Vigenere Cipher Cracking and Simplified Data Encryption Standard implementation

Abstract

Redact this part later

1 Vigenere Cipher Cracking

We are given the following cipher :

```
BQZRMQ KLBOXE WCCEFL DKRYYL BVEHIZ NYJQEE BDYFJO PTLOEM EHOMIC
UYHHTS GKNJFG EHIMK NIHCTI HVRIHA RSMGQT RQCSXX CSWTNK PTMNSW
AMXVCY WEOGSR FFUEEB DKQLQZ WRKUCO FTPLOT GOJZRI XEPZSE ISXTCT
WZRMXI RIHALE SPRFAE FVYORI HNITRG PUHITM CFCDLA HIBKLH RCDIMT
WQWTOR DJCNDY YWMJCN HDUWOF DPUPNG BANULZ NGYPQU LEUXOV FFDCEE
YHQUXO YOXQUO DDCVIR RPJCAT RAQVFS AWMJCN HTSOXQ UODDAG BANURR
REZJGD VJSXOO MSDNIT RGPUHN HRSSSF VFSINH MSGPCM ZJCSLY GEWGQT
DREASV FPXEAR IMLPZW EHQGMG WSEIXE GQKPRM XIBFWL IPCHYM OTNXYV
FFDCEE YHASBA TEXCJZ VTSGBA NUDYAP IUGTLD WLKVRI HWACZG PTRYCE
VNQCUP AOSPEU KPCSNG RIHLRI KUMGFC YTDQES DAHCKP BDUJPX KPYMBD
IWDQEF WSEVKT CDDWLI NEPZSE OPYIW
```

We know that this english message has been encrypted by a polyalphabetic substitution cipher, and that the encryption key is not longer than 10 characters. To break this cipher, one approach could be to try to guess the key length. To do so, we are looking for repeating sequences, which have a high probability of corresponding to a repetition of the key. This method is called kasiski examination.[1]

To perform this examination, I implemented a Python method which loops through the whole cipher, and searches for patterns which appear again in the rest of the cipher. If an occurence of the current pattern is found, the distance between the two patterns is added to the possible keys list. Note that all divisors of the distance are also added to the list, as the key could have repeated multiple times between the two occurences of the pattern. The get_divisors() subroutine does not return divisors which are greater than the maximum key length.

```python
def kasiski_examination(cipher):
    possible_key_lengths = []

    for pattern_length in range(MIN_PATTERN_LENGTH, MAX_PATTERN_LENGTH):
        for index in range(len(cipher)):
            substring = cipher[index:index+pattern_length]
            distance = cipher[index+pattern_length:].find(substring)

            if distance != -1:
                possible_key_lengths.extend(get_divisors(distance + pattern_length))
    probable_key_lengths = {length:possible_key_lengths.count(length) for length in
                                                 possible_key_lengths}

    return sorted(probable_key_lengths.items(), key=lambda x:x[1], reverse=True)
```

When executed on the given ciphertext, we get the following output.

```
key length : 2 -> 115 occurences
key length : 4 -> 115 occurences
key length : 8 -> 115 occurences
key length : 3 -> 57 occurences
key length : 6 -> 54 occurences
key length : 7 -> 31 occurences
key length : 9 -> 19 occurences
key length : 5 -> 2 occurences
key length : 10 -> 2 occurences
```

Which implies that the key is probably of length 2, 4 or 8. Since 2 and 4 are divisors of 8, it is safe to assume that they appear so often here because the get_divisors method returns all divisors smaller than the maximum key length. Our guess is therefore that the key is of length 8. Next, we perform a frequency analysis for each set of character with an equal position in the ciphertext, modulo the key length. These sets will be called columns from now on, because they each represent a column of the ciphertext when it is splitted every key length characters.

This frequency analysis is implemented in the following method.

```
def frequency_analysis(cipher, key_length):
    candidates = []

    for column in range(key_length):
        frequencies = char_frequency(message[column::key_length])
        column_candidates = []

        for most_common in frequencies[:5]:
            E_offset = (LETTERS.find(most_common[0]) - LETTERS.find(’E’)) % 26
            column_candidates.append(LETTERS[E_offset])
        print("key[{}] candidates : {}".format(column, column_candidates))
        candidates.append(column_candidates)

    return candidates
```

This method computes character frequency for each column, and returns the key character which would have encrypted this column if the most common character corresponds to the letter E.

```
key[0] candidate : [’B’]
key[1] candidate : [’D’]
key[2] candidate : [’A’]
key[3] candidate : [’A’]
key[4] candidate : [’E’]
key[5] candidate : [’T’]
key[6] candidate : [’C’]
key[7] candidate : [’Y’]
```

This gives us the following possible key : ”BDAAETCY”. We then try to decrypt the ciphertext with this key using the following method.

```
def poly_decrypt(cipher, key):

    # make sure key and cipher are in uppercase and without whitespace
    cipher = cipher.upper().replace(' ', '')
    key = key.upper().replace(' ', '')

    # expand the key so that it matches the length of the cipher
    expanded_key = ''.join(key[i \% len(key)] for i in range(len(cipher)))

    decrypted_message = ''

    for cipher_letter, key_letter in zip(cipher, expanded_key):
        decrypted_index = (LETTERS.find(cipher_letter) - LETTERS.find(key_letter)) \% 26
        decrypted_message += LETTERS[decrypted_index]

    return decrypted_message
```

When using this method on the given ciphertext with the key ”BDAAETCY”, we can do a first decryption of the message. If we format the output by columns, we get the following output

```
01234567
--------
ANZRIXIN
ALXESJAG
EIDKNFWN
ASEHEGLA
INEEXKWH
...
CRJPTRNA
LYDISKOG
ETSERRRE
CAWLEUCR
YPEOLFGY
```

This is not the original message, but we can clearly identify possible words in this text.

2 Simplified DES Implementation

3 Conclusion

References

[1] Wikipedia contributors. Vigenre cipher — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Vigen%C3%A8re_cipher, 2019. [Online].