Attacking Classic Crypto Systems - Lab 2
Solutions, Steps, Commands and Code

Moloy Banerjee
Reg: 2020831021

Overview

This report contains solutions, steps, commands, and code for the assigned tasks in Lab 2. The tasks include breaking a Caesar cipher and two substitution ciphers. All provided code samples are in **Python 3** and can be copied into a '.py' file to run.

—

1 Checkpoint 1: Caesar Cipher

1.1 Ciphertext and Approach

The Caesar cipher to be broken is:

`odroboewscdrolocdcwkbdmyxdbkmdzvkdpybwyeddrobo`

The approach used is **brute-force**. This involves trying all 26 possible shifts. The English-looking candidates are then printed, and the meaningful plaintext is chosen.

1.2 Python Program: Brute-Force Caesar Solver

This script, named `caesar_break.py`, iterates through all possible 26 shifts and prints the output for manual inspection.

```
# caesar_break.py
import string

cipher = "odroboewscdrolocdcwkbdmyxdbkmdzvkdpybwyeddrobo"
alpha = string.ascii_lowercase

def caesar(s, shift):
    out = []
    for ch in s:
        if ch in alpha:
            # Apply the reverse shift (decryption)
            out.append(alpha[(alpha.index(ch) - shift) % 26])
        else:
            out.append(ch)
    return "".join(out)

for shift in range(26):
    cand = caesar(cipher, shift)
    print(f"shift {shift}: {cand}")

# How to run:
# python3 caesar_break.py
```

1.3 Expected Outcome

One of the shifts will produce a readable English sentence, which is the decoded plaintext.
—

2 Checkpoint 2: Substitution Ciphers

2.1 Ciphertexts and Approach Summary

The solution is applied to two substitution ciphertexts (Cipher-1 and Cipher-2), which should be pasted into files named `cipher1.txt` and `cipher2.txt`.

The approach combines frequency analysis and a heuristic search algorithm:

1. **Frequency Analysis:** Initial guesses are made by mapping the most frequent ciphertext letters to common English letters (like 'e', 't', 'a', etc.).
2. **Hill-Climbing:** The mapping is iteratively improved using a score based on English word matches and vowel presence (a simplified fitness function). A hill-climbing algorithm is used to maximize this score by randomly swapping key mappings (mutating the key).
3. **Output:** The best plaintext candidate found after the iterations is produced.

2.2 Python Solver: Simple Hill-Climbing

This script, named `subcipher_break.py`, implements a basic hill-climbing approach for substitution cipher solving.

```
# subcipher_break.py
# Simple substitution cipher breaker using hill-climbing with English scoring

import math, random, string, sys
from collections import Counter

alphabet = string.ascii_lowercase
```

```
def load_cipher(filename):
    with open(filename) as f:
        # Load the cipher and remove all whitespace
        return "".join(f.read().lower().split())

# Basic English wordlist scoring for quick improvement
common_words = ["the", "and", "that", "have", "for", "not", "with", "you",
                "this", "but", "his", "from", "they", "she", "which"]

def score_plaintext(text):
    # Score by counting common words occurrences and letter frequency fitness
    score = 0
    for w in common_words:
        if w in text:
            score += 10 * text.count(w)

    # Letter frequency closeness to English (simplified: promote vowels)
    freq = Counter(c for c in text if c.isalpha())
    vowels = sum(freq[c] for c in "aeiou")
    score += vowels
    return score

def decrypt_with_key(cipher, keymap):
    # keymap is a list of 26 letters (plaintext alphabet)
    table = str.maketrans(alphabet, "".join(keymap))
    return cipher.translate(table)

def random_key():
    L = list(alphabet)
    random.shuffle(L)
    return L

def mutate_key(key):
    k = key[:]
    # Swap two random key positions
    i, j = random.sample(range(26), 2)
    k[i], k[j] = k[j], k[i]
    return k

def hill_climb(cipher, iterations=5000):
    best_key = random_key()
    best_plain = decrypt_with_key(cipher, best_key)
    best_score = score_plaintext(best_plain)

    for it in range(iterations):
        cand_key = mutate_key(best_key)
        cand_plain = decrypt_with_key(cipher, cand_key)
        cand_score = score_plaintext(cand_plain)

        if cand_score > best_score:
            best_key, best_score = cand_key, cand_score
            best_plain = cand_plain
            # print progress
            print(f"iter {it} improved score {best_score}: {best_plain[:120]}...")

    return best_plain, best_key, best_score

if __name__ == '__main__':
    if len(sys.argv) < 2:
        print("Usage: python3 subcipher_break.py cipherfile.txt")
        sys.exit(1)

    cipher = load_cipher(sys.argv[1])
    # Run hill climbing for 5000 iterations
    best_plain, key, sc = hill_climb(cipher, iterations=5000)

    print("\nBest score:", sc)
    print("Plaintext candidate:\n")
    print(best_plain)
    print("\nKey mapping (cipher->plain):")
    for c, p in zip(alphabet, "".join(key)):
        print(f"{c}->{p}", end=' ')
    print()

# How to run:
# python3 subcipher_break.py cipher1.txt > out1.txt
# python3 subcipher_break.py cipher2.txt > out2.txt
```

2.3 Ciphertext Comparison: Which is Easier to Break?

When comparing the two ciphertexts (Cipher-1 and Cipher-2), the decision on which is easier is based on the following criteria:

- **Method to Decide:** Compare the final scoring convergence, the number of recognizable English words found, and the overall clarity of the candidate plaintexts.
- **Reason:** Ciphertexts that are generally shorter or more repetitive are easier to break. More importantly, texts with higher word structure and common short words (e.g., "the," "and," "to") are easier because the frequency patterns and word boundaries strongly help guide the key mapping.

—

3 Deliverables and Conclusion

The final deliverables for this lab include:

1. Showing the working Python scripts and their outputs to the instructor.
2. Including the best plaintext candidates (from `out1.txt` and `out2.txt`) and explaining the steps and reasoning in the final report.