Hill Cipher

Criptography
Carmen Azorín Martí

March 22th 2024

1 Introduction

The code written in Python will use four external .txt files. These files contain an original message (*text.txt*), a string that represents the key (*key.txt*) and the modified message, after the desired encryption (*text_en.txt*) or decryption operation (*text_de.txt*).
In addition, the Python code will use the *Unicode* library that allows us to read a file containing letters of the Polish alphabet. Besides, it will also use the *codecs* library, used to open and read a file saving the original letters.
Since the algorithm needed to execute the Hill Cipher has to do matrix calculations, we are using the *numpy* and *math* libraries.
Suppose our *text.txt* file contains the message

act

which we want to encrypt. And the key we get from *key.txt* is

GYBNQKURP

Let's see how the encryption works.

2 Implemented functions

The first function implemented in Pyhton is called *removeUnnecesaryChars()* and receives a string as parameter. This line of text could contain characters other than letters, such as spaces, line breaks, etc. What the function returns is the string removing all non-alphabetic characters.

```python
def removeUnnecesaryChars(text):
    return ''.join(letter for letter in text if letter.isalpha())
```

An example of this function is exposed in the Caesar Cipher report.
Before explaining the functions that encrypt or decrypt a message, we need to explain the functions they will call. These functions do calculations of $\mathbf{Z_N}$ matrix operations.
The first of these is called *modInverse()* and, as its name indicates, calculates the inverse of a number modulo N. It therefore receives these two parameters.
As the space is $\mathbf{Z_N}$, the inverse X of the integer A must be an integer between 0 and N-1 that fulfills:

$$A \cdot X \mod N = 1$$

The inverse may not exist. In that case, the function returns -1.

```python
    def modInverse(A, N):

    for X in range(1, N-1):
        if ((A* X) % N == 1):
            return X
    return -1
```

For example, to calculate the inverse of 25 in $\mathbf{Z}/\mathbf{26Z}$, we should try the integers from 0 to 25 and check which one fulfills $25 \cdot X \equiv 1 \pmod{26}$. In fact, $25 \cdot 25 = 625 \equiv 1 \pmod{26}$, so $25^{-1} \equiv 25 \pmod{26}$.

Although in the project we are supposed to use 2x2 dimension matrices, I wanted to add 3x3 matrices to make the code more complete. In particular, this *inverseMatrixDim3()* function returns the inverse of a 3x3 matrix modulo 26. We know that if

$$A = \begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix}$$

is an invertible 3x3 matrix, then its inverse is

$$A^{-1} = \frac{1}{det(A)} \begin{pmatrix} ei - fh & ch - bi & bf - ce \\ fg - di & ai - cg & cd - af \\ dh - eg & bg - ah & ae - bd \end{pmatrix}$$

Here, $\frac{1}{det(A)}$ means the inverse in $\mathbf{Z}/\mathbf{mZ}$ if our matrix entries are in $\mathbf{Z}/\mathbf{mZ}$.
We take into account that the matrix A with entries in $\mathbf{Z}/\mathbf{mZ}$ has inverse if and only if $gdc(det(A), m) = 1$. Although in our function we will assume that this constraint has been checked before calling it.

Therefore, our function will calculate the determinant of the matrix A using *numpy* and calculate its inverse modulo $\mathbf{Z}/\mathbf{26Z}$ using the above function. It will then apply the above formula.
Each element of the calculated matrix must be in $\mathbf{Z}/\mathbf{26Z}$, so we have to calculate its value modulo 26. Also, as the formula indicates, each element will be multiplied by the inverse of the determinant of A.

```python
def inverseMatrixDim3(matrix):
    detMatrix = int(np.linalg.det(matrix))
    inverseDetMatrix = modInverse(detMatrix%26,26)

    a = matrix[0,0]
    b = matrix[0,1]
    c = matrix[0,2]
    d = matrix[1,0]
    e = matrix[1,1]
    f = matrix[1,2]
    g = matrix[2,0]
    h = matrix[2,1]
    i = matrix[2,2]

    inverse = np.array([[e*i-f*h,c*h-b*i,b*f-c*e],
                        [f*g-d*i,a*i-c*g,c*d-a*f],
                        [d*h-e*g,b*g-a*h,a*e-b*d]])

    for r in range(3):
        for c in range(3):
            inverse[r,c] = ((inverse[r,c]%26)*inverseDetMatrix)%26
    return inverse
```

An example of the use of the function would be: create a matrix of elements of $\mathbf{Z}/\mathbf{26Z}$ that we know has an inverse. Let

$$A = \begin{pmatrix} 6 & 24 & 1 \\ 13 & 16 & 10 \\ 20 & 17 & 15 \end{pmatrix} \longrightarrow det(A) = 441$$

Since $gdc(441, 26) = 1$ the matrix A has inverse in $\mathbf{Z}/\mathbf{26Z}$. We have that $441 \equiv 25 \pmod{26}$ and, as said before, $25^{-1} \equiv 25 \pmod{26}$.
We could define this matrix in Python as follows:

```python
np.array([[ 6, 24,  1],[13, 16, 10],[20, 17, 15]])
```

If we call the function with this matrix as an argument we receive the following:

```
array([[ 8,  5, 10],
       [21,  8, 21],
       [21, 12,  8]])
```

Let's check that the received function is the inverse in $\mathbf{Z}/\mathbf{26Z}$:

$$\begin{pmatrix} 6 & 24 & 1 \\ 13 & 16 & 10 \\ 20 & 17 & 15 \end{pmatrix} \cdot \begin{pmatrix} 8 & 5 & 10 \\ 21 & 8 & 21 \\ 21 & 12 & 18 \end{pmatrix} = \begin{pmatrix} 573 & 234 & 572 \\ 650 & 313 & 546 \\ 832 & 416 & 677 \end{pmatrix} \equiv \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \pmod{26}$$

The following function *inverseMatrixDim2()* does the same as the previous one but with 2x2 matrices. Bearing in mind that: If

$$A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

is an invertible 2x2 matrix, then its inverse is

$$A = \frac{1}{det(A)} \begin{pmatrix} d & -b \\ -c & a \end{pmatrix}$$

Therefore, we define this function analogously to the previous one. If we look at it, we save a loop, since the inverse matrix formula has a single number in each position and is less tedious to calculate.

```
def inverseMatrixDim2(matrix):
    inverse = np.array([[0,0],[0,0]])
    detMatrix = int(np.linalg.det(matrix))
    inverseDetMatrix = modInverse(detMatrix,26)
    inverse[0,0] = (matrix[1,1]*inverseDetMatrix)%26
    inverse[0,1] = (-matrix[0,1]*inverseDetMatrix)%26
    inverse[1,0] = (-matrix[1,0]*inverseDetMatrix)%26
    inverse[1,1] = (matrix[0,0]*inverseDetMatrix)%26
    return inverse
```

The following function is called *stringToMatrix()* and receives two parameters:

- text. corresponds to the string to be converted to an array of integers.
- dim1. corresponds to the number of columns the matrix is to have

The number of rows can be calculated from the quotient of the length of the string and the number of columns. In case the remainder is not null, it will be filled with random numbers. Therefore,

$$dim_0 = ceil\frac{len(text)}{dim_1}$$

Once the dimension of the matrix is known, we can generate a random $\mathbf{Z}/\mathbf{mZ}$ matrix, for the extra elements if any.
Each character in the text will be converted to an integer using the *ord()* method and its position in the alphabet will be calculated by subtracting *ord('a')*. Remember that we have converted the text to lowercase. Now we can add the integer corresponding to each letter to the matrix, from left to right and from top to bottom.

```
def stringToMatrix(text,dim1):
    text = text.lower()
    dim0 = math.ceil(len(text)/dim1)
    text_matrix = np.random.randint(0,25,(dim0, dim1))
    row = 0
    column = 0
    for character in text:
        number = ord(character) - ord('a')
        text_matrix[row,column] = number
        if(column == dim1-1):
            row += 1
        column = (column+1)%dim1
    return text_matrix
```

Let's show what happens if we pass the string "GYBNQKUR" to the *stringToMatrix()* function by giving it 3 columns.

```
    stringToMatrix("GYBNQKUR",3)
```

After passing the string to lower case, the number of rows will be calculated:

$$dim_0 = ceil\frac{len("GYBNQKUR")}{dim_1} = ceil\frac{8}{3} = 3$$

And a 3x3 matrix of random integers modulo 26, called text_matrix, is created. And now it would go in order of the text: strint with character = g,

$$\begin{aligned} ord(character) - ord('a') &= 103 - 97 \\ &= 6 \end{aligned}$$

So in the position (0,0) of the matrix a 6 is entered. Then character=y and in the position (0,1) a 24 is entered. Finally, we get

$$\begin{pmatrix} 6 & 24 & 1 \\ 13 & 16 & 10 \\ 20 & 17 & 15 \end{pmatrix}$$

The number in the last position (2,2) is random, in order to adjust the length of the string to the size of the matrix. The returned array is

```
array([[ 6, 24,  1],
       [13, 16, 10],
       [20, 17, 15]])
```

The next function is *isCorrectKey()* which receives two parameters:

- key. a string that is converted to an array and will be used for encrypting or decrypting
- dim. an integer indicating the size of the array.

The key is correct if :

- it can be converted to a square matrix, i.e. the same number of rows and columns (given by dim).
- the greatest common divisor of the corresponding matrix and 26 is 1.

Therefore, our method checks the first constraint. It then calculates the matrix associated to the key using stringToMatrix to check the second constraint.

```
def isCorrectKey(key,dim):
    if len(key) != dim*dim:
        return False
    key_matrix = stringToMatrix(key,dim)
    determinant = np.linalg.det(key_matrix)
    if(math.gcd(int(determinant),26) != 1):
        return False
    return True
```

In our example, the string "GYBNQKUR" is not a correct key, since $2 \cdot 2 < dim("GYBNQKURP") < 3 \cdot 3$, so it doesn't fulfill the first constraint. So, for encrypting and decrypting we are using the key "GYBNQKURP".

The following functions are actually the most important of the whole project. They are the functions that encrypt and decrypt depending on a key.
The *encryptText()* function receives three parameters:

- Text. the message to be encrypted
- Key. is the string used to encrypt the text by its matrix
- Dim. dimension of matrices

The Hill Cipher algorithm tells us to multiply each row of the original text matrix by the key matrix (in modulo 26, of course). This new vector can then be converted to letters and added to the encrypted text.

```
def encryptText(text, key, dim):
   encrypted_text = ""
   for row in stringToMatrix(text, dim):
      block = (np.matmul(row, stringToMatrix(key,dim)))%26
      for number in block:
         encrypted_text += chr(ord('a') + int(number))
   return encrypted_text
```

If we execute the code line

```
    encryptText("act","GYBNQKURP",3)
```

The text has only 3 letters, so it corresponds to a one-row matrix. This row is multiplied by the matrix calculated above:

$$\begin{pmatrix} 0 & 2 & 19 \end{pmatrix} \cdot \begin{pmatrix} 6 & 24 & 1 \\ 13 & 16 & 10 \\ 20 & 17 & 15 \end{pmatrix} = \begin{pmatrix} 406 & 355 & 305 \end{pmatrix} \equiv \begin{pmatrix} 16 & 17 & 19 \end{pmatrix}$$

Since in the alphabet the letters 16, 17 and 19 are q, r and t, respectively, the coded word ”act” would be ”qrt”. The *decryptText()* function receives three parameters:

- Text. the message to be decrypted
- Key. is the string used to decrypt the text by its matrix
- Dim. is the dimension of the matrix

To decrypt we will do the reverse of encrypting. That is to calculate the inverse of the key matrix and mutiply it by each row of the encrypted textx. This will result in a matrix of integers whose modulo 26 will correspond to the characters of the original message.

```
def decryptText(text, key, dim):
    decrypted_text = ""
    if dim==2:
        inverse_key_matrix = inverseMatrixDim2(stringToMatrix(key,2))
    elif dim == 3:
        inverse_key_matrix = inverseMatrixDim3(stringToMatrix(key,3))
    else:
        return "This dimension is not available yet"

    for row in stringToMatrix(text,dim):
        block = np.matmul(row, inverse_key_matrix)
        for number in block:
            decrypted_text += chr(ord('a') + int(number)%26)
    return decrypted_text
```

In our example, we would call the *decryptText()* function by passing the string ”qrt” and the key ”GYBNQKURP”.

```
    decryptText("qrt","GYBNQKURP",3)
```

We calculated above that the matrix corresponding to ”GYBNQKURP” was

$$\begin{pmatrix} 6 & 24 & 1 \\ 13 & 16 & 10 \\ 20 & 17 & 15 \end{pmatrix}$$

and its inverse was

$$\begin{pmatrix} 8 & 5 & 10 \\ 21 & 8 & 21 \\ 21 & 12 & 18 \end{pmatrix}$$

then, multiplying by the matrix of ”qrt”, we have

$$\begin{pmatrix} 16 & 17 & 19 \end{pmatrix} \cdot \begin{pmatrix} 8 & 5 & 10 \\ 21 & 8 & 21 \\ 21 & 12 & 18 \end{pmatrix} = \begin{pmatrix} 884 & 444 & 669 \end{pmatrix} \equiv \begin{pmatrix} 0 & 2 & 19 \end{pmatrix}$$

We receive the original matrix that corresponds to ”act”, as we wanted.

3 The menu

To get the message we want to encrypt or decrypt we have to read the *text.txt* file. This file can contain national characters, so to read it without problems, we use the *codecs open()* function. We save the message read from the file in the variable message and close the file reader. We also depurate the original text, to avoid mistakes.

```
    file = codecs.open('message.txt','r','utf-8')
    message = file.read()
    file.close()
    text = text.lower()
    text = unidecode(text)
    text = removeUnnecesaryChars(text)
```

To get the key we are going to use, we have to read *key.txt*. And also we calculate if this key is correct, supposing dimension 2. But we could change to 3 if we wanted to.

```
    file = codecs.open('key.txt','r','utf-8')
    key = file.read()
    file.close()

    key = key.lower()
    correct_key = isCorrectKey(key,2)
```

The menu will be the part of the code in charge of interacting with the user. The first thing we will do is ask the user to enter a number indicating whether they want to encrypt (1) or decrypt (0) a message. We will store the number entered in the encrypt_option variable.
Depending on the option chosen, the *encryptText()* or *decryptText()* function will be called and the result will be saved in *text_en.txt* or *text_de.txt*. In case the key is not valid, or the encryption option is not 0 or 1, a message will be launched indicating that no valid values have been entered.

```
encrypt_option = input('Press 1 for encrypting a message and press 0 for decrypting a
                                               message')
if encrypt_option == '1' and correct_key:
    file = open('text_en.txt', 'w')
    file.write(str(encryptText(text,key,2)))

elif encrypt_option == '0' and correct_key:
    file = open('text_de.txt', 'w')
    file.write(str(decryptText(text, key,2)))

else:
    print('The values introduced are not valid')

file.close()
```