UNIVERSIDADE DE COIMBRA

LEI – Tecnologia da Informação

Trabalho Prático 1

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE D
COIMBRA

2019219929 – Alexandre Leopoldo

2019216646 – António Correia

2019216764 – João Monteiro

Índice

Introdução

Este trabalho tem como objetivo desenvolver conhecimentos no âmbito da teoria da informação.

O trabalho aborda as informações adquiridas ao longo de várias semanas, usando como ferramentas as bibliotecas do Python: *numpy*, *matplotlib*, *huffmancodec* e *scipy*.

Exercício 1

```
def histograma(file, alfa, alt):
    plt.title(file)
    plt.bar(alfa, alt, 0.5)
    plt.xlabel("Alfabeto")
    plt.ylabel("Número de Ocorrências")
    plt.show()
```

A função *histograma* que recebe como parâmetros as variáveis *file*, que corresponde ao nome do ficheiro, *alfa* e *alt*, arrays de *numpy* correspondentes ao alfabeto e ocorrências de cada caracter respetivamente.

A função *bar()* da biblioteca *matplotlib* cria o gráfico.

```
def altura(P, alfa):
    dicti = dict(zip(alfa, [0] * len(alfa)))

    for i in P:
        dicti[i] += 1

    return dicti
```

A função *altura* recebe como parâmetros *P* e *alfa*, arrays de *numpy* contendo os conteúdos do ficheiro e o alfabeto correspondente, respetivamente.

O dicionário *dicti* é inicializado com os símbolos presentes em *alfa* e os seus *values* a 0.

É percorrida *P*, incrementado o *value* no *dicti* correspondente ao símbolo lido.

A função devolve um dicionário com cada símbolo de *alfa* e as suas ocorrências em *P*.

```python
def alfabeto(dim, P, file):
    array = []
    extensao = file.split(".")[1]

    if dim != 1:
        return unique(P)

    if extensao == "txt":
        for i in range(26):
            array += chr(97 + i)
        for i in range(26):
            array += chr(65 + i)

        return np.array(array)
    elif extensao == "bmp":
        return np.array(range(256))
    elif extensao == "wav":
        if P.dtype == "float32":
            return np.array(range(-1, 2))
        elif P.dtype == "int32":
            return np.array(range(-2147483648, 2147483649))
        elif P.dtype == "int16":
            return np.array(range(-32768, 32769))
        elif P.dtype == "uint8":
            return np.array(range(256))
```

A função alfabeto recebe como parâmetros *dim*, *P* e *file*, que são, respetivamente, o número de símbolos por índice, a informação e o nome do ficheiro.

Se a dimensão for diferente de 1, ou seja, a informação tem os símbolos agrupados 2 a 2 (como é pedido no ex. 3), é devolvida uma lista com todos os símbolos existentes em *P*.

No caso de a dimensão ser 1 verifica-se a extensão do ficheiro para o qual queremos criar o alfabeto e, dependendo do resultado, devolve o array correspondente:

- Se for um ficheiro de texto, o array possui todas as letras, tanto minúsculas como maiúsculas;
- Se for uma imagem, o array devolvido contém todos os valores entra 0 e 255;
- Se o ficheiro for de som, dependendo do *dtype* desse mesmo, devolve um array correspondente:
    - float32: [-1, 1]
    - int32: [-2147483648, 2147483648]
    - int16: [-32768, 32769]
    - uint8: [0, 255]

Exercício 2

```
def entropia(P, alfa, alt=None):
    H = 0

    if alt is None:
        alt = np.array(list(altura(P, alfa).values()))

    for i in range(len(alfa)):
        if alt[i] != 0:
            H += (alt[i] / len(P)) * math.log2(alt[i] / len(P))

    return round(-H, 4)
```

O limite mínimo teórico para o número medio de bits por símbolo é a entropia da informação.

A função *entropia* calcula a entropia de um conjunto de informação. Recebe como parâmetro uma lista com a informação a ser usada no cálculo, recebe também o alfabeto da informação e tem o parâmetro *alt*, uma lista com o número de ocorrências de cada símbolo que tem como valor predefinido *None*. Caso esta lista não seja recebida, a função calcula-a através da função altura.

A informação é percorrida através do *loop for* e é aplicada a fórmula da entropia.

É devolvido o valor da entropia arredondado com 4 casas decimais.

Exercício 3

```
def lerTexto(file):
    array = []

    with open(file, "r", encoding="UTF-8", errors="ignore") as f:
        for line in f:
            for char in line:
                if 65 <= ord(char) <= 90 or 97 <= ord(char) <= 122:
                    array += char

    return array
```

A função *lerTexto* recebe como parâmetro um ficheiro de texto e tem como objetivo devolver um array com os caracteres existentes no mesmo.

É percorrida cada linha do ficheiro, e consequentemente cada caracter nessa linha, se esse caracter não for um sinal de pontuação caracter com acento ou especial, ou seja, se pertencer ao intervalo de 65 a 90 ou 97 a 122 na tabela *ascii*, então esse caracter é adicionado ao array.

(A condição existente dentro do *with*, " *encoding = "UTF-8", errors = "ignore"* " foi a solução encontrada para no sistema MacOS ser possível ler caracteres que não pertencem ao UTF-8)

```python
def fonte(file, dim):
    extensao = file.split(".")[1]

    if extensao == "txt":
        P = lerTexto(file)
    elif extensao == "bmp":
        P = mpimg.imread(file)
    elif extensao == "wav":
        P = wavfile.read(file)[1]
    else:
        print("Extensão inválida.")
        exit(0)

    if dim == 1:
        return np.array(P).flatten()

    P = np.array(P).flatten()
    if P.size % 2 != 0:
        P = P[:-1]

    array = []

    for i in range(len(P) // 2):
        if extensao != "txt":
            array.append(chr(P[i * 2]) + chr(P[i * 2 + 1]))
        else:
            array.append(P[i * 2] + P[i * 2 + 1])

    return np.array(array)
```

A função fonte tem como objetivo devolver a informação "tratada" de modo a poder ser utilizada pelas funções que a sucedem.

Para tal, analisa a extensão do file (passado como parâmetro) e utiliza o método correto para o abrir e devolve uma lista de uma dimensão com a informação pretendida.

Caso o objetivo seja agrupar a informação 2 a 2, como pedido no ex. 5, se esta não tiver um comprimento de tamanho par retira o último elemento.

Por fim, agrupa os símbolos da informação e devolve um array com estes.

```
def readFile(file, dim):
    if dim > 2 or dim < 1:
        print("Dimensão inválida.")
        return

    informacao = fonte(file, dim)
    alfa = alfabeto(dim, informacao, file)

    alt = altura(informacao, alfa)
    alturas = np.array(list(alt.values()))

    if dim == 1:
        histograma(file, alfa, alturas)
    print(file + ":")
    print("Entropia -> " + str(entropia(informacao, alfa, alturas) / dim))
    print("Número médio de bits por símbolo -> " + str(bitsPorSimbolo(informacao, alt) / dim))
    print("Variância dos comprimentos dos códigos -> " + str(variancia(informacao)))
    print("")
```

A função *readFile* tem como argumentos uma *string*, o nome do ficheiro para ser lido, e a dimensão da informação.

Começa por verificar se a dimensão é 1 ou 2, caso contrário a função termina devolvendo um erro.

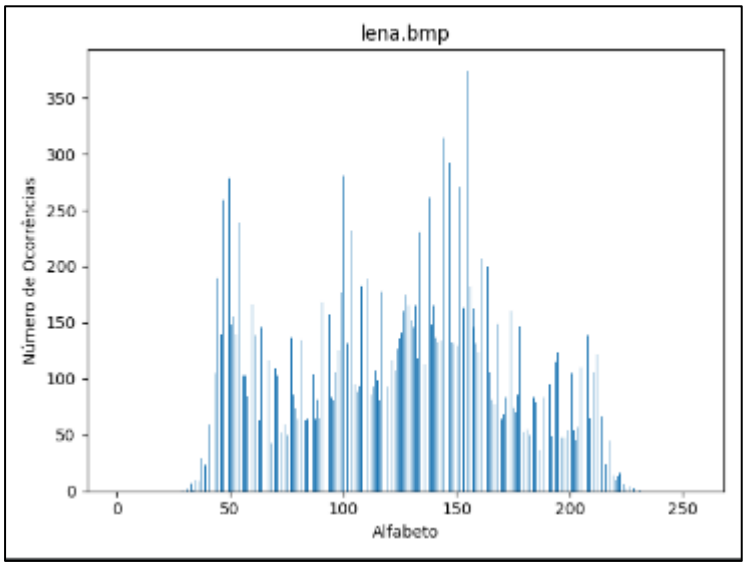
A função chama a função *fonte*, e armazena a informação na sua respetiva variável, e calcula o seu alfabeto.

Calcula também o número de ocorrências de cada símbolo.

Imprime a entropia, número médio de bits por símbolo e a variância dos comprimentos dos códigos, usando as funções anteriormente descritas.

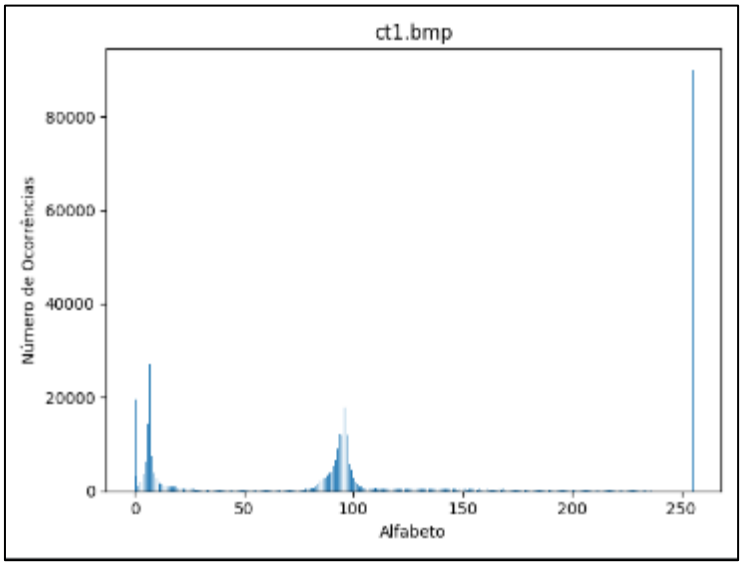
Se a dimensão for 1, também cria o histograma da informação e apresenta-o ao utilizador.

Resultados:



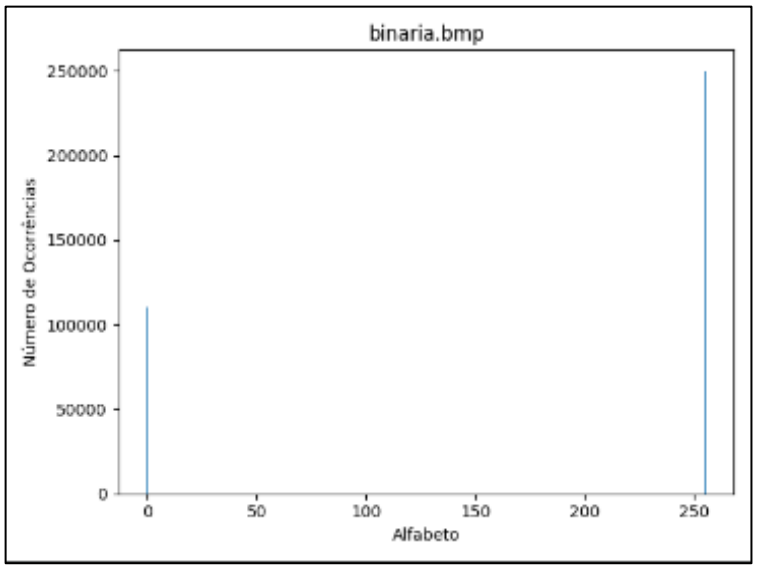
Entropia: 6.9153 bits/pixel

Possuindo apenas tons cinza, apresenta um histograma muito disperso logo a entropia é bastante elevada.



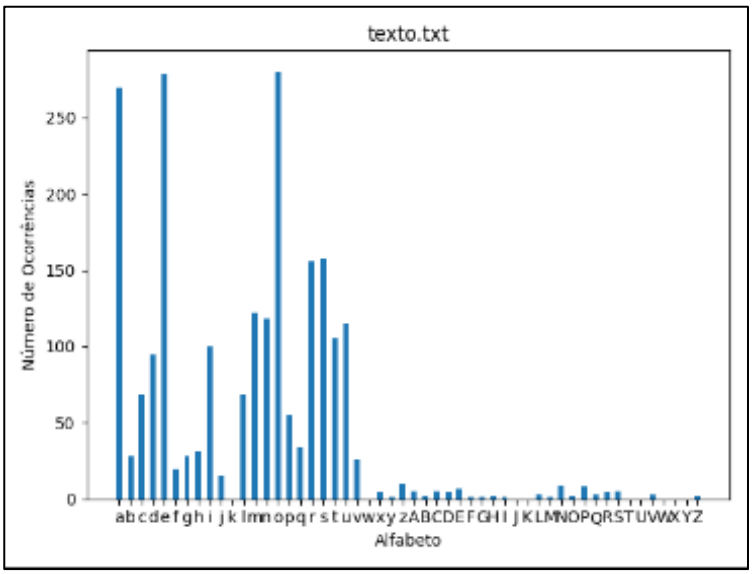
Entropia: 5.2900 bits/pixel

Apresenta um histograma onde a existência de pixéis pretos e de tons cinzentos predomina, desta forma é inferior à anterior.



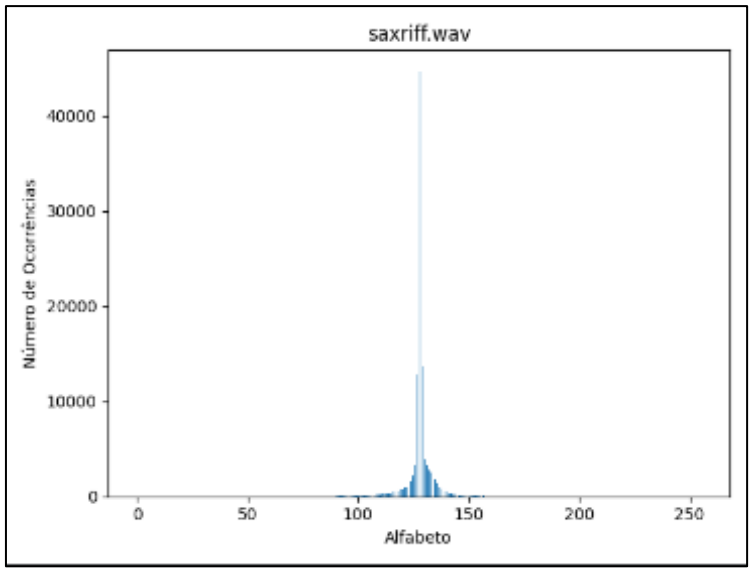
Entropia: 0.8886 bits/pixel

Uma vez que a imagem apenas possui pixéis pretos e brancos, então o histograma apenas apresenta duas barras. Desta forma a entropia é muito reduzida.



Entropia: 4.1969 bits/char

Apesar de alguma dispersão a entropia não é muito alta pois tem um grande alfabeto, sendo maioritariamente preenchido por letras minúsculas.



Entropia: 3.5356 bits/simb

Observa-se que a entropia neste ficheiro é diminuta pois encontra-se extremamente condensada no centro do seu alfabeto.

- **Será possível comprimir cada uma das fontes de forma não destrutiva?**

Para comprimir uma fonte de forma não destrutiva, é necessário reduzir o espaço ocupado pela mesma, de modo a que não seja perdida informação.

Uma vez que a informação está a ser codificada em 8 bits/símbolo, e que os valores calculados da entropia são inferiores a este valor, concluímos que é possível comprimir a fonte de forma não destrutiva.

- **Se Sim, qual a compressão máxima que se consegue alcançar?**

A compressão será máxima no ficheiro com menor entropia, sabendo que:

$$H_{max} = log_2(M)$$

Em que M é o número de elementos do alfabeto. Concluímos que a entropia máxima é igual a 8. Desta forma:

File	Entropia	Taxa de compressão Máxima
lena.bmp	6.9153	13.56%
ct1.bmp	5.2900	33.88%
binaria.bmp	0.8886	88.89%
texto.txt	4.1969	47.54%
Saxriff.wav	3.5356	55.81%

Observamos que a compressão máxima alcançada foi de 88.89%

Exercício 4

```
def bitsPorSimbolo(P, alt):
    bits = 0

    codec = hc.HuffmanCodec.from_data(P)
    symbols, lengths = codec.get_code_len()

    dicti = dict(zip(symbols, lengths))

    for i in dicti.keys():
        bits += alt[i] * dicti[i]

    return round(bits / len(P), 4)
```

Esta função utiliza a biblioteca *huffmancodec* para gerar duas listas *symbols* e *lengths*, as quais possuem respetivamente os símbolos presentes na informação *P*, recebida como parâmetro, e o número de bits necessários para representar cada símbolo.

De seguida cria um dicionário no qual as *keys* são os símbolos e os respetivos *values* são o número de bits necessários para representar esse mesmo símbolo, para não ser necessário estar constantemente a percorrer as listas criadas anteriormente.

Ao percorrer o dicionário, multiplica o número de ocorrências de cada símbolo (obtido da lista *alt* passada como parâmetro) pelo número de bits necessário para o representar e soma tudo para obter o total de bits usados para representar a informação codificada.

Por fim, divide o número anteriormente obtido pelo tamanho da informação para obter o número médio de bits que é necessário para representar cada símbolo.

```
def variancia(P):
    E1 = E2 = 0

    codec = hc.HuffmanCodec.from_data(P)
    symbols, lengths = codec.get_code_len()

    alt = list(altura(P, symbols).values())

    for i in range(len(lengths)):
        E1 += lengths[i]**2 * alt[i] / len(P)
        E2 += lengths[i] * alt[i] / len(P)

    return round(E1 - E2**2, 4)
```

A função *variancia* permite calcular a variância dos *lenghts* dos símbolos presentes na informação, *P*.

Tal como na função *bitsPorSimbolo* é utilizada a biblioteca *huffmancodec* para gerar duas listas *symbols* e *lengths*, que possuem respetivamente os símbolos presentes em *P* e o número de bits necessários para representar cada símbolo.

A variável *alt* através da função altura irá possuir uma lista com as ocorrências de cada símbolo. As três listas *symbols*, *lengths* e *alt* estão organizadas pelo mesmo índice. Desta forma é possível aplicar a fórmula da variância:

$$E[X] = \sum_{i=1}^{\infty} x_i p(x_i)$$

$$\operatorname{var}(X) = \mathrm{E}(X^2) - (\mathrm{E}(X))^2$$

File	Entropia	Huffman	Variância
lena.bmp	6.9153	6.9425	0.6394
ct1.bmp	5.2900	5.3097	7.4876
binaria.bmp	0.8886	1.000	0.000
texto.txt	4.1969	4.2173	1.8827
Saxriff.wav	3.5356	3.5899	7.7501

É possível concluir que os códigos de *Huffman* possuem valores pouco maiores que o limite mínimo teórico, não o conseguindo alcançar. Assim, apesar de preciso não é totalmente eficiente.

- **Será possível reduzir-se a variância?**

Para obter variância reduzida num código de *Huffman* é necessário colocar os símbolos combinados na lista usando a ordem mais elevada possível, ou seja, a variância será mínima quando, na criação da árvore, é dada prioridade aos símbolos com menor comprimento.

- **Se sim, como pode ser feito e em que circunstância será útil?**

Sim, pode ser feito. O agrupamento de dois símbolos deve ser colocado o mais perto da raiz. Será útil em situações onde a informação é constituída por uma grande quantidade de símbolos com probabilidades semelhantes.

Exercício 5

O exercício 5, faz uso da função *fonte*, anteriormente descrita no exercício 3, para agrupar os elementos da informação dois a dois.

File	Entropia	Entropia Agrupada
lena.bmp	6.9153	5.5965
ct1.bmp	5.2900	3.4861
binaria.bmp	0.8886	0.6919
texto.txt	4.1969	3.7543
Saxriff.wav	3.5356	2.4706

Desta forma, por análise dos valores obtidos da entropia agrupada dois a dois, concluímos que ocorre uma otimização do código uma vez que estes valores são menores que os calculados da entropia anterior.

Exercício 6a)

```
def infoMutua(query, target, step):
    array = []
    h_x = entropia(query, np.unique(query))

    for i in range(math.ceil((len(target) - len(query) + 1) / step)):
        window = target[i * step: i * step + len(query)]
        h_y = entropia(window, np.unique(window))

        c = np.c_[window, query]
        new_c = []
        for i in c:
            new_c.append(chr(i[0]) + chr(i[1]))

        h_c = entropia(new_c, unique(new_c))

        array.append(round(h_x + h_y - h_c, 4))

    return array
```

Sabendo que a informação mútua é calculada usando a fórmula:

$$I(X,Y) = H(X) - H(X|Y)$$

e que

$$H(X|Y) = H(X,Y) - H(Y)$$

então,

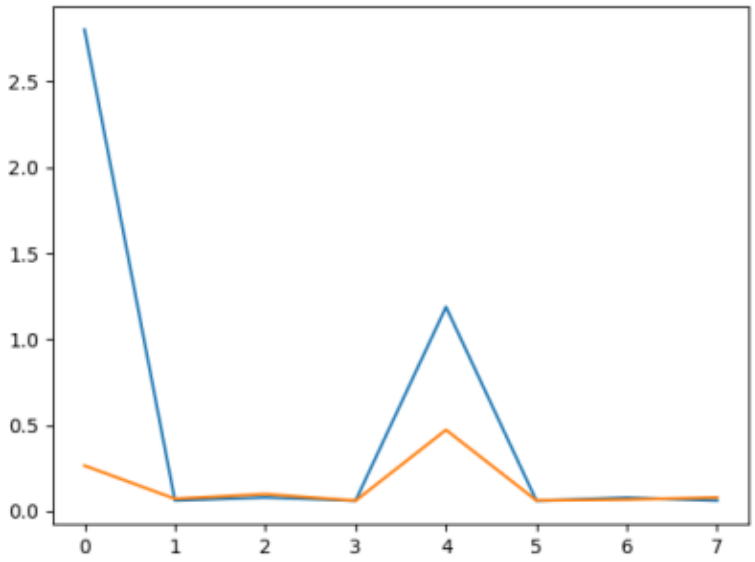
$$I(X,Y) = H(X) + H(Y) - H(X,Y)$$

A função inicia-se por calcular a entropia da *query*.

De seguida, é percorrido o *target* segundo o *step* enviado como parâmetro. É calculado a entropia de cada *window* do *target*, a entropia conjunta da *query* e da *window* e por fim a informação mútua destes.

A função termina devolvendo um array contendo estes valores da informação mútua.

Exercício 6b)



- Valores da informação mútua entre saxriff.wav e target01 - repeat.wav
- Valores da informação mútua entre saxriff.wav e target02 - repeatNoise.wav

Utilizando a função previamente definida, foram obtidos os resultados:

Informação Mútua (target01 - repeat.wav) -> [2.8027, 1.1893, 0.8595, 0.6931, 0.5989, 0.5394, 0.5019, 0.4788]

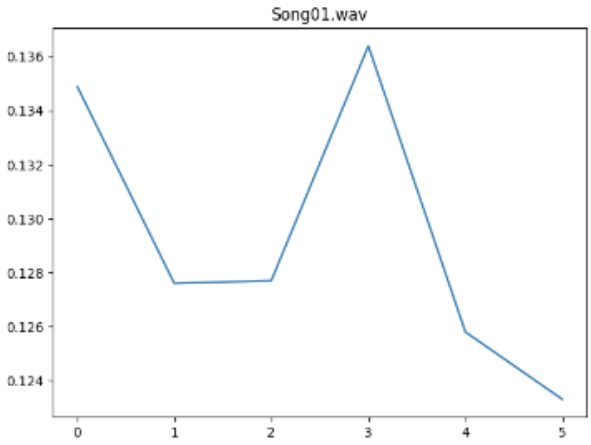
Informação Mútua (target02 - repeatNoise.wav) -> [0.2647, 0.2937, 0.2714, 0.2339, 0.2043, 0.1881, 0.1763, 0.1702]

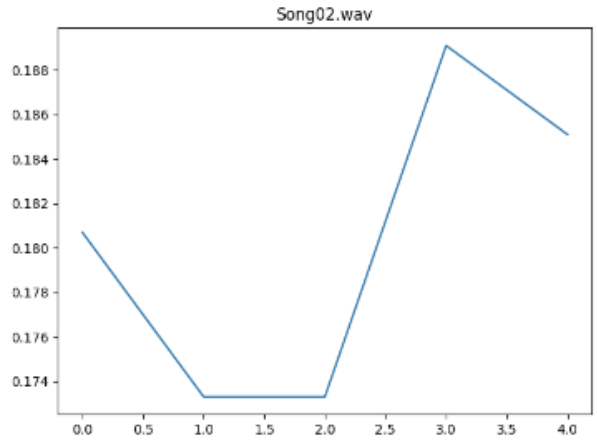
Podemos então concluir que o ficheiro áudio saxriff.wav e target01 -repeat.wav possuem maior informação mútua. Desta forma é possível determinar que ao adicionar ruido ao áudio, a informação comum entre ambos diminui.

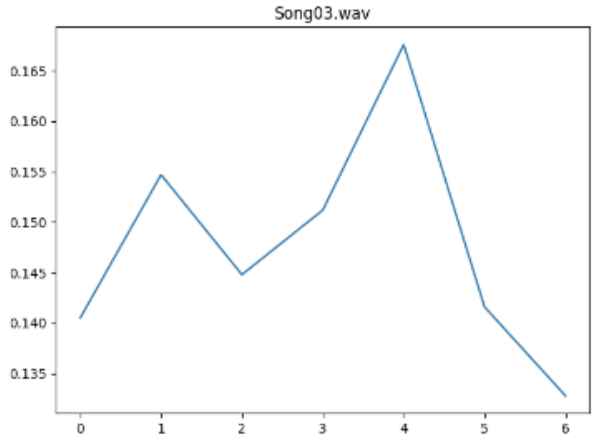
Exercício 6c)

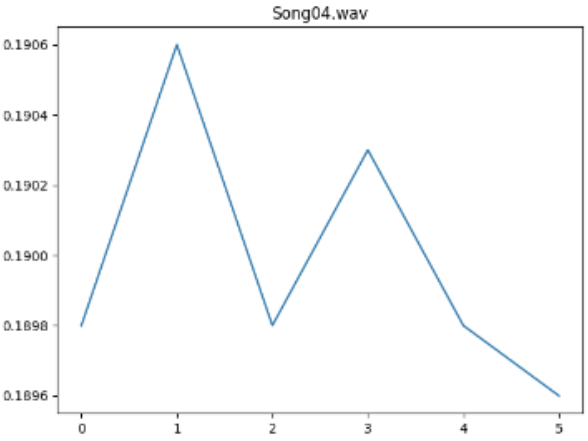
File	Informação mútua máxima
Song06.wav	3.5310
Song07.wav	3.5310
Song05.wav	0.5323
Song04.wav	0.1906
Song02.wav	0.1891
Song03.wav	0.1676
Song01.wav	0.1364

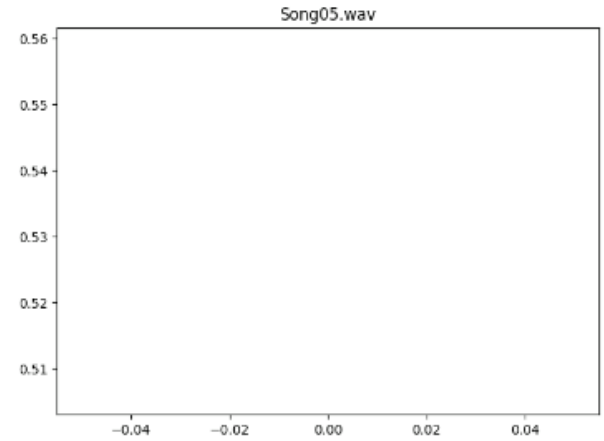
Desta forma podemos concluir que os ficheiros mais parecidos ao ficheiro saxriff.wav são os Song06.wav e Song07.wav, que na verdade são o próprio saxriff.wav com ruido.

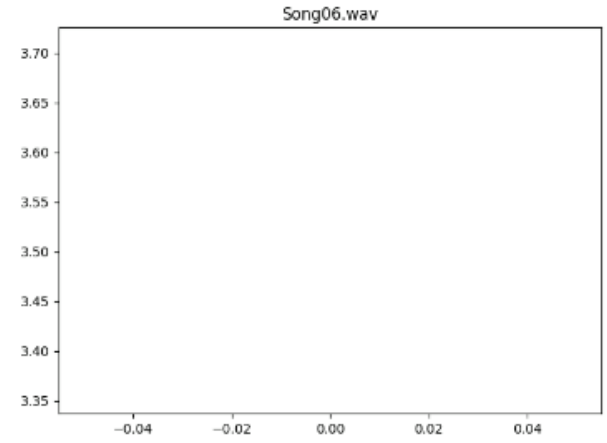

Song01.wav
0.136
0.134
0.132
0.130
0.128
0.126
0.124
0
1
2
3
4
5

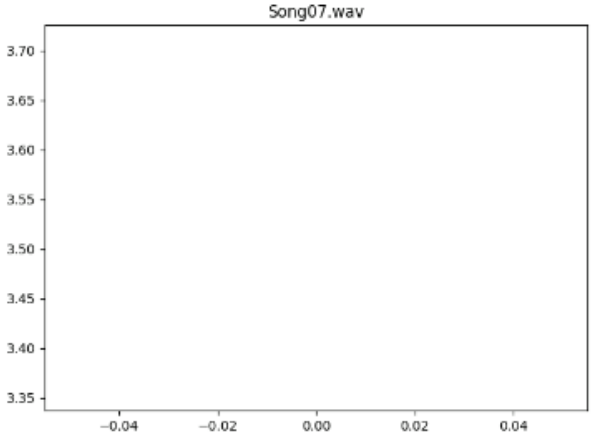

Song02.wav
0.188
0.186
0.184
0.182
0.180
0.178
0.176
0.174
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0


Song03.wav
0.165
0.160
0.155
0.150
0.145
0.140
0.135
0
1
2
3
4
5
6


Song04.wav
0.1906
0.1904
0.1902
0.1900
0.1898
0.1896
0
1
2
3
4
5


Song05.wav
0.56
0.55
0.54
0.53
0.52
0.51
−0.04
−0.02
0.00
0.02
0.04


Song06.wav
3.70
3.65
3.60
3.55
3.50
3.45
3.40
3.35
−0.04
−0.02
0.00
0.02
0.04


Song07.wav
3.70
3.65
3.60
3.55
3.50
3.45
3.40
3.35
−0.04
−0.02
0.00
0.02
0.04

Conclusão

Com este trabalho foram desenvolvidas capacidades no tratamento de dados, cálculo de entropia. Aprofundamos também o conhecimento da biblioteca *numpy*.

Concluímos também que, em certos casos, agrupando os símbolos dois a dois é possível reduzir ainda mais a entropia. A utilização de códigos *Huffman* apesar de alcançarem valores muitos próximos do limite mínimo teórico possuem algumas limitações.

Foi possível também aplicar conteúdos teóricos em situações práticas.