Project 1: Cryptanalysis of a class of Ciphers based on utilization of dictionaries and ngram frequency

Leo Ho, Matteo Mastrogiovanni, Nathan Cantu

Introduction

The team members for this project are Leo Ho, Matteo Mastrogiovanni and Nathan Cantu. Matteo worked on decrypting Test 1 when there were no random characters inserted. Leo worked on guessing plaintext in dictionary_1 from ciphertext with random characters inserted. Nathan worked on frequency analysis in Test 2. Every member worked on their corresponding section for the report.

Test 1: Cryptanalysis using a set of Dictionaries

Case 1: There is no random character inserted into the ciphertext

The first step in solving step 1 was to understand how the encryption worked and how to solve it without the random character insertion. It was also helpful in making sure our encryption algorithm used for testing worked properly. By the end of the project, not a lot of code from this section was used. It was more of a means of assistance in understanding how to use some basic functions on the cipher. The main goal of this section was to see if hardcoding a solution would work once random characters were implemented. We found that this was not the case and continued the project using other means.

For each item in the dictionary given, the approach would be to shift each character in the cipher until it was equal to the corresponding character in each dictionary. This was done in a single function that compared cipher[i] and plaintext[i] (for i=length of the ciphertext) and subtracted the difference between them. This distance was recorded in an array corresponding to the length of the cipher. The implementation of this code is below. Reindexing is used so that it is easy to subtract and add numbers to the characters without having to worry about ASCII values.

```
#reindex so that ' '=0, a=1,...,z=27
def reindex(text,int):
    original_index = ord(text[int]) - ord('a') + 1
    if text[int] == ' ':
        original_index = 0
    return original_index

#calculate the distance to get from cipher[i] to plaintext[i]
```

```
def compare_cipher_and_dict(ciphertext, dict):
    key = []
    for i in range(0,len(ciphertext)):
      newindex_cipher = reindex(ciphertext,i)
      newindex_plain = reindex(dict,i)
      difference = (newindex_cipher - newindex_plain + 27) % 27
      key.append(difference)
    return key
```

The next step of the algorithm was to find the proper key length, given the key that was the length of the cipher. This function cycled through the available values for t (24 to 1) in reverse order, so that the function didn't get caught up on duplicate numbers. It examined the key from [0:t] and [t:2*t+1] and confirmed if they were equal or not. This function returned the key length, which essentially provided us with the correct key that was used for encryption. The code for this is found below. Note, this implementation does not take into account a factoring problem. If the key length is 6, the program will most likely return 24 to the fact that it is testing 24 first. For this implementation, this does not actually matter as it will decrypt the cipher the same way.

```
#finding the key length by testing if arrays are equal
def find_key_length(keyArray):
    keyLength = 0
    keyGuess = []
    for t in range(24,1,-1):  #each t
        if keyArray[0:t+1] == keyArray[t:2*t+1]:
            keyLength = t
            keyGuess = keyArray[0:t]
            break
        else:
            keyLength = 0
            keyGuess = []
    return(keyLength,keyGuess)
```

The last part of the algorithm decrypted the cipher using the key that was found. This was done by shifting each character in the cipher by its corresponding number in the key. Once this was done, we could determine if the dictionary item used was the correct one or not, since it would either be a 1:1 duplicate or completely off. The program cycled through each item in the dictionary until it found one with the 1:1 match.

The code below shows the implementation of what is explained above. The program returns a key length of 0 when there is no key found from the code above. This is so the program continues to cycle through the dictionaries until it has a correct key.

```
#decryption scheme for when there is no random characters
def decrypt_norandom(cipher, keyguess):
    decrypted = ""
    i = 0
    while len(decrypted) < len(cipher):
        for j in range(0,len(keyguess)):
            if len(decrypted) < len(cipher):
                original_index = ord(cipher[i]) - ord('a') + 1
            else:
                break
            if cipher[i] == ' ':
                original_index = 0
            original_index -= keyguess[j]
            templetter = mod_encrypt(original_index)
            decrypted = decrypted + templetter
            i += 1
    return decrypted

#compare each plaintext to the cipher
#when a key is found, that plaintext is the answer
def test1_norandom(cipher,dict1):
    for i in range(0,len(dict1)):
        tempkey = compare_cipher_and_dict(cipher,dict1[i])
        keyLength,keyGuess = find_key_length(tempkey)
        if keyLength != 0:
            decrypted_text=decrypt_norandom(cipher,keyGuess)
            break
    return (decrypted_text, 100)
```

Again, this method was proved to be inefficient in determining a solution once random characters were introduced. This is because much of this algorithm revolves around the cipher and plaintext being the same length. It breaks once one random character is introduced.

Case 2: There are random characters inserted into the ciphertext

This approach is inspired by the n-gram frequency. An n-gram is a contiguous sequence of n items from a given sample of text or speech. According to Glottopedia, ngram frequency is the mean, or summed, frequency of all fragments of a word of a given length. For example, the word ‘dog’ will have two 2-grams ‘do’ and ‘og’.

This notion can somewhat be applied to guess the plaintext in dictionary_1 from the ciphertext. It is known that the key length (between 1 and 24) is much shorter than the plaintext’s length (500), so the key must be applied many times across the plaintext. Each character in the plaintext is shifted by an amount specified using a character in the key to get the character in the ciphertext. Thus, the shifts between each character in the plaintext and the cipher can be calculated to find the key character in that position.

However, there are random characters inserted into the ciphertext. These random characters are inserted randomly, and they do not correspond to any character in the plaintext, so the key sequence is separated.

k1	k2	k3	k4	k5	k6		k7	k8	k1		k2
c1	c2	c3	c4	c5	c6	**c7**	c8	c9	c10	**c11**	c12

Above is an example of a key of size 8 and the bold c is the random character inserted in a chunk of size 12. Since the key is applied multiple times across the plaintext to get the ciphertext, with the random characters inserted, there must be parts of the key that are repeated multiple times across the plaintext. This is where ngram will be applied to find a certain part of the key with length of n that is repeated the most across the plaintext. The ciphertext has r random characters inserted, so the plaintext will be compared r times to r substrings of similar length in the ciphertext, moving up by 1 character at a time, like a sliding window. Ngram of the keys will be calculated over all of the r comparisons to get the ngram that repeats the most (which is a part of the key that keeps repeating).

The code will choose each of the plaintext in dictionary_1 and run the ngram calculation algorithms to find the maximum number of times an ngram repeat in that plaintext. Since the plaintexts’ characters are quite different from one another and the distribution are also different, the correct plaintext will give the highest ngram values for a certain part of the key. In the code, ngram’s length was chosen to be 5 to make it 1% of the length of the plaintext. This is for the assumption that there should be around 5 ciphertext characters between any two random characters on average. If there are more random characters, ngram value can be decreased.

Code:

This code below creates a PlaintextShifts class which is used to store two variables: the plaintext which is chosen from dictionary_1, and r_shifts which is a list of all r shifts using the chosen plaintext to the cipher text.
There is a function to calculate the ngram frequency in the r_shifts list. The ngram is a sublist of each item in the r_shifts list with size of n. A dictionary was created to store all the data. The goal is to get the ngram with the largest number of appearances.

```
class PlaintextShifts:
    def __init__(self, plaintext, r_shifts):
        self.plaintext = plaintext
        self.r_shifts = r_shifts

    def find_ngram_freq(self, n):
        ngram_dict = {}
        for each in self.r_shifts:
            l = len(each)
            for i in range(0,l-n+1):
                ngram = tuple(each[i:i+n])
                if ngram not in ngram_dict:
                    ngram_dict[ngram] = 1
                else:
                    ngram_dict[ngram] += 1
        return ngram_dict
```

In order to get the r_shifts list, these below functions are used.

- calculateShifts() take two character from the plaintext and cipher at the same index and compute how many shifts it take to turn the plaintext character into the ciphertext character
- lstShifts() is mapping all the characters from the plaintext to the ciphertext with the calculateShift() function to get a list of all shifts corresponding to the difference between each character in both texts at the same index

```
# calculate how many shifts it take to turn pChar into cChar
def calculateShifts(pChar, cChar):
    p = get_char_ind(pChar)
    c = get_char_ind(cChar)
    if c >= p:
        return c - p
    return 27 - p + c
```

```
# make a list of all shifts of each character
def lstShifts(pText, cText):
    return list(map(calculateShifts, pText, cText))
```

Next, get_r_shifts() is used to get the r_shifts list for each plaintext in dictionary_1, put them in the PlaintextShifts class, then put them into a list. This final list will contain 5 PlaintextShifts object.

```
# if there are r random characters inserted, compare the plaintext to r
substrings
# of same length from the ciphertext, each time moving up one character
def get_r_shifts(c):
    r_shifts = []
    for p in dictionary_1:
        if len(p) >= len(c):
            continue
        shifts = []
        r = len(c) - len(p)
        for i in range(0,r+1):
            shifts.append(lstShifts(p, c[i:i+len(p)]))
        r_shifts.append(PlaintextShifts(p,shifts))
    return r_shifts
```

Below is the solver function utilizing all of the above. The function is getting the r_shifts list from the ciphertext, which will return 5 PlaintextShifts object. Each object will be computed to create the ngram_dict, and then get the ite with the largest value in its ngram_dict. The result is the plaintext which generates the item with the largest value among the 5.

```
# solver function
def solve_random_char_dict1(ciphertext):
    r_shifts = get_r_shifts(ciphertext)
    if len(r_shifts) == 0:
        return None
    # find the maximum ngram value of each plaintext and return the one
with the largest value
    # ngram value chosen as 5 in this case
    max_ngram = list(map(lambda x: (x.plaintext,
                        max(x.find_ngram_freq(5).values())), r_shifts))
    result = max(max_ngram, key=lambda x: x[1])
```

```
    return result[0]
```

Test 2: Cryptanalysis using a set of words

The first step towards solving the second case relied on using the technique known as the index of coincidence. For every possible t value (from 1 to 24) you have to iterate over the ciphertext and you have to find the index of coincidence for each substring of that ciphertext.

```
def get_key_length(cipher):
  ic_table=[]
  for t in range(1,24):  #try for each t
    sum=0.0
    avg=0.0
    for i in range(t): #go through t
      sequence=""
      for j in range(0, len(cipher[i:]), t):
        sequence += cipher[i+j]
      sum+=index_coincidence(sequence)
```

What this means is that for every substring, you have to count the occurrences of each letter in that substring and then times that count by the count minus 1. The total number of occurrences for each character is added to a sum. This sum is then divided by the length of the cipher times the length of the cipher minus 1.

```
def index_coincidence(cipher):

  N = float(len(cipher))
  sum = 0.0

  for letter in alphabet:
    sum+= cipher.count(letter) * (cipher.count(letter)-1)

  ic = sum/(N*(N-1))
  return ic
```

All of this follows the Index of coincidence formula shown here:

$$\mathbf{IC} = \frac{\sum_{i=1}^{c} n_i(n_i - 1)}{N(N-1)/c}$$

When you have the index of coincidence for each substring, that value is then added to a sum, and the sum is then divided by the t value since there were t substrings taken. The average for that t value is then added to a table which at the end, houses every ic average for every value of t. The table is then sorted, with the highest valued average being at the first position. Because that t has the highest average number of occurrences, that is our best guess as to what the length of the key is. The second best guess is the value in the second position in the sorted list. Because the average number of occurrences is roughly equivalent to when that t value is doubled or tripled etc, if the first position “best guess” t value is divisible by the second, then we determine that it is more likely that the second “best guess” is actually the length of the key.

```
      sum+=index_coincidence(sequence)

    avg=sum/t
    ic_table.append(avg)

best_guess = ic_table.index(sorted(ic_table, reverse = True)[0])
second_best_guess = ic_table.index(sorted(ic_table, reverse = True)[1])

#if the first guess is divisible by the second, return the second
if best_guess % second_best_guess == 0:
    return second_best_guess
else:
    return best_guess +1
```

Taking the length of the key we can then try to decipher the actual key values. In order to do this, we have to iterate over the ciphertext, starting at index 0, jumping to every character that is included in the shift t. For example, if the t was 4, you would start at index 0 and push that character into a dictionary and then jump 4 indexes forward in the text to get the next character, which is then put into the dictionary. This continues until the ciphertext ends. When the ciphertext ends, you then have a dictionary that lists the count of occurrences for each character that was included in the shift.
On the next iteration, you iterate the start value 1 index to the right, and then you perform the same shift character occurrences. You continue doing this, shifting the start

value 1 index to the right, for t iterations to make sure that you have every character in the ciphertext included in your analysis.

```
def decipher(cipher, t,alpha_frequencies):
    keys = []
    for jump in range(t):
        dic = {}
        freq = {}
        total_values=[]
        total = 0
        for i in range(jump,len(cipher),t):
            if cipher[i] in dic:
                dic[cipher[i]] += 1
            else:
                dic[cipher[i]] = 1
```

During each in range(t) iteration, we have to then in turn figure out the frequency of each letter in the encrypted text. To do this, we have to start by totaling the number of occurrences in the entire ciphertext and then dividing the number of occurrences of each character by the total number in order to get that character's frequency in that shift t.

```
for value in dic.values():
    total += value
for key, value in dic.items():
    frequency = value/total
    if key not in freq:
        freq[key] = frequency
        #['a':0.25]
```

Once that is accomplished for every character, you can move on to comparing the encrypted frequencies to general English letter frequencies. To start, you take the general English letter frequencies in the order of the alphabet and then multiply each character's encrypted frequency with the frequency of its general letter frequency. After you perform that for each letter, you then add all of those values up and you get the overall accuracy rate for that particular t shift. Originally the t starts at 0, which is when every letter cipher frequency is matched exactly with its character general frequency, but as t increases, the general frequency positions stay the same but they are then multiplied by the letter that is in that position because of a shift number t. You do this until the number t has been reached.

```
        total = 0
        freq = sorted(freq.items())
        order_of_chars =[]
        for i in freq:
            order_of_chars.append(alpha_frequencies[i[0]])
        for i in range(len(freq)):
            total = 0
            for j in range(len(freq)):
                val = freq[j][1]
                total += order_of_chars[j]*val
            total_values.append(total)
            first_char = freq[0]
            for k in range(len(freq)):
                if k == len(freq)-1:
                    freq[k] = first_char
                else:
                    freq[k] = freq[k+1]
```

The shift with the highest accuracy rate represents the ith key-value.

```
        mx=0
        mx_index=0
        for i in range(len(total_values)):
            if total_values[i] > mx:
                mx = total_values[i]
                mx_index = i
        keys.append(mx_index)
    return keys
```

You perform all of this for every range(t) iteration, shifting the start value one index to the right. Once you have every key value, then you can go onward to decrypting the ciphertext.

Going into the decryption, the first step is to get the alphabet index of each ciphertext character. You iterate over the key_length and for every iteration, you apply the key value that is in proportion to the character index that you are currently on. For example, the first index would have had the first key-value shifted onto it initially and the second would have had the second key-value and so on until it repeats. You subtract the key-value shift from the character to try to get the original plaintext.

```
def decrypt(cipher,keyguess):
    decrypted = ""
    i = 0
    while len(decrypted) < len(cipher):
        for j in range(0,len(keyguess)):
            if len(decrypted) < len(cipher):
                original_index = ord(cipher[i]) - ord('a') + 1
            else:
                break
            if cipher[i] == ' ':
                original_index = 0
            original_index -= keyguess[j]
            templetter = get_char(original_index)
            decrypted = decrypted + templetter
            i += 1
    return decrypted
```

Unfortunately, although the function to calculate the key length worked as planned, the function to decipher the ciphertext did not return the accurate key values and thus the ciphertext was still unsolved.

Encryption Scheme

We developed an encryption algorithm for the purpose of testing our solutions. This encryption algorithm follows the specifications detailed in the project manual. The full encryption scheme can be found in encryption.py, attached in the Github Repository. This code can also be used in a terminal. When run, the terminal will prompt the user to input the probability of a random character insertion (a number between 0 and 1). It will then perform a coin flip to determine whether to generate a cipher from test 1 or test 2. If test 1 is used, it will randomly pick an item from the dictionary. If test 2 is used, the program will generate a random text using the provided word bank and cut off the length at 500. The program will then run the encryption algorithm.

First, a random t is generated with bounds of (1,24). This is used to generate a random key of length t, with each value having bounds of (0,26). The encryption algorithm will then travel through the length of the message. For each letter, it will perform a coin flip to determine whether or not to insert a random letter. This coin flip algorithm will randomly pick a number between 0 and 1. If the number is below the given random character probability provided by the user, the algorithm will insert a random character. If the coin flip number is above that probability, encryption will continue.

The character of the message will be indexed according to its ASCII value so that it is easy to do calculations. If the character is “ “, its index is 0. The new text is then

generated by adding the corresponding key value to the indexed message character. The index is then converted back into a character, as this process is easier than dealing with ASCII values all the time. The encryption algorithm then returns the ciphertext once it has completed this process for every character.