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

Introduction

Team Members:

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

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:

```
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 con_value variables since those are the variables that enhance the 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. But, when it increased to 0.75, the probability of successfully analyzing and getting the plaintext was considerably reduced. Hence, we worked on a 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 the and on analysis if the distribution of a specific character was to be greater than the distribution of the character in the ciphertext, then we reduce the number of the distribution and the entire ciphertext by the (distribution of ciphertext - max frequency for the specific character in the list of all plaintexts provided). Based on this, we then make comparisons of both the distributions and start removing those plaintexts that have the distribution of any of their characters greater than the distribution of the corresponding character in the ciphertext in sorted order. This thereby helped us obtain better results for those ciphertexts that were slightly greater in length than the plaintexts. For very long plaintext though, which were almost two times larger than the ciphertext, we weren't able to obtain a perfect cryptanalysis strategy, but thought of implementing the chi square test which is also the same method that we thought of implementing for test 2. The chi square part hasn't been completely implemented, but it does produce output though the output isn't as expected.

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 the and on analysis if the distribution of a specific character was to be greater than the distribution of the character in the ciphertext, then we reduce the number of the distribution and the entire ciphertext by the (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, then 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. Hre 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)
```

Hence, in this way, we were able to obtain considerable, though not 100% of cryptanalysis.