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.

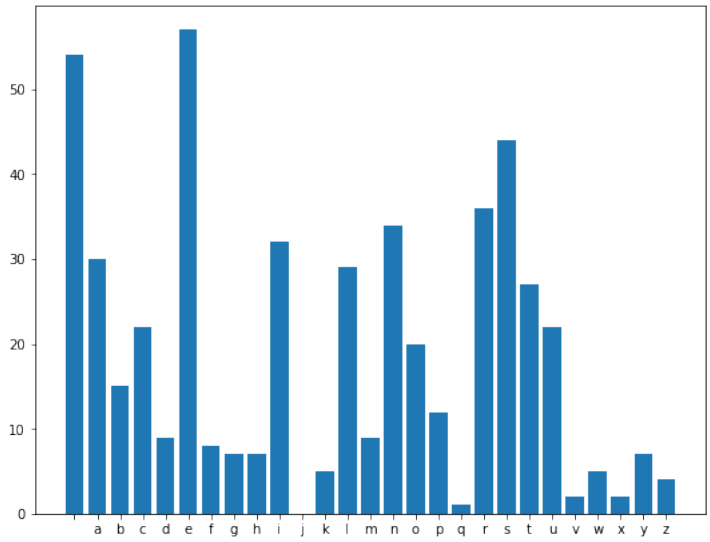

Figure 1: Dictionary 1 - Plaintext 1 Character Frequency

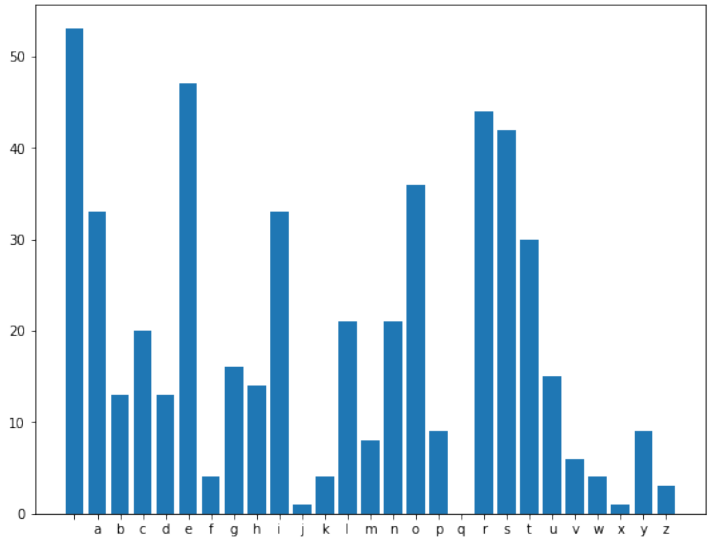

Figure 2: Dictionary 1 - Plaintext 2 Character Frequency

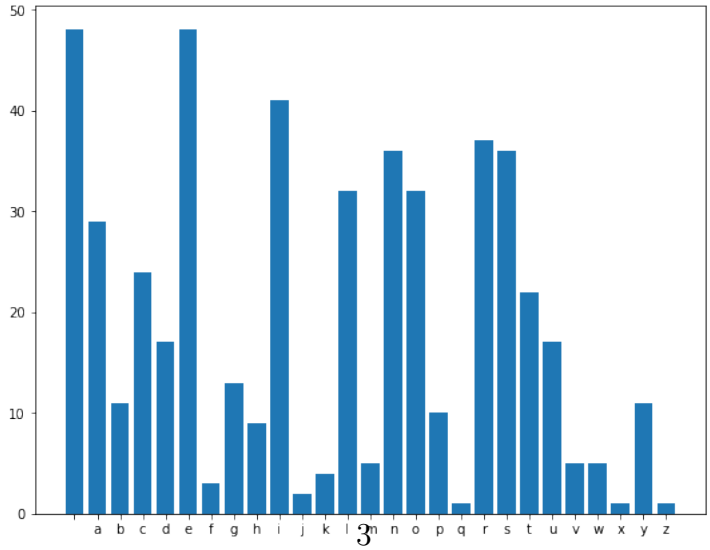

Figure 3: Dictionary 1 - Plaintext 3 Character Frequency

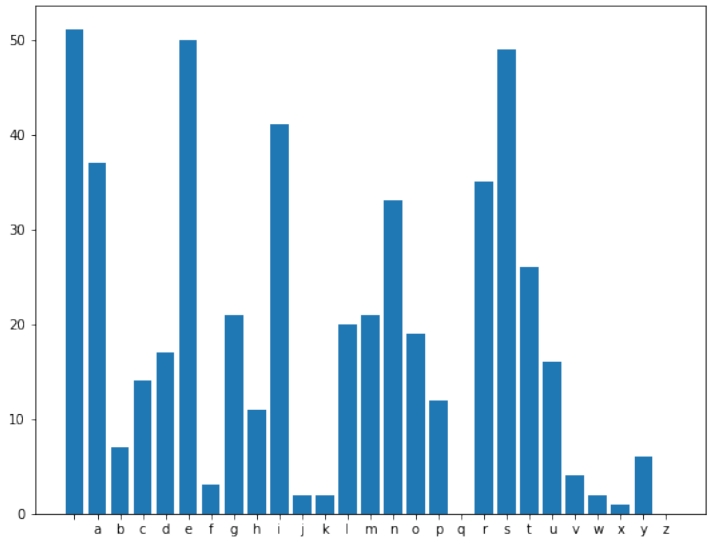

Figure 4: Dictionary 1 - Plaintext 4 Character Frequency

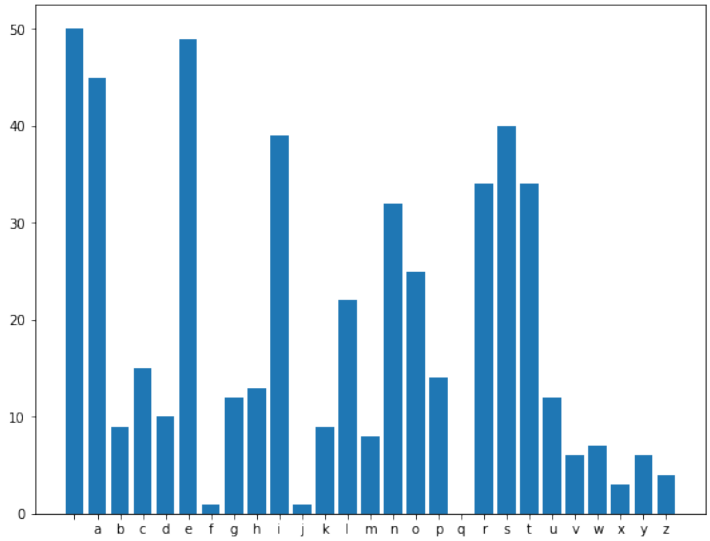

Figure 5: Dictionary 1 - Plaintext 5 Character Frequency

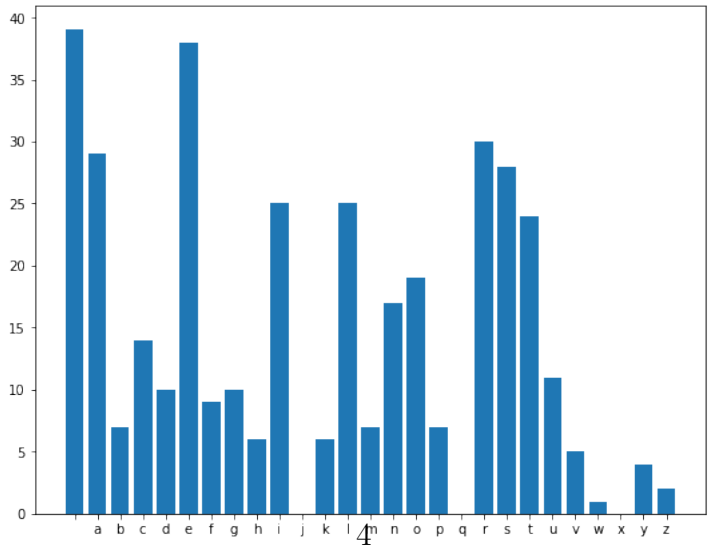

Figure 6: Dictionary 2 Character Frequency

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 **??** 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.

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

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

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

/* Step 2: Grouping stage */
hashmap inSymbolMap = hashmap()
for (i = 0; i < 27; i++) do
    int curGroup = i / 2;
    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 ∈ 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 = ∞
int minCandPtxtIdx = −1
for string sCandPtxt ∈ 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]
```