Cryptanalysis of a class of ciphers based on character frequency, grouping, and Levenshtein distance

Guandi Wang & Daniel DiPietrantonio

March 20, 2022

1 Introduction

This project was completed by Guandi Wang and Daniel DiPietrantonio. Guandi worked heavily on our submission for test 1, and Daniel made minor edits and test scripts to optimize his work. Both team members worked on solutions for test 2, however Guandi's approach proved to be more successful and thus that is what we submitted. Strategies were formulated and devised by both team members. The report was written by both team members.

We are submitting five files for this assignment: the report file, test1 implementation source, test1 implementation binary, test 2 implementation source, and test2 implementation binary. As indicated by the labels, we took a different approach for solving test 1 and test 2. The approach for test 1 was successful, showing a minimal error rate even as the probability of randomness rises. Some errors can be found once the probability approaches 0.5, however it is not unlikely to have successful results even with a probability set to 0.75. The approach for test 2 was successful as well, but less so than test 1. The approach manages to find most of the words with 0 probability of randomness, but struggles as the probability of randomness is raised.

2 Informal description of approach

Since we took separate approaches for test 1 and test 2, this section is divided into two subsections. The first talks about our approach, and the second talks about our approach for test 2. Throughout both susections, we talk heavily about character distributions. Included here are plots that show the character distributions of the five plaintexts in dictionary 1 as well as the entire vocabulary in dictionary 2.

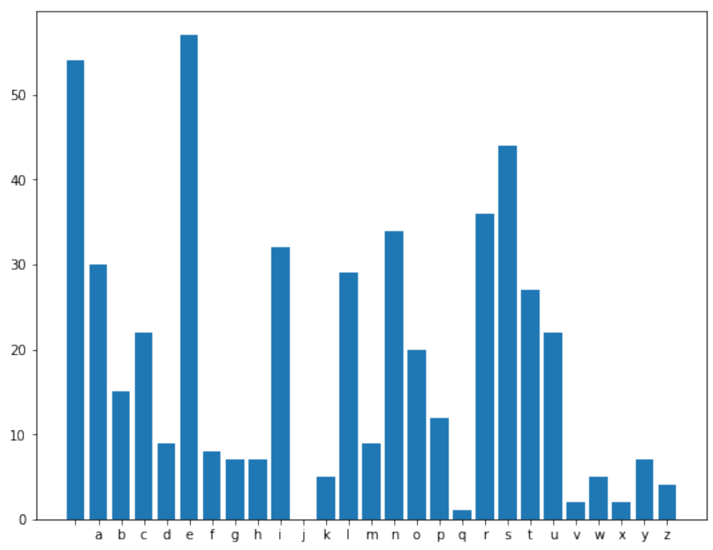

Figure 1: Dictionary 1 - Plaintext 1 Character Frequency

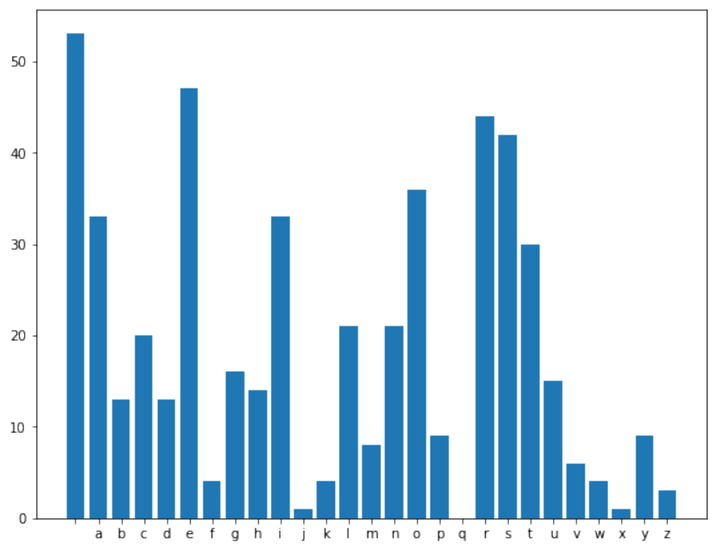

Figure 2: Dictionary 1 - Plaintext 2 Character Frequency

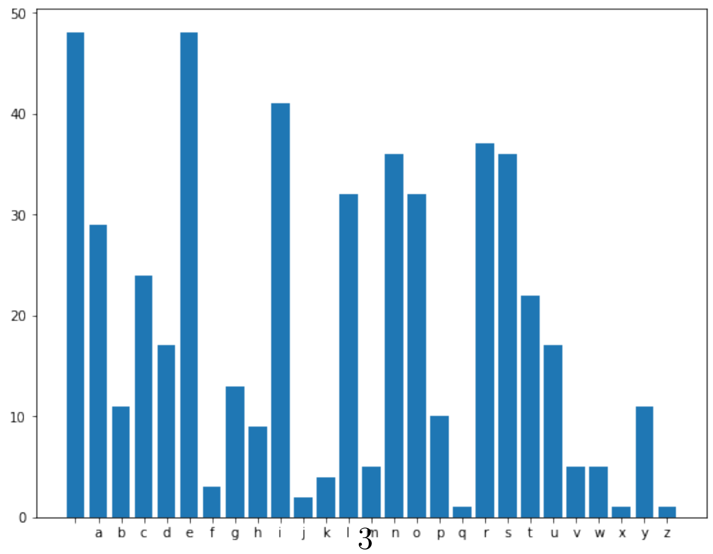

Figure 3: Dictionary 1 - Plaintext 3 Character Frequency

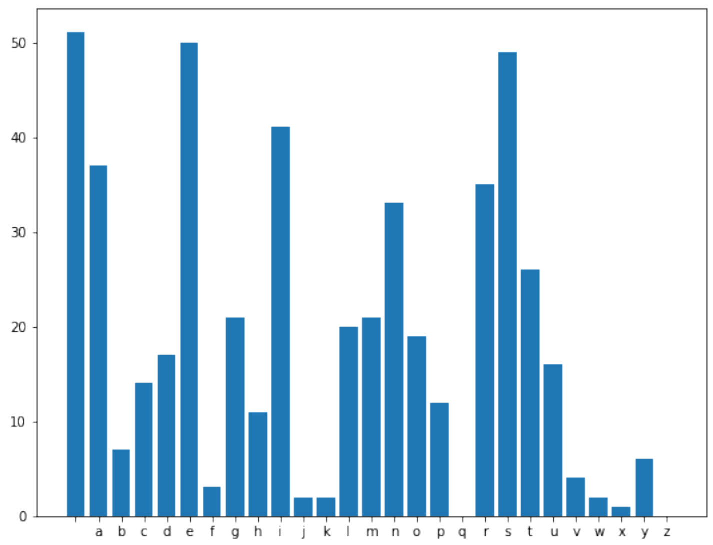

Figure 4: Dictionary 1 - Plaintext 4 Character Frequency

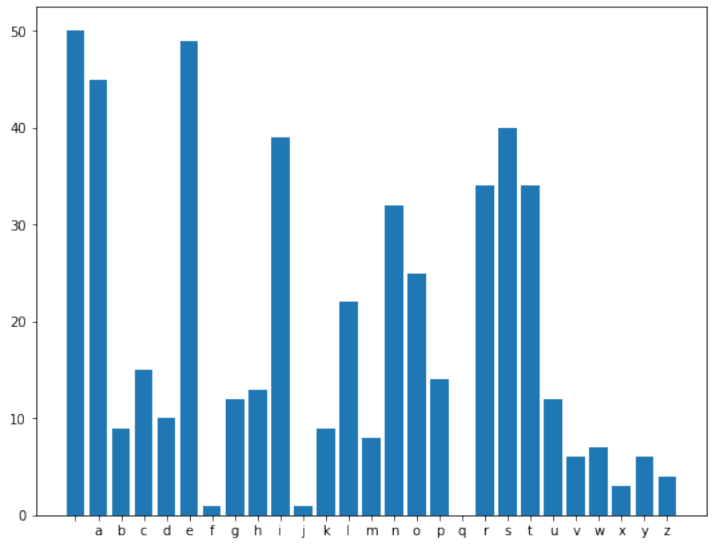

Figure 5: Dictionary 1 - Plaintext 5 Character Frequency

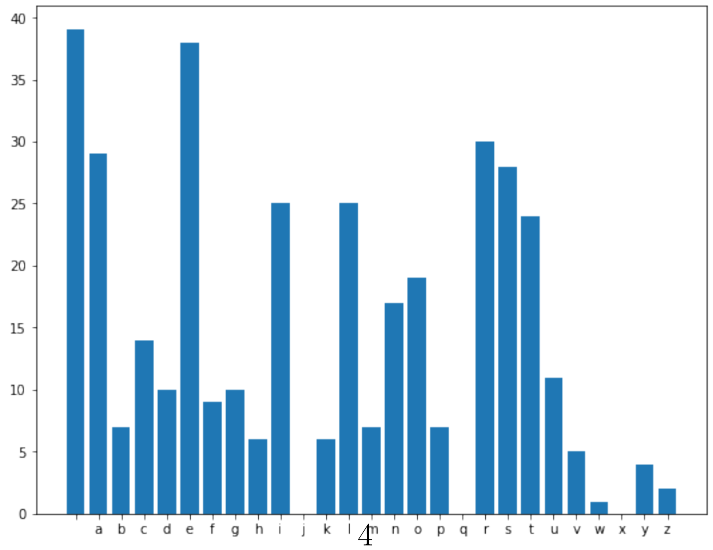

Figure 6: Dictionary 2 Character Frequency

2.1 Approach 1 - Test 1

At a high level, our approach to test 1 was the following:

1. Get the character frequency distribution of the encrypted input text.
2. Map characters with similar frequencies to the same symbol (grouping stage).
3. For each character in the input, change that character to the symbol it maps to. We refer to this new symbol-substituted input as the *symbolized input*.
4. Repeat the above three steps for each candidate plaintext in dictionary 1, creating five symbolized candidate plaintexts.
5. For each symbolized candidate plaintext, compute the Levenshtein distance (wiki) between it and the symbolized input.
6. Output the answer as the candidate plaintext with the minimum Levenshtein distance as computed in step 5.

With a high level understanding of the approach, we can now go into more detail on our implementation. Our approach for test 1 was based on the character frequency distribution of the potential input strings. First, we performed a character frequency analysis of each character in the encrypted input string. This is done by iterating through the input one character at a time, and keeping track of how many times we encountered that counter via a hash map.

After this, we perform a "grouping" step. The inspiration behind this step is the following observation: as more and more random characters are inserted into the encrypted text, the encrypted text's character distribution will become more and more uniform. Therefore, it is not enough to simply match the most commonly occuring character in the encrypted input to the most commonly occuring character in each of the five plaintexts. To fix this, the grouping step matches characters that have similar frequencies to the same symbol. For example, Figure 1 shows that 'g' and 'h' occur with very simliar frequency in plaintext 1 of dictionary 1. In the grouping step, they would be mapped to the same symbol, α. A hashmap data structure holds the mapping between a character in the input text and its symbol. For our implementation, we found that a group size of two works best. To make the groups, we sort the characters by frequency, and group them in pairs of two, from least frequent to most frequent. Since there are an odd number of characters in our input alphabet (26 letters and space), the most frequently observed character is mapped to its own symbol.

Once we have our input symbol mapping, we create the *symbolized input*, which is simply a string S of length L (where L is the length of the input) where the ith character of S is equal to the symbolized mapping of the ith character of the input.

Upon computing our symbolized input string, we now repeat the character frequency analysis, grouping, and symbolized mapping stages (steps $1-3$ in the high level overview) for each of the five candidate plaintexts in dictionary 1, thus giving us one symbolized input and five symbolized candidate plaintexts.

For each symbolized candidate plaintext, we compute the Levenshtein distance between that symbolized candidate plaintext and the symbolized input string. There were many string comparison algorithms to choose from in this step, however we chose the Levenshtein distance because it outputs the minimum number of character changes between the two strings which is a great metric to compare encrypted text to plaintext. Once we have these five Levenshtein distances, we simply take the minimum of them and output the candidate plaintext whose symbolized plaintext generated that minimum value.

This process is unique in that it doesn't attempt to morph the encrypted text into the plaintext but instead attempts to morph each candidate plaintext into the encrypted text, using grouping to adjust for randomness that is introduced in the encryption scheme.

2.2 Approach 2 - Test 2

At a high level, our approach to test 2 was the following:

1. Store candidate vocabularies and combine candidate vocabularies into aggregated vocabulary sentence.
2. Get the character frequency distribution of the aggregated vocabulary sentence.
3. Map characters with similar frequencies to the same symbol (grouping stage).
4. For each character in the candidate vocabularies, change that character to the symbol it maps to. We refer to this new symbol-substituted vocabulary as the *symbolized vocabulary*. Store the results in an unordered map so that we can retrieve original vocabulary later.
5. Get the character frequency distribution of the encrypted text.
6. Decrypt the input with different character mapping and find the best-matching combination of vocabularies:
    (a) Iteratively choose one character in frequency distribution of input and map it to space in decrypted text.
    (b) Map remaining characters with similar frequencies to the same symbol (grouping stage).
    (c) For each character in the input, change that character to the symbol it maps to. We refer to this new symbol-substituted input as the *symbolized input*.
    (d) In each iteration, tokenize the sinput on space and find the best-matching word in vocabularies for each token according to Levenshtein distance between the token and the vocabularies.
    (e) Record the total Levenshtein distance converting to sentence and combined the best-matching vocabularies into a sentence.
    (f) Choose the sentence that has lowest total Levenshtein distance.
7. Output the answer as the sentence that has lowest total Levenshtein distance.

Our approach for test 2 was generally similar to our approach in test 1. However, the difference in problem setting requires us to modify some details to adapt.

To start with, we are not provided with plaintexts but provided with vocabularies that make up the plaintext. To learn about the distribution of characters of vocabularies, we combined the vocabularies into an aggregated sentence joined by space and performed character distribution analysis on the aggregated sentence. Then we remove the space from the distribution and group the characters with similar frequencies.

Once we have our vocabularies symbol mapping, we create *v symbolized vocabularies*, that we can refer to later.

For our encrypted input, we perform character distribution analysis and store a list of characters sorted by frequency. Since finding the spaces in plaintext is essential in test 2 and that it is difficult to correctly find the spaces from input, we iteratively assign one of the characters as space. Then, in each iteration, we group and label characters according to frequency as in test 1 and get a *symbolized input*. Then we tokenize the *symobolized input* on spaces and get a list of symbolized tokens. For each symbolized token, we compare it with *symbolized vocabularies* individually to find the best-matching vocabulary with least Levenshtein distance. We then record the total Levenshtein distance between symbolized tokens and its corresponding best-matching vocabularies and the sentence they created. After all iterations, the sentence with lowest Levenshtein distance is chosen as our output.

3 Formal description of approach

Much like the previous section, this section is divided into two subsections, one for each approach we took.

3.1 Approach 1 - Test 1

Note that we do not include the implementation of the LevenshteinDistance function as its implementation is trivial based on the Wikipedia article linked above.

Input: encryptedInput

```
1: list candidatePlaintexts = [candidates from dictionary 1]

2: /* Step 1: Get character frequency of input */
3: hashmap inputCharFreqMap = hashmap()
4: for char ∈ encryptedInput do
5:     inputCharFreqMap[char]++
6: end for
7: list inCharsByFreq = sort(input chars by frequency)

8: /* Step 2: Grouping stage */
9: hashmap inSymbolMap = hashmap()
10: for ($i = 0; i < 27; i{+}{+}$) do
11:     int curGroup = i / 2;
12:     char curGroupSymbol = 'a'+curGroup
```

```
    inSymbolMap[inCharsByFreq[i]] = curGroupSymbol
end for

/*Step 3: Get symbolized input*/
*string* symbolizedInput = string([inSymbolMap[c] for c in input])

/*Step 4: Get symbolized candidate plaintexts*/
*list* symbolizedCandidatePtxt = []
for text $\in$ candidatePlaintexts do
    *string* symbolizedText = /* Repeat the code above in steps 1-3 */ symbolizedCandidatePtxt.append(symbolizedText)
end for

/*Step 5: Get Levenshtein distance of all symbolized candidates with symbolized input*/
*int* minLDistance = $\infty$
*int* minCandPtxtIdx = $-1$
for string sCandPtxt $\in$ symbolizedCandidatePtxt do
    *int* LevDistance = LevenshteinDistance(sCandPtxt, symbolizedInput)
    if LevDistance ¡ minLDistance then
        minLDistance = LevDistance
        minCandPtxtIdx = idx(sCandPtxt)
    end if
end for

/*Step 6: Output the candidate */
Output candidatePlaintexts[minCandPtxtIdx]
```

3.2 Approach 2 - Test 2

Note that as above, we do not include the implementation of the LevenshteinDistance function.

Input: encryptedInput

```
/* Step 1: Store candidate vocabularies in aggregated sentence */
*string* candidateVocab = [words in dictionary 2]

/* Step 2: Get the character frequency distribution of candidateVocab */
*hashmap* candidateVocabFreqDist = hashmap()
for char $\in$ candidateVocab do
    inputCharFreqMap[char]++
end for

/* Step 3: Grouping stage */
*hashmap* candSymbolMap = hashmap()
for $(i = 0; i < 26; i{+}{+})$ do
    *int* curGroup = i / 2;
```

```
        char curGroupSymbol = ‘a‘+curGroup
        candSymbolMap[inCharsByFreq[i]] = curGroupSymbol
    end for

    /* Step 4: Create symbolized vocabulary */
    hashmap symbolizedWordsMap = hashmap()
    string symbolizedVocab = ””
    for word ∈ candidateVocab do
        string symbolizedWord = ””
        for char ∈ word do
            symbolizedWord += candSymbolMap[char]
        end for
        symbolizedVocab += symbolizedWord
        symbolizedWordsMap[symbolizedWord] = word
    end for

    /* Step 5: Cet character frequency distribution of input */
    hashmap inputFreqDist = hashmap()
    for char ∈ encryptedInput do
        inputFreqDist[char]++
    end for

    /* Step 6: Decrypt input to find combination of vocabularies */
    string curDecryptGuess = ””
    int minLDistance = $\infty$
    string minLDistanceText = ””
    for curSpaceGuess ∈ ”abcd...xyz” do
        for char ∈ encryptedInput do
            curDecryptGuess += candSymbolMap[char]
        end for
        curDecryptGuess.removeAll(curSpaceGuess)
        for word ∈ symbolizedVocab do
            int ldist = LevenshteinDistance(word, encryptedInput)
            if ldist < minLDistance then
                minLDistance = ldist
                minLDistanceText = curDecryptGuess
            end if
        end for
    end for

    /*Step 7: Output the best guess */
    Output minLDistanceText
```

4 Performance Analysis

This section analyze the performance of the approaches mentioned above.

4.1 Approach 1 - Test 1

To test on the performance, a script(testCorrectness.py) has been created to iteratively choosing the plaintext in dictionary_1, encrypt the plaintext, run the executable of Approach 1, compare the result with chosen plaintext.

4.1.1 Plaintext Selection

The script iteratively choose a plaintext from dictionary 1.

4.1.2 Plaintext Encryption

The script encrypt the plaintext according to encryption scheme provided by handouts.

4.1.3 Execution

The script runs the executable of Approach 1 and retrieve the results from stdout.

4.1.4 Comparison

The script compare the result with chosen plaintext and record the number of errors.

4.1.5 Repetition

Repeat 4.1.1 to 4.1.5 for different randomness p.

4.1.6 Results

We ran the above experiment 75 times. The first five tests had probability 0, the next five probability 0.05, then 0.1, increasing in increments of 0.05 until we reached 0.75. The results are captured in the table below. Of the 75 tests, 5 failed, indicating an approximately 94% success rate.

Randomness(p)	Tests Run	Successful Guesses
0.00	5	5
0.05	5	5
0.10	5	5
0.15	5	5
0.20	5	5
0.25	5	5
0.30	5	5
0.35	5	4
0.40	5	5
0.45	5	5
0.50	5	5
0.55	5	4
0.60	5	5
0.65	5	4
0.70	5	3
0.75	5	5

Table 1: Test 1 Performance

4.2 Approach 2 - Test 2

Since no plaintext is provided for Test 2, the performance of Approach 2 is assessed by the accuracy in finding the chosen words from dictionary in correct sequence. To test on the performance, a script(test2Correctness.py) has been created to generate plaintext, encrypt the plaintext, execute Approach 2 executable, and calculate the Levenshtein distance with plaintext.

4.2.1 Plaintext Generation

The script repeatedly and randomly choose a word from dictionary 2, join into a sentence with space, until the length of the generated sentence is larger than L. Then, the script splits the plaintext on space to generate a list of words (plaintextWords).

4.2.2 Plaintext Encryption

The encryption process is similar to the process in Test 1. Encrypt the plaintext with randomness p.

4.2.3 Execution

The script runs the executable of Approach 2 and retrieve the results from stdout.

4.2.4 Calculate Distance

The script splits the result on space to generate a list of words(decryptedWords). Calculate the Levenshtein distance between decryptedWords and plaintextWords.

4.2.5 Summary

Repeat 4.2.2 to 4.2.4 for couple hundreds of times. Calculate the mean and minimum Levenshtein distance.

4.2.6 Repetition

Repeat 4.2.1 to 4.2.5 for different randomness p.

4.2.7 Results

The results of running the experiment can be seen in the table below.

Randomness(p)	Average Distance in Words	Minimum Distance in Words	Num Words
0.00	10.66	6	53
0.05	14.54	5	53
0.10	19.02	3	53
0.15	25.98	8	53
0.20	33.65	10	53
0.25	37.48	15	53

Table 2: Test 2 Performance