Group Work - SymEnA

Frequency Analysis Decryption

Group and Activity description

Group Members:
- Michele Bortolato, 885031
- Ziyang Fu, 884958
- Mattia Smania, 884969

We approached this project mostly working in parallel on a specific part of the code, that was then reviewed by everybody, comparing the performance and quality of everybody's work, this allowed us to agree on the best solution which will be the basis for everybody to start working on the next section.

Introduction to the Project

- Goals of the project: Creation a code able to decrypt a ciphertext using frequency analysis

- Why it is important: To understand how information can be extracted from an apparently incomprehensible message and the importance of a good encryption to ensure secrecy.

- What we did in a nutshell: We wrote a python code that automatize frequency analysis and tries to decrypt an encrypted message starting from the ciphertext and an unencrypted text used as a sample of the english language

The group work

0) Files loading and text cleaning

First of all we start by loading the subsequent files:
- Ciphertext, the text we want to decrypt;
- Corpus, a text large enough to be representative of the English language.

Then in order to have uniform texts and to facilitate the following tasks, we cleaned both texts from: numbers, punctuations and special characters.
Here's the function we used for doing so:

```
def text_cleaning (txt):
    cleaned_text = re.sub('[^a-zA-Z]+', ' ', txt)
    return cleaned_text
```

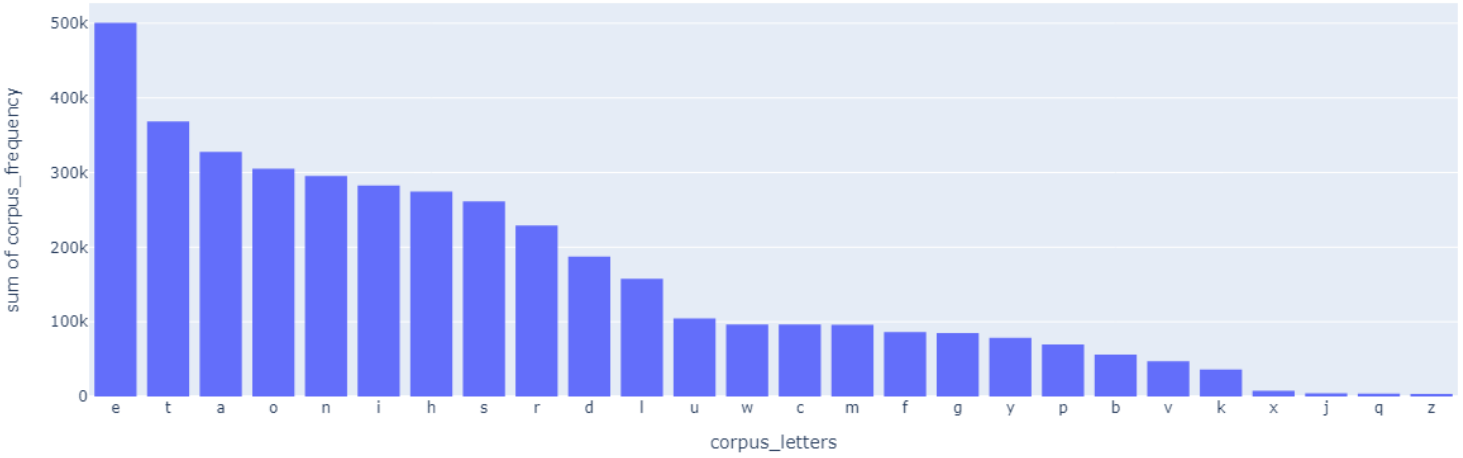
1) Single letters frequency analysis & manual decryption

We start the by counting how many times each letter appears both in the ciphertext and in the corpus and by creating a dictionary that maps as key the letter and as its value its frequency in the text

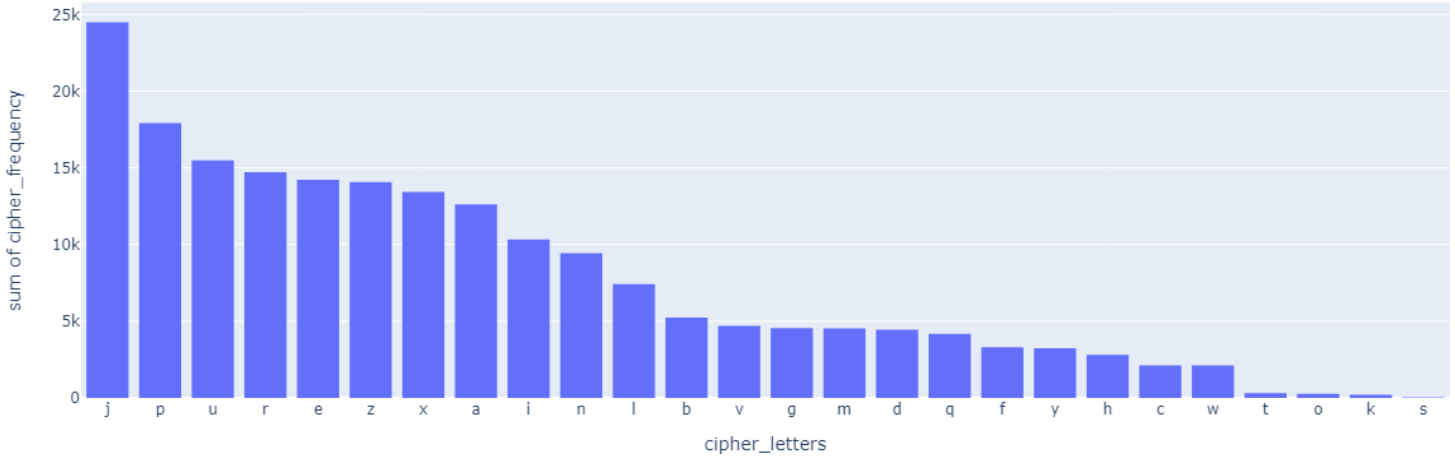
```
def getLetterCount(message):

    letterCount = {'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 letter in message:
        if letter in alphabet:
            letterCount[letter] += 1
    return letterCount
```

1.1) Plot the distribution and see which are the most frequent letters

English language letter distribution (from corpus):



Ciphertext letter distribution:



1.2) Compare the different distribution between the corpus and the ciphertext

	corpus_letters	corpus_frequency	cipher_letters	cipher_frequency
0	e	500633	j	24485
1	t	368470	p	17920
2	a	327849	u	15485
3	o	305299	r	14719
4	n	295527	e	14219
5	i	282496	z	14081
6	h	274387	x	13421
7	s	261541	a	12622
8	r	229242	i	10325
9	d	187310	n	9440
10	l	157631	l	7410
11	u	104543	b	5244
12	w	96528	v	4693
13	c	96354	g	4555
14	m	96170	m	4521
15	f	86385	d	4448
16	g	84993	q	4177
17	y	78626	f	3299
18	p	69704	y	3246
19	b	56170	h	2802
20	v	47157	c	2111
21	k	35977	w	2110
22	x	7812	t	306
23	j	4085	o	262
24	q	3752	k	199
25	z	3432	s	42

In this case the key is: ‘jpurezxainlbvgmdqfyhcwtoks’

1.3) Frequency decryption key and applying it on the ciphertext

As result of the frequency analysis we obtain our first candidate decryption key, by mapping each letter in the ciphertext with its corresponding decryption letter, the one that shows a similar frequency through the English corpus.

The output plaintext is the following:

```
'yrhsoiers tnrougn tne yreshdeit, to use yayer aid yeichls, aid to examhie tne arthcles yut hi as ekhdeice. tnehr dutp was to judge iot falselp, but justlp [...]'
```

It's far from perfect but good enough to understand several words and make manual adjustments until it is completely decrypted.

The function we used for applying the decryption key on the ciphertext is the following:

```
def key_application (msg, key):
    alphabet = 'abcdefghijklmnopqrstuvwxyz'
    msg = msg.translate(str.maketrans(key, alphabet))
    msg = msg.replace('\n'," ")
    return msg
```

1.4) Manual substitutions

Based on the previous resulting text we identified the misplaced letters and manually replaced them with the correct ones.

```
manual_attempt = key_application(ciphertext, starting_key)
manual_attempt = manual_attempt.replace('y', 'P')
manual_attempt = manual_attempt.replace('p', 'Y')
manual_attempt = manual_attempt.replace('h', 'I')
manual_attempt = manual_attempt.replace('i', 'N')
manual_attempt = manual_attempt.replace('n', 'H')
manual_attempt = manual_attempt.replace('k', 'V')
manual_attempt = manual_attempt.replace('v', 'K')
```

```
'PRISONERS THROUGH THE PRESIDENT, TO USE PAPER AND PENCILS, AND TO EXAMINE THE ARTICLES PUT IN AS EVIDENCE. [...]'
```

Since the replace function is case sensitive, in order to avoid the lines of code replacing the already replaced letters, we decided to write substituted letters in uppercase.

2) Bigrams frequency analysis & automatic decryption

To automate the previous letters replacing process we worked on bigrams (couples of adjacent letters) and on their frequency.

The working flow is quite simple: starting from the frequency key, we generate neighbouring keys of it and we compute a score (based on the frequency of the bigrams in plaintext resulting from applying these keys on the original ciphertext and the frequency of bigrams in the chosen English corpus) and the one with the highest score at the end of the this process will be our final candidate decryption key.

2.1) Bigrams

We iterate through both the ciphertext and the corpus to create dictionaries similar to those of the letters frequency, but this time we do it for the bigrams (set of 2 adjacent letters).

We used the following functions:

- for obtaining all the bigrams of a given word:

```
def letterNGrams(msg, n):
   return [msg[i:i+n] for i in range(len(msg) - (n-1))]
```

- for iterating through a list of words and creating a dictionary containing all the bigrams of the list and their frequency:

```
def score_text_bigram(txt):
    #list of words
    txt_words = re.findall(r"[\w']+", txt)
    #bigrams dictionary
    bigrams = {}
    for i in txt_words:
        for element in letterNGrams(i,2):
            if element not in bigrams:
                bigrams[element] = 1
            else:
                bigrams[element]+= 1

    sorted_bigrams = dict(sorted(bigrams.items(),
key=operator.itemgetter(1),reverse=True))
    return sorted_bigrams
```

2.2) Scoring systems

We needed to implement a scoring system to evaluate how good the decryption is, based on the bigram frequency.

In particular we tried using 2 different score computations in order to choose the one that gives us the best results:

- **Chi-Squared:**

$$\chi^2 = \sum \frac{(obs.freq - exp.freq)^2}{exp.freq}$$

Unfortunately our implementation took too long to compute and slowed the code.

- **Logarithm method**

$$S = \sum(obs.freq * log(exp.freq))$$

We found it to be quite effective and our implementation of it has been time efficient so we ended up using this score
https://towardsdatascience.com/applications-of-mcmc-for-cryptography-and-optimization-1f99222b7132

Starting from the key generated from the letters frequency analysis we generate neighbouring keys, permutations, up to 2 letter changes per permutation, of the it.

- Function for generating neighbouring keys:

```
def permute(s):
    for i,j in combinations(range(len(s)), 2):
        res = list(s)
        res[i], res[j] = res[j], res[i]
        yield ''.join(res)
```

- Applying each neighbouring to the ciphertext, and computing the logarithmic score on the resulting ‘plaintext’:

```
for i in range(attempts):
        print("iter", i)
        p=list(permute(best_key))

        best_candidate=candidate_plaintext

        for k in p:

            candidate_plaintext = key_application(ciphertext, k)
            score= get_key_score(candidate_plaintext,scoring_parameter)
```

If the score improves, the swap has led to a better key so we keep it as best_key and perform on it the same permutation process we applied to the starting frequency key.

```
            if score>max_score:
                max_score=score
                best_key=k
                best_candidate=candidate_plaintext
```

After a given number of iterations we stop the process and print the result: the best candidate key and the best candidate plaintext.

We managed to get really close to the perfect decryption.
The only letters that resulted swapped with each other were the J with the Q and the Z with Y. Apart from these 4 letters the text was perfectly readable.

The reason for this swap was our scoring system, that being purely statistical, attributed to the plaintext we got the same exact score that a perfect plaintext would have gotten.
The same output was achieved also on the trial text uploaded on moodle.

Conclusions

Frequency analysis is a useful tool to decrypt a ciphertext but we think it's a bit too "statistic-based".
It won't work effectively on shorter text, where other methods are to be preferred.
And even on longer texts it's very dependent on which scoring system you use and the trust in how "common" the n-gram frequency of the plaintext is.

Bibliography

- **https://jeremykun.com/tag/ngrams/**
- **https://towardsdatascience.com/applications-of-mcmc-for-cryptography-and-optimization-1f99222b7132**

Appendices

- Code fragments: provided in the project description
- Configuration files:
- Whatever else is needed to make your experiments reproducible:
    - Ciphertext
    - Corpus