Cryptanalysis of a Class of Ciphers based on Frequency Analysis and the Chi Square Test

Introduction

Team Members and Responsibilities:

Jake Nemiroff (jrn8168@nyu.edu)
Staford Titus S (ss14958@nyu.edu)

Staford was in charge of researching how a frequency of character analysis using a chi squared test might be useful in terms of decoding ciphertext that was ambiguous. He also worked on writing out the algorithm that would be used for decryption.

Jake also researched how the chi squared test and frequency analysis could be used to decode plaintext in certain edge cases. He also worked on implementing the algorithm that Staford had written out.

This project was written using the python language. The math and random packages were imported to help with the implementation.

We had first started implementing a regular frequency analysis algorithm, however we soon realized we needed to augment our approach with statistical analysis when faced with the case where none of the plaintext samples provided in dictionary_1.txt could be eliminated based on the amount of random characters generated. (More on this in the next section).

Approach Explanation: (More English than Pseudocode)

For this project, we first worked on implementing the Encryption Algorithm on our end. We built a generate_key function that generates the key which is a set of 27 distinct numbers, which is then passed to the encryption function defined below. Here is a snippet of our Encryption Algorithm:

```python
def generate_key(plaintext):

   key = []

   for char in plaintext:
       key = random.sample(key_vals, len(key_vals))

   return key

def find_char(c, key):

   if c == ' ':
       letter = 0
   else:
       letter = ord(c) - ord('a') + 1

   final = key[letter]

   if final == 0:
       char = ' '
   else:
       char = list(alphabet.keys())[final]

   last = char

   return last

def encrypt(message, key):

   ciphr_ptr = 0
   msg_ptr = 0
   num_rand_characters = 0
   L = 500
   ciphertext = []

   probabilities = [0, 0.05, 0.15, 0.25, 0.35, 0.45, 0.55, 0.65, 0.75]
```

```
prob_of_random_ciphertext = random.choice(probabilities)

# print('KEY: ', key)


while ciphr_ptr < L + num_rand_characters:
    # coin_generation_algorithm(ciphertext_pointer,L)
    coin_value = random.uniform(0,1)

    if prob_of_random_ciphertext < coin_value and coin_value <= 1:
        j = message[msg_ptr]
        char = find_char(j, key)
        # print(char)

        ciphertext.append(char)
        msg_ptr += 1

    if 0 <= coin_value and coin_value <= prob_of_random_ciphertext:
        random_char = random.choice(list(alphabet.keys()))
        ciphertext.append(random_char)
        num_rand_characters += 1

    ciphr_ptr += 1

return ''.join(ciphertext)
```

We then moved on to work on the decryption function. Hence, for this, we first identified that since, random characters were being added to the ciphertext, the probability of success would rest on the prob_of_random_ciphertext and the coin_value variables since those are the variables that determine the amount of randomness in the ciphertext. Thus, we first wrote our program to work for only the basic substitution cipher where every character in the plaintext space is replaced by a single character in the ciphertext space. That is, for every p[i], in the plaintext, there is a unique mapping to c[i'] in the ciphertext. It was easy when the ciphertext length was the same length as of the plaintext and when the prob_of_random_ciphertext was in the lower sub-zero ranges like 0.05, 0.15, 0.25, and such. We start by calculating the length of the ciphertext. We know that if the length is equivalent to the lengths of the plaintext samples found in dictionary_1.txt, no random characters have been added. This is the easiest case. We can simply build our distribution of letters and perform a frequency analysis comparing our ciphertext to each of the plaintext samples. Whichever distribution of plaintext characters is equivalent to the distribution of ciphertext characters is our original message. This part of our algorithm runs in O(n) time. But, when the prob_of_random_ciphertext value is increased to 0.75, the probability of successfully analyzing and getting the plaintext was considerably reduced. Our strategy at this point is to calculate the maximum character frequency for each of the plaintext messages and reduce the ciphertext frequency by at least the difference between the maximum frequency and the current ciphertext frequency for that character. For example, if out of all plaintext messages found in dictionary_1, the maximum amount of Z's is 57, we know that if our ciphertext has 92 Z's, we can reduce it to at least 57 without altering the integrity of the message. We perform this operation for each letter in our ciphertext to simplify the next process as much as possible. After getting rid of as many random characters as we can, we take a look at the frequency of each character in the ciphertext and compare it to the frequency of each character in the plaintext. If any character is more frequent in the plaintext than in the ciphertext, that plaintext message can be disqualified as a possible original message.This thereby helped us obtain better results for those ciphertexts that were slightly greater in length than the plaintexts. This works in some cases where the ciphertext contains characters where the frequency of that character was not falsely augmented by the random generation part of the encryption algorithm. If we are lucky and a single plaintext message can be discerned by this process, we return it. We traverse the frequencies from Z to A as there are far larger frequencies at the end of the alphabet for each plaintext. This means that we will be able to disqualify plaintext messages far quicker if we start at the end of the alphabet rather than the beginning. This part of our algorithm runs in O(n^2) time. However, this is not terrible as there will be at most 135 comparisons (27 letter checks multiplied by 5 plaintext messages). However, for particularly large plaintexts, which were almost two times larger than the ciphertext, we weren't able to obtain a perfect cryptanalysis strategy, but implemented the chi square test which is also the same method that we thought of implementing for test 2. The chi square test doesn't always provide accurate results as the sample size was too small for it to be effective.

Approach Explanation: (More Pseudocode than English)

As explained above, we made use of frequency analysis. Hence, we first tried it for ciphertext without the addition of random characters as follows where we just compared the distributions/frequencies with those of the plaintexts provide and returned the one which matched:

```
if cipher_count == plaintext_count:

       for i in range(len(dictionary_distribution_mapping)):

           if ciphertext_distribution ==
list(dictionary_distribution_mapping.values())[i]:

               return plaintext_dictionary[i]
```

Then, we started to work on the following code which would update the ciphertext length as well as the ciphertext distribution if the ciphertext length was larger than that of the plaintext. Here, we worked on a specific logic where if the length of the ciphertext was greater than length of the plaintext, then the frequency/distribution of each of the characters of the plaintext were compared to that of the ciphertext. Here, we obtain the distributions, sort them and if the distribution of a specific character was to be greater than the distribution of the character in the ciphertext, we reduce the number of that particular character found in the distribution and the entire ciphertext length using: (distribution of ciphertext - max frequency for the specific character in the list of all plaintexts provided). The pseudocode is as follows:

```
max_freq = [max(i) for i in zip(*dictionary_distribution_mapping.values())]

       # print('max freq: ',max_freq)

       for i in range(len(alphabet)):

           if ciphertext_distribution[i] > max_freq[i]:
               cipher_count -= (ciphertext_distribution[i] - max_freq[i])
               ciphertext_distribution[i] = max_freq[i]
```

Thus, we perform the following operation of comparing the distributions of the updated ciphertext with that of the plaintexts and remove those plaintexts that have specific character distributions greater than that of the ciphertext. This way the ciphertext that remains is output. If more than one remains, then we perform more comparison operations, but that doesn't provide the desired results. Here is the code for what was explained above:

```
for plaintext_distribution in dictionary_distribution_mapping.values():

           for i in range(len(alphabet) - 1, -1, -1):

               if ciphertext_distribution[i] < plaintext_distribution[i]:
                   print(ciphertext_distribution[i], plaintext_distribution[i])

                   y=find_key(dictionary_distribution_mapping, plaintext_distribution)

                   print('ITEM TO DELETE: ', y, '\n')

                   x =
possible_plaintexts.remove(find_key(dictionary_distribution_mapping,
plaintext_distribution))
                   break
```

All that the find_key function does is as follows:

```
def find_key(input_dict, value):

   return next((k for k, v in input_dict.items() if v == value), None)
```

Now, if the possible_plaintext list still contains more than one plaintext, then we continue to the next step, i.e., performing chi square analysis.

```
chi_square_statistic = {}

           for index, plaintext in enumerate(possible_plaintexts):

               dictionary_distribution_mapping[possible_plaintexts[index]] =
build_distribution(plaintext)
```

Our approach for determining the chi square statistic is as follows: We first loop through each remaining plaintext message and build distributions for each of those. Technically we could reuse the ones calculated previously but since the time complexity doesn't change it makes little difference. This also helped us organize this part of our code better.

```
for plaintext_distribution in dictionary_distribution_mapping.values():

               chi = 0

               for index, letter in enumerate(alphabet.keys()):

                   expected_letters = len(ciphertext) *
expected_letter_frequency[letter]

                   diff = pow((expected_letters - plaintext_distribution[index]), 2)

                   if plaintext_distribution[index] == 0:
                       chi += 0

                   else:
                       chi += diff / plaintext_distribution[index]


               chi_square_statistic[chi] = plaintext_distribution
```

Next we looped through each plaintext distribution. We set an initial chi value to zero and traverse each letter in our alphabet. ([a-z] plus ' '). We multiply the expected percentage of frequency of letters in the English by the length of the ciphertext and then subtract each actual frequency from the plaintext messages. We then square the result and divide by the actual frequency to get our chi square statistic. We have a quick check to avoid a divide by zero error since there are certain letters that do not appear in certain plaintexts.

```
chi_square_statistic_sorted = sorted(list(chi_square_statistic.keys()))
          min_chi_val = chi_square_statistic_sorted[0]

          result = find_key(dictionary_distribution_mapping,
chi_square_statistic[min_chi_val])

          return result
```

Finally, we determine the minimum chi square statistic and return the plaintext associated with that score. We are able to obtain considerable, although not a 100% accurate result of deciphering the original plaintext message using our cryptanalysis strategy.