Cryptanalysis of A Class of Ciphers Based on Linear Regression, Standard Deviation, and Chi-Squared Statistic

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.

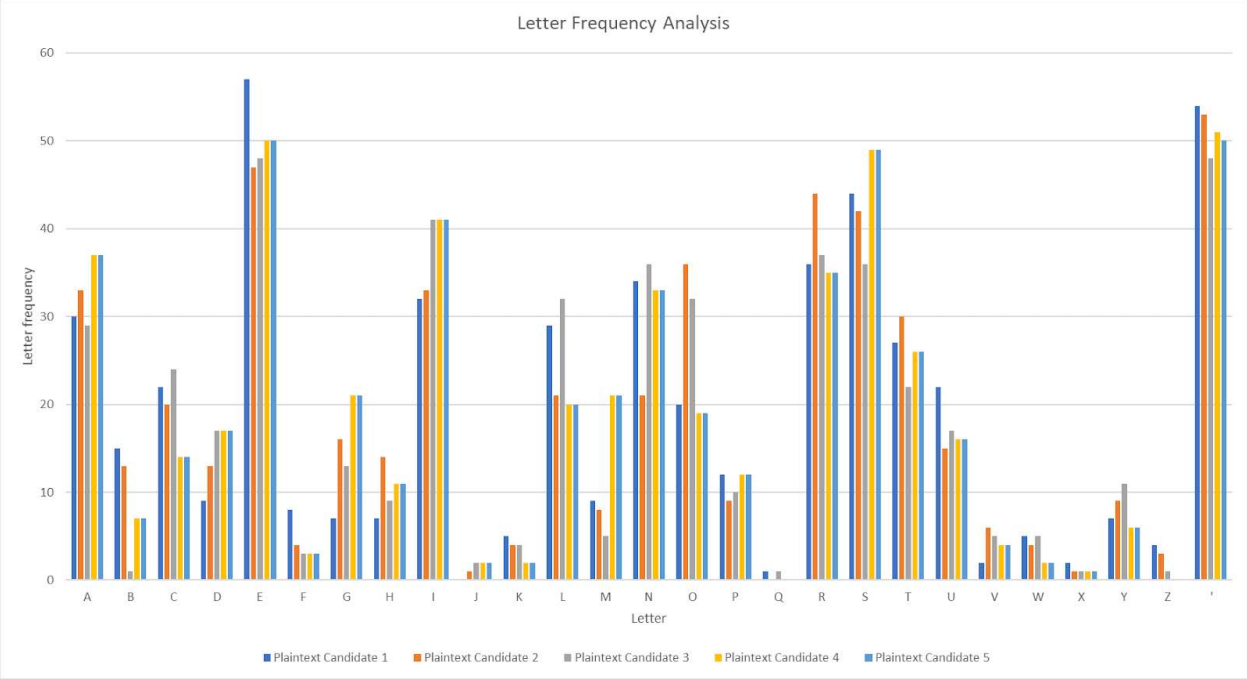
Decryption Schemes:

In the case where there is no noise added in the ciphertext, one approach to determining the plaintext that was encrypted is to analyze the letter frequencies. For example, the letter "e" is the most common letter in the English language, therefore, if there is a substantially large ciphertext with no noise added, one can make an educated guess that the most common letter in the ciphertext maps to "e". This partially elucidates the key that was used to encrypt the message. Interestingly, we also know that the last letter in the ciphertext must be deterministically mapped to the last letter in the plaintext, since the encryption algorithm ends when it maps the last letter. This can also provide some insight into what key was used to perform the encryption.

These ideas led us towards the path of analyzing letter frequencies, however, we noticed that many of the plaintexts shared similar frequencies, making it difficult to distinguish which plaintext the ciphertext derived from based on the frequencies of the letters alone.

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.

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 (CrightonDasariVo-percentile_matching-decrypt-source.py) - Pseudocode
    - 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.

Statistical Pattern Matching

This method works by looking for statistical similarities between certain subsections of the plaintext and the ciphertext. In the case of no random characters, the position of a character in the plaintext has the same position in the ciphertext. If we isolate a certain part of the plaintext that has interesting statistical patterns, we can expect to see those same statistical patterns in the ciphertext. Now, we must define what statistical patterns are interesting. We will start by describing the logic for a scenario with no random characters and later introduce a generalization that will expand this method for randomness.

Certain words can have interesting and unique patterns based on the dispersion of certain letters. For example, the word “analgia” has the same letter, the letter ‘a’, appear three times. Once in the beginning, middle, and very end. We can compute the dispersion of the letter ‘a’ by computing the standard deviation of the normalized indices of the letter ‘a’ in “analgia”. Normalized indices are used to scale the indices between 0 and 1. For example, the indices of ‘a’ in “analgia” are [0, 2, 6]. Normalizing this results in [0, 0.2, 0.8]. Taking the standard deviation of this new array, we get a distinct value that describes the dispersion of a particular letter in a particular word.

Now, assume that we have encrypted a plaintext that contains the word analgia. Since we know the different plaintexts that were used, we can identify the position of the word “analgia” in the plaintext. Then we can identify where we expect to find the word in the ciphertext. Since there is no randomness, we can assume that the encrypted word appears at the same index in the ciphertext as it appears in the plaintext. We can then look in that window for a letter that appears at least 3 times. If we find such a letter, we can compute the standard deviation of the normalized indices in that subsection of the ciphertext. In the case of no randomness, we expect the standard deviations in the ciphertext slice and plaintext slice to be equivalent.

Given the conditions of the problem, we don’t actually know if “analgia” is in the ciphertext, but since we know the subsection of the ciphertext where we expect to find it and the statistical properties that subsection has, we can make an educated guess as to whether or not the word “analgia” actually appears in the ciphertext. If the word does appear, then we can say that the ciphertext likely belongs to the plaintext that also contains the word “analgia”.

So far we have discussed a single word from a single plaintext. When we have multiple plaintexts, we can identify words in the plaintexts that have at least one character repeated three times. We can also note down their positions in the plaintext and their statistical properties, as described above. When we want to find out which plaintext the ciphertext is derived from, all we have to do is identify the plaintext that contains the closest statistical match across the words identified based on the conditions above.

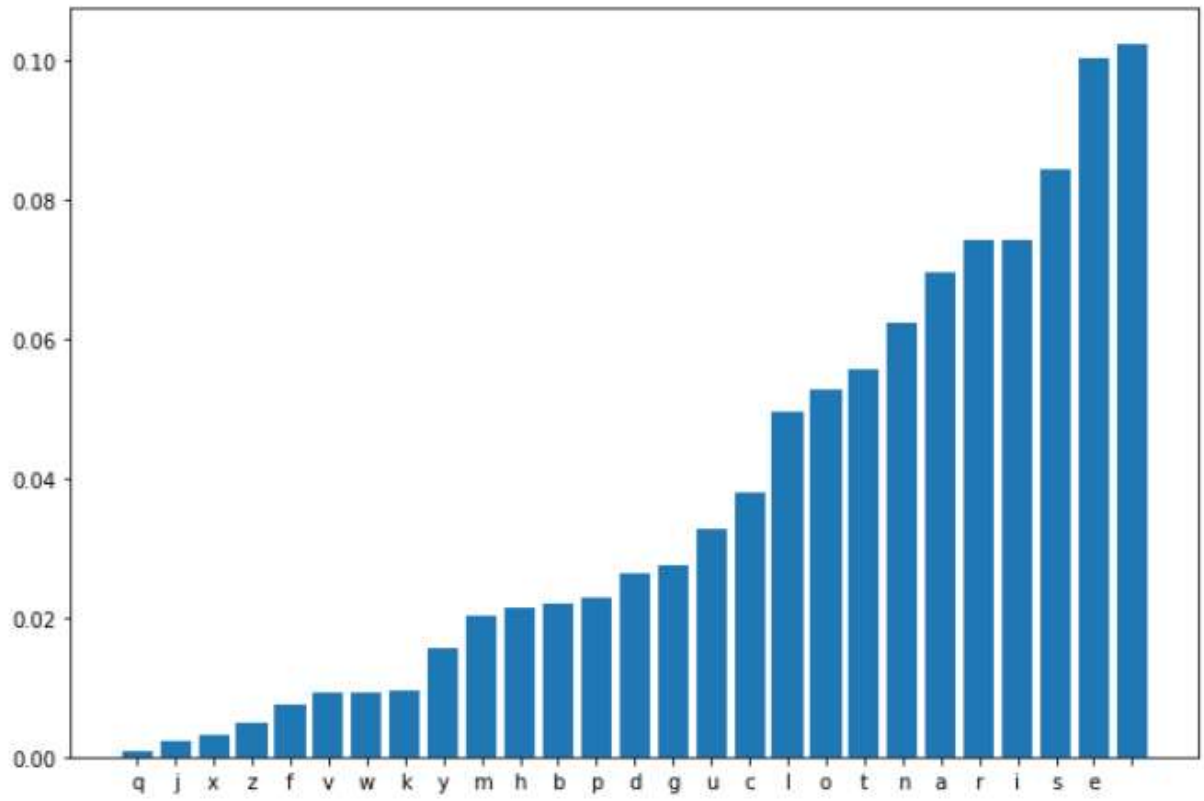
This adequately describes the decryption scheme when there is no randomness in the encryption algorithm. Since the positions of the words we identify in the plaintext are the same as the positions of the words in the ciphertext, we know exactly where the words should show

up in the ciphertext. As we introduce randomness, however, we introduce the ability for encrypted characters in the ciphertext to not have the same index as they had in the plaintext. This introduces the question of how we can identify where we expect to find a word in the ciphertext when random characters can be added.

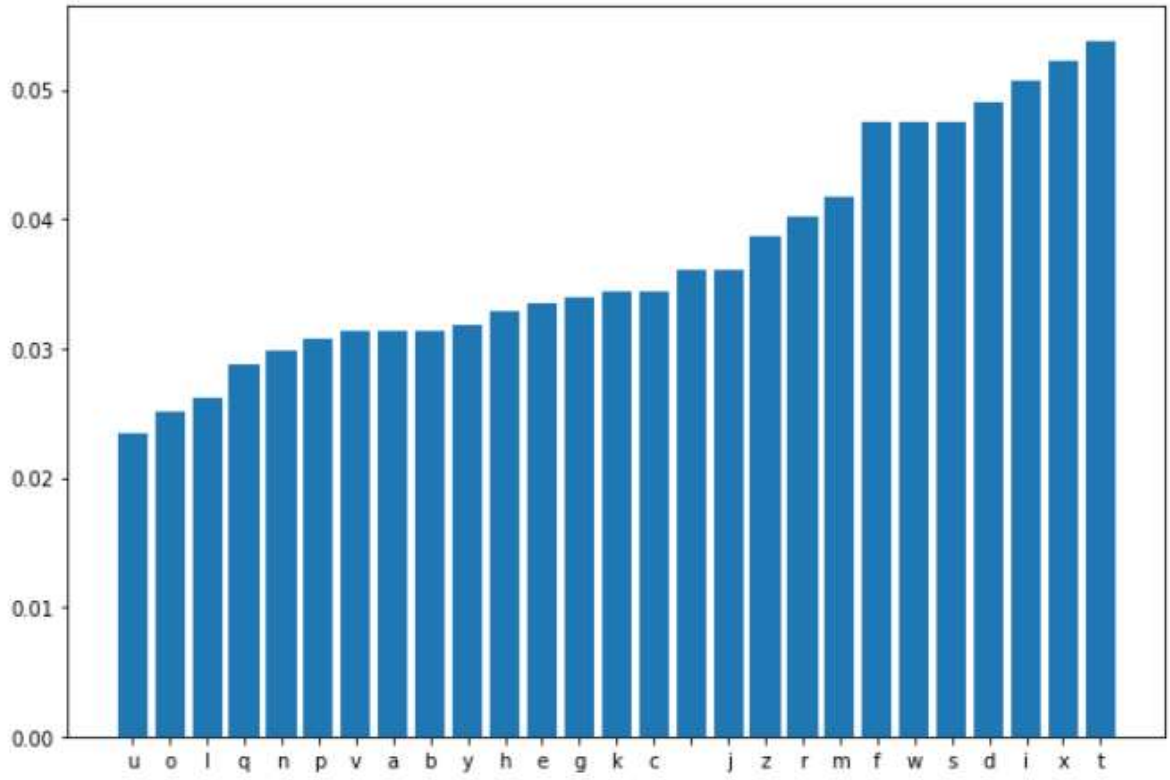
It turns out that the position of a determiniscally encrypted character in the ciphertext is related to the probability of adding a random character and the position of that character in the plaintext. We can exploit this information and train a linear regression model based on the combination of these features to predict where the expected position of a plaintext character is in the ciphertext.

Using this new linear regression model, we can pinpoint where we expect a word in the plaintext to appear in the ciphertext and determine whether or not the ciphertext and plaintexts are statistical matches, based on the standard deviation of letters that appear at least three times in each text, respectively.

But now, we have introduced a new problem. We are not allowed to know the probability of adding a random character that was used to encrypt the plaintext when we attempt to decrypt it. However, this is closely related to another statistical feature in the distribution of the characters. Shown below is a rough estimate of the frequencies of the characters in the english language:



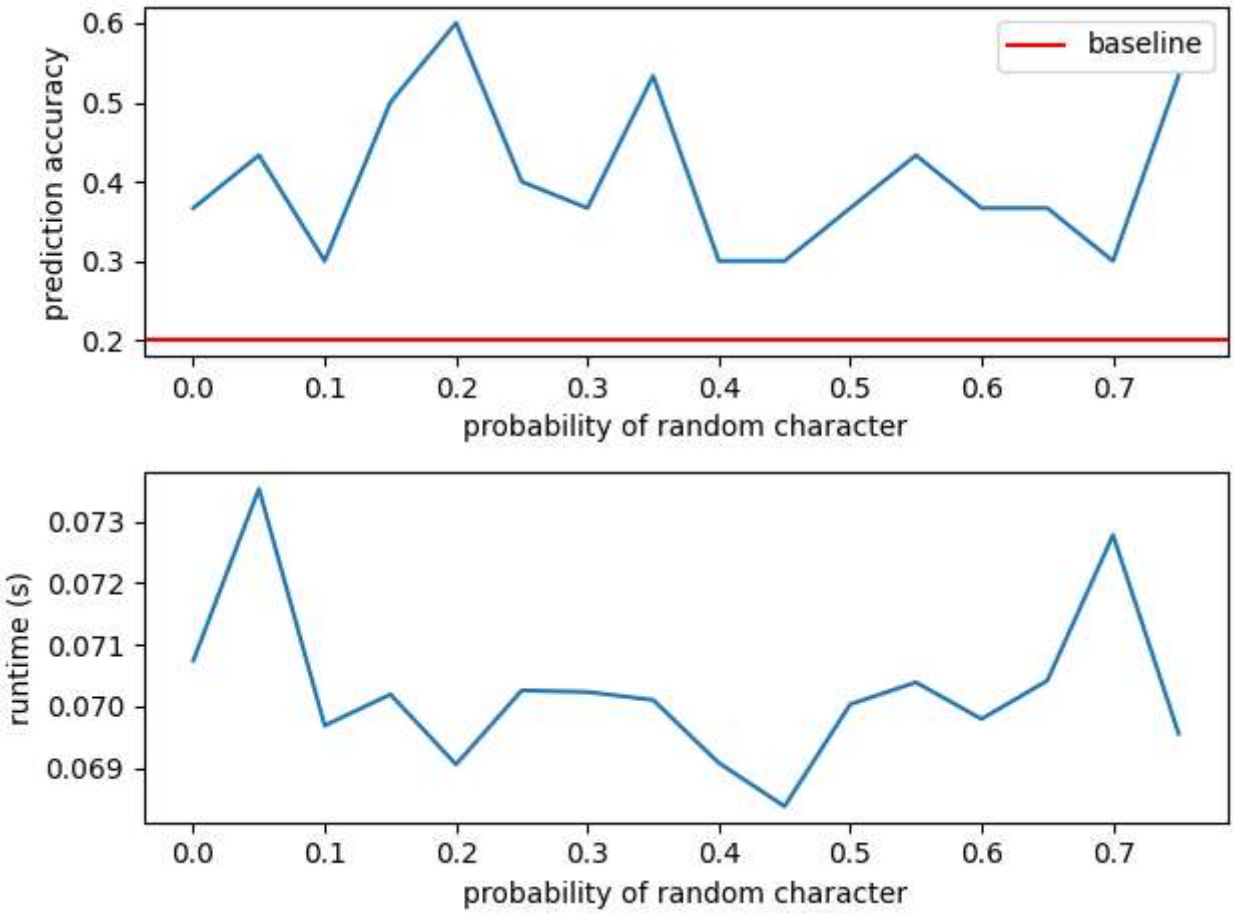
Here, we see that the distribution, once sorted in ascending order, is distinctly skewed to the right. Now, let's look at the frequency distribution of letters in a ciphertext that was encrypted with a probability of random characters of 0.75:



Here, we see that the distribution is much less asymmetric and approaches a normal distribution. The degree of this asymmetry can be measured using a statistical concept called the skew. The skew of a frequency distribution is directly proportional to the probability of randomness, making this a good feature to use in the linear regression, in place of the probability of adding a random character. The skew is calculated using a built-in function from Scipy, a common scientific computing Python library.

Once all the statistical properties of every possible subsection of the ciphertext is computed, we can identify which plaintext the ciphertext belongs to by selecting the plaintext that contains the closest number of matching statistical subsections.

The following is a performance profile of this method at varying levels of randomness used during encryption. The levels of randomness are sampled from a discrete list of values from 0 to 0.75 in steps of 0.05. 30 trials are performed for each discrete value.



Here, we see that the prediction accuracy is consistently above the baseline of 0.2, which is the expected performance if one randomly guesses the plaintext, However, we do not see a noticeable difference in performance as the probability of random character increases. One would expect that with less randomness, the decryption should be easier, however that doesn't seem to be the case. We also see that the runtime also remains relatively consistent across the trials as well.

A outline of the algorithm is provided here:

1. Use the encryption algorithm to train a linear regression model using the skew and character positions to predict new character positions on the ciphertext. (Note: During the actual evaluation, we are not allowed access to the encryption algorithm. This step is done once before the evaluation from generated training data.)
2. Identify interesting words (words that contain at least 3 of the same character) across all the plaintexts
    a. interesting _words = []
    b. For each plaintext in plaintext dictionary::
        i. For each word in plaintext dictionary:

            1. If word contains 1 letter that appears 3 times:
                a. interesting_words.append((word, letter, position in ciphertext, associated plaintext ID))
3. Compute statistical similarity based on dispersion of letters in each previously identified word
    a. Plaintext_scores = {1: 0, 2: 0, 3: 0, 4: 0, 5: 0}
    b. For each word in interesting_words:
        i. Compute relative positions of letter that appears at least 3 times
        ii. Plaintext_std = Compute standard deviation of these relative positions
        iii. Get start and end indices of plaintext word
        iv. Compute skew of frequency distribution of ciphertext
        v. Cipher_start, cipher_end = model.predict(skew, start/end indices)
        vi. Cipher_slice = ciphertext[cipher_start:cipher_end]
        vii. Cipher_std = Compute statistical properties of cipher_slice
        viii. Plaintext_scores[word.plaintext_id] += abs(cipher_std - plaintext_std)
4. Check which plaintext has the lowest statistical difference between its associated interesting words and the ciphertext:
    a. Prediction = None
    b. minScore = 999
    c. For each plaintext, score in plaintext_scores.items():
        i. If score < minScore:
            1. minScore = score
            2. Prediction = plaintext
5. Return prediction

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