Cryptanalysis of a class of ciphers based on Letter Frequency

Introduction

This report demonstrates a cryptanalysis approach of breaking a variation of the mono-alphateic substitution cipher where characters are randomly inserted into the ciphertext. A high level description of the crypoanalyis approach is illustrated below, followed by a more detailed description of the technical code/details. Additionally, we explain an alternative approach for solving problem 1 that is theoretically faster, although it still requires more work to be ready for use. This project was worked on by Ahmed Khater, Leo Liu , and Samuel Ballan . All team members worked together to develop an understanding of the problem and develop possible solutions. Ahmed Khater created the encryption, decryption, and testing libraries that we used to develop the program. Leo Liu developed an approach to solving problem 1 that we ended up not using (but is included for reference), and worked on the portion of this report describing `decrypt1.` Samuel Ballan developed the algorithms and code used in the `decrypt1` and `decrypt2` procedures, and wrote the portion of this report describing `decrypt2`.

High Level Approach:
First, a description of the problem we are trying to solve: our program is provided a ciphertext. The encryption scheme used to produce the ciphertext is computationally indistinguishable from a monoalphabetic cipher that randomly and uniformly inserts random characters into its output with some known probability.

At the highest level, our program attempts to decrypt a ciphertext using a particular set of parameters, and then calculates a score for the attempt. The parameters are repeatedly changed, and the attempt with the highest score is kept and returned. More concretely, there are two main approaches used by the program, corresponding to the two dictionaries provided. For each dictionary, we produce a set of *noisytexts*, which are partial decryptions of the ciphertext. Under ideal conditions that we'll explain below, the *noisytext* is computationally indistinguishable from the plaintext message with random characters randomly and uniformly inserted into it with some probability. We can directly compare *noisytexts* to the entries in the first dictionary, and if we find a good enough match we return it. Otherwise, we use the second dictionary to generate our best possible guess, and return it. All comparisons are done using a Levenshtein distance, which is defined as the minimum number of edits required to change one string into the other. [1]

In the first approach, we try to match the ciphertext to one of the five messages in the first dictionary. Many different *histkeys* (described below) are applied to the ciphertext to create *noisytexts*. These are directly compared to entries in the first dictionary and a quality rating is calculated based on how close the *noisytexts* are to the plaintexts. We then use this quality score to decide whether or not we need to try again with our second approach.

The *HistKeyGen* is responsible for generating guesses (called *histkeys*) which are lists of characters sorted from most to least frequent. The first *histkey* will therefore start with the most frequent character in the second dictionary, and end with the least frequent character. As we iterate through the *HistKeyGen*, we will see *histkeys* that are progressively more and more different from the first *histkey*.

In the second approach, we try to match the ciphertext to a string of words that we generate using the second dictionary. To do this we (A) generate a *histkey*, (B) use the key to transform the ciphertext into a *noisytext*, and (C) find words from the second dictionary inside the *noisytext*.

Steps (A) and (B) are very similar to our first approach. We instantiate a *HistKeyGen* for the second dictionary by concatenating the words in the second dictionary into a space-separated string. This string is meant to approximate the distribution of characters and spaces in the ciphertext. For each *histkey* we produce, we use the character distribution from the ciphertext to produce a decryption key and a *noisytext*.

In the next step (C) we find matches between substrings of the *noisytext* and words in the second dictionary. A naive approach would split the *noisytext* by spaces, and compare each *noisytext* substring to the words in the second dictionary. This cannot work because space characters can be pseudo-randomly inserted just like any other character, and so the splits would not be valid. Instead, we do the following procedure:

1. We examine the characters up to the next space, find the best match in the second dictionary, and create a score for the match.
2. We combine the substring from step 1 with both the space following it and all the characters after that space up until the space after that. For instance, if the *noisytext* were “cryp tography”, we would first search our dictionary for “cryp” and then for “cryp tography”. We create a score for this substring too.
3. If the score of our second substring is higher than that of the first, we keep the second substring and repeat looking ahead for an even longer substring at step 2. Otherwise, we return the found substring as a match, save its score, and continue with a fresh substring at step 1.
4. When we get to the end of the *noisytext*, we concatenate all the found words together to create a guess at the plaintext, and we combine the scores together into an aggregate score of the whole guess.

Low Level

The program prompts the user for a ciphertext, and then prepares to process it with the `decrypt1` and `decrypt2` procedures which make use of `dictionary1` and `dictionary2` respectively.

Since the two approaches both use a common key guess generation algorithm, `HistKeyGen`, it will be described in detail before discussing specifics on either approach.

HistKeyGen

The *HistKeyGen* algorithm takes in a string of the reference text, and two whole number parameters for the *tolerance* and the *first_chunk_size*. It first calculates the frequency of each character in the reference text, and ranks them in descending order of frequencies. Then, chunks where the difference in frequencies between two characters adjacent in frequencies that vary by no more than *tolerance* number of occurrences are made. Lastly, we create permutations of these chunks and join them together to create likely keys for the monoalphabetic substitution. The *i-th* character of the key indicates the guess for the character encoded by the *i-th* frequent character in the ciphertext.

More specifically, the permutations are generated (and therefore fed into the decryption algorithm) in order such that the chunks with more frequently occurring characters are permuted first, before chucks with less frequently occurring characters. Together with *first_chunk_size*, which makes it so that the most frequent characters are always in a chunk, these frequent characters are almost always going to be correctly decrypted by some of the keys. Since these characters represent a very large portion of the ciphertext, having a key where these characters are decrypted correctly is very important. The space character, which is one such frequent character, must also be decrypted correctly by the key for problem 2, so that the spaces may be used as possible separators of words with random letters inserted.

HistKeyGen is implemented as a Python Iterator class, Python style pseudocode as follows:

```
class HistKeyGen:
    def __init__(reference_text, tolerance, first_chunk_size):
        chunks = create_chunks(reference_text, tolerance, first_chunk_size)
        chunk_permutations = list of lists containing permutations of a chunk
        # so chunk_permutations[i] contains permutations of the i-th chunk
        chunk_permutation_indices = [0] * number of chunks
        # Think of the indices as a little-endian number in a mixed radix numeral
        # system, where each numerical place corresponds to a chunk, and the place
        # value corresponds to which permutation of that chunk we're using.

    def create_chunks(reference_text, tolerance, first_chunk_size):
        calculate character frequencies
        sort characters by frequency, in descending order
        for char, next_char in sorted_frequencies:
            if difference in frequency(char, next_char) < tolerance:
                put char and next_char in the same chunk
            else:
                put next_char into a new chunk
        return list of chunks

    def __next__():
        # returns the next possible key
        if there are more keys:
            increment the chunk_permutation_indices mixed radix numeral by 1
            assemble the key represented by that number key
            return the key
        else:
            we've gone through all likely keys, stop the iterator
```

We also use the Levenshtein distance, which we import as a library.

`decrypt1` and `decrypt2` are both built with similar architecture as well, and make use of the `ray` library for multiprocessing. The overall architecture is as follows: keys are generated by `HistKeyGen`, and batches of keys of size `CHUNK_SIZE` are processed in parallel with the a function named `decryption_with_histkey()` (`decrypt1` and `decrypt2` both have their own unique versions of this function with the same name and overall functionality, but different internal logic) until we have tried `KEY_LIMIT` number of guesses of keys. Then, the message with the best `quality` rating by `decryption_with_histkey()` is chosen as the best guess for the original plaintext.

For `decrypt1`, its `decryption_with_histkey()` undoes the monoalphabetic substitution according to the given key, and then takes the Levenshtein distance between this string and one of the five candidate plaintexts in `dictionary1`. With a perfect monoalphabetic substitution key, this distance would be the number of random characters that were inserted during the encryption process. This distance is also expected to be smaller for closer guesses, of both key and message. A `quality` rating is calculated simply as the quotient of the distance and length of the ciphertext. A lower numerical value represents a better guess.

Different from decrypt2, decrypt1 finds the best key and its associated quality value for all five candidate plaintexts in dictionary1. The five minimum quality values are then checked against each other for outliers significantly smaller than the others, which indicates that the key (and therefore the plaintext) associated with that quality value is very likely the actual key (and plaintext). This plaintext is returned.

Decrypt1

```
def decryption_with_hist_key(message, ciphertext, histkey):
    find character frequencies of ciphertext
    generate substitution key from frequencies and histkey
    # undo the monoalphabetic substitution done in the encryption process
    noisy_text = perform_substitution(ciphertext, substitution_key)
    distance = levenshtein_distance(message, noisy_text)
    quality = distance / length of ciphertext
    return message, quality, substitution_key
```

If our confidence in the guess produced by `decrypt1` is too low, in the case that no outlier was found, we proceed with `decrypt2`. `decrypt2` makes use of the `HistKeyGen` library described above to generate its `histkeys`, and processes batches of them in parallel using a remote `ray` function called `decryption_with_histkey`, just like `decrypt1`. `decrypt2`'s `decryption_with_histkey` works in a fundamentally different way from `decrypt1's` function by the same name, because it doesn't have any fully formed dictionary messages to compare against the ciphertext. Instead, it needs to compare various substrings of the ciphertext with the words in `dictionary2`. These comparisons are performed by a function called match_closest_word, described here:

```
def match_closest_word(substring, dictionary_words):
    closest_word = "" # No match found
    closest_distance = len(substring)  # Every character was not found

    for word in dictionary_words:
        distance = Levenshtein.distance(substring, word)
        if distance < closest_distance):
            closest_word = word
            closest_distance = distance

    return (closest_word, closest_distance)
```

The `decrypt2` `decryption_with_histkey` uses this function in order to evaluate the quality of the matches of individual words. The `quality` metric is defined as `Lev.distance(substring, dictionary_word) / len(substring).` For each batch of `histkeys,` a guesses are created using `decryption_with_histkey`, and the best guess is kept.

(function level indent removed)

```
def decryption_with_histkey(ciphertext, histkey, dictionary2_words):

find character frequencies of ciphertext
generate substitution key from frequencies and histkey
noisystring = perform_substitution(ciphertext, substitution_key)

start_pointer = 0
end_pointer = start_pointer + 1
lookahead_pointer = end_pointer

message = []
match_qualities = []

while end_pointer < len(noisystring):
    Move end_pointer forward to next space

    if noisystring[end_pointer] == " ":
        substring = noisystring[start_pointer:end_pointer]
        match_result = match_closest_word(substring, dictionary2_words)

        lookahead_checked = False
        while lookahead_checked == False:
            lookahead_pointer = end_pointer + 1
            Move lookahead_pointer forward to next space

            if noisystring[lookahead_pointer] == " ":
                lookahead_substring = noisystring[start_pointer:lookahead_pointer]
                lookahead_result = match_closest_word(lookahead_substring,
                                                      dictionary2_words)

                if lookahead_result.quality < match_result.quality:
                    match_result = lookahead_result
                    end_pointer = lookahead_pointer
                else:
                    lookahead_checked = True
            else:
                lookahead_checked = True


        match_qualities.append(match_result.quality)
        message.append(match_result.found_word)
        start_pointer = end_pointer + 1
        end_pointer += 2
    else:
        leftover_character_num = the difference between the current length of
                                our output and the known length of the true plaintext
        alternate_dictionary_words = a truncated version of dictionary2, where words
                                   are shortened to fit the leftover_character_num

        substring = noisystring[start_pointer:end_pointer]
        match_result = match_closest_word(substring, alternate_dictionary_words)

        end_pointer += 1

return " ".join(message), match_result
```

The result is returned, and if the `quality` of this guess is found to be better than the previous guess, it replaces the previous guess. `decrypt2` finishes by returning the best guess to the user.

Possible Improvements:
Since the random characters are inserted into the plaintext, a weighted Levenshtein distance between the *noisytext* and the guess of the message would be a better metric than an unweighted Levenshtein distance. The weight would be infinity for insertions into the *noisytext*, 0 for deletions, and 1 for edits.

Other methods:

One attempted method for decrypting plaintexts from dictionary 1 that we chose not to use due to technical difficulties is in `decrypt1_alternate.py`.

Informally, and from a very high level perspective, this method creates "fingerprints" of the ciphertext and the five candidate plaintexts, and see if the "fingerprint" of the plaintexts could possibly be the "fingerprint" of the ciphertext with randomly inserted characters

References:

[1]: https://en.wikipedia.org/wiki/Levenshtein_distance