Vigenère encrypting

Camille BARDON - Marion BIBES - Paul DALOUS

March 5, 2019

Contents

Introduction

Many encryption methods can be forcibly cracked. This technique consists of using all possible combinations until a satisfactory result is obtained. However, for long coded messages, it may require a huge calculation power and take a long time to complete. Therefore, we will present here an easier method to break the code of relatively short message, with a small unknown key.

For all of the following, we estimate that the text is long enough and the key is short enough to be find. Otherwise, all that is presented here will not work.

Our work was entirely inspired by Xavier Dupre's article about Vigenere cipher. You will find it here. Nevertheless, we added some features to make his method more reliable.

It is customary for a code to be clear and properly commented so that anyone can easily use it again. But sometimes you have to choose between clarity and efficiency. We therefore preferred to optimize our scripts to make them as fast as possible, to respond to future issues.

1 Vigenère cipher

The Vigenère cipher is a polyalphabetical substitution cipher. This indicates that the same letter in the plain text (the original text) can be encrypted by different letters. Actually, the Vigenère cipher is a succession of Caesar's code, with a different shift for each letter, which is determined by a key.

To encrypt a message, it is necessary to have the Vigenère table (cf fig. 1) and a key. The letters of the key will be taken one after the other to get the current line in the table, which corresponds with the shift of the Caesar's cipher.

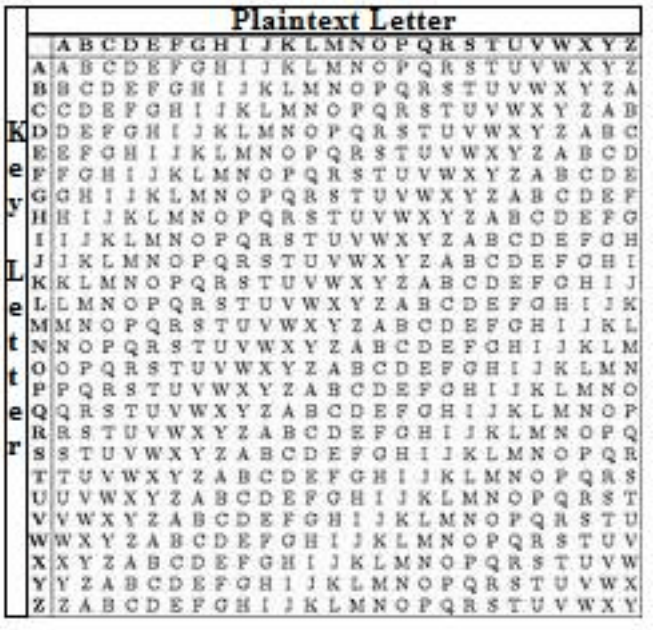

Figure 1: Vigenère table

The encrypted letter is obtained by crossing the corresponding column with the current line.

key : PANGRAM
plain text : THE QUICK BROWN FOX JUMPS OVER THE LAZY
repeated key :PAN GRAMP ANGRA MPA NGRAM PANG RAM PANG

Similarly, from an encrypted message and a key, it is possible to find the clear message.

Mathematically, if letters are associated with numbers (A = 0, B = 1 ... Z = 25), we have for the i^{th} letter of the message :

(% corresponding to *modulo*)

See the python script **Code_vigenere.py.** to see how we implemented this in python.

Plain text letter	Key letter	Encrypted letter
T	P	I
H	A	H
E	N	R
Q	G	W
U	R	L

Table 1: Example of encrypting

$$\left\| \begin{array}{l} \text{Cipher [i] = (Plain [i] + Key[i]) \% 26} \\ \text{Plain [i] = (Cipher [i] - Key [i]) \% 26} \end{array} \right\|$$

2 Decoding without key

Polyalphabetic encryption resists word frequency analysis, therefore another way to break the code has to be defined.

2.1 Determining the length of the key

In 1854, Babbage and Kasiski developped the pattern analysis method

- **Method description**

Charles Babbage was a polymath who originated the concept of a digital programmable computer. In 1854 Friedrich Kasisk - a german cryptographer - studying Babbage's work, discovered a method to find out the length of the key by recognizing patterns in the cipher text.
Although Vigenère's cipher flattens out the frequency of the letters in the cipher text by using a different shift for each letter, there is one main weakness to the security : the fact that the key is repeated.
For instance, if we use the keyword 'KEY', then the keystream will be KEYKEY... **This means that every third letter is encrypted using the same shift**. Essentially, we have 3 Caesar Ciphers, which can each individually be broken by frequency analysis. **The hard part is thus working out the length of the keyword.**

As an example, consider what we get when we encode the plaintext "maths is short for mathematics" using the keyword 'KEY'. We then get the ciphertext shown in the table below. The important things to notice are the two bits that are bolded. Due to the repeating nature of the key, both times we see "MATH", it is encrypted in the same way to "WERR".

M	**A**	**T**	**H**	S	I	S	S	H	O	R	T	F	O	R	**M**	**A**	**T**	**H**	E	M	A	T	I	C	S
W	**E**	**R**	**R**	W	G	C	W	F	Y	V	R	P	S	P	**W**	**E**	**R**	**R**	I	K	K	X	G	M	W

Since the repeating are 15 letters apart, **we know that the length of the key must be a factor of 15**. That is, the key must be one of the lengths 15,

5 or 3 (or 1 then it would be a simple Caesar Shift).

However it is possible that repeating strings of letters in the cipher text could be coincidence. If we look at the intercepted message below then we can see two sets of repeating strings **QUCE** and **VHVS**.

"**VHVS**SP**QUCE**MREGTDVB**VHVS**URQGIBDUGRNICJ**QUCE**CER"

As before, the gap between the "**VHVS**" pair is 18, suggesting a key length of 18, 9, 6, 3, 2 or 1. The gap between the "QUCE" pair is 30, which suggests a key length of 30, 15, 10, 6, 5, 3, 2 or 1. So looking at both together the most likely key length is 6 or possibly 3.
A generalisation of this example gives us that the length of the key is the GCD (Greatest common divisor) of the distances between all patterns. We choose the greatest common divisor, and not only a common divisor, for more security. Even if the key are lower than the GCD, if we take a key with the GCD as its length, the result will be a repetition of the key. This is suitable for the rest of the method. On the contrary, if the key is actually the GCD and if we take a lower common divisor, the key will not be in its entirety. This is why we always take the GCD of the distance between patterns as the length of the key. Then, the longer the message the more repeated n-patterns there are likely to be, and the more confident you can be about the length of the keyword.

- **Discussion about hypothesis and limits of this method**

The most important hypothesis we have made here is that **the length of the key must be (very) short compared to the length of the message**. Not making this hypothesis will lead us to deeper considerations that are out of the scope here, however we will keep this in mind for later.
Second, we also made the assumption that **the same consecutive letters are resulting from the same part of the key on other plain letters.** This will again, lead us to out of scope considerations here. To overcome this second problem the solution is to calculate several GCDs and to exclude those who doesn't fit. Then the likelier GCD is retained.

See the python script **Get_key_length.py.** to see how we implemented this method in python.

2.2 Decrypting the key

Once the length of the key has been determined, we want to determine the right letters. To do so we used a frequency analysis method, described below.

2.2.1 Frequencies analysis method

We can use the frequency method described in Xavier Dupré's article.

Since we know the length of the key, we know which letter of the key codes which letters of the message. By performing a frequency analysis on each position (all letters coded by the first letter of the key for example), we obtain the coded letter corresponding to the E, assuming that it is the most frequent letter. Indeed, in french, the E frequency is 15,87% (9,42% for the second letter, which is A). This allows us to determine the shift of Caesar's code and thus the corresponding letter.

However, to do so, the text must be long enough compared to the key length to have a workable frequency analysis.

If the frequencies aren't workable, we can retain several possible letters for each letter of the key, then apply the following method. Indeed even if the frequencies analysis doesn't give plainly the key it still permits a selection of the more likely letters.

See the python script **Frequence_analysis.py.** to see how we implemented this method in python.

2.2.2 Forcing the code

This method tests all the possible solution resulting from the frequencies analysis. In the first place, the key composed of all the first candidate letters resulting from the frequencies analysis is tested with the following process (We will call it key 0). If key 0 isn't conclusive, we test all the keys different from key 0 by one letter, by switching to the second candidate letter from the frequencies analysis method. Then we graduate, all the keys different from key 0 by two letters, and so on. We only work with the two first candidate letters.

3 Verifying the validity of a key

First of all, the words of the text are sorted by decreasing length. The longest words are fewer in the dictionary, therefore, faster to look for. Depending on the length of the key they are two options. The first one is if the key is shorter than the longest word : we assume that the longest word should be in the dictionary, so, we check for it. If not, then we suppose the key is not the good one. This method is very efficient because of the scarcity of long words. As a matter of fact, we can presume that the probability for too long words to differ just from one letter is really low. The second solution is for a longer key than the longest word : we find in the text all the big words and each of then must be found in the dictionary. We presume the text long enough to have enough words to cover all the letters of the key we are validating.

The main problem with this method is if a special word is in the text, for instance a proper noun, then if it's counted as a long word, it will never be found in the dictionary. So, maybe we could have give a certain freedom to the algorithm by accepted a certain number of wrong words, but those acceptable wrong words can also be due to a wrong letter on the key.

An other way to validate the key we first experienced was to find five words that are following each others on the first twenty words of the decoded text. If the words were findable in the dictionary we assumed they were correct. Indeed we made the hypothesis that the probability to have 5 existing words following but not being the actual words of the original text was really low.
To see this more in detail see the code bellow that assert if a message is 'reliable' :

```
class Verify_text():
    """ Verify if words of a text are in a dictionary with a certain level of confidence """

    def is_word_in_dictionnary(self, dictionary, word):
        """ Verify if a word is in a dictionary
            :param dictionary: pd.DataFrame - a data frame with all words of a certain length
            :param word:       string - a word to be checked
            :return:           bool - True if the word is in the dictionary """
        return word in dictionary[len(word)]

    def is_message_in_dictionnary(self, dictionary, message_token):
        """ Verify if text is 'reliable'. Reliable means that it contains a sufficient amount of words that are actually real words
            (in the dictionnary). We start with a confidence level of 0 and then increase it as we find words that are in the
            dictionary. We start with the longest words first. If the confidence level exceeds 10 we return true (false otherwise).
            :param dictionary:     pd.DataFrame - a data frame with all words of a certain length
            :param message_token:  string - a tokenized message to be checked
            :return:               bool - True if the confidence level exceeds 10 """

        message_token.sort(key=lambda word: len(word), reverse=True) # Sort the token by descending order of word length

        loop_counter = 0
        confidence_level = 0

        while (confidence_level < 10) and (loop_counter < 100): # loop_counter 100
            word = message_token[loop_counter]

            if self.is_word_in_dictionnary(dictionary, word):
                confidence_level += 1

            loop_counter += 1

        if confidence_level >= 10:
            return (True)
        else:
```

```
35                 return (False)
```

4 To go further

We said earlier that the text has to be long enough and the key short enough. What if it is not ? Actually, the Vernam cipher can be connect to the Vigenère cipher. This will result as a Vigenère cipher where the key is as long as the text and where the components are randomly chosen. The system offers a theoretical absolute security, as Claude Shannon proved in 1949.

Using the Vigenère cipher, we also can use a corpus of text as the key, indicating to our correspondent which book the key is in and where it begins. In the same way as described above, it is piratically impossible to decode, as the number of possibilities of key is finite but extremely huge.