Five Ways to Crack a Vigenère Cipher

brought to you by The Mad Doctor ("madness")

This is just a review of five nice ways to break a Vigenère cipher. It assumes that you are using a computer and can write simple code. The examples in this paper are in Python 3 (for Python 3, `/` and `//` behave differently, so be careful).

The Vigenère cipher

The *Vigenère cipher* is a periodic polyalphabetic substitution cipher. The key is a string of characters. To explain how the cipher works, let's first replace the characters of the key and the characters of the plaintext by integers, where A=0, B=1, ..., Z=25. The length of the key let's call the *period* or L. So the key is just a set of numbers k_0, k_1, ..., $k_{L\text{-}1}$. Next take the plaintext and express it also as a list of numbers: p_0, p_1, p_2, ... The text is encrypted by adding a number from the key modulo 26 to a number from the plaintext, where we run through the key over and over again as needed as we run through the plaintext. As an equation, the i^{th} character is encrypted like this:

$$c_i = (p_i + k_{i \bmod L}) \bmod 26$$

After that, the ciphertext is expressed as letters instead of numbers.

Here is a Python routine that encrypts a text:

```
ALPHABET = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'

def encrypt(plaintext,key):
    ciphertext = ''
    for i in range(len(plaintext)):
        p = ALPHABET.index(plaintext[i])
        k = ALPHABET.index(key[i%len(key)])
        c = (p + k) % 26
        ciphertext += ALPHABET[c]
    return ciphertext
```

Decryption is simply the inverse. In other words, instead of adding, we subtract. Here is some code:

```
def decrypt(ciphertext,key):
    plaintext = ''
    for i in range(len(ciphertext)):
        p = ALPHABET.index(ciphertext[i])
        k = ALPHABET.index(key[i%len(key)])
        c = (p - k) % 26
        plaintext += ALPHABET[c]
    return plaintext
```

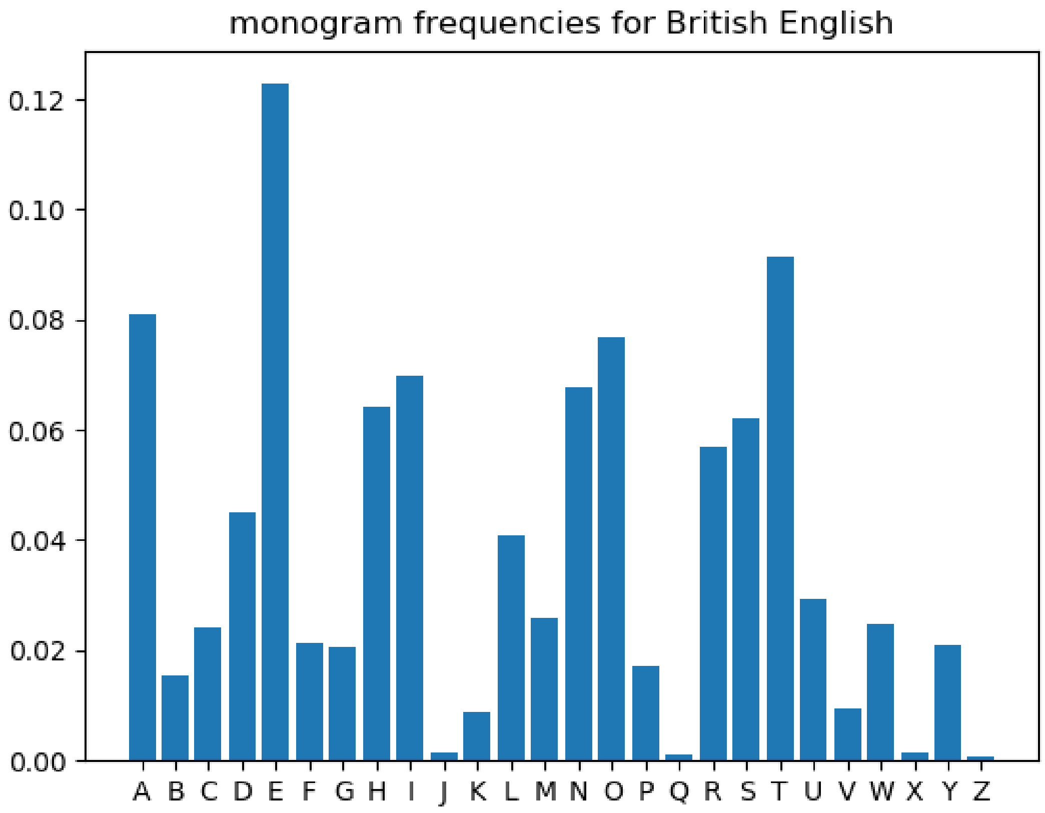
Frequency tables

For everything we do from this point forward, we are going to need to know the frequencies of letters as they are used in English. We are also going to need to know the frequencies of combinations of letters. It is up to you whether you want to use *digrams* (two-letter combinations) which are not so good, or *trigrams* (three-letter combinations) which are better, or *tetragrams* (four-letter combinations) which are just fine. In this paper I am going with tetragrams.

To build your frequency tables, start with a large piece of English text. I used twelve British novels (remember that American spelling is different) that I strung together. To build my *monogram* (single-letter) frequency table, I used code similar to this:

```
monofrequencies = [0]*26
for char in text:
    x = ALPHABET.index(char)
    monofrequencies[x] += 1
for i in range(26):
    monofrequencies[i] = monofrequencies[i] / len(text)
```

Here is a graph of the monogram frequencies:



To build the tetragram frequency table, the code was similar to this:

```
tetrafrequencies = [0]*26*26*26*26
for i in range(len(text) - 3):
    x = (ALPHABET.index(text[i])*26*26*26 +
         ALPHABET.index(text[i+1])*26*26 +
         ALPHABET.index(text[i+2])*26 +
         ALPHABET.index(text[i+3]))
    tetrafrequencies[x] += 1
 for i in range(26*26*26*26):
     tetrafrequencies[i] = tetrafrequencies[i] / (len(text)-3)
```

Fitness

Fitness is a way to quantify how closely a piece of text resembles English text. One way to do this is to compare the frequencies of tetragrams in the text with the frequency table that we built in the last section. It turns out that throwing in a logarithm helps, too. The basic idea is to start with zero and add the log of the value from our table for each tetragram that we find in the text that we are evaluating, then divide by the number of tetragrams to get an average. The average is more useful than the total because it allows our programs to make decisions independent of the length of the text. Defined in this way, the fitness of English texts is typically around -9.6.

In equation form the fitness is

$$F(t) = (1/N)\ \Sigma_{\text{tetragrams}} \log f_{\text{English}}(\text{text}[i,...,i+3])$$

Some Python code:

```
from math import log
def fitness(text):
    result = 0
    for i in range(len(text)-3):
        tetragram = text[i:i+4]
        x = (ALPHABET.index(tetragram[0])*26*26*26 +
             ALPHABET.index(tetragram[1])*26*26 +
             ALPHABET.index(tetragram[2])*26 +
             ALPHABET.index(tetragram[3]))
        y = tetrafrequencies[x]
        if y == 0:
            result += -15 # some large negative number
        else:
            result += log(y)
    result = result / (len(text) - 3)
    return result
```

Finding the period (keylength): index of coincidence

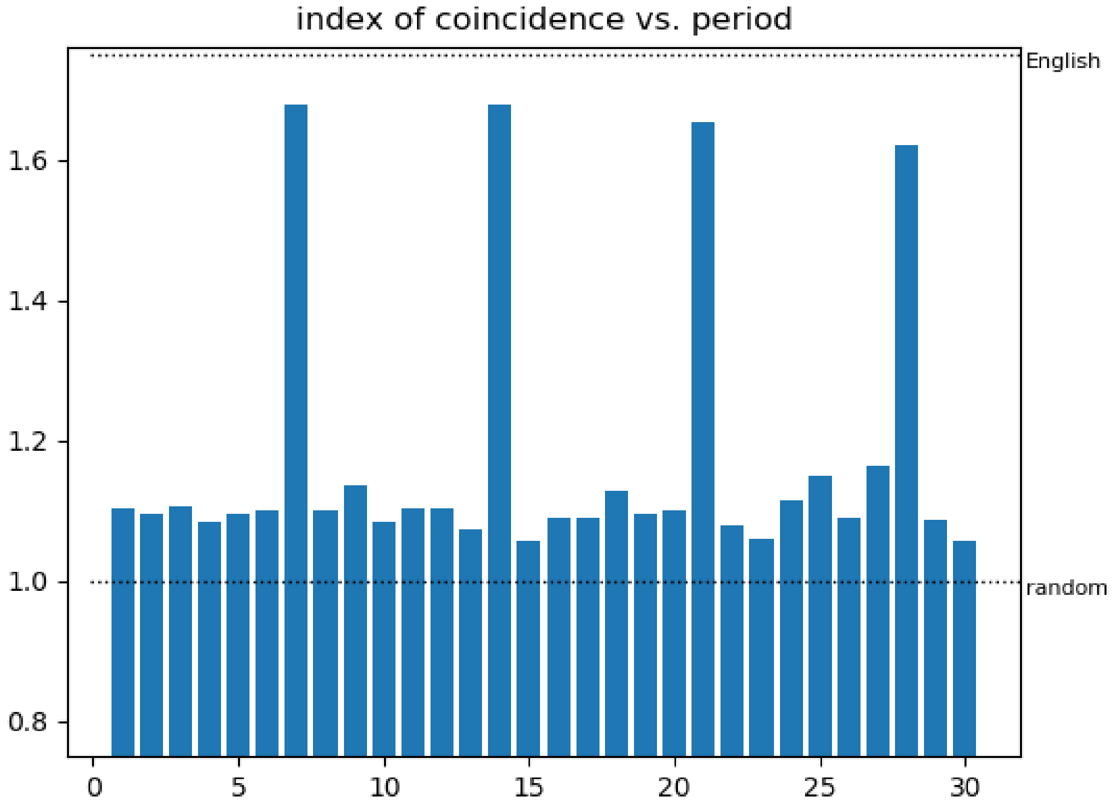
The Kasisky method for finding the period involves finding groups of characters that reoccur in the ciphertext. The distances between repeating groups are multiples of the period. This is fine and good, but we have a more modern way to find the period: the index of coincidence.

The *index of coincidence* (IoC) measures the likelihood that any two characters of a text are the same. A concise formula for the IoC is

$$I = 26\ \Sigma_{i=\mathrm{A}}^{\mathrm{Z}}\ n_i\ (n_i - 1)\ /\ N\ (N - 1)$$

where n_i are the counts of the letters in the text, and N is the total number of characters. Notice that I use a normalization factor of 26 which does not appear in Friedman's original definition. With this normalization, a random text has an IoC close to 1, while English text is close to 1.7.

To use the IoC to find the period of the cipher, we cut the ciphertext into *m* slices, where each slice contains every *m*th letter. Then we find the IoC for each slice and average them. We do this for various choices of *m*. The smallest *m* with an average IoC close to 1.7 is our period. For example, here we see the IoC versus the number of slices *m*. The period for this example is 7.



To put it all together, here is some sample Python code that finds the period:

```
def index_of_coincidence(text):
    counts = [0]*26
    for char in text:
        counts[ALPHABET.index(char)] += 1
    numer = 0
    total = 0
    for i in range(26):
        numer += counts[i]*(counts[i]-1)
        total += counts[i]
    return 26*numer / (total*(total-1))

found = False
period = 0
while not found:
    period += 1
```

```
        slices = ['']*period
        for i in range(len(ciphertext)):
            slices[i%period] += ciphertext[i]
        sum = 0
        for i in range(period):
            sum += index_of_coincidence(slices[i])
        ioc = sum / period
        if ioc > 1.6:
            found = True
```

Now we have enough tools to start attacking the Vigenère cipher.

Method #1: Brute force

The *brute-force* method, also called *exhaustive search*, simply tries every possible key until the right one is found. If we know the period from the IoC test above, then we can at least reduce the options to keys with the correct length. Otherwise, we start at length 1 (same as a Caesar cipher), and work our way up. We know when to stop when the fitness of the decryption is close to the fitness of typical English text.

Here is a snippet of code that assumes that we have already found the period and that it is 3. Yes, the "else" statements are in the right places. You will have to be clever to write your own code that works for any period, or which runs through all periods until it finds the key.

```
    key = ['']*3
    for key[0] in ALPHABET:
        for key[1] in ALPHABET:
            for key[2] in ALPHABET:
                pt = decrypt(ciphertext,key)
                fit = fitness(pt)
                if fit > -10:
                    break
            else:
                continue
            break
        else:
            continue
        break
    plaintext = decrypt(ciphertext,key)
```

The brute-force method is only useful if the key is short. For keys longer than 4 characters, it takes far too long to execute.

Method #2: Dictionary attack

If we are confident that the key is an English word, then we can do a dictionary attack. A *dictionary attack* tries every word from a list until it finds one that produces an acceptable fitness for the decrypted text. Knowing the period can speed up the process by allowing us to only try words with the correct length.

Coding this is very simple:

```
words = open('words.txt').read().upper().split('\n')
for key in words:
    pt = decrypt(ciphertext,key)
    fit = fitness(pt)
    if fit > -10:
        break
plaintext = decrypt(ciphertext,key)
```

The usefulness of this method depends on the quality of the word list. If the list is long, then the program can take too much time. If the list is too short, then the key might not be in it.

Method #3: Using a crib

If we know a bit of the plaintext, we can use it as a *crib* to try to find the key. Basically, we run from the start of the ciphertext and subtract the crib from it at different positions until we get a result that looks like a key. Here's a piece of code that prints all of the results of the subtractions. It is up to you to look at the results and pick out the one with the key in it.

```
for i in range(len(ciphertext)-len(crib)):
    piece = ciphertext[i:i+len(crib)]
    decryptedpiece = decrypt(piece,crib)
    print (decryptedpiece)
```

Method #4: Variational method

Here is a variational method that works even for shorter ciphertexts. For each letter of the key, we vary it and choose the option that gives the best fitness of the decrypted text. It loops over all the letters of the key until the fitness can no longer be improved. Here is some code to illustrate the method:

```
from random import randrange
key = ['A']*period
fit = -99 # some large negative number
while fit < -10:
    K = key[:]
    x = randrange(period)
    for i in range(26):
        K[x] = ALPHABET[i]
        pt = decrypt(ciphertext,K)
        F = fitness(pt)
        if (F > fit):
            key = K[:]
            fit = F
plaintext = decrypt(ciphertext,key)
```

You will find that this method is very fast, and works for very short ciphertexts (sometimes as short as 5 times the period).

Chi-squared statistic

For the last method, we need a way to decide if two monogram (single-letter) frequency tables are similar. One tool to do that is the *chi-squared statistic*, which measures the difference between two ordered sets of numbers. It is defined as

$$\chi^2 = \Sigma\ (n_i - N_i)^2 / N_i$$

where the n_i are the numbers in the set that we are testing, and the N_i are numbers from a standard set that we want to match. For our case, the N_i are the entries in the frequency table that we made for English earlier. The n_i are the frequencies that we want to compare to English. A small value for the chi-squared statistic indicates a good fit.

Inner product

While the chi-squared statistic is a fine measure to use, I prefer the inner (dot) product. The *inner product* of two ordered lists of numbers (*vectors*) is simply the sum of the products of the numbers:

$$\mathbf{x}\cdot\mathbf{y} = \Sigma\ x_i\, y_i$$

If you have studied vector spaces, you would know that the cosine of the angle between two vectors is

$$\cos\theta = \mathbf{x}\cdot\mathbf{y} / \sqrt{\mathbf{x}\cdot\mathbf{x}}\ \sqrt{\mathbf{y}\cdot\mathbf{y}}$$

This number can provide a good way to determine if a monogram frequency table is close to the English table. A value around or above 0.9 is a good match. Here is some code:

```
from math import sqrt
def cosangle(x,y):
    numerator = 0
    lengthx2 = 0
    lengthy2 = 0
    for i in range(len(x)):
        numerator += x[i]*y[i]
        lengthx2 += x[i]*x[i]
        lengthy2 += y[i]*y[i]
    return numerator / sqrt(lengthx2*lengthy2)
```

Method #5: Statistics-only attack

This method is useful for longer ciphertexts. It uses only statistics to crack the Vigenère cipher. First, find the period with the index of coincidence as shown above. Then, construct frequency tables for each of the slices of the ciphertext that were created when the period was found. Those tables are each rotated until they match well to the monogram frequencies of English. To determine the matching, the cosine of the angle (see the last section above) is used.

This code constructs the frequency tables for the slices and then finds the shifts that give good matches (cosine > 0.9) to English. Those shifts determine the key.

```
frequencies = []
for i in range(period):
    frequencies.append([0]*26)
    for j in range(len(slices[i])):
        frequencies[i][ALPHABET.index(slices[i][j])] += 1
    for j in range(26):
        frequencies[i][j] = frequencies[i][j] / len(slices[i])

key = ['A']*period
for i in range(period):
    for j in range(26):
        testtable = frequencies[i][j:]+frequencies[i][:j]
        if cosangle(monofrequencies,testtable) > 0.9:
            key[i] = ALPHABET[j]

plaintext = decrypt(ciphertext,key)
```

If you implement this attack, you will find that it is lightning-fast. However, it is only reliable for ciphertexts that are at least 100 times as long as the period.