Project 1: Cracking Classical Codes

This project asks you to crack several classical codes. What you should learn from this is that crypt-analysis is possible and that you do not know enough to invent your own codes. Other than that, it is an intellectual exercise strengthening your capability to draw conclusions from clues and to work towards a goal in a systematic manner. You can use any programming language and any information you might find on the web. Just be careful about using code downloaded from the web, you might be using products produced by newbies.

Challenge

You are given several challenge texts that you have to read without guessing passwords or passphrases. You can assume that the clear text is written in a English, but it might be slightly pre-processed. For instance, it might only contain the letters of the text in capitals, or it might be the original text. However, the texts will not contain keyword substitutions. The crypto used are included in the following list:

- Substitution Ciphers
  - Caesar
  - General ciphers
- Stream ciphers
  - XORing with a key-pad consisting of a repeating string
  - Key-pad consisting of pseudo-random symbols. (You should not be able to break this code, though NSA would, since I am not using a good random number generator.)

Hand-in

The first 100 letters or so of each cipher text that you cracked together with the password or passphrase.

A short description of the tools that you used.

Hints

The internet is full of useful links that describe how to crack codes. The first step in general is looking at the file with the cipher text. Often, the file is a binary file and not a text file, so you want to be careful about how you open them programmatically. You then write tools that calculate the frequency distribution of symbols at certain off-sets, calculate Gini impurities of inequality, Friedman statistics, and numbers of coincidences in dependence on displacement.

Cracking Vigenère

Cracking Vigenère usually proceeds by first obtaining the length of the key. A number of approaches can be taken.

First, the Kasiski method looks for repeated substrings in the cypher text. These can appear by happenstance, but more frequently, because a substring in the plain text was repeated at a distance that is the key length or a multiple of the key length. The cryptanalysis gathers all differences between repeated substrings (for example of length 3 or of length 4), factors them, and looks for factors that are more frequent. The most frequent factor is likely to be the length of the key.

A much simpler method is to assume a certain key length *n* and then divide the cipher text into *n* different substrings formed from the symbols at a position x where x = i mod n for i = 0, 1, 2, … , *n*-1. If *n* is a multiple of the key-length, then the *n* substrings will show the same frequency distribution as the original plain text. We can measure the inequality in various ways:

- We calculate the combined entropy of the *n* substrings
- We calculate the Gini impurity (NOT the Gini coefficient). Given a substring, we calculate the frequency of all letters in a substring. Let p_i be the relative frequency of the *i*-th letter. The Gini impurity is then

$$\sum_{i=1}^{J} p_i(1 - p_i)$$

  and the Gini impurity of the *n* substrings is the average of the individual Gini impurities.
- The Friedman test does not divide the cipher into a number of substrings, but rather calculates an index of coincidence directly and estimates the key length from there using certain constants. For monocase English, the formula is

$$\frac{0.067 - 0.0385}{\frac{\sum_{i=1}^{26} n_i(n_i-1)}{N(N-1)} - 0.0385}$$

  where *N* is the number of letters in the cipher text block, *c* the number of letters in the alphabet (e.g. 26 for monocase English) and p_i the relative frequencies of the letter in the cipher. The first constant (0.067) corresponds to the coincidence index in normal English text and the second constant is merely the inverse of 26.

_____CAESAR.py_____

```
import string

def create_table(shift):
    dicc = {}
    for x in string.ascii_letters:
        if ord(x) >= 97 and ord(x) <= 122:
            y = (ord(x)-97+shift)%26+97
            dicc[x]=chr(y)
        elif ord(x) >= 65 and ord(x) <= 90:
            y = (ord(x)-65+shift)%26+65
            dicc[x]=chr(y)
        else:
            raise ValueError
    return dicc

def pretty_print_table(dicc):
    for x in string.ascii_letters:
        print(x, dicc[x])


def caesar_encode(in_file, out_file, shift):
    dicc = create_table(shift)
        with open(in_file, encoding="utf-8") as in_file, open(out_file, "w", encoding="utf-8") as
out_file:
        for line in in_file:
            output_line = []
            for element in line:
                if element in dicc:
                    output_line.append(dicc[element])
                elif element == "\n":
                    pass
                else:
                    output_line.append(element)
            print("".join(output_line), file=out_file)
    print("hecho")

def caesar_decode(in_file, out_file, shift):
    decode_shift = 26-shift
    dicc = create_table(decode_shift)
        with open(in_file, encoding="utf-8") as in_file, open(out_file, "w", encoding="utf-8") as
out_file:
        for line in in_file:
            output_line = []
            for element in line:
                if element in dicc:
                    output_line.append(dicc[element])
                elif element == "\n":
                    pass
                else:
                    output_line.append(element)
            print("".join(output_line), file=out_file)
```

```
    print(“hecho")
```

_____vigenere.py_____

```
    import collections

    def change(letra, key):
        if 65 <= ord(letra) <= 90:
            return chr((ord(letra)-65+ord(key)-65)%26+65)
        else:
            return letra

    def unchange(letra, key):
        if 65 <= ord(letra) <= 90:
            return chr((ord(letra)-65+91-ord(key))%26+65)
        else:
            return letra

    def vigenere_encode(in_file_name, out_file_name, key):
        with open(in_file_name) as in_file, open(out_file_name, "wt") as out_file:
            pos = 0
            for line in in_file:
                cadena = []
                for letra in line:
                    kl = key[pos]
                    pos += 1
                    if pos >= len(key):
                        pos = 0
                    cadena.append(change(letra, kl))
                print("".join(cadena), file=out_file, end='')

    def vigenere_decode(in_file_name, out_file_name, key):
        with open(in_file_name) as in_file, open(out_file_name, "wt") as out_file:
            pos = 0
            for line in in_file:
                cadena = []
                for letra in line:
                    kl = key[pos]
                    pos += 1
                    if pos >= len(key):
                        pos = 0
                    cadena.append(unchange(letra, kl))
                print("".join(cadena), file=out_file, end='')
```

_____xor.py_____

```
import random

password = "Boulderdash"
```

```
def decipher(nameIn, nameOut):
    toDecode = bytearray()
    result = bytearray()
    with open(nameIn, "rb") as inFile, open(nameOut, "wb") as outFile:
        for line in inFile:
            for x in line:
                toDecode.append(x)
        for i in range(len(toDecode)):
            result.append(toDecode[i] ^ ord(password[i%len(password)]))
        outFile.write(result)
```