Jeffry Jimenez
CSE 331.01
Professor Rahmati
Fall 2020

Report

TASK1:

Encoder.py consist of two simple functions, of is the generate_table() which generates 26X26 table that will be use to encode the plaintext. The function that does all the work in this file is generate_cypher() which takes two inputs, plaintext and cipher_key. In this function the cipher key is repeated to match the length of the plaintext. At index (i), cipher_key[i] and plaintext[i] are use in the generated table as indexes in other to find the corresponding cipher.

TASK2:

Decoder.py generates a table in the same manner as TASK1. The function that does the heaving lifting in this file is decode_cypher() which takes the cipher text and the key as its inputs. Just like in TASK1 the cipher key is elongated but this time to match the length of the cipher text. However, unlike TASK1, at each index (i) in the cipher key we find index row (x) in the table such that table[x][i] = cipher_text[i]. that (x) represents the letter in the plain text at index (i).

TASK3:

CrackVigenere-wKey.py takes a cipher text and the length the key used to encode the plaintext; the function then generates the corresponding key decodes the cipher text. With the length of the key known we can break the cipher text into (key size) different spans. Each span is different, the first span which starts at index 0 will have a char from the cipher text located at a multiple of the key length. For example, say the key length is 3, the first span will have chars from the cipher text at index 0, 3, 6, 9 … The same idea applies the other spans generated though they start at different indexes. This will mean that every letter within a span has been encoded with the same letter from the cipherkey, or in other words each span is a Caesar cipher. The getSpan() function call in the crack() function returns an array of spans generated from the cipher text. Now that we have a list of Caesar ciphers what's left to do is use frequency analysis to find the most probable letter that was used to encode each Caesar cipher. The getChar() function call does this in the crack() function. The getChar() function takes a span as input and shifts its letter 26 times, if a shift of 1 (a) -> (z) and (b) -> (a), if to 2 (a) -> (y), (b) -> (z) and so on. Each time a shift of letters is made, a chi square test is generated with the frequency of shifted letters of the span and the expected frequency being that of the English language. The result of the test into an array of length 26 where every index correlate to a shift of letters. When all the shifts are made, the index with the minimum chi square test tells us that this shift is the one that looks more English like, so that index is casted into a char and returned. After going through all the spans, the most likely key is generated, all that is left is use the functions from TASK2 to decode the cipher text.

TASK4:

crackVigenere_Friedman.py takes a cipher text and produces a key and its plaintext. The getkeylength() function call the main function returns the most likely key length. The getkeylength() function uses Friedman's Index of Coincidence to accomplish its task. In the function a span is created for every possible key length, and for each possible span from a specific key length the Index of Coincidence is calculated by a sub function called getIndexOfCoincendece() which implements the IC = $\frac{\sum_{i=1}^{c} n_i(n_i-1)}{N(N-1)}$, the results of the spans generated by the same key length are added together and then divided by the key length, this results in the average Index of Coincidence of the spans for that specific key length, the average is store in the ic[]. After all the averages of the Indexes of Coincidence have been calculated all that is left is to find the most suitable keylength in ic[]. Before finding the most likely length I converted ic[] into a dictionary ic_dictionary, where key= length of a key, and value = Index of Coincidence. To choose the most favorable index in ic_dictionary, the function searches for the value closest Index of Coincidence to 0.067. When such value is found its key does not necessarily contain the correct length, however the correct length if most likely to be a factor of the found key. The function call to getFactors() takes care of finding the factors of the key we found. Not all factors of this number are suitable to be the correct length. To pick the correct factors we loop through the factor list and if the current factor exists within the ic_dictionary, check the value of ic_dictionary[factor] is larger than both the values of its neighbors, this will indicate that spans created with that key length are closer to the standard Frequency for English letters than its neighbors. From Those suitable factors the one with the highest index of coincidence is chosen as our length. Once the length is found TASK3 takes care of the rest and finds the KEYWORD.