Monoalphabetic cipher decryption method based on linear regression and standard deviation

Github Link: **https://github.com/bvo4/CS6903-Project-1**

Introduction

Team members:

- Alvin Crighton
  - Worked on: Percentile Matching (Python) based on Brandon and Rohin's findings
- Rohin Dasari
  - Worked on: Linear regression Model & Skew Model (Python)
- Brandon Vo
  - Worked on: Percentile analysis and Chi-Square Statistic (C++)

Encryption Scheme:

Given the pseudocode for the encryption scheme, we each created our own encryption methods based on the pseudocode provided in the project. Our encryption schemes used the C++ library <math.h> and the Python library random in order to implement a pseudo random number generator based on uniform distribution. This allowed us to implement a random coin generation algorithm and a random letter with equal probability for all letters for an accurate noise generator for the encryption scheme.

Our encryption algorithms were coupled with the possibility of random letters to be inserted for each iteration based on a set variable. Once our encryption algorithms were capable of generating a string of ciphertext based on the plaintext input, we used our own encryption schemes and attempted to develop our own decryption methods.

Decryption Schemes:

Upon receiving the plaintext_dictionary, we conducted a letter-based frequency analysis on all five candidate plaintext messages to use as reference for what the ciphertext may hold. While the analysis is unable to take into account the possibility of which letters are randomly generated and which letters are encoded, there is a distinctly imbalanced distribution of letters as shown in the graph below. While all five plaintext candidates have very similar distributions, it identifies and allows us to establish a default mapping of each plaintext letter might map to in regards to the ciphertext.

Plaintext #1 Frequency table:

A	30	B	15	C	22
D	9	E	57	F	8
G	7	H	7	I	32
J	0	K	5	L	29
M	9	N	34	O	20
P	12	Q	1	R	36
S	44	T	27	U	22
V	2	W	5	X	2
Y	7	Z	4	‘ ‘	54

Plaintext #2 Frequency table:

A	33	B	13	C	20
D	13	E	47	F	4
G	16	H	14	I	33
J	1	K	4	L	21
M	8	N	21	O	36
P	9	Q	0	R	44
S	42	T	30	U	15
V	6	W	4	X	1
Y	9	Z	3	‘ ‘	53

Plaintext #3 Frequency table:

A	29	B	11	C	24
D	17	E	48	F	3

G	13	H	9	I	41
J	2	K	4	L	32
M	5	N	36	O	32
P	10	Q	1	R	37
S	36	T	22	U	17
V	5	W	5	X	1
Y	11	Z	1	‘ ‘	48

Plaintext #4 Frequency table:

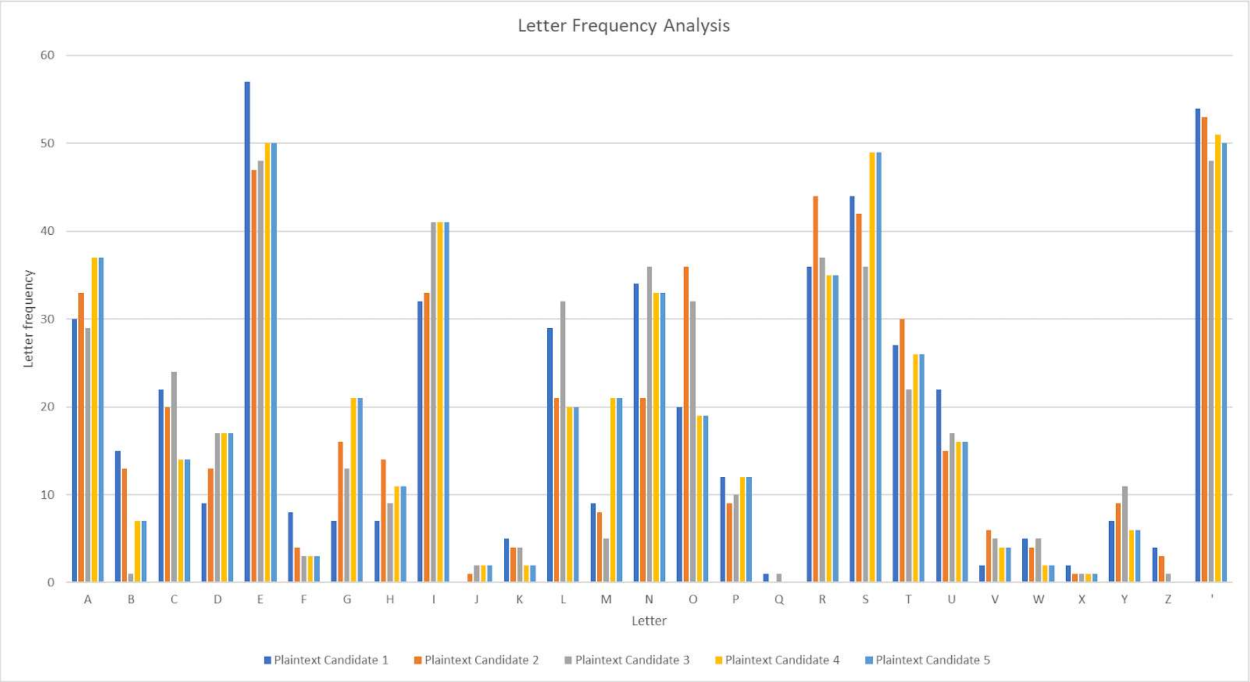
A	37	B	7	C	14
D	17	E	50	F	3
G	21	H	11	I	41
J	2	K	2	L	20
M	21	N	33	O	19
P	12	Q	0	R	35
S	49	T	26	U	16
V	4	W	2	X	1
Y	6	Z	0	‘ ‘	51

Plaintext #5 Frequency table:

A	37	B	7	C	14
D	17	E	50	F	3
G	21	H	11	I	41
J	2	K	2	L	20

M	**21**	**N**	**33**	**O**	**19**
P	**12**	**Q**	**0**	**R**	**35**
S	**49**	**T**	**26**	**U**	**16**
V	**4**	**W**	**2**	**X**	**1**
Y	**6**	**Z**	**0**	**‘ ‘**	**50**

When measuring the frequency appearances of each letter for all five plaintext candidates, it can be noted that, in terms of analyzing the entire plaintexts themselves, the overall distribution of letters are closely matching for two or more plaintext candidates. In addition, one letter will always be known because the last letter of the ciphertext will always be matched with the last letter of one of the five plaintext candidates. This is because if a random letter is generated, the encryption scheme would still continue for the same letter being analyzed, meaning that the last letter of the ciphertext will never be a randomly generated letter.



Upon analyzing the letter distribution for each plaintext candidate, we came upon two possible methods to decrypt a ciphertext:

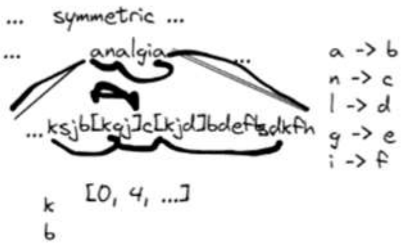
Decryption Methods

- Decrypt the ciphertext by deciphering the encryption key
  - Can be done by deciphering the key length and understanding what ciphertext letters were mapped out to which plaintext message.

    - Could greatly increase the accuracy of ciphertext decryption measures if successful.
    - Would make it possible to guess which letters were randomly generated and which letters were part of the input.
    - However, this would be substantially more difficult to execute properly due to the addition of random noise generated during the encryption process.
    - Higher probability of randomly generated letters makes it exponentially more difficult to guess the encryption key used which becomes more difficult as the probability of random generation increases.
    - As a result, the accuracy of decrypting the ciphertext would depend heavily on the accuracy of the key used, making this method unfeasible.
- Decrypt the ciphertext based on distribution analysis and pattern recognition
    - This method works based on the plaintext attack method.
    - For each keyword found in the ciphertext, the corresponding plaintext from each candidate is analyzed by each letter based on a matching pattern.
    - This method was more flexible to noise generation as opposed to the previous method.
    - This method was later chosen as the method used for the decryption scheme.
- Standard Deviation Matching
    - Similar to pattern recognition but we use any matches we've found to build a linear regression model showing the relationship of keywords between the ciphertext and plaintext based on letter frequency
    - At the same time, calculate the standard deviation the letter frequencies have as the skew for the linear regression model. This skew will be used to predict the number of randomly generated letters found in the ciphertext.
    - Determine the best-fitting match based on which letter has the smallest skew calculated
    - Afterwards, calculate the displacement found in the ciphertext crested from the concatenation of randomly generated letters
- Percentile Matching (percentile_matching.py)
    - The ciphertext has a probability of being longer than the plaintext.
    - However, in this case we assume the coin generation algorithm is normally distributed. This means that we can guess the letters where the key was used and not randomly added.
    - We would compare the positions of certain characters in the plain text and ciphertext, use the Chi-squared statistic to determine if the current position we are comparing in the ciphertext is close to the position in the plain text.
    - The closest one will be counted in the statistic and sum the Chi-squared result from all the possible characters that could be from the plaintext.
    - The lowest summation would be recorded and will tell us the possible plain text the ciphertext derived from.

1. solving for the key
    - metrics don't scale with noise
2. directly find patterns in the ciphertext that show up in the plaintext

1-(len(plaintext)/len(ciphertext))



Pattern-Recognition based decryption

The decryption method through pattern-recognition meant that each ciphertext output was matched with its plaintext input from all five candidates to see which corresponding plaintext word had the strongest match.

For example:

1	2	3	4	5	6	7
A	N	A	L	G	I	A

- Excluding the usage of noise generation, if the word Analgia was to be encrypted:
    - There would be 3 letters, two of which are located at the beginning and end of the word
    - The letter ‘A’ is repeated 3 times while every other letter is distinct.
    - There would be four additional, distinct letters, each of which that are distinct from one another, but would be placed in an order such that *N* is placed between two A’s. L succeeds after A, G is placed after L, etc.

This decryption method is effective for a standard monoalphabetic substitution cipher; however, it becomes unreliable once random letter concatenation is introduced. Because random letters can be inserted into the ciphertext, this shifts the rest of the ciphertext messages to the right based on the number of random letters generated. In addition, because white spaces can be inserted into the message or substituted as well, this means that the only method of matching each ciphertext word with its plaintext counterpart would be by matching and comparing the range of where the plaintext could be located at.

As such, the possible range of letters being searched within the plaintext candidate’s substring had to include a margin of error based on the number of concatenated letters. This amount could be found because for Dictionary_1.txt, every plaintext candidate was set to a fixed value of 500 characters while the ciphertext must be atleast 500 characters or more. The margin of error was calculated as 1/(Ciphertext.length-plaintext.length).

In this method of decryption, the substring of the ciphertext would be obtained and analyzed for any matching patterns. The substring was obtained through a sliding window whose range depended on the size of a keyword in the plaintext along with the margin of error calculated. For each iteration of the sliding window, the substring of the ciphertext was taken and matched with the plaintext candidates in order to check to see which plaintext candidate has a more accurate match to the ciphertext based on the placement of letters and frequency of letters for that particular match.

Standard Deviation Matching

Similar to pattern matching, this method works based off matching searching works based off the position of each keyword in the plaintext relative to their predicted position in the ciphertext.

However, this method differs by putting more focus on the scaling displacement created by the concatenation of random letters. While searching and matching for keywords for each plaintext candidate, the displacement for each keyword is calculated and computed as part of a regression model. This regression model creates multiple slices of the ciphertext and checks how far these keywords were displaced from their original position. This displacement is then taken and used to calculate the skew for the regression model. This skew was calculated using a built-in skew function from Python's libraries.

In addition to calculating skew, the algorithm would also keep track of the letter frequency for each matching keywords. This is because the encryption scheme either inserts an additional letter or substitutes every particular letter with its matching key, meaning that letters are always added to the ciphertext, never taken out of. This means that for every plaintext letter, the matching encrypted variant of that letter will either match the frequency of its plaintext counterpart or have more due to the concatenation of an extra letter.

As a result, the decryption algorithm will keep track of the frequencies for the keywords found and check if the hypothesized matching letter has a frequencies equal to or more than the frequencies of its plaintext counterpart, allowing the algorithm to dismiss possible guesses that do not fit this requirement. The decryption algorithm takes advantage of this feature by matching looking at each keyword in the plaintext candidates and seeing if within a sliding window's range that there are a particular set of letters that match or exceed the amount of appearances found in the ciphertext within a relatively matching position. These letters allow the algorithm to check the # of appearances shown in the ciphertext and their position relative to the plaintext message to determine if there was a random letter possibly concatenated within that range. This allows the algorithm to formulate a standard deviation for each letter where the algorithm will make a match based on which letter has the smallest standard deviation from its plaintext counterpart.

Once the skew was calculated, this was used to provide a rating for each plaintext candidate. After a keyword was found and had its skew calculated, the decryption algorithm would continue with each string from the plaintext candidate, searching for more keywords as the sliding window moved rightward.

The algorithm repeats this process of calculating its regression mode, the skew from the displacement observed, and observes and matches the frequencies of every substring until it has fully read and analyzed all five plaintext candidates. Once it has finished reading the five candidates, the linear regression model will be able to use the skew generated from successive attempts at calculating displacement as a way to make the final best possible guess for which plaintext candidate is the correct input.

Percentile Matching/Chi-Square Statistic (Scrapped)

When developing on the C++ version, the first attempted method was to determine the best possible match for each letter based on how often they appear in the ciphertext compared to the plaintext. Using the frequencies from the set of letters found from both strings of the plaintext and ciphertext, they were analyzed as to what best possible matches each set of letters could be.

One attempted method was to analyze the entire string of each plaintext candidate and compare them to the ciphertext whereupon they would each be calculated into a percentile distribution of letters. This method was attempted because an analysis of the letter frequencies showed that, while the inclusion of noise generation makes it difficult to match based on the specific frequency of each letter, the percentile distribution remains consistent despite the noise generation.

While the noise generation is unpredictable and volatile, its impact on the letter frequency for all letters should be small in comparison to the total number of appearances each letter makes. Percentile distribution was an attempt to match each letter from the ciphertext to a plaintext based on which set of letters had the closest match in terms of percentiles. Unfortunately, percentile distribution wasn't able to be used because if two plaintext letter or ciphertext letters had the same frequency of distribution, then there was no effective tie-breaker solution for either identical set.

The Chi-Square statistic is a method used to calculate the probability of distribution from one set of data to the distribution of that same variable in another set of data. In the context of the monoalphabetic substitution cipher, the Chi-Square statistic could be used to identify which subset of letters from the ciphertext could match the set of letters from the plaintext based on how statistically close they are.

Letters with matching frequencies would be identified by having a Chi-Square statistic of 0 which increases as the difference in frequencies increase as shown by the equation below.

$$X^2(C,E) = \sum_{i=a}^{i=" "} \frac{(C_i - E_i)^2}{E_i}$$

```
for (i = a['a'] to i = a[' '])
{
sum += (# frequency in frequency_map[i] - frequency of frequency_PT[i])^2/
(# of frequency in frequency_map[i])
}
```

- Where C represents the Ciphertext
  - C_i represents the frequency of a particular letter found in ciphertext C
- E represents the plaintext
  - E_i represents the frequency of a particular letter found in plaintext E
- Where i represents the set of all letters from a to whitespace.

The Chi-Square statistic was an attempt to conduct a basic level of analysis and matching for the frequencies of all letters. Unfortunately, the nature of noise generation meant that the distribution of letters were not accurately represented for the ciphertext which would muddle the letter frequencies found for each ciphertext input. As such, using Chi-Square statistics was unfortunately not feasible due to being unable to solve tie-breaker situations where it could not be accurately determined if two letters having matching frequencies were influenced by random letter concatenation.

Percentile Matching

We first take all the plain texts and map the relative positions of each letter into a dictionary. This dictionary will be used to compare with the relative positions in the ciphertext.

The coin_generation_algorithm was assumed to be normally distributed in order for this method. When using a normal distribution, the addition of random letters would not affect the relative position of letters in a ciphertext where no random letters were added. For example, if the letter ‘u’ was in the 60th percentile of the plaintext, the mapped letter of ‘u’ using the encryption key on the ciphertext would likely be at the 60th percentile of the ciphertext with normally distributed letters being added in the ciphertext. Using this knowledge, we can try to guess the plaintext by comparing the percentiles of each letter. However, this would be difficult since we don’t have the key. One solution to this is to derive a possible key. A key was generated by comparing frequencies of the letters in the ciphertext with the frequencies of letters in the English language. However, there is a limitation to this technique as we are only limited to the letters in the ciphertext which gives us a small sample size to work with.

An approximate key was generated and used to decipher the whole ciphertext. Even though the random letters would be deciphered, it would not be an issue since it will not be used. The relative positions in the deciphered text were calculated. Now that there are the plaintext's relative positions and the deciphered text's relative positions, we can compute the Chi-squared statistic to see how far off the possible letters are. Each letter and its relative positions from the plain text were compared to the same letter and its relative positions in the deciphered text. However, the deciphered text could be longer than the plaintext. To combat this issue, the comparison is compared with the plaintext. The expected value is from the plaintext's relative position and the observed value would be the deciphered text's relative position. The position with the smallest Chi-squared statistic is used and added to a total Chi-squared for that plaintext letter comparison at that relative position. The smallest total Chi-squared for a plaintext is considered to be the guess.

Sources:

https://mathweb.ucsd.edu/~crypto/java/EARLYCIPHERS/Monoalphabetic.html#:~:text=To%20break%20a%20monoalphabetic%20substitution,letters%20with%20the%20same%20pattern

https://mathweb.ucsd.edu/~crypto/java/EARLYCIPHERS/breakmono.html

http://practicalcryptography.com/cryptanalysis/text-characterisation/chi-squared-statistic/#:~:text=The%20Chi%2Dsquared%20Statistic%20is,some%20higher%20number%20will%20result

https://www.tapatalk.com/groups/crypto/the-index-of-coincidence-the-chi-test-the-kappa-t238.html