 0.]

[0. 0. 1. 0. 1. 0.]

[0. 0. 0. 0. 0. 0.]]

Recovered s,t:

[1. 0. 1. 0. 0. 0. 0. 1. 1. 0. 1. 1. 1. 1. 1. 1. 1. 0. 1. 0. 0.] [1. 1. 1. 0. 1. 0. 0. 1. 1. 0. 0. 1. 0. 1. 1. 1. 1. 0. 0. 1. 0.]

password recovered is: Potter

References

1. Philip N. Klein, “Coding the matrix: Linear algebra through computer science applications”, Newtonian Press, 2013.
2. <https://realpython.com/linear-regression-in-python/>].
3. Wikipedia contributors, “Secret sharing”, Wikipedia, The Free Encyclopedia. May 20, 2020, at 16:27 UTC. Available at: <https://en.wikipedia.org/w/index.php?title=Secret_sharing&action=history> . Accessed June 26, 2020.
4. A. Shamir, ["How to share a secret"](https://cs.jhu.edu/~sdoshi/crypto/papers/shamirturing.pdf)  *Communications of the ACM*, vol. 22, no. 11, pp. 612-613, Nov. 1979.
5. G. R. Blakley, ["Safeguarding Cryptographic Keys"](https://pdfs.semanticscholar.org/32d2/1ccc21a807627fcb21ea829d1acdab23be12.pdf), *Managing Requirements Knowledge, International Workshop on AFIPS*. vol. **48**, pp. 313–317, 1979.