Ataques a cifrados polialfabéticos

Usando el método de Kasiski y el Índice de Coincidencia

Adrián

Índice

1. Introducción

El objetivo es automatizar la tarea de descifrado de un mensaje encriptado mediante un cifrado polialfabético. La obtención del periodo la hemos realizado para un cifrado polialfabético arbitrario y la obtención del texto plano y la llave nos hemos centrado en el cifrado Vigenère.

En este caso la implementación se ha llevado a cabo en *Python* mediante tres módulos:

period_polialphabetic_cipher.py obtiene el periodo de un cifrado polialfabético

crack_vigenere.py obtiene el texto plano y llave de un cifrado Vigenère

main.py combina los dos módulos anteriores y ofrece una interfaz mediante argumentos por línea de comandos.

Además hemos añadido una implementación del cifrado de Vigenère en *vigenere_cipher.py* con soporte para el alfabeto inglés y español.

2. Obtención del periodo en cifrados polialfabéticos

Dado un texto encriptado mediante un cifrado polialfabético vamos a *automatizar* la obtención del periodo de dicho cifrado. Para ello utilizaremos dos técnicas: el método de Kasiski y el índice de coincidencia. No haremos mucho énfasis en la explicación de estos métodos y nos centraremos principalmente en la implementación.

En *period_polialphabetic_cipher.py*, la clase *PolialphabeticCipher* es la que se encarga de la obtención del periodo. Para no tener problemas, nada más empezar limpiamos el texto, esto es, lo pasamos a mayúsculas y removemos todos los caracteres que no pertenezcan al alfabeto (que dependerá del lenguaje que se haya seleccionado).

```
class PolialphabeticCipher(object):
[...]

def __init__(self,text,language='English'):
    [...]
    self.text = self._clean_text(text)

def _clean_text(self,text):
    """Get the characters that belong to the alphabet"""
    clean_text = ""
    for letter in text.upper():
        if letter in self.alphabet:
            clean_text = ''.join((clean_text,letter))

    return clean_text
```

2.1. Método de Kasiski

Resumidamente, el método de Kasiski consiste en hallar los trigramas del texto cifrado que se repiten y usando la distancia entre las distintas repeticiones obtener información sobre el periodo.

Primero calculamos los trigramas del texto, los ordenamos según el número de repeticiones y eliminamos aquellos que solo aparezcan una vez.

```
def _get_3_grams(self):
    """Get the 3-grams that are repeated with their occurrences"""
    three_grams = {}
    for i in range(len(self.text)-3+1):
        currentgram = self.text[i:i+3]
        three_grams[currentgram] = three_grams.get(currentgram,0) + 1

    three_grams = sorted(three_grams.items(), key=itemgetter(1), reverse=True)
    three_grams = [(gram,occurrences)
                               for gram,occurrences in three_grams
                               if occurrences > 1]

    return three_grams
```

Para cada trigrama, calculamos la distancia que hay entre cada una de sus repeticiones y factorizamos esa distancia, almacenando todos los factores. Una vez recorrido todos los trigramas, contabilizamos todos los factores que han aparecido. Los 5 factores más frecuentes son los posibles periodos del texto cifrado.

```
def kasiski_method(self):
    """Compute the possible periods using Kasiski's method."""
    ngrams = self._get_3_grams()
    if not ngrams:
        print("Kasiski's method failed: no 3-grams found. Aborting...")
        return

    periods = []
    for ngram,occurrences in ngrams:
        next_pos = 0
        for i in range(occurrences-1):
            current_pos = self.text.find(ngram,next_pos)
            next_pos = self.text.find(ngram,current_pos+1)
            distance = next_pos - current_pos

            for period in range(2,distance+1):
                if distance % period == 0:
                    periods.append(period)

    periods = Counter(periods).most_common(5)
    total_occurrences = sum([occurrences for _,occurrences in periods])
    periods = [(period,occurrences/total_occurrences)
                        for period,occurrences in periods]
```

```
        return periods
```

En muchas descripciones del método de Kasiski se hace el máximo común divisor de las distancias entre trigramas para hallar una estimación del periodo. El problema es que si hay una repetición de trigrama debida al azar, el máximo común divisor puede resultar 1 en este caso. Como queremos automatizar el proceso, hemos optado por un método más robusto: elegir como periodos posibles aquellos que aparecen con más frecuencia como factores en las distancia de trigramas.

Nótese que *kasiski_method()* devuelve una lista de hasta 5 periodos y su probabilidad de ser el periodo real. La probabilidad de un periodo la hemos calculado como el cociente entre el número de repeticiones del periodo (como factor de la distancia) entre el número totales de factores.

2.2. Índice de Coincidencia

Para utilizar el índice de coincidencia (IC), necesitamos saber sobre que lenguaje está escrito el texto que fue cifrado. Por ello, en el constructor de la clase *PolialphabeticCipher* fijamos el lenguaje con el que vamos a trabajar:

```
class PolialphabeticCipher(object):
    """Guess the period of a polialphabetic cipher"""
    english_ic = 0.066895
    spanish_ic = 0.076613

    def __init__(self,text,language='English'):
        if language == 'English':
            self.language = 'English'
            self.alphabet = string.ascii_uppercase
            self.language_ic = PolialphabeticCipher.english_ic
        elif language == 'Spanish':
            self.language = 'Spanish'
            self.alphabet = "ABCDEFGHIJKLMNÑOPQRSTUVWXYZ"
            self.language_ic = PolialphabeticCipher.spanish_ic
        else:
            raise ValueError('Language must be English or Spanish')

        self.text = self._clean_text(text)

    [...]
```

Aunque nosotros solo lo hemos hecho para los idiomas inglés y español, añadir uno más es muy fácil: solo hay que añadir el IC asociado al lenguaje y el alfabeto sobre el que trabaja, el programa se encargará del resto.

El método que usa el IC se basa en dos partes: primero compara para distintos periodos el IC medio con el IC del lenguaje y despues compara el IC del texto cifrado con el valor esperado del IC dado un periodo.

En ambos casos utilizaremos el cálculo del índice de coincidencia de un texto:

```
def _ic(self,text):
    """Calculate the index of coincidence"""
    ic = 0
    n = len(text)
    for f in Counter(text).values():
        ic +=  (f*(f-1))/(n*(n-1))

    return float("{0:.6f}".format(ic))
```

2.2.1. Comparando con el IC del lenguaje

Dada una lista de periodos (que pueden ser los obtenidos por el método de Kasiski), calculamos el IC medio para cada periodo de la siguiente forma:

- Dividimos el texto en tantas subsecuencias como valor del periodo p, donde cada subsecuencia i es de la forma

$$c_i c_{i+p} c_{i+2p} c_{i+3p} \cdots$$

- Para cada subsecuencia calculamos el IC y hacemos la media.

```
    def _avg_ic(self,period):
        """Calculate the average of the ic of the period-subsequences"""
        subsequences = [[] for i in range(period)]
        for pos,letter in enumerate(self.text):
            subsequences[pos%period].append(letter)
        subsequences = [''.join(subseq) for subseq in subsequences]
        avg_ic = sum([self._ic(subseq) for subseq in subsequences])/period

        return float("{0:.6f}".format(avg_ic))
```

Comparamos cada IC medio con el IC del lenguaje ya que aquel periodo que tenga un IC medio más próximo, más probable de que sea el periodo real del cifrado polialfabético. Para comparar simplemente hallamos la diferencia entre cada IC medio y el IC del lenguaje y le asociamos a cada periodo la probabilidad de ser el periodo real.

Esta probabilidad la calculamos como el cociente entre el inverso de la diferencia y el inverso de la suma de las diferencias. Utilizamos inversos ya que una diferencia menor esta asociada a una mayor probabilidad y viceversa. Todo esto se recoge en la primera parte del método *ic_ method()*

```
def ic_method(self,periods=None):
    """Compute the possible periods using the index of coincidence"""
    # part 1
    if periods is None:
```

```
        periods = range(1,21)

    avg_ics = []
    for period in periods:
        avg_ics.append( (period,self._avg_ic(period)) )
    difference_respect_language_ic = [(period,abs(avg_ic-self.language_ic))
                                                    for period,avg_ic in avg_ics]
    total_diff = sum([1/diff for period,diff
                                in difference_respect_language_ic])
    p1 = [(period,1/diff/total_diff )
            for period,diff in difference_respect_language_ic]

    # part 2
    [...]
```

Nótese que *p1* es la variable que contiene a la lista de periodos junto con su respectiva probabilidad de ser el periodo real del cifrado polialfabético.

2.2.2. Comparando con el IC del texto

En este caso comparamos el IC del texto con el valor esperado del IC (expIC) para distintos periodos, viendo cual se acerca más.

El cálculo del valor esperado del IC para una cifrado de periodo d se calcula en la siguiente función:

```
def _exp_ic(self,period):
    """Calculate the expected value of the IC for a cipher of period d"""
    d = period
    n = len(self.text)
    return 1/d*(n-d)/(n-1)*self.language_ic + (d-1)/d*n/(n-1)*1/len(self.alphabet)
```

La segunda parte de la función *ic_method()* compara de forma análoga a la primera parte: calcula las diferencias y a cada periodo le asocia una probabilidad utilizando dichas diferencias.

Al final, se combinan las probabilidades obtenidas en la parte 1 y en la parte 2 y se devuelve la lista de periodos junto con su probabilidad.

```
def ic_method(self,periods=None):
    """Compute the possible periods using the index of coincidence"""
    # part 1
    [...]

    # part 2
    text_ic = self._ic(self.text)
    periods_ic = [(period,self._exp_ic(period)) for period in periods]
    difference_respect_text_ic = [(period,abs(period_ic-text_ic))
                                            for period,period_ic in periods_ic]
    total_diff = sum([1/diff for period,diff
```

```
                                in difference_respect_text_ic])
    p2 = [(period,1/diff/total_diff )
            for period,diff in difference_respect_text_ic]

    periods_with_probability = [(period,p1[index][1]+p2[index][1])
                                        for index,period in enumerate(periods)]
    periods_with_probability.sort(key=itemgetter(1),reverse=True)
    total_prob = sum([probability for _,probability
                                    in periods_with_probability])
    periods_with_probability = [(period,probability/total_prob)
                                for period,probability in periods_with_probability]

    return periods_with_probability
```

2.3. Obtención del periodo combinando ambos métodos

Para aprovechar la detección de ambos métodos, primero obtenemos una lista de periodos con el método de Kasiski y le aplicamos a dicha lista el método que utiliza los IC.

Solo nos queda combinar las probabilidades que obtenemos de ambos métodos. En el caso del idioma inglés basta asociarle a ambos métodos la misma ponderación y se obtienen buenos resultados. Sin embargo para el idioma español tras una serie de pruebas hemos visto que el método que usa los IC no siempre arroja buenas resultados y por eso le asignamos un menor peso a la hora de combinar las probabilidades.

Estos cálculos son llevados a cabo por *guess_ period()* junto algunas normalizaciones de probabilidades y transformaciones de probabilidades a porcentajes sobre 100 que son más entendibles para el usuario final. Esta función retorna una lista de hasta 5 periodos con su porcentaje de ser el periodo del cifrado polialfabético. Basta obtener de esta lista el primer elemento para obtener la mejor estimación del periodo.

```
def guess_period(self):
    """Guess the period of the polialphabetic cipher using Kasiski's and IC method"""
    kasiski = self.kasiski_method()
    kasiski.sort()
    if not kasiski:
        return
    periods = [period for period,_ in kasiski]
    ic = self.ic_method(periods)
    ic.sort()

    if self.language == 'Spanish':
        weight = 0.1
    else:
        weight = 1
    guessed_periods = [(period,kasiski[index][1]+weight*ic[index][1])
                                for index,period in enumerate(periods)]

    guessed_periods.sort(key=itemgetter(1),reverse=True)
```

```
        total_prob = sum([probability
                                    for _,probability in guessed_periods])
        guessed_periods = [(period,"{0:.2f}%".format(100*probability/total_prob))
                                   for period,probability in guessed_periods]

        return guessed_periods
```

3. Obtención del texto plano y llave en el cifrado Vigenère

Dado un texto encriptado mediante el cifrado de Vigenère y conocida la longitud de la clave o período, vamos a descifrar el texto. La idea es bien sencilla: si partimos el texto en tantas subsecuencias (donde la subsecuencia i tiene la forma $c_i c_{i+p} c_{i+2p} c_{i+3p} \dots$) como longitud de la llave y en cada subsecuencia conocemos el descifrado de una sola letra, podemos conocer el descifrado de toda la subsecuencia. La explicación es que en cada subsecuencia el cifrado es del tipo César y conociendo el descifrado de un carácter podemos obtener el desplazamiento usado para cifrar.

Nuestro problema se reduce entonces en conocer el descifrado de un carácter. Para ello vamos a utilizar que la distribución de letras de cada subsecuencia debe ser similar a la distribución de letras del lenguaje y vamos a asociar a la letra más común de cada subsecuencia la letra más común del lenguaje.

La clase que va a realizar este trabajo es *CrackVigenere*. Como hemos comentado, necesitamos conocer el lenguaje el cual estaba escrito el texto que fue cifrado. Además para evitarnos problemas, igual que en el caso anterior, limpiamos el texto quedándonos solo con los caracteres que pertenecen al alfabeto en el cual trabajamos.

```
class CrackVigenere(object):
    """Automatic and manual decoder/solver for Vigenere's cipher """
    def __init__(self,text,language='English'):
        if language == 'English':
            self.alphabet = string.ascii_uppercase
            self.language_most_common_letters = "ETAOI"
        elif language == 'Spanish':
            self.alphabet = "ABCDEFGHIJKLMNÑOPQRSTUVWXYZ"
            self.language_most_common_letters = "EAOSR"
        else:
            raise ValueError('Language must be English or Spanish')

        self.original_text = text
        self.text = self._clean_text(text)
```

El método que va a llevar a cabo el descifrado es *decrypt_ text()*. Este método mejora la idea de asociar la letra más frecuente de la subsecuencia a la letra más frecuente del idioma. El problema de esta idea es que mientras que para textos largos suele ser cierto, para textos cortos la letra más frecuente del idioma puede corresponder no a la letra más frecuente de la subsecuencia, sino a la segunda más frecuente o a la tercera, etc...

Para solucionar esto, en primer lugar suponemos que la letra más frecuente del idioma tiene que corresponder a una de las 5 letras más comunes de una subsecuencia dada. Para elegir entre estas 5 letras, las valoramos según el número de *aciertos* que produzcan.

Una letra de la subsecuencia decimos que produce k aciertos si con el desplazamiento obtenido por dicha letra desciframos las 5 letras más comunes de la subsecuencia, entre ellas hay k letras que están en las 5 letras más comunes del lenguaje.

Supondremos que la letra con más *aciertos* en el cifrado esta asociada a la letra más frecuente del idioma y así obtendremos el desplazamiento para descifrar toda la subsecuencia.

Además, *decrypt_text()* tiene una opción de descifrado *manual* en el que va mostrando al usuario en cada subsecuencia las 5 letras más comunes y la probabilidad (en términos de porcentaje) de que cada una de ellas sea el cifrado de la letra más común del alfabeto y le pide al usuario que introduzca manualmente que letra utilizar para descifrar.

```
def decryptText(self,period,manual=False):
    """Decrypt a text encrypted with Vigenere's without knowing the key. """
    subsequences = [[] for i in range(period)]
    for pos,letter in enumerate(self.text):
        subsequences[pos%period].append(letter)
    subsequences = [''.join(subseq) for subseq in subsequences]

    keyword = ""
    subsequences_decrypted = ['' for _ in subsequences]
    for index_subseq, subseq in enumerate(subsequences):
        # mcl(s) = most common letter(s)
        mcls_ciphertext = Counter(subseq).most_common(5)
        mcls_ciphertext.sort(key=itemgetter(1,0),reverse=True)

        mcl_plaintext = self.language_most_common_letters[0]
        encryptions_of_mcl_plaintext = []
        for index_letter,letter in enumerate(mcls_ciphertext):
            encrypted_mcl_plaintext = mcls_ciphertext[index_letter][0]
            offset = (self.alphabet.index(encrypted_mcl_plaintext)
                            - self.alphabet.index(mcl_plaintext))%len(self.alphabet)
            matches = 0
            for letter in mcls_ciphertext:
                pos = self.alphabet.index(letter[0])
                decrypted_letter = self.alphabet[(pos-offset)%len(self.alphabet)]
                if decrypted_letter in self.language_most_common_letters:
                    matches += 1
            encryptions_of_mcl_plaintext.append(
                                        (encrypted_mcl_plaintext,matches))

        encryptions_of_mcl_plaintext.sort(key=itemgetter(1),reverse=True)
        total_matches = sum([matches for _,matches
                            in encryptions_of_mcl_plaintext])
        encryptions_of_mcl_plaintext = [(letter,
```

```
                                    "{0:.2f}%".format(100*m/total_matches))
                                    for letter,m
                                    in encryptions_of_mcl_plaintext]

            if manual:
                print("Possible encryptions of {0} with their probability: {1}".format(
                            mcl_plaintext,encryptions_of_mcl_plaintext))
                while True:
                    encrypted_mcl_plaintext = input(
                            "Encryption of {0}: ".format(mcl_plaintext)).upper()
                    if encrypted_mcl_plaintext in self.alphabet:
                        break
                    else:
                        print("Bad character found. Type a letter.")
            else:
                encrypted_mcl_plaintext = encryptions_of_mcl_plaintext[0][0]

            offset = (self.alphabet.index(encrypted_mcl_plaintext)
                        - self.alphabet.index(mcl_plaintext))%len(self.alphabet)
            keyletter = self.alphabet[offset-1]
            keyword += keyletter
            if manual:
                 print("Key: {0}{1}\n".format(keyword,'?'*(period-len(keyword))))

            subseq_deciphered = subseq
            for pos, letter in enumerate(self.alphabet):
                subseq_deciphered = subseq_deciphered.replace(
                        letter,self.alphabet[(pos-offset)%len(self.alphabet)].lower())

            subsequences_decrypted[index_subseq] = subseq_deciphered
            # end for

        subsequences_decrypted_mixed = zip_longest(
                                        *subsequences_decrypted,fillvalue='')
        subsequences_decrypted_mixed = [''.join(subseq_mix) for subseq_mix
                                        in subsequences_decrypted_mixed]
        plaintext = ''.join(subsequences_decrypted_mixed)
        plaintext = self._rebuilt_text(plaintext,self.original_text)

        return keyword, plaintext
```

4. Modo de uso

Para probar el programa, es recomendable utilizar *main.py* que se encarga de ir llamando a los distintos métodos correctamente y además dispone de una interfaz fácil de utilizar. *main.py* funciona a traves de argumentos por línea de comandos. Con el argumento *–help* podemos ver las opciones que tenemos:

```
[adrian@archPC trabajo]\$ python3 main.py --help
    usage: main.py [-h] [-m] [-spa] [-i INPUT_FILE] [-o OUTPUT_FILE]
```

```
crack a Vigenère's cipher

optional arguments:
  -h, --help            show this help message and exit
  -m, --manual          interacts with the user
  -spa, --spanish       suppose the ciphertext is in Spanish
  -i INPUT_FILE, --input-file INPUT_FILE
                        the input file with the encrypted text
  -o OUTPUT_FILE, --output-file OUTPUT_FILE
                        the output file with the decrypted text
```

Si no se seleccionan las opciones *-i* o *-o*, se utilizará la entrada o salida por consola respectivamente. También dispone de una opción manual donde nos irá mostrando información adicional y el usuario tomará la elección del periodo y la elección del descifrado de la letra más frecuente del idioma. Lo vemos con un par de ejemplos.

5. Ejemplos

5.1. Tears in Rain

Vamos utilizar como texto el monólogo *Tears in Rain* de la película *Blade Runner*:

```
I've seen things you people wouldn't believe.
Attack ships on fire off the shoulder of Orion.
I watched C-beams glitter in the dark near the Tannhauser Gate.
All those moments will be lost in time, like tears...in...rain.
Time to die.
```

Ciframos este texto con la clave *ROY*. Podemos utilizar para ello el programa *vigenere_ cipher.py* o cualquier otra herramienta que trabaje con el alfabeto del inglés. Obtenemos así el texto cifrado:

```
A'kd ktdf igacfk nnm edgekw lnmacf'i awahwkd.
Sissrj kwhhh nf uhjt nxu szt rzdtdsdj de Gghgc.
H opsuwdv R-awplk vkaiswg hf igw szjz mwpq lwd Lpmfwzmhdj Vzlt.
Zda szdrw bnetmlh vaak tt kghs ac sabd, dxjw idsgr...ac...qsxm.
Lxlw in vxd.
```

Ejecutando *main.py* con las opciones por defecto (idioma inglés, modo automático y entrada y salida por consola) nuestro programa es capaz de descifrar el texto automáticamente:

```
[adrian@archPC trabajo]\$ python3 main.py
    If you want to:
            - Read/write from/to a file.
```

```
        - Interact with the decryption process.
        - Decrypt texts that are in Spanish (English by default).
use command-line arguments. For more information, type:
        python3 main.py --help

Introduce the ciphertext: A'kd ktdf igacfk nnm edgekw lnmacf'i
awahwkd. Sissrj kwhhh nf uhjt nxu szt rzdtdsdj de Gghgc. H opsuwdv
R-awplk vkaiswg hf igw szjz mwpq lwd Lpmfwzmhdj Vzlt. Zda szdrw
bnetmlh vaak tt kghs ac sabd, dxjw idsgr...ac...qsxm. Lxlw in vxd.
Key: ROY
i've seen things you people wouldn't believe. attack ships on
fire off the shoulder of orion. i watched c-beams glitter in the
dark near the tannhauser gate. all those moments will be lost
in time, like tears...in...rain. time to die.
```

5.2. La vida es sueño

Vamos a utilizar ahora un texto en castellano bastante conocido: el mónologo de Segismundo en *La vida es sueño*.

```
Sueña el rey que es rey, y vive
con este engaño mandando,
disponiendo y gobernando;
y este aplauso, que recibe
prestado, en el viento escribe,
y en cenizas le convierte
la muerte, ¡desdicha fuerte!
¿Que hay quien intente reinar,
viendo que ha de despertar
en el sueño de la muerte?
Sueña el rico en su riqueza,
que mas cuidados le ofrece;
sueña el pobre que padece
su miseria y su pobreza;
sueña el que a medrar empieza,
sueña el que afana y pretende,
sueña el que agravia y ofende,
y en el mundo, en conclusion,
todos sueñan lo que son,
aunque ninguno lo entiende.

Yo sueño que estoy aqui
destas prisiones cargado,
y soñe que en otro estado
mas lisonjero me vi.
¿Que es la vida? Un frenesi.
¿Que es la vida? Una ilusion,
una sombra, una ficcion,
y el mayor bien es pequeño:
```

```
que toda la vida es sueño,
y los sueños, sueños son.
```

Cifrando este texto con la clave VIDA, obtenemos:

```
odiow no sah uva nw sah, c weei
dlv itpn iñcjrp ijqewvhp,
zqwqlvmfjms z cxffñveñzx;
c foci bmtevox, uva aideki
qñnwuwms, fj no wenqul nwdñqff,
u nq davmawb of yxqwenvua
te nqnvua, ¡mitzqgiw ñyfñci!
¿rqn lbu zyjav mñpnqua aijjjv,
wenqel zyf dj hf znwqaaxbñ
nq fh byfkx hf hj pvaaxf?
odiow no sels fj by sezyfvj,
uva uet ydmewmst hn sgñngf;
odiow no qlkvf ndi qwmida
by nebisej c tq yscñndb;
odiow no rqn e namvbñ npqendb,
odiow no rqn egwve z maiuavhf,
odiow no rqn ehñjzjw h sgavhf,
u nq fh uyñzx, iñ yxqdhdwjlv,
xpzxw tqnrbj ts rqn wpj,
jyñndi ñevkvjx op avxjavhf.

ux wvaws rqn itpxc bndm
eabxbo yvjoqsñab gbñoeel,
h wpkn uva nq ppas foceel
uet hqwpjrisl ui we.
¿zyf ab ob rqhb? qv jsavite.
¿zyf ab ob rqhb? qve jhdwjlv,
yñw bsnxae, vjj jjylmpj,
h im ijcpñ kmfj nw qazyfkx:
uva csew te weme fo byfkx,
c mlb wvawst, odiolb wpj.
```

Ejecutamos *main.py* con la opción *–spanish*, y obtenemos:

```
[adrian@archPC trabajo]\$ python3 main.py --spanish
    Introduce the ciphertext: odiow no sah uva nw sah, c weeidlv
    itpn iñcjrp ijqewvhp,zqwqlvmfjms z cxffñveñzx;c foci bmtevox,
    uva aidekiqñnwuwms, fj no wenqul nwdñqff,u nq davmawb of
    yxqwenvuate nqnvua, ¡mitzqgiw ñyfñci!¿rqn lbu zyjav mñpnqua
    aijjjv,wenqel zyf dj hf znwqaaxbñnq fh byfkx hf hj pvaaxf?odiow
    no sels fj by sezyfvj,uva uet ydmewmst hn sgñngf;odiow no qlkvf
    ndi qwmidaby nebisej c tq yscñndb;odiow no rqn e namvbñ
    npqendb,odiow no rqn egwve z maiuavhf,odiow no rqn ehñjzjw
    h sgavhf,u nq fh uyñzx, iñ yxqdhdwjlv,xpzxw tqnrbj ts rqn wpj,
```

```
jyñndi ñevkvjx op avxjavhf.ux wvaws rqn itpxc bndmeabxbo
yvjoqsñab gbñoeel,h wpkn uva nq ppas foceeluet hqwpjrisl ui
we.¿zyf ab ob rqhb? qv jsavite.¿zyf ab ob rqhb? qve jhdwjlv,yñw
bsnxae, vjj jjylmpj,h im ijcpñ kmfj nw qazyfkx:uva csew te weme
fo byfkx,c mlb wvawst, odiolb wpj.
Key: VIDA
sueña el rey que es rey, y vivecon este engaño mandando,disponiendo
y gobernando;y este aplauso, que recibeprestado, en el viento escribe,
y en cenizas le conviertela muerte, ¡desdicha fuerte!¿que hay quien
intente reinar,viendo que ha de despertaren el sueño de la muerte?
sueña el rico en su riqueza,que mas cuidados le ofrece;sueña el
pobre que padecesu miseria y su pobreza;sueña el que a medrar
empieza,sueña el que afana y pretende,sueña el que agravia y ofende,
y en el mundo, en conclusion,todos sueñan lo que son,aunque
ninguno lo entiende.yo sueño que estoy aquidestas prisiones cargado,
y soñe que en otro estadomas lisonjero me vi.¿que es la vida? un
frenesi.¿que es la vida? una ilusion,una sombra, una ficcion,y el mayor
bien es pequeño:que toda la vida es sueño,y los sueños, sueños son.
```

5.3. La vida es sueño (reducido)

Vamos a utilizar el mismo monólogo de *La vida es sueño*, pero un fragmento mucho menor para que el descifrado automático falle y sea necesaria una intervención manual.

```
¿Que es la vida? Un frenesi.
¿Que es la vida? Una ilusion,
una sombra, una ficcion,
y el mayor bien es pequeño:
que toda la vida es sueño,
y los sueños, sueños son.
```

Cifrando este texto con la clave SUEÑO, obtenemos:

```
¿kpj si ev awst? pr thxijhx.
¿kpj si ev awst? pro xepxweg,
pro iihggp, ñif txvxndc,
r zp aprkw pxxi jh fxmzsdi:
mzs jiyf zp odio tm ñzsdi,
t pdi mpjcem, ñzsdiñ xdc.
```

Ejecutamos *main.py* con la opción *–spanish*, y obtenemos:

```
[adrian@archPC trabajo]\$ python3 main.py --spanish
    Introduce the ciphertext: ¿kpj si ev awst? pr thxijhx.¿kpj si ev awst?
    pro xepxweg,pro iihggp, ñif txvxndc,r zp aprkw pxxi jh fxmzsdi:
    mzs jiyf zp odio tm ñzsdi,t pdi mpjcem, ñzsdiñ xdc.
    Key: SLEÑS
```

```
¿qee eo lk viza? en fñewese.¿qee eo lk viza?
ena elesiln,ena oovbrw, uwa fecmioj,y ñl mwyyr beew es meaueko:
aue pona lw vrda as cueko,i loo seeñls, cuekoc soj
```

No hemos obtenido el texto cifrado completamente pero hemos obtenido una buena aproximación. Observando el texto vemos que la primera palabra parece que es *que* y la cuarta palabra parece que es *vida*, esto es:

$$Dec_2(P) = E \rightarrow Enc_2(E) = Z$$

$$Dec_5(S) = D \rightarrow Enc_4(E) = T$$

Haciendo estos cambios con el análisis manual obtenemos:

```
[adrian@archPC trabajo]\$ python3 main.py --spanish -m
    Introduce the ciphertext: ¿kpj si ev awst? pr thxijhx.¿kpj si ev awst?
    pro xepxweg,pro iihggp, ñif txvxndc,r zp aprkw pxxi jh fxmzsdi:mzs
    jiyf zp odio tm ñzsdi,t pdi mpjcem, ñzsdiñ xdc.
    Possible periods: [(5, '27.30%'), (2, '19.59%'), (4, '19.01%'),
    (10, '17.91%'), (20, '16.19%')]
    Introduce period: 5

    Possible encryptions of E with their probability: [('X', '30.00%'),
    ('I', '20.00%'), ('M', '20.00%'), ('E', '20.00%'), ('R', '10.00%')]
    Encryption of E: X
    Key: S????

    Possible encryptions of E with their probability: [('P', '25.00%'),
    ('I', '25.00%'), ('M', '25.00%'), ('Ñ', '12.50%'), ('V', '12.50%')]
    Encryption of E: Z
    Key: SU???

    Possible encryptions of E with their probability: [('J', '33.33%'),
    ('Z', '16.67%'), ('R', '16.67%'), ('X', '16.67%'), ('A', '16.67%')]
    Encryption of E: J
    Key: SUE??

    Possible encryptions of E with their probability: [('S', '33.33%'),
    ('W', '25.00%'), ('O', '16.67%'), ('H', '16.67%'), ('D', '8.33%')]
    Encryption of E: S
    Key: SUEÑ?

    Possible encryptions of E with their probability: [('X', '22.22%'),
    ('I', '22.22%'), ('P', '22.22%'), ('S', '22.22%'), ('D', '11.11%')]
    Encryption of E: T
    Key: SUEÑO

    Key: SUEÑO
    ¿que es la vida? un frenesi.¿que es la vida? una ilusion,una sombra,
    una ficcion,y el mayor bien es pequeño:que toda la vida es sueño,y
    los sueños, sueños son....
```

```
Try again [y/N]: N
Key: SUEÑO
¿que es la vida? un frenesi.¿que es la vida? una ilusion,una sombra,
una ficcion,y el mayor bien es pequeño:que toda la vida es sueño,y
los sueños, sueños son.
```