Hill Cipher: Theoretical Overview

Hill Cipher

The Hill cipher is a polygraphic substitution cipher based on linear algebra, invented by Lester S. Hill in 1929. It was the first polygraphic cipher in which it was practical to operate on more than three symbols at once.

Encryption

To encrypt a message, each block of n letters (considered as an n-component vector) is multiplied by an invertible $n \times n$ matrix, against a modulus that depends on the size of the alphabet (26 for the English alphabet). Each letter is represented by a number modulo 26, with A=0, B=1, ..., Z=25.

For example, consider the message 'ACT' and the key matrix:

$$\begin{pmatrix} 6 & 24 & 1 \\ 13 & 16 & 10 \\ 20 & 17 & 15 \end{pmatrix}$$

Since 'A' is 0, 'C' is 2, and 'T' is 19, the message can be written as the vector:

$$\begin{pmatrix} 0 \\ 2 \\ 19 \end{pmatrix}$$

The enciphered vector is then obtained by matrix multiplication:

$$\begin{pmatrix} 6 & 24 & 1 \\ 13 & 16 & 10 \\ 20 & 17 & 15 \end{pmatrix} \begin{pmatrix} 0 \\ 2 \\ 19 \end{pmatrix} == \begin{pmatrix} 67 \\ 222 \\ 319 \end{pmatrix} \equiv \begin{pmatrix} 15 \\ 14 \\ 7 \end{pmatrix} (\mathrm{mod}\, 26)$$

which corresponds to a ciphertext of 'POH'.

Decryption

To decrypt a message, we turn the ciphertext back into a vector and multiply it by the inverse of the key matrix used for encryption.

For example, using the same key matrix as above, the inverse matrix is:

$$\begin{pmatrix} 6 & 24 & 1 \\ 13 & 16 & 10 \\ 20 & 17 & 15 \end{pmatrix}^{-1} \equiv \begin{pmatrix} 8 & 5 & 10 \\ 21 & 8 & 21 \\ 21 & 12 & 8 \end{pmatrix} (\mathrm{mod}\, 26)$$

If we have the ciphertext 'POH', we can decrypt it as follows:

$$\begin{pmatrix} 8 & 5 & 10 \\ 21 & 8 & 21 \\ 21 & 12 & 8 \end{pmatrix} \begin{pmatrix} 15 \\ 14 \\ 7 \end{pmatrix} == \begin{pmatrix} 260 \\ 574 \\ 539 \end{pmatrix} \equiv \begin{pmatrix} 0 \\ 2 \\ 19 \end{pmatrix} (\mathrm{mod}\, 26)$$

which gets us back to 'ACT'.

Choosing the Key Matrix

Not all matrices can be used as key matrices in the Hill cipher. The determinant of the matrix must be non-zero and must not have any common factors with the modular base (26 for the English alphabet). This ensures that the matrix is invertible, which is necessary for decryption.

For example, the determinant of the 3x3 key matrix above is:

$$|\begin{pmatrix} 6 & 24 & 1 \\ 13 & 16 & 10 \\ 20 & 17 & 15 \end{pmatrix}| == 6(16 \cdot 15 - 10 \cdot 17) - 24(13 \cdot 15 - 10 \cdot 20) + 1(13 \cdot 17 - 16 \cdot 20) = 441 \equiv 25 (\mod 26)$$

Since 25 is prime with 26 (i.e., they have no common factors), this matrix can be used for the Hill cipher.

Variants and Extensions

The basic Hill cipher is vulnerable to known-plaintext attacks because it is completely linear. However, it can be combined with other non-linear operations to increase its security. For example, the MixColumns step in AES is a matrix multiplication, and the Twofish cipher uses a combination of non-linear S-boxes with a carefully chosen matrix multiplication.

The Hill cipher can also be extended to work with larger blocks of letters by using larger key matrices. Hill himself invented a machine that mechanically implemented a 6 x 6 version of the cipher, which was very secure. However, the machine was unable to change the key setting, limiting its use in practice.

Listing 1: hill-cipher.py

```
import numpy as np
import string
import math


def modinv(num, mod):
    for i in range(1, mod):
        if (num * i) % mod == 1:
            return i
    return None

def modinv_matrix(matrix, mod):
    det = int(round(np.linalg.det(matrix)))
    inv_det = modinv(det, mod)
    adj_mat = (np.linalg.inv(matrix).T * det).T
    mat_inv = (inv_det * adj_mat) % mod
    return np.round(mat_inv).astype('int')

class HillCipher:

    _alphabet = string.ascii_uppercase + '1234567890 .-:$'
    mod = len(_alphabet)


    def _char_to_num(self, char):
        return self._alphabet.index(char)


    def _num_to_char(self, num):
        return self._alphabet[num]

    def __init__(self, key: str, alphabet=None):
        self.key = key
        self._key_size = math.ceil(math.sqrt(len(key)))

        if alphabet:
            self._alphabet = alphabet
            self.mod = len(alphabet)

```

```
        self.encryption_key = np.array([self._char_to_num(char) for char in key.upper()] + [26] * (self._key_size**2 - len(self.key)), dtype='int').reshape(self._key_size, self._key_size)

        det = int(round(np.linalg.det(self.encryption_key))) % self.mod
        if det == 0 or math.gcd(det, self.mod) != 1:
            raise ValueError('Invalid Key: Key Matrix is not invertible modulo 29')

        self.decryption_key = modinv_matrix(self.encryption_key, self.mod)

    def encrypt(self, plaintext: str):
        _plaintext = [self._char_to_num(c) for c in plaintext.upper()]
        while len(_plaintext) % self._key_size != 0:
            _plaintext.append(len(self._alphabet)-1)

        _ct = []
        for i in range(0, len(_plaintext), self._key_size):
            block = np.array(_plaintext[i:i+self._key_size])
            e_block = np.dot(self.encryption_key, block) % self.mod
            _ct.extend(e_block)

        return "".join([self._num_to_char(n) for n in _ct])


    def decrypt(self, ciphertext: str):
        _ciphertext = [self._char_to_num(c) for c in ciphertext.upper()]
        if len(_ciphertext) % self._key_size != 0:
            raise ValueError("Ciphertext length must be a multiple of key size")

        _pt = []
        for i in range(0, len(_ciphertext), self._key_size):
            block = np.array(_ciphertext[i:i+self._key_size])
            d_block = np.dot(self.decryption_key, block) % self.mod
            _pt.extend(d_block)

        while _pt and _pt[-1] == len(self._alphabet)-1:
            _pt.pop()

        return "".join([self._num_to_char(n) for n in _pt])

```

```
78
79 KEY = 'DEVANSH'
80 PLAINTEXT = "DATE:2024-07-26$CNS:PRACTICAL THREE"
81
82 cipher = HillCipher(KEY)
83 print(f'[KEY]\n{cipher.key}')
84 print(f'[PLAINTEXT]\n{PLAINTEXT}')
85 print(f'[ENCRYPTION KEY]\n{cipher.encryption_key}')
86 e_text = cipher.encrypt(PLAINTEXT)
87 print(f'[ENCRYPTED]\n{e_text}')
88 print(f'[DECRYPTION KEY]\n{cipher.decryption_key}')
89 d_text = cipher.decrypt(e_text)
90 print(f'[DECRYPTED]\n{d_text}')
91
```

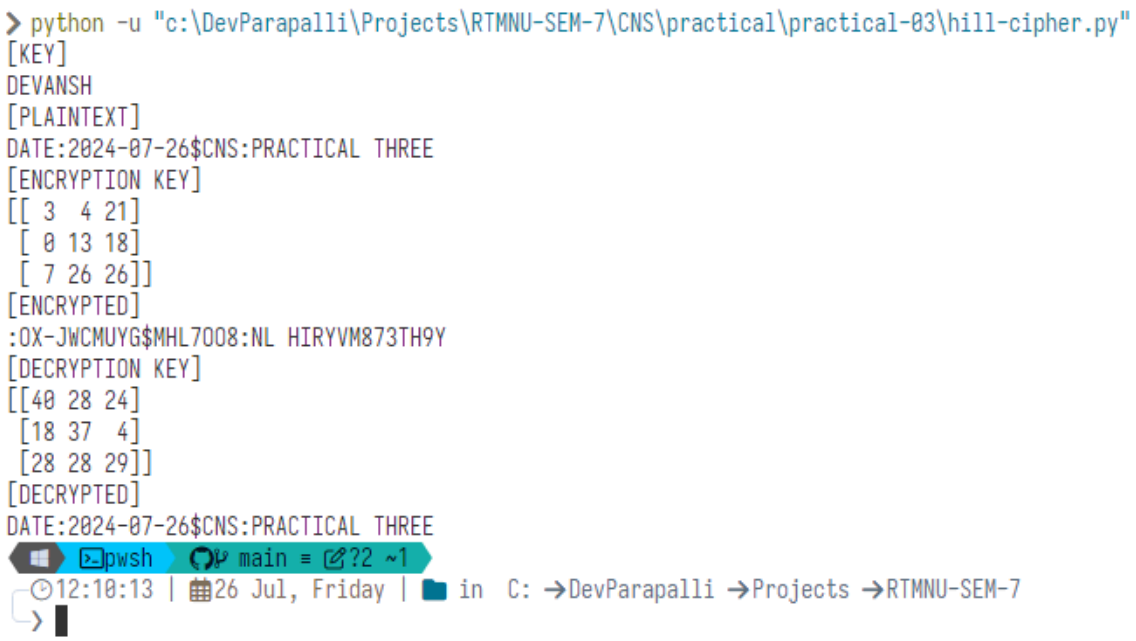

Figure 1: Output