Cryptanalysis of substitution ciphers

30.10.2021

GROUP MEMBERS:

PULIVENDULA REDDY GOWTHAM(AM.EN.U4CSE19143)

CHANDRASEKHAR MADAN(AM.EN.U4CSE19167)

LOHITH KUMAR REDDY (AM.EN.U4CSE19131)

DUTTA SAI TARUN(AM.EN.U4CSE19119)

Cryptanalysis of Substitution ciphers:

- A substitution cipher is a method of encrypting units of plaintext to be replaced with cipher text following an encryption scheme where the units may be single letters, pairs of letters, triplets of letters or a combination of all.
- The cipher text is decrypted by performing the inverse substitution. The number of keys possible with the substitution cipher is higher, around 2^88 possible keys.
- Substitution ciphers can be deciphered by exhaustively searching through key space for the key that produces the decrypted text most closely resembling meaningful word, but an efficient algorithm would be to exploit patterns and redundancy in the patterns to significantly narrow their search.
- The substitution cipher which operates on single letters, it is termed a simple substitution cipher and that operates on larger groups of letters is termed poly-graphic.
- A **monoalphabetic cipher** uses fixed substitution over the entire message, whereas a polyalphabetic cipher uses a number of substitutions at different positions in the message
- This encryption type varies with what kind of substitution is considered. We have Simple Substitution: Entire plaintext is rotated by a fixed key.
- **Homophonic Substitution**: Single plaintext letters are replaced by different cipher text letters. They are generally much more difficult to break than standard substitution ciphers.
- **Polygraphic substitution**: Pairs of letters are substituted is bigraphic.
- **One-Time Pad**: Plaintext is paired with a random secret key and each character of the plaintext is encrypted by combining it with the corresponding bit or character from the pad using modular addition.

Algorithm - Decrypt an L-symbol cipher text

INPUT:

1. 100 characters cipher text (c) which includes {<space>, a, b ... z} only.
2. No of keys symbols (t) used to encrypt the given cipher text, maximum up to 18 keys preferred.
3. Dictionary 1 - Plaintexts obtained as a sequence of space-separated English words.
4. Dictionary 2 – set of English words

OUTPUT:

The most probable plaintext from Dictionary 1 or Dictionary 2.

ALGORITHM:

For Dictionary 1:

1. Import the Dictionary 1 in memory.
2. Read the next 100 character sentence from Dictionary 1, say p
3. Perform an operation (Cipher Text (c) – Plaintext (p)) to get a string of 100 numbers.
4. Calculate the number of Unique Numbers in this string, say n.
5. If n == t,

   (Yes)

   If (The unique symbols are repeated as much as t)

   (If t=5, L=100, there must be 5 unique symbols repeating 20 times) (Yes)

   This is the most likely required plaintext, we print it and terminate the program.

6. If n != t, continue from step 2
7. If no matches are found in Dictionary 1, we check Dictionary 2.

For Dictionary 2:

We consider a Cartesian product of all words in the dictionary as multiple possible plaintext (permutations)

1. Import Dictionary 2 in memory.
2. Read next word from the Dictionary 2.
3. Append the word to a string, say s
4. Perform the operation (Cipher Text – Plain text).
5. Calculate the number of Unique Numbers in this string, say n If n <= t continue steps 1 through 4.

If n > t, discard this and go to next permutation.

6. When the string append is of 100 characters If n == t,

(Yes)

If (The unique characters are repeated as much as t)

(If t=5, L=100, there must be 5 unique symbols repeating 20 times) (Yes)

This is the most likely required plaintext, we print it and terminate the program.

If n != t, discard this and go to next permutation

Backtrack if the current permutation no longer is the probable candidate for plaintext.

CODE:

```
from collections import Counter
from collections import defaultdict
import time

characterMap = {' ': 0, 'a': 1, 'c': 3, 'b': 2, 'e': 5, 'd': 4, 'g': 7,
'f': 6, 'i': 9, 'h': 8, 'k': 11, 'j': 10, 'm': 13, 'l': 12, 'o': 15, 'n':
14, 'q': 17, 'p': 16, 's': 19, 'r': 18, 'u': 21, 't': 20, 'w': 23, 'v':
22, 'y': 25, 'x': 24, 'z': 26}

sentenceMap = defaultdict()

file1 = open('dictionary1.txt', 'rb')
file2 = open('Dictionary2-1.txt', 'rb')

plaintext = []

for line in file2:
    plaintext.append(line.rstrip('\r\n'))

key = input("Enter your the value of keylength")

remainder = 100 % key
quotient = 100 / key
repeatElement = key - remainder

cipherText = raw_input("Enter the cipher text :").strip().lower()
cipherMap = map(lambda x: characterMap[x], cipherText)

print (cipherMap)

for i in range(len(plaintext)):
    # creating mapping for dictionary 2
```

```
    sentenceMap[plaintext[i]] = plaintext[0:i] + plaintext[i+1:]

def dictionary1():
    '''
    Find the plaintext from dictionary 1, by checking the number of unique
shifts
    '''
    start_time = time.time()
    for line in file1:
        line = 'disported patient breveting strikingly filliping catwalk
anticline towery unfits ebullience lurers d'
        plainTextMap = map(lambda x: characterMap[x], line)
        frequencyMap = [a_i - b_i if a_i - b_i > 0 else a_i - b_i + 27 for
a_i, b_i in zip(cipherMap, plainTextMap)]
        print (frequencyMap)
        print (plainTextMap)
        print (Counter(frequencyMap).keys())
        break
        if len(Counter(frequencyMap).keys()) == key:
            print ("Plaintext is :")
            print (line)
            print("--- Time required to find plaintext in seconds %s ---"
% (time.time() - start_time))
            return True
    return False

def dictionary2(graph, start, path=[]):
        '''
       First generate a sentence using backtracking, and then check
whether it is our possible plaintext candidate
        '''
        path = path + [start]
        if len(" ".join(path)) >= 100:
            return " ".join(path)[0:100]
        plainTextMap = map(lambda x: characterMap[x], " ".join(path))
```

```
        frequencyMap = [a_i - b_i if a_i - b_i > 0 else a_i - b_i + 27 for
a_i, b_i in zip(cipherMap, plainTextMap)]
        if len(Counter(frequencyMap).keys()) <= key:
             for node in graph[start]:
                 if node not in path:
                     result = dictionary2(graph, node, path)
                     if result:
                         plainTextMap = map(lambda x: characterMap[x],
result)
                         frequencyMap = [a_i - b_i if a_i - b_i > 0 else
a_i - b_i + 27 for a_i, b_i in zip(cipherMap, plainTextMap)]
                         if len(Counter(frequencyMap).keys()) == key and
Counter(frequencyMap).values().count(quotient) == repeatElement:
                             print ("Plaintext is :")
                             print (result)
                             print("--- Time required to find plaintext in
seconds %s ---" % (time.time() - start_time))
        else:
             return()

if dictionary1():
    # if plaintext is found in dictionary 1 then exit here
    exit()
else:
     for i in plaintext:
         # Create a sentence of 100 characters and test whether it will
match with ciphertext
         start_time = time.time()
         dictionary2(sentenceMap, i)
```

ENCRYPTION CODE:

```
characterMap = {' ': 0, 'a': 1, 'c': 3, 'b': 2, 'e': 5, 'd': 4, 'g': 7,
'f': 6, 'i': 9, 'h': 8, 'k': 11, 'j': 10, 'm': 13, 'l': 12, 'o': 15, 'n':
14, 'q': 17, 'p': 16, 's': 19, 'r': 18, 'u': 21, 't': 20, 'w': 23, 'v':
22, 'y': 25, 'x': 24, 'z': 26}


inv_map = {v: k for k, v in characterMap.items()}


plaintext = 'saintliness wearyingly shampoo headstone syrian elapse
between eigenstate suspends deferentially amu'


t = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]


rotatedArray = (map(lambda x: characterMap[x], plaintext))


for i in range(len(rotatedArray)):


    rotatedArray[i] = (t[i % len(t)] + rotatedArray[i]) % 27


rotatedText = map(lambda x: inv_map[x], rotatedArray)


print (" ").join(rotatedText)


'''
tclrymkqixtbzifs lrlm cwmbosstajheitvrrjauavnbpciqbrviecgw
jfpcinhgqwybvhdxvusiseuchjgguisukdpqzbdqz
icymshbgi
fnrtjebpiyiqgwegqudrfcvywjpjdyigchfuccelbkqwyavksxfbuyqfucwybihdkjxhdfmnr
xavkieecweerwdmq
having developed methods for measuring the data against those rules stage
five allows the data quail
```

```
tclrymkqixtbzifs lrlm cwmbosstajheitvrrjauavnbpciqbrviecgw
jfpcinhgqwybvhdxvusiseuchjgguisukdpqzbdqz
'''
```

OUTPUTS:

```
charanpuli@linux:~/Desktop/c$ python si.py
Enter your the value of keylength5
Enter the cipher text :icymshbgi fnrtjebpiyiqgwegudufcywjpjdyigchfuceelbkqwyavksxfbuyqfucwybihdkjxhdfmnr xavkieecweerwdmq
[9, 3, 25, 13, 19, 8, 2, 7, 9, 0, 6, 14, 18, 20, 10, 5, 2, 16, 9, 25, 9, 17, 7, 23, 5, 7, 21, 4, 21, 6, 3, 25, 23, 10, 16, 10, 4, 25,
 9, 7, 3, 8, 6, 21, 3, 5, 5, 12, 2, 11, 17, 23, 25, 1, 22, 11, 19, 24, 6, 2, 21, 25, 17, 6, 21, 3, 23, 25, 2, 9, 8, 4, 11, 10, 24, 8,
 4, 6, 13, 14, 18, 0, 24, 1, 22, 11, 9, 5, 5, 3, 23, 5, 5, 18, 23, 4, 13, 17]
[5, 21, 6, 24, 4, 17, 9, 2, 5, 27, 17, 13, 25, 11, 5, 18, 9, 16, 7, 7, 4, 22, 2, 3, 23, 20, 14, 4, 2, 13, 12, 16, 12, 1, 2, 3, 19, 27
, 9, 1, 21, 23, 21, 12, 14, 23, 18, 5, 2, 8, 16, 3, 2, 27, 10, 27, 19, 23, 19, 9, 12, 22, 5, 24, 7, 25, 23, 5, 14, 13, 3, 13, 13, 10,
 3, 21, 25, 24, 20, 22, 18, 22, 22, 7, 10, 26, 27, 27, 18, 27, 18, 5, 20, 24, 5, 26, 22, 25]
[4, 9, 19, 16, 15, 18, 20, 5, 4, 0, 16, 1, 20, 9, 5, 14, 20, 0, 2, 18, 5, 22, 5, 20, 9, 14, 7, 0, 19, 20, 18, 9, 11, 9, 14, 7, 12, 25
, 0, 6, 9, 12, 12, 9, 16, 9, 14, 7, 0, 3, 1, 20, 23, 1, 12, 11, 0, 1, 14, 20, 9, 3, 12, 9, 14, 5, 0, 20, 15, 23, 5, 18, 25, 0, 21, 14
, 6, 9, 20, 19, 0, 5, 2, 21, 12, 12, 9, 5, 14, 3, 5, 0, 12, 21, 18, 5, 18, 19, 0, 4]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27]
```

Dictionary-1:

```
charanpuli@linux:~/Desktop/c$ python encryption_generator.py
ukijevypeuifzlauqmukytbvrnmnfthhexaccwsxfkfehhjt adhjwolofvdhxhqwoisilailpyfkjuvyfjhcnhdhgsygvdskbft
```

Dictionary-2

```
charanpuli@linux:~/Desktop/c$ python encryption_generator.py
ueovcsrrk xgepgucpwrtaezqguztvdnmkpuxuvogabxqfuiprduznkjhycnkvjjpkkvbecxkhbaatkviziokijynyalfsudhkra
```