Project 1 (Cryptanalysis of a class of ciphers)

Submitted by:

Michael Chen ([mc7232@nyu.ed](mailto:mc7232@nyu.ed))

Yin Tian Chen ([ytc344@nyu.edu](mailto:ytc344@nyu.edu))

Robert Frost ([rf1327@nyu.edu](mailto:rf1327@nyu.edu))

Table of Contents

[1. Introduction 3](#_Toc116294119)

[1.1 Team Members 3](#_Toc116294120)

[1.2 Approach 3](#_Toc116294121)

[2. Informal Explanation 3](#_Toc116294122)

[2.1 Naïve Approach 3](#_Toc116294123)

[2.2 Statistical Approach 4](#_Toc116294124)

[3. Rigorous Explanation 4](#_Toc116294125)

[3.1 Naïve Approach 4](#_Toc116294126)

[3.2 Statistical Approach 5](#_Toc116294127)

[4. Conclusion 7](#_Toc116294128)

[5. References 8](#_Toc116294129)

# Introduction

## 1.1 Team Members

Robert Frost implemented our solution for the known plain text attack and our attempt to use statistical analysis for the second decryption approach. Michael formulated a solution utilizing a hash table to track character differences and a brute force approach to deal with random insertions. Yin Tian helped Robert design the statistical analysis approach. All three of us contributed to the report, with Robert writing the initial draft and the others helping to edit and refine.

## 1.2 Approach

Our group designed to methods for extracting the plaintext, one naïve method that relies upon known plaintexts and a second statistical method. The naïve approach works for the first test when one does not consider the random insertions. The second approach, while we believe has merit, did not fully solve test two.

# 2. Informal Explanation

## 2.1 Naïve Approach

The naïve approach relies on knowing the full universe of potential plain texts that might be enciphered to create the cipher text. It also relies on the encryption key being linearly and cyclically applied. The algorithm iterates through all potential key lengths and all potential plain texts. For a given key length k, it then selects each “kth” character from both the plain text and cipher text. It compares how much each character in the plain text would have to shift to achieve the cipher text. If all shifts are the same, we have found a potential key length. All possible key lengths are then used to build a candidate key from each plain text and the cipher text. That key is then applied to the entire message to determine if we have found the actual correct plain text. This approach scales only moderately well as the universe of plain text candidates grows. The algorithm completes in four seconds at a thousand candidates, 21 seconds at ten thousand and jumping to nearly 5 minutes for a hundred thousand.

## 

## 2.2 Statistical Approach

Our second approach attempts to use the normal distribution of characters within an English text and repeated N-grams within the cipher text to attempt to identify the key without relying on knowledge of the plain text. This is because there are far too many combinations of words (roughly 4\*10^106) from the second dictionary to allow the first approach to be successful. First, our algorithm finds all character sequences of length three that repeat in the cipher text. It then calculates the gaps between those characters repeating, theorizing that a repeat might indicate the same three-character plain text being enciphered by the same part of the key. All found gaps are then compared to all possible key lengths, assuming that key lengths that “fit” more gaps are more likely to be the key length and returning a prioritized list of key lengths to attempt.

Each potential key length is then used to build groupings of characters which are then brute forced for potential key values. The generated decryption attempts are analyzed to determine rough similarity to English and each grouping of characters adds potential key values. Finally, each individual key value possibility is attempted to generate full decrypted texts which are again analyzed. The candidate plain text which scores the most similar to English is returned as the plain text guess.

# 3. Rigorous Explanation

## 

## 3.1 Naïve Approach

As described, the first helper function groups all “kth” characters from a string, simply getting the character at index “x” and subsequently adding k to x until x is greater than the text length.

def get\_all\_k\_th\_chars(text, k):

x = 0

return\_string = ''

while x < len(text):

return\_string.append(text[x])

x = x + k

return return\_string

The second function, calculate shifts, determines how much a character at index i must be shifted to match the character at the same index in a different string.

alphabet = ' abcdefghijklmnopqrstuvwxyz'

def calculate\_shifts(string\_one, string\_two):

differences = []

for i in [0, len(string\_one)]:

difference = alphabet.index(string\_one[i]) - alphabet.index(string\_two[i])

if difference < 0:

difference = 27+difference

differences.append(difference)

return difference

The third function in determining valid key lengths for a cipher text and plain text pair simply combines the two functions and examines the output.

def key\_length\_brute\_force(cipher\_text, plain\_text):

possible\_lengths = []

for possible\_length in [1, max\_length]:

cipher\_chars = get\_all\_i\_th\_chars(cipher\_text, possible\_length)

plain\_chars = get\_all\_i\_th\_chars(plain\_text, possible\_length)

shifts = calculate\_shifts(plain\_chars, cipher\_chars)

if all shifts are the same:

possible\_lengths.append(possible\_length)

return possible\_lengths

The key length brute force function is applied to all possible plain texts. If it finds potential key lengths, the algorithm rebuilds the potential key. This is done below, by iterating up to the potential key length characters in both texts and determining how much the characters were shifted between the two.

def build\_key(cipher\_text, plain\_text, length):

key = []

for i in [0, length]:

key.append(calc\_shift(plain\_text[i], cipher\_text[i]))

return key

Finally, the key is applied to the entire cipher text to check if it matches the candidate plain text. If successful, the plain text is returned and the function terminates. The final function is shown below.

def test\_one(cipher\_text, all\_plain\_texts):

for plain\_text in all\_plain\_texts:

possible\_key\_lengths = brute\_key\_len(cipher\_text, plain\_text)

for key\_len in possible\_key\_lengths:

possible\_key = build\_key(cipher\_text, plain\_text, key\_len)

if decrypt(cipher\_text, possible\_key) == plain\_text:

return d

## 3.2 Statistical Approach

For our attempted statistical approach, first the algorithm finds repeated N-grams, by default N=3.

def find\_repeat\_sequences(cipher\_text, length = 3):

repeated\_sequences = []

for i in [0, len(cipher\_text) – length]:

sequence = cipher\_text[i:i+length]

for j in [i+1, len(cipher\_text)-length]:

if cipher\_text[j:j+length] == sequence:

repeated\_sequences.append(sequence)

return repeated\_sequences

We then find the sizes of the gaps between sequences and their repetitions in the cipher text.

def find\_gaps\_between\_repeats(cipher\_text, repeated\_sequences, length=3):

gaps = []

for sequence in repeated\_sequences:

mark = None

for i in [len(cipher\_text)-length]:

if cipher\_text[i:i+length] == sequence:

if mark:

gaps.append(i-mark)

mark = i

return gaps

The gap sizes are then utilized to determine candidate key lengths. Each gap is compared to potential key lengths, if the key length divides the gap evenly, there is a chance the two sequences were the same string in the plain text encountering the same key material during a cycle.

def analyze\_gaps(gaps):

key\_lengths = {}

for potential\_length in [1, max\_key\_length]:

key\_lengths[potential\_length] = 0

for gap in gaps:

if gap % potential\_length == 0:

key\_lengths [potential\_length] += 1

sort key\_lengths using the hashtable values

return key\_lengths

The last function performs several steps to combine the work done to this point and a candidate key length. First, it creates a number of groups of characters equal to the candidate key length, so that one group would have all characters enciphered by the first key value, another would have all characters enciphered by the second, and so on.

def candidate\_freq\_analysis(key\_length, cipher\_text):

groups = []

for i in range [0, key\_length]:

new\_group = get\_all\_k\_th\_chars(cipher\_text, key\_length, i)

groups.append(new\_group)

It then attempts to decrypt each group using one of the 27 possible alphabet values that might have been used to encipher the text. The decrypted characters are analyzed to determine how similar the characters are to normal English distributions which is what we would expect. The scoring function and frequency analysis code was taken from an open source github repository and is cited below. Finally, each of the potential key values is sorted based on how correct of an English distribution they produced, with only high scoring values carried forward.

for group in groups:

scores = []

for subkey in range(0,28):

pt\_candidate = decrypt(group, subkey)

score = frequency\_score(pt\_candidate)

scores.append((score, subkey))

scores.sort()

best\_score = scores[0][0]

subkeys = [all subkeys that produced best\_score]

test\_key.append(subkeys)

Following this step, we now have potential values for each character of a potential key. We then build all possible combinations using all of the potential values, leaving a list of possible keys of the given length. Each possible key is then used to decrypt the full text, and each decryption is again analyzed for its closeness to English. The best scoring decryption is returned along with its score.

for key in all\_keys:

pt\_candidate = decrypt(cipher\_text, key)

score = frequency\_score(pt\_candidate)

if score > best\_score:

best\_score = score

best\_pt = pt\_candidate

return (best\_score, best\_pt)

Finally, all of the above processes are repeated for all possible key lengths in priority order based on the gap analysis. The best scoring plain text from all key lengths is then returned for presentation to the user.

repeated\_sequences = find\_repeat\_sequences(cipher\_text)

gaps = find\_gaps\_between\_repeats(cipher\_text, repeated\_sequences)

key\_length\_candidates = analyze\_gaps(gaps)

all\_scores = []

for key\_len\_candidate in key\_length\_candidates:

all\_scores.append(candidate\_freq\_analysis(key\_len\_candidate, cipher\_text))

best = sorted(all\_scores)

return best

# 4. Conclusion

Our naïve approach performed satisfactorily in the simple case of test one but would not achieve the proper execution times for test two. However, we were unable to make a full performance statistical analysis function properly. Possible problem errors, outside of implementation error, might include poor selection of frequency scoring shorthand or improper methodology for determining individual key values. Any purely statistical approach will suffer potential issues in decryption correctness, so improving a correct implementation could also involve aspects of a dictionary attack and grammatical analysis if the attacker knows the cipher text is particular language.

# 5. References

Sweigart Al. *freqAnalysis.Py*. Github: 2013. Retrieved October 3, 2022. <https://github.com/asweigart/codebreaker/blob/master/freqAnalysis.py>

Aldrrab, Nada and May, Jonathan. *Can Sequence-to-Sequence Models Crack Substitution Ciphers?*. Retrieved October 2, 2022. <https://aclanthology.org/2021.acl-long.561.pdf>