

# DAT510 - Assignment 1

Frydis Jrgensen

September 16, 2020

## Abstract

A one-paragraph summary of the entire assignment -your procedure, results, and analysis.

## Part 1

### The plaintext message you managed to decipher

The plaintext message I got when I decrypted the cipher was:

*AN ORIGINAL MESSAGE IS KNOWN AS THE PLAINTEXT WHILE THE CODED MESSAGE IS CALLED THE CIPHERTEXT THE PROCESS OF CONVERTING FROM PLAINTEXT TO CIPHERTEXT IS KNOWN AS ENCIPHERING OR ENCRYPTION RESTORING THE PLAINTEXT FROM THE CIPHERTEXT IS DECIPHERING OR DECRYPTION THE MANY SCHEMES USED FOR ENCRYPTION CONSTITUTE THE AREA OF STUDY KNOWN AS CRYPTOGRAPHY SUCH A SCHEME IS KNOWN AS A CRYPTOGRAPHIC SYSTEM OR A CIPHER TECHNIQUE USED FOR DECIPHERING A MESSAGE WITHOUT ANY KNOWLEDGE OF THE ENCIPHERING DETAILS FALL INTO THE AREA OF CRYPTANALYSIS CRYPTANALYSIS IS WHAT THE LAY PERSON CALLS BREAKING THE CODE THE AREAS OF CRYPTOGRAPHY AND CRYPTANALYSIS TOGETHER ARE CALLED CRYPTOLOGY*

**Describe the strategy you employed, show the details for each of the steps of that strategy, describe any programs you wrote, show sample output of these programs, and show how you transformed that output into your solution.**

To solve the Poly-alphabetic Ciphers I started off by determine what kind of cipher algorithm that is used. The best-known polyalphabetic cipher is called The Vigenre cipher. It is mainly two steps to decrypt this cipher. We have to identify the length of the key, and then find the actual key.

To find the length of the key I used a test based on the taking the factors of the periods. I needed to find strings that repeated trough the ciphertext and chose to add strings of length 3 and longer. Therefor my outer for-loop starts at 3, as shown in Figure 1. The algorithm searches through the whole ciphertext and if a string with length longer than 3 occurs more than once, we add it to the list *wordsFrequently*. This part of the algorithm did I found

```

def findKeyLength(input):
    wordsFrequently = {}
    wordsIndex = []
    for i in range(3, int(len(input)/2)):
        for j in range(0, len(input)-i):
            sub = input[j:j+i]
            count = input.count(sub)
            if count >= 2 and sub not in wordsFrequently:
                wordsFrequently[sub] = count
                wordsIndex.append(findFactorialsOnWordIndex(input, sub))
    possibleKeyLengths = Counter(np.concatenate(wordsIndex))
    occurrences = 0
    key = 0
    dict_keys = list(possibleKeyLengths.keys())
    dict_values = list(possibleKeyLengths.values())
    for i in range(0, len(dict_values)):
        if dict_values[i] >= occurrences and dict_keys[i] > key:
            occurrences = dict_values[i]
            key = dict_keys[i]
    return int(key)

```

Figure 1: the function to find key length

here [REF to <https://cs.stackexchange.com/questions/79182/im-looking-for-an-algorithm-to-find-unknown-patterns-in-a-string>].

To get the index I used the function *findIndexOfDuplicates* that is highly inspired of one of the answers in this stackoverflow question. [REF <https://stackoverflow.com/question-of-duplicates-items-in-a-python-list>]. Figure 2 shows the function *findIndexOfDuplicates* where the indexes of the substrings is added to a list and returned.

This method is used in the function *findFactorialsOnWordIndex* as shown in Figure 3. I then used the distance between the indexes of the repeated words and factorize it. I get the factors when the modulo of the index distance and *i* is zero. We got information that the key would be no larger than 10, therefore I only collected the factors in the range of 2 to 10.

```

def findIndexofDuplicates(input, sub):
    start = -1
    indexes = []
    while True:
        try:
            index = input.index(sub, start + 1)
        except ValueError:
            break
        else:
            indexes.append(index)
            start = index
    return indexes

```

Figure 2: the function to find the indexes of duplicated strings

```

def findFactorialsOnWordIndex(input, index):
    factorials = []
    wordsIndex = findIndexofDuplicates(input, index)
    for i in range(0, len(wordsIndex)):
        if i > 0:
            indexDiff = wordsIndex[i] - wordsIndex[i-1]
        for i in range(1, indexDiff + 1, 1):
            if indexDiff % i == 0:
                if i <= 10 and i > 1:
                    factorials.append(i)
    return factorials

```

Figure 3: the function to find factorials

The next step in Figure 1 to find the key length is to count the occurrence of the factors. The method I used to find the key length is based on using the highest factor that most often occurs. In this case both 2, 4 and 8 occurred 105 times. The if-sentence at the end of Figure 1 sets the key to be the highest factor that most often occurs, witch in this case is 8.

Now that we know the length of the key we have to find the actual key. We know the period of the Vigenere cipher now is 8, witch means that we have 8 caesar ciphers to break. I chose to use Chi-squared statistic. Chi-squared statistic measures how similar two categorical probability distributions are. In this case I wanted to compare the frequency distribution of the ciphertext characters and the frequency distribution of english.

As can see in the first for-loop in Figure 4, I split my cipher in 8 strings, so I get a string witch consist of index [0,8,16,24 ...], [1,9,17, 25 ..], [2,10, 18, 26, ...] and so on. Then I count the occurrence of every letter in each of these 8 strings. To do that I use the function *countLetters* as shown in Figure 5. When I have the letter occurrences I send the letter occurrences to performe the chi-square statistic and compare it to the english letter frequency.

The chi-square statistic is based on the formula:

$$\sum_{i=Z}^{i=A} (Ci - Ei)^2 / Ei$$

Where i is going through all of the letters in the english alphabeth so that *Ca* is the count of letter the A, and *Ea* is the expected count of the letter A. As shown in Figure 6 we loop through every letter inside a loop of every letter and we do once for each of the 8 strings we have gotten previously. The chi-square statistic says that the letter with the lowest sum in each of the strings is a part of our key. In this example the *chiSquare* function returns an array of [1,3,11,0,4,10,2,24]. This array tells us at wich index the letters lays on. Since index starts at zero, The first letter will be B. The key is therefore, as we find in the last for-loop in Figure 4, when we convert the indexes back to letters **BDLAEKCY**.

With the knowledge of what the key is, the only thing that is lest is to decrypt the actual text.

```

def findKey(input):
    keyLength = findKeyLength(input)
    keyLengthSortedText = []
    letterOccurance = []
    for i in range(0, len(input)):
        if len(keyLengthSortedText) < keyLength:
            keyLengthSortedText.append(input[i])
        else:
            keyLengthSortedText[i % keyLength] += input[i]
    for i in range(0, keyLength):
        letterOccurance.append(countLetters(keyLengthSortedText[i]))
    indexes = chiSquare(letterOccurance, keyLength, len(input))
    key = []
    alphabet = list(englishFreq.keys())
    for i in range(len(indexes)):
        key.append(alphabet[indexes[i]])
    return key

```

Figure 4: the function to find the key

```

def countLetters(input):
    charFrequency = {'A': 0, 'B': 0, 'C': 0, 'D': 0, 'E': 0, 'F': 0,
                    'G': 0, 'H': 0, 'I': 0, 'J': 0, 'K': 0, 'L': 0, 'M': 0,
                    'N': 0, 'O': 0, 'P': 0, 'Q': 0, 'R': 0, 'S': 0, 'T': 0,
                    'U': 0, 'V': 0, 'W': 0, 'X': 0, 'Y': 0, 'Z': 0}
    for i in input:
        if i in charFrequency:
            charFrequency[i] += 1
        else:
            charFrequency[i] = 1
    return charFrequency

```

Figure 5: the function count the occurrence of letters

```

def chiSquare(letterOccurance, keyLength, inputLength):
    freqVal = list(englishFreq.values())
    sum = []
    for i in range(0, keyLength):
        temp = []
        values = list(letterOccurance[i].values())
        for j in range(26):
            result = 0
            for k in range(26):
                result += ((values[((j+k)%26)] -
                    (freqVal[k]*inputLength)**2)/(freqVal[k]*inputLength))
            temp.append(int(result))
        sum += [temp]
    indexSmallest = []
    for i in range(0, len(sum)):
        smallest = min(sum[i])
        indexSmallest.append(sum[i].index(smallest))
    return indexSmallest

```

Figure 6: the function to find the smallest index

## Part 2

The result of test cases in Tasks 1 and 2; The bits making up the keys of the SDES and TripleDES in Task 3; Describe the filtering strategy you used to know that the keys are correct.

## Conclusion

A short paragraph that restates the objective from your introduction and relates it to your results and discussion and describes any future improvements on your techniques that you would recommend.