Esercizi Set 1

Lorenzo Macchiarini

7 Aprile 2021

Esercizio 2.2: Indici di Coincidenza

a. Prendo $i \in \mathbb{Z}_{26}$ e so che $f_i = \sum_{j=1}^{n} Y_j$ con $Y_j = \begin{cases} 1 & \text{if } X_j = i \\ 0 & \text{if } X_j \neq i \end{cases}$.

Analizzo il valore atteso di f_i e sfruttando la linearità del valore atteso ottengo:

$$\mathbb{E}[f_i] = \mathbb{E}[\sum_{j=1}^{n} Y_j] = \sum_{j=1}^{n} \mathbb{E}[Y_j] \tag{1}$$

Per la definizione di p_i, probabilità che un carattere in x sia i, ho che $p_i = \mathbb{E}[Y_j]$. Quindi ottengo::

$$\sum_{j=1}^{n} \mathbb{E}[Y_j] = \sum_{j=1}^{n} p_i = n * p_i \tag{2}$$

b. Dimostro che la varianza di f_i calcolata per definizione come $\text{var}(f_i) = \mathbb{E}[(x - \mathbb{E}[x])^2]$ può essere espressa come $\text{var}(f_i) = \mathbb{E}[f_i^2] - \mathbb{E}[f_i]^2$ e che quindi $\mathbb{E}[f_i^2] = \text{var}(f_i) + \mathbb{E}[f_i]^2$.

$$\begin{aligned} var(f_i) &= \mathbb{E}[(f_i - \mathbb{E}[f_i])^2] = \mathbb{E}[f_i^2 - 2x\mathbb{E}[f_i] + \mathbb{E}[f_{i2}]] \\ &= \mathbb{E}[f_i^2] - 2\mathbb{E}[f_i]\mathbb{E}[f_i] + \mathbb{E}[f_{i2}] = \mathbb{E}[f_i^2] - \mathbb{E}[f_i]^2 \end{aligned} \tag{3}$$

c. Poiché se i caratteri x_j sono estratti in maniera i.i.d. ho che f_i è una distribuzione binomiale, so che la varianza $\text{var}(f_i) = n * p_i * (1 - p_i)$. Quindi, sapendo che $\mathbb{E}[f_i^2] = \text{var}(f_i) + \mathbb{E}[f_i]^2$ posso scrivere:

$$\mathbb{E}[f_i^2] = var(f_i) + \mathbb{E}[f_i]^2 = np_i^2 + np_i(1 - p_i) = n^2p_i^2 + np_i - np_i^2 \tag{4}$$

Quindi studiando il valore di $\mathbb{E}[f_i(f_i - 1)]$ noto che:

$$\begin{aligned} \mathbb{E}[f_i(f_i - 1)] &= \mathbb{E}[f_i^2 - f_i] = \mathbb{E}[f_i^2] - \mathbb{E}[f_i] \\ &= n^2p_i^2 + np_i - np_i^2 - np_i = np_i^2(n - 1) \end{aligned} \tag{5}$$

d. Sappiamo che $I_c(x) = \sum_{i \in \mathbb{Z}_{26}} \frac{f_i(f_i-1)}{n(n-1)}$, studiamo il valore di $\mathbb{E}[I_c(x)]$:

$$\begin{aligned}\mathbb{E}[I_c(x)] &= \mathbb{E}[\sum_{i \in \mathbb{Z}_{26}} \frac{f_i(f_i-1)}{n(n-1)}] = \sum_{i \in \mathbb{Z}_{26}} \mathbb{E}[\frac{f_i(f_i-1)}{n(n-1)}] \\ &= \sum_{i \in \mathbb{Z}_{26}} \frac{1}{n(n-1)} \mathbb{E}[f_i(f_i-1)] = \sum_{i \in \mathbb{Z}_{26}} \frac{np_i^2(n-1)}{n(n-1)} = \sum_{i \in \mathbb{Z}_{26}} p_i^2\end{aligned} \tag{6}$$

e. Per la definizione di prodotto scalare ho che

$$| < p, q > | = \|p\|_2 \cdot \|q\|_2 * cos(\theta) \tag{7}$$

avendo p,q due vettori generici e definendo θ l'angolo compreso fra quest'ultimi. Tale angolo può essere complicato da calcolare a seconda dello spazio in cui i vettori hanno valori, però sappiamo che se $\theta = 0, cos(\theta) = 1$ e quindi è massimo. Questo ci permette di dire che se i vettori p e q sono paralleli il valore di $| < p, q > |$ è massimo.

Possiamo anche notare che

$$| < p, q > | = p_1q_1 + ... + p_nq_n \tag{8}$$

Sia k il valore dello shift che massimizza il prodotto scalare fra p e q. Per assurdo ipotizzo che $\exists k' : | < p, q > | = \|p\|_2 \cdot \|q\|_2$ e che quindi è anch'esso massimo. Ipotizzo, senza perdita di generalità, che le frequenze massime in p e q sono rispettivamente p_i e q_j. Sappiamo che k è il valore per cui $(p_1 + k)q_1 + ... + (p_n + k)q_n$ è massimo e che quindi fa coincidere i valori di p_i e q_j moltiplicandoli insieme. Poichè $k' \neq k$ ho che i valori di p_i e q_j non coincidono e che quindi $| < p, q > |$ non è massimo, trovando l'assurdo.

Esercizio 2.3: Un Crittogramma Vigenère

L'algoritmo di cifratura di Vigenère è polialfabetico a sostituzione. L'idea è quella di eseguire degli shift nel plaintext in modo che a stesse lettere nel plaintext vengano associate diverse lettere del ciphertext. L'algoritmo cifra il plaintext sommandolo modulo 26 ad una chiave ripetuta più volte ottenendone una lunga quanto il plaintext stesso.

plaintext	d	a	t	a	s	e	c	u	r	i	t	y	a	n	d	p	r	i	v	a	c	y	+
key	d	s	p	d	s	p	d	s	p	d	s	p	d	s	p	d	s	p	d	s	p	d	=
ciphertext	g	s	i	d	k	t	f	m	g	l	l	n	d	f	s	s	j	x	y	s	r	b	

Questo permette di avere un numero di alfabeti uguale alla lunghezza della chiave originaria rendendo più complicato un attacco basato sulle frequenze.

La ripetizione della chiave però introduce delle ridondanze nel ciphertext che possono essere sfruttate in un attacco.
L'attacco al cifrario è composto dalle seguenti fasi:

1. determinare la lunghezza m della chiave usata
2. determinare ognuno degli m caratteri della chiave

La lunghezza della chiave potrebbe essere potenzialmente la stessa del plaintext, quindi la sua ricerca potrebbe essere computazionalmente onerosa. Possiamo notare però che ad intervalli multipli della lunghezza della chiave possono ripetersi in modo casuale dei trigrammi specifici nel ciphertext. Sfruttando questa caratteristica del cifrario, possiamo utilizzare il *test di Kasiski* che ci permette di diminuire lo spazio delle possibili lunghezze delle chiavi all'insieme dei divisori delle distanze fra i trigrammi uguali. Estraggo quindi dal ciphertext fornito tutti i trigrams che si ripetono almeno una volta e calcolo la distanza fra ogni ripetizione. Al termine dell'esecuzione *ngramsAppearances* conterrà l'insieme di coppie *trigrams : lista di distanze di apparizione*.

```
ct = "EIVDMAO...TZOGUX".lower()
ngramsAppearances = {}
i = 0
while i+3 <= len(ct):
    gram = ct[i:i+3]
    if(gram not in ngramsAppearances):
        ngramsAppearances[gram] = []
        j = i+3
        while j+3 <= len(ct):
            if(gram == ct[j:j+3]):
                ngramsAppearances[gram].append(j-i)
            j += 1
        if(len(ngramsAppearances[gram]) == 0):
            ngramsAppearances.pop(gram)
    i += 1
```

I risultati ottenuti da questa esecuzione sono:

```
{
   "bvr":[30, 105, 150, 300, 390],
   "nae":[285, 345, 415, 615],
   "tam":[195, 210, 435],
   "dyt":[90, 315],
   "eim":[250, 449],
   "vru":[105, 150],
   "lqc":[120, 135],
   "vpt":[195],
   "ptr":[195],
   ...
   "vra":[90]
}
```

Possiamo quindi estrarre i fattori che compongono queste distanze e analizzare quelli che sono più frequenti. In particolare otteniamo che i 10 più frequenti sono riportati nella Tabella 1.

fattore	numero occorrenze
3	58
5	55
15	52
2	27
6	25
10	23
9	17
7	12
21	11
4	10

Tabella 1: Numero di occorrenze dei 10 fattori più frequenti nel test di Kasiski

Empiricamente notiamo che il valore corretto di m è 15, quindi ora dovremo trovare i valori della chiave k lunga m. Per fare ciò inizialmente analizziamo il ciphertext e lo disponiamo in una matrice di dimensione $15 \times \lceil len(ciphertext)/15 \rceil$. Ogni colonna della matrice sarà composta da blocchi di 15 caratteri adiacenti del ciphertet, mentre le righe saranno composte da caratteri del ciphertext cifrati usando uno stesso shift, poichè la lettera della chiave con cui sono cifrati è la stessa. Possiamo qunidi analizzare i valori degli indici di coincidenza per verificare che la lunghezza della chiave sia corretta. Se la lunghezza della chiave fosse sbagliata i valori degli indici sarebbero casuali e $\simeq 0.038$ mentre se la lunghezza è corretta i valori risultano $\simeq 0.065$. Infatti, come possiamo vedere nella Tabella 2, i valori sono $\simeq 0.065$.

Il codice utilizzato per estrarre i valori è il seguente, in cui *txt* è un dizionario contenente la matrice del ciphertext disposto per colonne, *freq* è una lista che conta il numero di occorrenze per ogni lettera all'interno di ogni riga e *coincIndex* associa ad ogni riga il relativo valore dell'indice di coincidenza.

```
coincIndex = {}
for i in range(m):
    freq = {j:txt[i].count(j) for j in txt[i]}
    coincIndex[i] = 0
    for j in freq:
        coincIndex[i] +=
            freq[j]*(freq[j]-1)/(len(txt[i])*(len(txt[i])-1))
```

Infine per ottenere la chiave conoscendo ciphertext e lunghezza della stessa, devo studiare con quanti shift delle lettere ottengo le frequenze più vicine a quelle della lingua inglese per ogni riga della matrice. Per fare questo quindi devo fare 25 shift per ogni riga e confrontare ogni risultato con le frequenze della lingua inglese tramite i prodotti scalari. Facendo il prodotto scalare fra i due

riga	indice di coincidenza
0	0.0613
1	0.0666
2	0.0513
3	0.0571
4	0.0862
5	0.0783
6	0.0867
7	0.0677
8	0.0613
9	0.0666
10	0.092
11	0.0746
12	0.0508
13	0.082
14	0.0738

Tabella 2: Verifica che gli ordini di coincidenza per m = 15 sono $\simeq 0.065$

vettori di frequenze ottengo una metrica che, massimizzata, permette di capire quale shift rende i due vettori più vicini. Per fare questo ho implementato tale ricerca con il seguente codice:

```
key = []
scalProds = []
for i in range(m):
    distr = {ord(j)-97:txt[i].count(j)/len(txt[i]) for j in txt[i]}
    scalarProds = []
    for k in range(26):
        scal = 0
        newDistr = {}
        for j in range(26):
            if((j+k)%26 in distr):
                newDistr[j] = distr[(j+k)%26]
            else:
                newDistr[j] = 0
        for j in range(26):
            scal += newDistr[j]*engFreq[j]
        scalarProds.append(scal)
    key.append(chr(scalarProds.index(max(scalarProds))+97))
    scalProds.append(max(scalarProds))
```

In questo caso *engFreq* è la distribuzione delle lettere nella lingua inglese, *distr* è la distribuzione delle lettere per la riga i-esima della matrice, nel ciclo interno faccio 25 shift creando per ognuno la distribuzione *newDistr* shiftata, quindi applico il prodotto scalare fra *newDistr* e *engFreq* salvandone il valore in *scalarProds*. Infine massimizzo i prodotti scalari della riga ottenendo il valore più

corretto del carattere i-esimo della chiave k.
I valori massimi dei prodotti scalari ottenuti per ogni riga sono riportati nella Tabella 3.

indice riga	carattere trovato	valore prodotto scalare
0	p	0.0633
1	e	0.0696
2	r	0.0609
3	m	0.0631
4	u	0.0716
5	t	0.0699
6	a	0.0739
7	t	0.0676
8	i	0.0634
9	o	0.0661
10	n	0.0708
11	c	0.0691
12	i	0.0574
13	t	0.0738
14	y	0.0704

Tabella 3: Tabella contenente i valori massimi dei prodotti scalari delle righe e i relativi caratteri della chiave trovati

Esercizio 3.1: Analisi delle frequenze di un testo

Il testo da analizzare è il primo capitolo di Moby Dick che ho riportato in lettere minuscole. Ho quindi estratto l'insieme di parole contenute nel testo considerando parole distinte quelle separate da segni di interpunzione o da spazi o caratteri diversi da lettere.

```
f = open("mobydick.txt", "r")
wordSet = []
for line in f:
    currentWord = ""
    for letter in line:
        if letter.isalpha():
            currentWord += letter
        else:
            if currentWord != "":
                wordSet.append(currentWord)
                currentWord = ""
f.close()
```

Ho quindi estratto gli n-grams dalle parole trovate creando un dizionario Python così composto:

```
ngrams = {
    i : {
        "totalNum" : # intero che conta il numero totale di i-grams
        "set" : { # set di coppie (i-gram, # occorrenze dell' i-gram)
            gram : numero occorrenze
        }
    }
    ... # per i = 1 ... 4
}
```

Il codice che ricava tali informazioni estrae gli n-gram da ogni parola, se tale n-gram non è presente nel dizionario lo aggiunge ed infine incrementa il contatore sia dell'n-gram relativo sia del numero totale di n-gram contenuti nel testo.

```
ngrams = {}
for n in range(1,5):
    ngrams[n] = {}
    ngrams[n]["totalNum"] = 0
    ngrams[n]["set"] = {}
    for word in wordSet:
        i = 0
        while i+n <= len(word):
            gram = word[i:i+n]
            if gram not in ngrams[n]["set"]:
                ngrams[n]["set"][gram] = 0
            ngrams[n]["set"][gram] += 1
            i += 1
            ngrams[n]["totalNum"] += 1
```

In questo modo ho potuto ottenere l'istogramma delle frequenze delle lettere all'interno del testo utilizzando la libreria *matplotlib*. Ho stampato il relativo istogramma, mostrato in Figura 1, colorando con un colore più scuro le lettere con frequenza maggiore.

Utilizzando tale dizionario ho potuto estrarre anche le distribuzioni empiriche degli n-grams. I risultati ottenuti sono in linea con le aspettative delle frequenze degli n-grams della lingua inglese. Riporto per ogni n solamente i 5 n-grams più frequenti ed in particolare per n = 1 ho incluso anche i valori della distribuzione della lingua inglese mostrando quanto la stima ottenuta sia vicina al valore reale di tale distribuzione. Infine per ogni n ho calcolato i valori degli indici di coincidenza e dell'entropia di Shannon.
L'*indice di coincidenza* definisce la probabilità che due elementi di un vettore x presi randomicamente siano uguali ed è definito come:

$$I_c(x) = \sum_i \frac{f_i * (f_i - 1)}{n * (n - 1)} \tag{9}$$

dato x un vettore (nel nostro caso il testo), i appartenente all'alfabeto (nel nostro caso l'insieme di n-grams), f_i le frequenze relative dell'i-esimo elemento

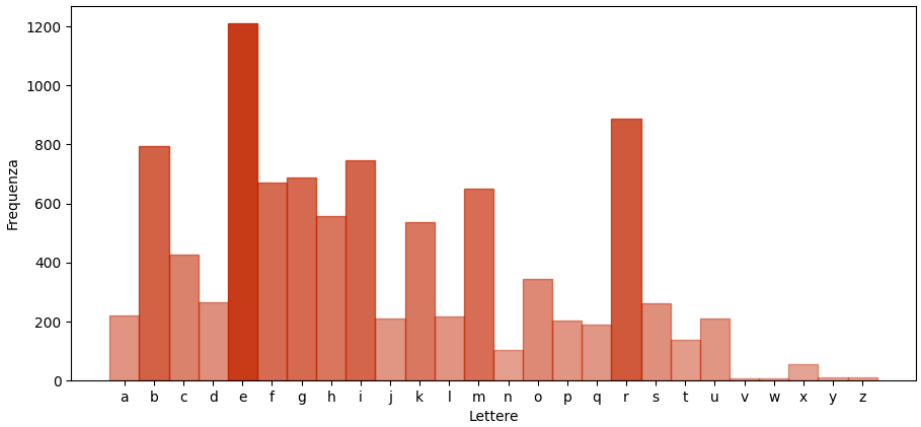

Figura 1: Istogramma che mostra la frequenza delle 26 lettere all'interno del primo capitolo di Moby Dick

n-grams	1	distr	engDistr	2	distr	3	distr	4	distr
I	e	0.1258	0.127	th	0.0417	the	0.0356	that	0.0087
II	t	0.0921	0.0906	he	0.0329	and	0.0198	ther	0.0079
III	a	0.0826	0.0817	in	0.0258	ing	0.0145	ever	0.0076
IV	o	0.0775	0.0751	er	0.0234	hat	0.0082	here	0.0073
V	s	0.0715	0.0633	an	0.0232	her	0.0082	thin	0.0051

Tabella 4: Tabella che mostra la distribuzione dei primi 5 n-grams più frequenti al variare di n e della distribuzione caratteristica della lingua inglese per n = 1

dell'alfabeto e n la cardinalità dell'alfabeto. Il valore dell'indice di coincidenza mostra quanto il testo sia una sequenza casuale di caratteri o sia una sequenza generata seguendo la distribuzione della lingua inglese. Infatti, per un alfabeto di lettere, se $I_c(x) \simeq 0.065$ il testo è generato nel secondo modo, mentre se $I_c(x) \simeq 0.038$ è stato generato nel primo. Possiamo infatti vedere nella Tabella 5 che l'indice di coincidenza del primo capitolo di Moby Dick, avendo come alfabeto l'insieme delle lettere, è $I_c(x) = 0.066 \simeq 0.065$.

L'*entropia di Shannon* misura il grado di incertezza su un vettore x, ovvero aumenta se l'insieme dei possibili valori di x aumenta o se la distribuzione dei valori di x è vicina a quella uniforme. Tale entropia è definita come:

$$H(p) = -\sum_i \frac{f_i}{n} \log_2 \frac{f_i}{n} \tag{10}$$

Possiamo vedere nella Tabella 5 che aumentando n e quindi lo spazio dei possibili valori di x, aumenta anche il valore dell'entropia.

Ho ottenuto entrambi i valori usando le frequenze calcolate per ogni n-gram e per tutti gli n-gram e svolgendo i relativi calcoli.

```
for n in range(1,5):
    ngrams[n]["indexCoincidence"] = 0
    ngrams[n]["shannonEntropy"] = 0
    for val in ngrams[n]["set"]:
        ngrams[n]["indexCoincidence"] +=
            ((ngrams[n]["set"][val])*(ngrams[n]["set"][val]-1))/
           ((ngrams[n]["totalNum"])*(ngrams[n]["totalNum"]-1))
        ngrams[n]["shannonEntropy"] -=
            (ngrams[n]["set"][val]/ngrams[n]["totalNum"])*
           math.log2(ngrams[n]["set"][val]/ngrams[n]["totalNum"])
```

In questo modo ho ottenuto un dizionario Python composto nel seguente modo:

```
ngrams = {
    i : {
        "totalNum" : # intero che conta il numero totale di i-grams
        "set" : { # set di coppie (i-gram, # occorrenze dell' i-gram)
            gram : numero occorrenze
        }
        "indexCoincidence" : # valore dell'indice di coincidenza
        "shannonEntropy" : # valore dell'entropia di Shannon
    }
    ... # per i = 1 ... 4
}
```

Ho quindi ottenuto i valori di entropia e indici di coincidenza mostrati nella Tabella 5.

n-grams	1	2	3	4
Indice di coincidenza	0.0660	0.0102	0.0033	0.0009
Entropia di Shannon	4.163	7.296	9.537	10.469

Tabella 5: Tabella che mostra i valori degli indici di coincidenza e dell'entropia di Shannon per gli n-grams ottenuti dal primo capitolo di Moby Dick

Esercizio 3.2: Cifrario di Hill

Il cifrario di Hill è un cifrario monoalfabetico a blocchi. Viene utilizzato un solo alfabeto ma in questo caso ogni lettera del ciphertext dipende da tutte le lettere di un blocco di plaintext. Questo permette di avere minori evidenze statistiche all'interno del ciphertext, rendendo molto difficile l'utilizzo di attacchi basati

sull'analisi delle frequenze.
Nel cifrario di Hill:

- il plaintext in input viene diviso in blocchi da m lettere $[p_1, ..., p_m] \in \mathbb{Z}_{26}^m$,
- anche il ciphertext è divisio in blocchi e ogni elemento nel blocco viene generato secondo la formula: $c_i = k_{i1} * x_1 + ... + k_{im} * x_m \, mod \, 26$,
- la chiave del cifrario è una matrice $K = [k_{ij}]$ con $K \in \mathbb{Z}_{26}^{m \times m}$

La cifratura di un blocco di plaintext P si ottiene quindi come $C = K \cdot P mod 26$, mentre la decifratura si ottiene come $P = K^{-1} \cdot C \, mod \, 26$. Questo implica che la chiave debba essere invertibile modulo 26, ovvero che $MCD(\, det(K), 26) = 1$.
Nell'implementazione dell'algoritmo in Python i metodi principali sono i seguenti:

- *generateKey()* che in modo randomico genera una chiave composta da interi $\in [0, 25]$ e tale che $MCD(\, det(K), 26) = 1$;

```
def generateKey():
    k = np.random.randint(lenAlphabet, size=(m, m))
    while(math.gcd(int(round(np.linalg.det(k))),lenAlphabet)!=1):
        k = np.random.randint(lenAlphabet, size=(m, m))
    return k
```

- *encryptHill()* che permette di cifrare un plaintext pt con la chiave k facendo il prodotto scalare fra pt e k modulo 26;

```
def encryptHill(pt, k):
    ct = np.dot(k, pt)%lenAlphabet
    return ct
```

- *decryptHill()* che permette di decifrare un ciphertext ct con la chiave k facendo il prodotto scalare fra l'inversa della matrice della chiave k e ct modulo. La funzione in questo caso permette di prendere in ingresso anche k1, l'inversa della matrice della chiave k, evitando di calcolare l'inversa ad ogni iterazione.

```
def decryptHill(ct, k, k1 = None):
    if(k1 is decryptHill.__defaults__[0]):
        k1 = np.array(Matrix(k).inv_mod(lenAlphabet))
    pt = np.dot(k1, ct)
    return pt%lenAlphabet
```

Il main prevede che la stringa di plaintext venga definita in *ptText*; se a lunghezza del testo non è un multiplo di m, vengono aggiunti $m - len(\, ptText \,) \, \% \, m$ caratteri di padding. Quindi divido il plaintext in blocchi di m caratteri che converto in interi da 0 a 25.

```
lenAlphabet = 26 # Lunghezza dell'alfabeto (lettere minuscole)
m = 10 # Dimensione dei blocchi

ptText = "datasecurityandprivacy"
while(len(ptText) % m != 0):
    ptText+="a"
len = len(ptText)//m # Numero di blocchi di plaintext
# ptText = "datasecurityandprivacyaaaaaaaa"

ptBlocks = np.zeros(shape=(len, m), dtype = np.int32)
for i in range(len):
    for j in range(m):
        ptBlocks[i][j] = ord(ptText[i*m+j])-97
```

Quindi genero la chiave e, utilizzando la funzione di cifratura *encryptHill*, cifro ogni blocco di plaintext trovato con la chiave ottenuta. Nella stringa di ciphertext risultante possiamo notare come questo algoritmo differisce da un semplice algoritmo monoalfabetico a sostituzione: dato che gli ultimi caratteri del plaintext sono *aaaaaaaa*, il secondo algoritmo avrebbe generato un ciphertext in cui gli ultimi caratteri sarebbero stati tutti uguali, il ciphertext del primo algoritmo invece ha come ultimi caratteri *oweameug*, grazie al principio di diffusione applicato nella cifratura, rendendo quindi quasi impossibile risalite alle caratteristiche del plaintext a partire dal ciphertext.

```
k = generateKey()
ctBlocks = np.zeros(shape=(len,m), dtype = np.int32)
for i in range(len):
    ctBlocks[i] = encryptHill(ptBlocks[i], k
# ctText = "dnmjupjvxetdeyflcrqzskoweameug"
```

Infine calcolo l'inversa della matrice K modulo 26 e decripto ogni blocco di ciphertext con tale matrice chiamando la funzione *decryptHill*.

```
k1 = np.array(Matrix(k).inv_mod(lenAlphabet))
ptFound = np.zeros(shape=(len, m), dtype = np.int32)
for i in range(len):
    ptFound[i] = decryptHill(ctBlocks[i], k, k1)
# ptFoundTxt = "datasecurityandprivacyaaaaaaaa"
```

Come mostrato in precedenza, con questo algoritmo è complicato svolgere un attacco basato sulle frequenze. Vediamo però che $C = K \cdot P \; mod \, 26$ può essere scritta come $K = C \cdot P^{-1} \; mod \, 26$ nel caso in cui riuscissimo ad ottenere una P invertibile modulo 26. Quindi in questo caso è possibile svolgere un attacco known plaintext in cui possiamo definire la matrice $P^* = [\, P_1 \mid ... \mid P_m \,]$ in cui dispongo sulle colonne i plaintext che ho a disposizione. Verifico che $MCD(\, det(P^*), 26) = 1$ quindi calcolo il prodotto scalare mostrato in precendenza. Questo tipo di attacco è permesso dal fatto che la relazione fra P e C è di

tipo lineare, che quindi è facilmente invertibile se conosciamo entrambe queste matrici.
La funzione *attackHill()* implementata prende in ingresso tutte le coppie di blocchi plaintext-ciphertext disponibili, calcola P^* scegliendo m blocchi di P che rendono P^* invertibile modulo 26, quindi calcola la chiave k come mostrato in precedenza.

```
def attackHill(ptBlocks, ctBlocks):
    pt = np.zeros(shape=(m, m), dtype = np.int32)
    ct = np.zeros(shape=(m,m), dtype = np.int32)
    for i in range(m):
        for j in range(m):
            pt[i][j] = ptBlocks[j][i]
            ct[i][j] = ctBlocks[j][i]
    i = m
    while(math.gcd(int(round(np.linalg.det(pt))), lenAlphabet) != 1 and
         i < len):
        for j in range(m):
            pt[j][i%m] = ptBlocks[i][j]
            ct[j][i%m] = ctBlocks[i][j]
        i += 1
    if not i < len:
        return None
    ptInv = np.array(Matrix(pt).inv_mod(lenAlphabet))

    k = np.dot(ptInv,ct)%lenAlphabet
    return k
```