Universidade de Coimbra
Faculdade de Ciências e Tecnologia
Licenciatura em Engenharia Informática
Teoria da Informação – 2019/2020
2º Ano – 1º Semestre

Relatório
Trabalho Prático nº1

Entropia, Redundância e Informação Mútua

Coimbra, 6 de novembro de 2020

2018285164	Eduardo F. F. Cruz	eduardo.cruz@student.uc.pt
2019216314	Gonçalo Marinho Barroso	uc2019216314@student.uc.pt
2017263654	Inês Ferreira Mendes	inesmendes@student.dei.uc.pt

Introdução

Neste trabalho, realizado no âmbito da cadeira de Teoria da Informação, focámo-nos em alguns tópicos fundamentais de teoria da informação, nomeadamente, informação, redundância, entropia e informação mútua. Recorremos à linguagem de programação Python e, fazendo uso dos módulos *numpy*, *mathplotlib* e *scipy,* importámos, processámos e analisámos os dados de ficheiros de vários formatos (.txt, .bmp e .wav), de acordo com o requisitado no enunciado do Trabalho-Prático 1.

Breve descrição da solução

Começámos por importar os módulos, já mencionados acima, necessários à resolução do trabalho. Foram então desenvolvidas três funções, *loadTxtData*, *loadImgData* e *loadAudioData,* que foram utilizadas para carregar os dados de ficheiros de texto, image e áudio, respetivamente, para arrays (*numpy*). Calculámos um Alfabeto para cada Fonte de dados através da análise do tipo e do número de bits (*dtype*) dos dados da correspondente Fonte. A partir dos dados importados dos ficheiros e guardados em arrays (Fonte) e dos respetivos Alfabetos gerados, recorrendo à função *getOccurrenceData,* desenvolvida por nós, calculámos e obtivemos o número de amostras consideradas, da Fonte, e um array onde cada índice possuí o registo de ocorrência de cada símbolo do Alfabeto, na Fonte. Através do Alfabeto e do array com o registo de ocorrências, e a partir da função *viewHistogram*, obtivemos o gráfico da distribuição estatística (histograma) de cada Fonte. Foi desenvolvida a função *getEntropy* para calcular o valor das Entropias das várias Fontes. Esta função recebe um array com o registo de ocorrências e o valor do número de amostras a considerar da Fonte, e devolve o valor da Entropia e um array que contém a probabilidade para cada símbolo desse Alfabeto.

Foi-nos disponibilizado o ficheiro 'huffmancodec.py', com funções de codificação de Huffman, que importámos como módulo para termos acesso direto a essas funções. Criámos a função *huffmanCodeAverageLength* que permite, utilizando os métodos *from_data* e *get_code_len* de 'huffmancodec.py', obter um array com os comprimentos (número de bits) dos códigos de Huffman gerados para cada símbolo do Alfabeto, que, juntamente com o array das probabilidades obtido anteriormente, é necessário ao cálculo do número médio de bits por símbolo codificado com o codec de Huffman, para cada uma das fontes de informação, que é devolvido pela função. Com este valor médio e com a Entropia calculada anteriormente é obtida a redundância do

código de Huffman para cada Fonte, o que nos permite medir a eficiência da codificação de Huffman para a respetiva Fonte.

Para visualizarmos o efeito do agrupamento de símbolos na Entropia desenvolvemos uma função *pairData* que recebe uma Fonte e o respetivo Alfabeto, converte cada par se símbolos num só equivalente a (1ºsímbolo*tamanhoDoAlfabeto+2ºsímbolo) e devolve tanto um array com os símbolos resultantes desse processo de emparelhamento (nova Fonte) como o novo Alfabeto para os símbolos emparelhados. A forma como o emparelhamento é feito dependendo do tipo dos dados da Fonte passada como argumento (há dois casos possíveis: serem caracteres, ou não). De seguida é calculado o array com as ocorrências de cada símbolo, do novo Alfabeto, na Fonte resultante, com a função *getOccurrenceArray* já utilizada anteriormente, e através desse array obtiveram-se os valores de Entropia.

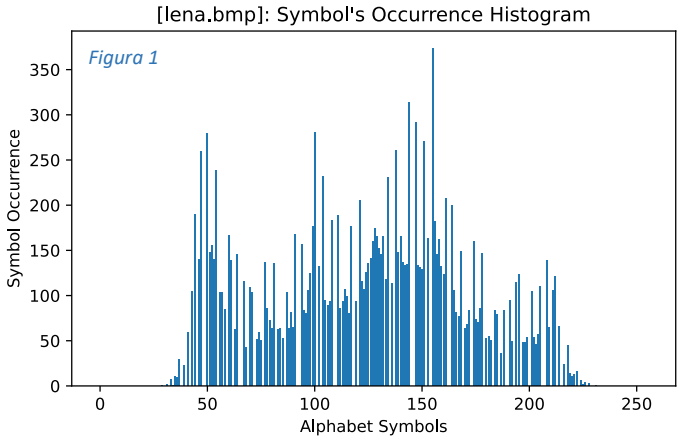
No exercício 6, alínea b), importámos os dados dos dois ficheiros áudio a serem utilizados como target, recorrendo à função *loadAudioData.* Foi desenvolvida a função *getMutualInfoArray* que devolve um array com a evolução da Informação Mútua para um dado um query (neste caso, 'saxriff.wav') e target. Nesta função é aplicada uma janela deslizante no target e é calculada a Informação Mútua entre o conjunto de dados dessa janela e o query (que possui o mesmo tamanho da janela). Para calcular este valor da Informação Mútua é utilizada outra função, *getWindowMutualInfo*. Nesta função foram percorridos os arrays com amostras da query e com amostras da janela de target, e, em cada iteração foi incrementada a ocorrência conjunta na matriz de ocorrências conjuntas. Com os valores já calculados da entropia da query, da janela e com a entropia conjunta é devolvida por essa função, *getWindowMutualInfo,* a Informação Mútua para a janela passada como argumento. Após terem sido obtidos os arrays com a evolução da Informação Mútua, foram calculados os arrays com os instantes de tempo para cada valor de Inf. Mútua. Destes dois arrays resultaram dois gráficos representativos da evolução da Inf. Mútua ao longo do tempo.

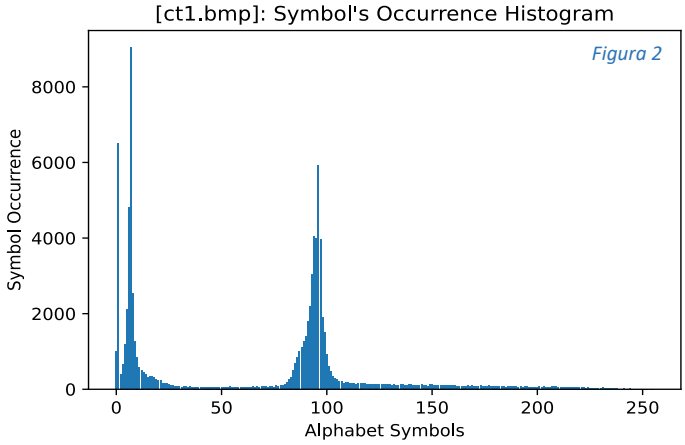
No exercício 6, alínea c), foi implementada uma função *musicIdentifier* onde utilizámos as funções descritas para a alínea b) para obtermos o array com evolução da Inf.Mútua para cada um dos sete sons importados. A estes arrays aplicámos a função do *numpy*, *max,* que nos permitiu obter e consequentemente guardar num array, os valores máximos de Informação Mútua para cada target diferente.

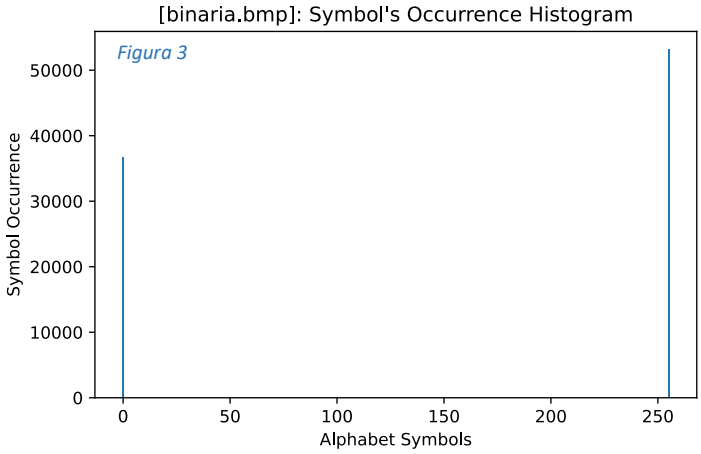
De notar que, nos casos em que o som era stereo, foram consideradas apenas as amostras do canal esquerdo de cada ficheiro.

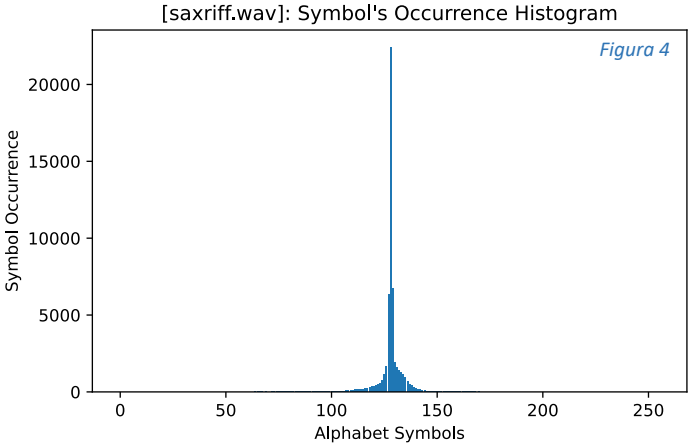
Resultados e Observações

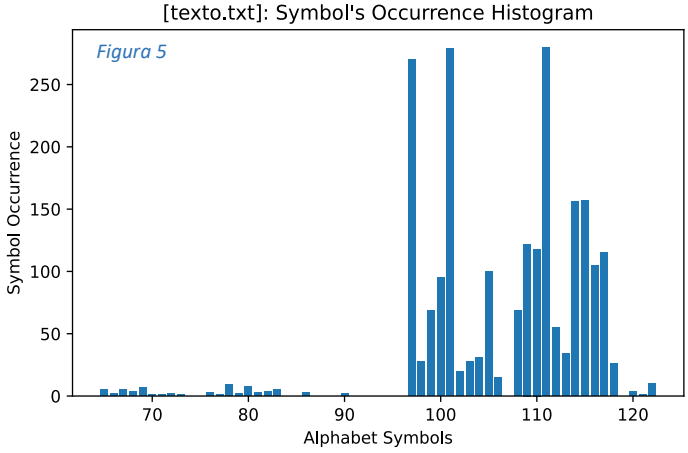
Usando as funções desenvolvidas nas alíneas 1) e 2) obtivemos os Histogramas (distribuição estatística) seguintes que nos permitem visualizar o número de ocorrências, na Fonte, dos vários símbolos pertencentes ao Alfabeto. Destas alíneas obtivemos vários resultados, entre eles, a Entropia (limite mínimo para o número médio de bits por símbolo) para cada uma das Fontes de dados, apresentados na Tabela 1, página 5.











Número de bits por símbolo (sem qualquer codificação) de acordo com o tipo de dados dos ficheiros:

- .bmp : 8 bits/símbolo
- .wav : 8 bits/símbolos
- .txt (ASCII): 8 bits/símbolo

(calculados a partir do atributo *dtype*, inerente a objetos do *numpy*)

Figura	Ficheiro	Entropia	Taxa de Compressão
1	**lena.bmp**	**6.915** ≈ 7 bits/símbolo	(1-6.915/8)*100= **13.6%**
2	**ct1.bmp**	**5.972** ≈ 6 bits/símbolo	(1-5.972/8)*100= **25.4%**
3	**binaria.bmp**	**0.976** ≈ 1 bits/símbolo	(1-0.976/8)*100= **87.8%**
4	**saxriff.wav**	**3.531** ≈ 4 bits/símbolo	(1-3.531/8)*100= **55.9%**
5	**texto.txt**	**4.197** ≈ 5 bits/símbolo	(1-4.197/8)*100= **47.5%**

Tabela 1

Observando os histogramas, para cada Fonte, conseguimos perceber que existem símbolos que são (muito) mais prováveis de ocorrer do que outros e sabemos que a informação para codificar um símbolo é tanto menor, quanto maior for a probabilidade desse símbolo ocorrer , daí e tendo em conta que a entropia veiculada na Fonte equivale à informação média, podemos esperar que dos histogramas cuja distribuição é mais uniforme resultem entropias maiores, e vice versa. Ou seja, a entropia pode ser utilizada, por exemplo, como uma medida de incerteza ou como uma medida de dispersão da distribuição. Estas conclusões são facilmente interpretadas nos resultados que obtivemos.

O ficheiro 'lena.bmp' consiste numa imagem em tons de cinza onde cada símbolo é representado com 8 bits, ou seja, temos um Alfabeto que compreende valores inteiros de 0 a 255, inclusive (2**8=256 símbolos). Visualizando o respetivo histograma, constatamos que existe grande dispersão que, como pudemos confirmar pelo resultado obtido, se traduziu num valor de Entropia mais elevado (6.915 bits/símbolo). Por ser a Fonte de dados com a distribuição mais uniforme (comparativamente com as restantes quatro Fontes analisadas), é, consequentemente, aquela que possui o maior limite mínimo (teórico) para o número médio de bits por símbolo codificado.

O Alfabeto da Fonte do ficheiro 'ct1.bmp', à semelhança do ficheiro 'lena.bmp', contém os valores inteiros de 0 a 255. No entanto, esta imagem em tons de cinza tem 4 canais (rgba); cada píxel tem um valor comum aos 3 canais rbg, logo, na implementação do código, por uma questão de eficiência computacional, considerámos apenas o canal r. O canal alfa foi ignorado já que é irrelevante para obter os resultados pretendidos com este trabalho. Obtivemos o histograma da Figura 2 onde nos apercebemos da existência de 3 picos mais salientes que correspondem a 3 símbolos (aprox.) do Alfabeto que têm uma probabilidade maior de ocorrer na Fonte (a cor preta, e outros dois tons de cinzento que se distinguem bem

na visualização da imagem). Por observação percebemos que existe uma maior variância do número de ocorrências nesta Fonte, comparativamente com a Fonte 'lena.bmp'. Este é o fator chave que justifica a diferença de 1 bit/símbolo entre a entropia de 'lena.bmp' e de 'ct1.bmp'.

Na imagem 'binaria.bmp' estão presentes apenas duas cores, o branco e o preto. À semelhança do ficheiro 'ct1.bmp', a imagem possui 4 canais, logo processámo-la da mesma forma (considerámos o canal r e ignorámos os restantes). Por intuição chegamos imediatamente à conclusão de que esta imagem poderá ser codificada utilizando apenas 1 bit/símbolo (0 para cor preta, 1 para cor branca, por exemplo). Para corroborar esta afirmação, calculámos e obtivemos a Entropia de 0.976 bits/símbolo. Pode colocar-se a questão: "Porque razão não se obteve um valor de Entropia exatamente igual a 1bit/símbolo?", a resposta é, porque embora próxima, a Probabilidade de ocorrer píxel preto é diferente da Probabilidade de ocorrer píxel branco; se as duas probabilidades fossem iguais a 0.5 teríamos a Entropia exatamente igual a 1 bit/símbolo. De notar que a pior situação, em qualquer caso, é considerar que todos os símbolos têm igual probabilidade de ocorrer. A melhor situação seria considerar que não existe incerteza nenhuma, aí teríamos Entropia igual a 0bits/símbolo.

O áudio saxriff.wav, teoricamente, poderá ser codificado, no mínimo, com um número médio de 3.531 bits/símbolo o que se traduziria numa taxa de compressão de 55.9%. O pico visível no histograma representa o silencio existente no áudio.

A Fonte 'texto.txt' contém vários caracteres ASCII. O seu Alfabeto consiste nos valores inteiros dos caracteres ASCII do conjunto [A-Za-z]. Os caracteres que não pertencem a [A-Za-z] foram descartados da Fonte. No histograma da Figura 5 percebemos que os caracteres [a-z] (minúsculos) ocorrem mais frequentemente que os caracteres [A-Z] (maísculos). A Fonte, ao ser codificada por um codec como o de Huffman, onde os símbolos mais prováveis de ocorrer são codificados com um menor número de bits e os menos prováveis de ocorrer são codificados com mais bits, terá os caracteres do conjunto [a-z] codificados com um menor número de bits e os do conjunto [A-Z] com um maior número de bits.

Após a análise dos resultados por nós obtidos, podemos afirmar que, todas as Fontes consideradas anteriormente podem ser comprimidas de forma não-destrutiva. Isto é possível para cada uma das Fontes porque os valores obtidos de Entropia são inferiores ao número médio de bits por símbolo sem qualquer codificação. A compressão máxima é alcançada quando se consegue codificar a Fonte com um número médio de bits/símbolo igual ao valor da Entropia veiculada. Assim, a compressão máxima que se consegue alcançar para cada Fonte foi calculada e registada na última coluna da Tabela 1.

A Tabela 2 foi preenchida com os resultados obtidos na resolução do exercício 4. Observando esses resultados verificamos que os valores calculados para o número médio de bits por símbolo codificado através do codec Huffman, se aproximam bastante dos valores calculados anteriormente para a Entropia. Como seria de esperar, todos esses valores, para cada Fonte, são superiores ao das respetivas Entropias. Com estes valores calculámos também o valor da redundância pois permite, de certa forma, medir a eficiência da codificação realizada.

Ficheiro	Entropia bits/símbolo	Nº médio de bits/símbolo codificado com Huffman Codec	Variância Codificação Huffman	Redundância Codificação Huffman
lena.bmp	**6.915**	**6.943** bits/símbolo	**0.639**	**0.027**
ct1.bmp	**5.972**	**6.008** bits/símbolo	**5.202**	**0.035**
binaria.bmp	**0.976**	**1.000** bits/símbolo	**0.0**	1-0.976= **0.024**
saxriff.wav	**3.531**	**3.584** bits/símbolo	**7.718**	**0.053**
texto.txt	**4.197**	**4.217** bits/símbolo	**1.883**	**0.020**

Tabela 2

A variância dos códigos de Huffman depende da forma como os símbolos (nós) são (re)ordenados na árvore binária de Huffman, a cada "iteração"/passo. Dependendo do processo de ordenação utilizado, podemos obter códigos de Huffman completamente diferentes para a mesma Fonte de dados. Para conseguirmos códigos de Huffman de variância mínima, é necessário posicionar os símbolos combinados no lugar mais elevado possível da lista, o que no final resultará numa árvore binária de Huffman com menor profundidade. Comparando um código de Huffman de variância mínima com outro código de Huffman obtido de maneira diferente, para a mesma Fonte, percebemos que são idênticos em termos de redundância, ou seja, o comprimento médio do código permanece inalterável, no entanto, a variância dos dois códigos é significativamente diferente. É efetivamente possível diminuir a variância utilizando o processo descrito acima.

Tenhamos em consideração o protocolo TCP. Este protocolo de transporte possuí vários mecanismos de controlo de fluxo e congestão, entre os quais, a aplicação de um buffer de tamanho limitado. O objetivo deste buffer é atenuar a variação do bit rate, ou seja, seria conveniente que a taxa de injeção de bits fosse aproximadamente constante, logo, quanto maior for a variância dos códigos mais complicada se torna esta limitação do buffer. Este é claramente um exemplo de uma situação em que seria útil ter códigos de Huffman de variância mínima.

Na Tabela 3, abaixo, apresentamos os resultados obtidos para as Entropias das Fontes importadas dos diversos ficheiros, cujos símbolos foram agrupados 2 a 2. A partir da observação desta Tabela torna-se evidente a vantagem de estudar várias variáveis em simultâneo, ou seja, em agrupamento, dado que permite, geralmente, obter um valor de Entropia inferior ao obtido para a Fonte sem a aplicação de qualquer agrupamento de símbolos. Esta condição só não se verificaria se os casos fossem independentes. É fundamental ter em consideração as implicações inerentes ao agrupamento de símbolos. Do agrupamento dos símbolos 2 a 2 resulta um novo Alfabeto cujo número de símbolos (emparelhamentos) passa a ser $(2^8)^2$, sendo 2^8 o número de símbolos únicos (não

emparelhados). Generalizando, se o Alfabeto tiver *m* símbolos, o agrupamento dos símbolos *n* a *n* implica um novo Alfabeto com m^n símbolos, ou seja, para concluir, o número de símbolos do Alfabeto cresce exponencialmente o que implica maiores necessidades de memória, o que eventualmente poderá tornar esta técnica inviável.

Ficheiro	Entropia sem agrupamento bits/símbolo	Entropia com agrupamento bits/símbolo
lena.bmp	**6.915**	**5.597**
ct1.bmp	**5.972**	**4.481**
binaria.bmp	**0.976**	**0.542**
saxriff.wav	**3.531**	**2.890**
texto.txt	**4.197**	**3.734**

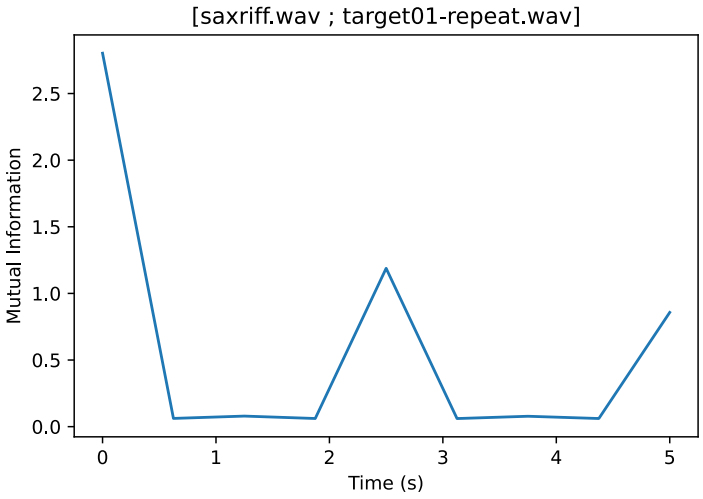
Tabela 3

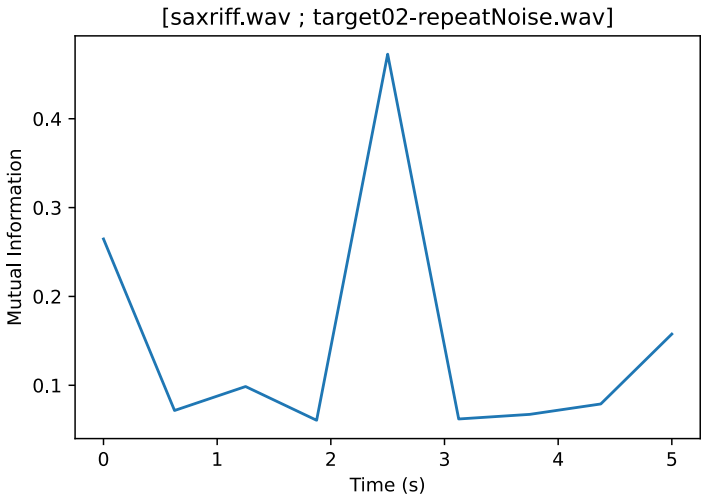
Para o exercício 6 considerámos o ficheiro 'saxriff.wav' como query (sinal a pesquisar). Em particular para a alínea b) determinámos a variação da informação mútua entre o query e os targets 'target01-repeat.wav' e 'target02-repeatNoise.wav', aplicando um passo de ¼ do comprimento do query, no deslize da janela.

Do cálculo da Inf. Mútua entre o query e o target01, para cada janela, resultou o seguinte array: [**2.803 0.062 0.079 0.061 1.188 0.061 0.078 0.061 0.857**].

Já para o target02 obtivemos: [**0.265 0.072 0.099 0.061 0.473 0.062 0.067 0.079 0.158**]

A seguir apresentamos os gráficos de evolução da informação mútua ao longo do tempo para cada target, respetivamente:





Para melhor entendermos e interpretarmos estes dados, devemos começar por reproduzir e ouvir cada um deles. Ao ouvirmos o áudio do ficheiro 'target01-repeat.wav' percebemos que consiste numa repetição consecutiva, três vezes, do aúdio de 'saxriff.wav' (query). Sabemos que a janela deslizante utilizada tem o tamanho do query, daí, seria de esperar três picos (dado que o target é aproximadamente o query repetido 3 vezes), que foi exatamente o que obtivemos no gráfico. Sendo a Informação Mútua calculada para uma janela deslizante de valores, os picos traduzem o momento em que as amostras do intervalo da janela deslizante do target apresentam a maior similaridade com as amostras do query. Por observação do gráfico e tendo em consideração a maneira como a janela desliza ao longo do target, conseguimos perceber com exatidão a razão de dada evolução do gráfico ao longo do tempo. Entre as três repetições do query que perfazem o target existem dois momentos de silêncio que estão claramente manifestados no gráfico entre os intervalos (aproximados) de tempo [0.6-1.9] e [3.1-4.4]. Este silêncio justifica estes dois intervalos em que o valor da informação mútua se mantém baixa e constante. O comportamento deslizante da janela justifica a descida, ou crescimento, gradual que está explícito no gráfico. O áudio 'target02-repeatNoise.wav' é constituído por três partes, a primeira e a última parte são, ao ouvido, semelhantes a 'saxriff.wav' com ruído e a parte do meio é igual mas sem ruído, ou seja, a única diferença de target02 para target01 é o ruído introduzido na primeira e última parte do aúdio. Assim, por observação do gráfico percebemos que a presença de ruído afeta significativamente o valor obtido para a Informação Mútua, mas, não nos impossibilita conseguir distinguir os instantes do início da janela de target para os quais as amostras apresentam maior similaridade com as da query, ou seja, mesmo existindo ruído presente nas amostras, através do cálculo da Informação Mútua, conseguimos identificar a presença da query no target. O pico do meio é tão acentuado em relação aos outros dois pois corresponde à parte do áudio equivalente a 'saxriff.wav', sem ruído (audível).

Na alínea c) desenvolvemos um pequeno simulador de identificação de música com base nas conclusões retiradas da alínea acima (utilizámos a informação mútua para detetarmos similaridades entre músicas, neste caso). Utilizando o ficheiro 'saxriff.wav' como query e os ficheiros Song*.wav como targets, determinámos a evolução da informação mútua e de seguida calculámos o valor de Informação Mútua máxima para cada um destes ficheiros. Os resultados obtidos foram os seguintes:

==========Results for saxriff.wav==========
[Song07.wav]: Informação Mútua Máxima de **3.531** bits/símbolo
[Song06.wav]: Informação Mútua Máxima de **3.531** bits/símbolo
[Song05.wav]: Informação Mútua Máxima de **0.532** bits/símbolo
[Song04.wav]: Informação Mútua Máxima de **0.191** bits/símbolo
[Song02.wav]: Informação Mútua Máxima de **0.189** bits/símbolo
[Song03.wav]: Informação Mútua Máxima de **0.168** bits/símbolo
[Song01.wav]: Informação Mútua Máxima de **0.136** bits/símbolo
==========

Reproduzindo e ouvindo todos os áudios podemos categorizá-los da seguinte forma:

- 'Song07.wav' e 'Song06.wav': são exatamente iguais a 'saxriff.wav'.
- 'Song05.wav': é igual a 'saxriff.wav' mas com ruído.
- Restantes ficheiros: não possuem qualquer parecença, que seja clara, com 'saxriff.wav'.

(de notar que a categorização foi feita com base numa interpretação sonora).

Tendo em conta a categorização feita acima e observando os valores obtidos para a informação mútua máxima entre 'saxriff.wav' e os outros ficheiros Song*.wav, conseguimos corroborar a validade destes resultados. Por exemplo, a inf. mútua máxima entre 'saxriff.wav' e os ficheiros 'Song07.wav' e 'Song06.wav' foi **3.530** bits/símbolo que é exatamente igual à Entropia veícula na Fonte para o ficheiro 'saxriff.wav', isto corrobora a afirmação de que o áudio em 'Song07.wav' e 'Song06.wav' é manifestamente igual ao áudio em 'saxriff.wav'. A Inf. Mútua Máxima entre o ficheiro 'Song05.wav' e 'saxriff.wav' é relativamente mais pequena do que as obtidas para os ficheiros cujo áudio é exatamente igual a 'saxriff.wav', mas, como podemos verificar ao analisarmos os valores da Inf. Mútua Máx. para os ficheiros que, sonoramente, nada têm a ver com 'saxriff.wav', mesmo com a existência de ruído, como já tínhamos concluído para a alínea b) do exercício 6, continuamos a conseguir identificar a existência do som (neste caso, do 'saxriff.wav'), no áudio ('Song05.wav').

Concluímos que a Informação Mútua permite medir, com precisão, a similaridade entre conjuntos de dados. Daí poder ser aplicada a instrumentos de deteção e identificação musical como por exemplo, o Shazam.

Conclusão

Através da observação e análise dos resultados práticos obtidos neste trabalho e tendo em conta os conhecimentos teóricos consolidados até ao momento, podemos afirmar que todos os resultados apresentados se encontram de acordo com o que seria espectável (e desejável).

Código Desenvolvido:

```
import matplotlib.image as mplimg
import scipy.io.wavfile as spiiowf
import matplotlib.pyplot as plt
import numpy as np
import huffmancodec as hc
```

```
dataPath='data/' #global

def loadTxtData(filename):
    f=open(dataPath+filename,'r')
    txt=list(f.read())
    f.close()
    return np.asarray(txt,dtype=np.str_)
def loadImgData(filename):
    return mplimg.imread(dataPath+filename)
def loadAudioData(filename):
    return spiiowf.read(dataPath+filename)

def flattenData(data):
    #convert multidimensional data array to unidimensional array
    if(data.ndim>1):
        return data.flatten()
    else: #if data array is already unidimensional
        return data

def getOccurrenceArray(data,alphabet): #return occurrences array
    #Note: alphabet is always unidimensional
    occurrences=np.zeros(len(alphabet))#create zeros array with
len==len(alphabet)

    if (data.dtype=='<U1'): #text has to analyzed differently
        for sample in data:
            occurrences[alphabet==ord(sample)]+=1
    elif (data.dtype=='<U2'):
        lenA=(90-65+1)+(122-97+1) #size of alphabet
        for sample in data:
            s=ord(sample[0])*lenA+ord(sample[1])
            occurrences[alphabet==s]+=1
    else:
        for sample in data:
            occurrences[sample]+=1

    return occurrences

def viewHistogram(alphabet,occurrences,title):
    plt.figure()
    plt.bar(alphabet,occurrences)
    plt.xlabel("Alphabet Symbols")
    plt.ylabel("Symbol Occurrence")
    plt.title(title)
    plt.tight_layout()

def getEntropy(nSamples,occurrences): #return entropy value and
probability's array
    probability=occurrences[occurrences>0]/nSamples #calculate
probability only for nonzero occurrence values

    return (-np.sum(probability*np.log2(probability))),probability
#consider only nonzero probabilities
```

```
def huffmanCodeAverageLength(data,dataA,p): #returns average length
and variance
    #Huffman Coding
    codec=hc.HuffmanCodec.from_data(data.flatten())
    symbolsList,lengthsList=codec.get_code_len() #get sub-set of
Alphabet Symbols with respective Huffman codewords lengths
    lengthsArray=np.array(lengthsList)

    l=np.sum(p*lengthsArray) #average length=
P(a1)*length_c(a1)+P(a2)*length_c(a2)+...+P(an)*length_c(an)
    v=(np.sum(p*(lengthsArray-l)**2))/np.sum(p) #weighted sample
variance (variancia ponderada)
    return l,v

def pairData(data,alphabet): #grouping source's data in pairs
    dataLen=len(data)
    alphabetLen=len(alphabet)
    if(dataLen%2==1): #in case dataLen is odd..ignore last element
from data
        data=data[:dataLen-1]

    if (data.dtype=='<U1'):
        ##group source samples as pairs
        pairedData=np.char.add(data[::2],data[1::2])
        ##generate alphabet with paired symbols

pairedAlphabet=np.concatenate([np.arange(65*alphabetLen+65,90*alphabet
Len+90+1,dtype=np.uint16),np.arange(97*alphabetLen+97,122*alphabetLen+
122+1,dtype=np.uint16)]) #ASCII from 65-90 + from 97-122

    else:
        ##group source samples as pairs
        pairedData=data[::2]*alphabetLen+data[1::2]
        ##generate alphabet with paired symbols
        pairedAlphabet=np.arange(0,alphabetLen**2,dtype=np.uint16)
#bottom 0 because source datatype is always unsigned int8 (if not char
type)

    return pairedData,pairedAlphabet

def getWindowMutualInfo(queryData,queryEntropy,windowData,alphabet):
#returns mutual information value between query and window's target
data
    #notice that queryData and windowData always have the same length
    alphabetLen=len(alphabet)
    #calculate window's entropy
    windowO=getOccurrenceArray(windowData,alphabet)
    windowEntropy,p=getEntropy(len(windowData),windowO)

    conjuntasO=np.zeros((alphabetLen,alphabetLen))#matriz de
ocurrences conjuntas
    for i in range(len(queryData)):
        conjuntasO[queryData[i],windowData[i]]+=1 #increment
```

```
occurrence of (queryData[i],windowData[i])

    conjuntaEntropy,p=getEntropy(len(queryData),conjuntasO) #no need
to flatten windowO. len(queryData) is equal to the number of samples
conjuntas

    return queryEntropy+windowEntropy-conjuntaEntropy #informacao
mutua para query e janela

def getMutualInfoArray(queryData,targetData, alphabet, step):
    #query
    queryLen=len(queryData) #number of samples query's audio
    queryO=getOccurrenceArray(queryData,alphabet)
    queryEntropy,p=getEntropy(len(queryData),queryO)
    #target
    targetLen=len(targetData) #number of samples of target's audio
    #target's window qnt
    windowsQnt=int(np.floor((targetLen-queryLen)/step)+1)

    #array to save mutual info for each window
    mutualInfoArray=np.zeros(windowsQnt)

    for i in range(0,windowsQnt):

mutualInfoArray[i]=getWindowMutualInfo(queryData,queryEntropy,targetDa
ta[i*step:queryLen+i*step],alphabet)

    return mutualInfoArray

def plotMutualInfo(t,mutualInfo,title):
    plt.figure()
    plt.plot(t,mutualInfo)
    plt.xlabel("Time (s)")
    plt.ylabel("Mutual Information")
    plt.title(title)

def musicIdentifier(query,alphabet,step,queryName):
    maxArr=np.zeros(7) #7 == quantity of targets to be analyzed
    for i in range(1,8):
        targetName='Song0'+str(i)+'.wav'
        [fs,target]= loadAudioData(targetName)
        if(target.ndim>1): #if audio is stereo consider left channel
            target=target[:,0]
        mutualInfo=getMutualInfoArray(query,target,alphabet,step) #get
array with mutual information
        print('\n[{} ; {}]:'.format(queryName,targetName))
        print('Evolução da Inf. Mútua: ',end='')
        print(mutualInfo) #print array with mutual info per sliding
window
        maxArr[i-1]=np.max(mutualInfo)
        print('Informação Mútua Máxima = {} '.format(maxArr[i-1]))

#plotMutualInfo(t,mutualInfo,'Inf({};{})='.format(queryName,targetName
```

```
))

    #print conclusion results
    sortedMaxArr=np.sort(maxArr)[::-1]   #descending order
    print('\n'+5*'='+'Results for '+queryName+5*'=')
    for inf in sortedMaxArr:
        i=np.where(maxArr==inf)[0][0]+1
        targetName='Song0'+str(i)+'.wav'
        print('[{}]: Informação Mútua Máxima de {}
bits/símbolo'.format(targetName,inf))
    print(35*'=')

def main():
    #================================================
    #image
    print(50*'-')
    filename='lena.bmp'
    tag='['+filename+']'
    lenaData=loadImgData(filename) #load data from file
    nBits=int(str(lenaData.dtype)[4:]) #get number of bits from data
type
    lenaData=flattenData(lenaData) #make data unidimensional
    lenaA=np.arange(0,2**nBits,dtype=lenaData.dtype) #img alphabet
from 0 to (2**nBits)-1. Note: arange(start,end[not inclusive],step) =>
end=2**nBits
    lenaO=getOccurrenceArray(lenaData,lenaA) #calculate symbol
occurrence array and source's number of samples
    print(tag+': {} bits/symbol'.format(nBits)) #print number of bits
from data type
    # using the huffman codec..calculates and prints average number of
bits per symbol + variance of codewords lengths (bits/symbol)
    viewHistogram(lenaA,lenaO,tag+': Symbol\'s Occurrence Histogram')
#plots histogram
    entropy,probabilities=getEntropy(len(lenaData),lenaO)
    print(tag+': Entropy for this source = {}
bits/symbol'.format(entropy))
    #ex4
    l,v=huffmanCodeAverageLength(lenaData,lenaA,probabilities)
    print(tag+': Huffman Codec: average of {} bits/symbol'.format(l))
    print(tag+': Huffman Codec: variance = {}'.format(v))
    print(tag+': Redundancy of the Huffman Code = {}
bits/symbol'.format(l-entropy)) #a measure of the efficiency of this
code (Huffman) is its redundancy
    #ex5
    lenaPairedData,lenaPairedA=pairData(lenaData,lenaA) #get source's
samples and alphabet's symbols, grouped in pairs
    lenaPairedO=getOccurrenceArray(lenaPairedData,lenaPairedA)
    entropy,probabilities=getEntropy(len(lenaPairedData),lenaPairedO)
    print(tag+': Entropy for this GROUPED source = {}
bits/symbol'.format(entropy/2))
    print(50*'-')
    #------------------------------------------------
    filename='ct1.bmp'
```

```
    tag='['+filename+']'
    ct1Data=loadImgData(filename)
    nBits=int(str(ct1Data.dtype)[4:])
    ct1Data=flattenData(ct1Data[:,:,0]) #make data unidimensional
    ct1A=np.arange(0,2**nBits,dtype=ct1Data.dtype) #img alphabet from
0 to (2**nBits)-1
    ct1O=getOccurrenceArray(ct1Data,ct1A) #ct1 is a monocromatic image
with rgb+alpha channels, so rgb channels will have the same value. For
that reason, we'll only consider the r [0] channel values for
efficiency purposes
    print(tag+': {} bits/symbol'.format(nBits))
    viewHistogram(ct1A,ct1O,tag+': Symbol\'s Occurrence Histogram')
    entropy,probabilities=getEntropy(len(ct1Data),ct1O)
    print(tag+': Entropy for this source = {}
bits/symbol'.format(entropy))
    #ex4
    l,v=huffmanCodeAverageLength(ct1Data,ct1A,probabilities)
    print(tag+': Huffman Codec: average of {} bits/symbol'.format(l))
    print(tag+': Huffman Codec: variance = {}'.format(v))
    print(tag+': Redundancy of the Huffman Code = {}
bits/symbol'.format(l-entropy)) #a measure of the efficiency of this
code (Huffman) is its redundancy
    #ex5
    ct1PairedData,ct1PairedA=pairData(ct1Data,ct1A)
    ct1PairedO=getOccurrenceArray(ct1PairedData,ct1PairedA)
    entropy,probabilities=getEntropy(len(ct1PairedData),ct1PairedO)
    print(tag+': Entropy for this GROUPED source = {}
bits/symbol'.format(entropy/2))
    print(50*'-')
    #-------------------------------------------------
    filename='binaria.bmp'
    tag='['+filename+']'
    binariaData=loadImgData(filename)
    nBits=int(str(binariaData.dtype)[4:])
    binariaData=flattenData(binariaData[:,:,0])
    binariaA=np.arange(0,2**nBits,dtype=binariaData.dtype) #img
alphabet from 0 to (2**nBits)-1
    binariaO=getOccurrenceArray(binariaData,binariaA) #binaria is a
binary image with 4 channels. There's only rgb(0,0,0) and
rgb(255,255,255) colors. For that reason, similarly to ct1 image,
we'll only consider the r [0] channel values for efficiency purposes
    print(tag+': {} bits/symbol'.format(nBits))
    viewHistogram(binariaA,binariaO,tag+': Symbol\'s Occurrence
Histogram')
    entropy,probabilities=getEntropy(len(binariaData),binariaO)
    print(tag+': Entropy for this source = {}
bits/symbol'.format(entropy))
    #ex4
    l,v=huffmanCodeAverageLength(binariaData,binariaA,probabilities)
    print(tag+': Huffman Codec: average of {} bits/symbol'.format(l))
    print(tag+': Huffman Codec: variance = {}'.format(v))
    print(tag+': Redundancy of the Huffman Code = {}
bits/symbol'.format(l-entropy)) #a measure of the efficiency of this
```

```
code (Huffman) is its redundancy
    #ex5
    binariaPairedData,binariaPairedA=pairData(binariaData,binariaA)

binariaPairedO=getOccurrenceArray(binariaPairedData,binariaPairedA)

entropy,probabilities=getEntropy(len(binariaPairedData),binariaPairedO
)
    print(tag+': Entropy for this GROUPED source = {}
bits/symbol'.format(entropy/2))
    print(50*'-')
    #===================================================
    #audio
    filename='saxriff.wav'
    tag='['+filename+']'
    [saxriffFs,saxriffData]= loadAudioData(filename)
    nBits=int(str(saxriffData.dtype)[4:]) #bits quantization (unsigned
8 bits integer =>sample values from 0 to 2**bitsQ )
    saxriffData=flattenData(saxriffData[:,0])
    saxriffA=np.arange(0,2**nBits,dtype=saxriffData.dtype)
    saxriffO=getOccurrenceArray(saxriffData,saxriffA)
    print(tag+': {} bits/symbol'.format(nBits))
    viewHistogram(saxriffA,saxriffO,tag+': Symbol\'s Occurrence
Histogram')
    entropy,probabilities=getEntropy(len(saxriffData),saxriffO)
    print(tag+': Entropy for this source = {}
bits/symbol'.format(entropy))
    #ex4
    l,v=huffmanCodeAverageLength(saxriffData,saxriffA,probabilities)
    print(tag+': Huffman Codec: average of {} bits/symbol'.format(l))
    print(tag+': Huffman Codec: variance = {}'.format(v))
    print(tag+': Redundancy of the Huffman Code = {}
bits/symbol'.format(l-entropy)) #a measure of the efficiency of this
code (Huffman) is its redundancy
    #ex5
    saxriffPairedData,saxriffPairedA=pairData(saxriffData,saxriffA)

saxriffPairedO=getOccurrenceArray(saxriffPairedData,saxriffPairedA)

entropy,probabilities=getEntropy(len(saxriffPairedData),saxriffPairedO
)
    print(tag+': Entropy for this GROUPED source = {}
bits/symbol'.format(entropy/2))
    print(50*'-')
    #===================================================
    #text
    filename='texto.txt'
    tag='['+filename+']'
    textoData=loadTxtData(filename)

textoData=textoData[((textoData>='a')&(textoData<='z'))|((textoData>='
A')&(textoData<='Z'))] #since we're not considering ascii letters
outside [A-Za-z]..we remove them from textoData for efficiency
```

```
purposes (since we have to pass the data without those 'special'
characters, as argument, to the Huffman codec later)
    textoData=flattenData(textoData)

textoA=np.concatenate([np.arange(65,91,dtype=np.uint8),np.arange(97,12
3,dtype=np.uint8)]) #texto Alphabet filled with decimal ascii
values(8bits) of chars [A-Za-z]
    textoO=getOccurrenceArray(textoData,textoA)
    print(tag+': ASCII = 8 bits/symbol')
    viewHistogram(textoA,textoO,tag+': Symbol\'s Occurrence
Histogram')
    entropy,probabilities=getEntropy(len(textoData),textoO)
    print(tag+': Entropy for this source = {}
bits/symbol'.format(entropy))
    #ex4
    l,v=huffmanCodeAverageLength(textoData,textoA,probabilities)
    print(tag+': Huffman Codec: average of {} bits/symbol'.format(l))
    print(tag+': Huffman Codec: variance = {}'.format(v))
    print(tag+': Redundancy of the Huffman Code = {}
bits/symbol'.format(l-entropy)) #a measure of the efficiency of this
code (Huffman) is its redundancy
    #ex5
    textoPairedData,textoPairedA=pairData(textoData,textoA)
    textoPairedO=getOccurrenceArray(textoPairedData,textoPairedA)

entropy,probabilities=getEntropy(len(textoPairedData),textoPairedO)
    print(tag+': Entropy for this GROUPED source = {}
bits/symbol'.format(entropy/2))
    #Note: dtype of textoData is Unicode. The ASCII value of a
character is the same as its Unicode value.
    print(50*'-')
    #--------------------------------------------------
    #ex6
    #TESTE de informacao mutua q está no enunciado do trabalho
    '''
    q=np.array([2 ,6 ,4 ,10 ,5 ,9, 5, 8 ,0 ,8],dtype=np.uint8)
    t=np.array([6, 8, 9, 7, 2, 4, 9 ,9 ,4 ,9 ,1 ,4 ,8 ,0 ,1 ,2 ,2 ,6
,3 ,2 ,0, 7, 4, 9, 5, 4, 8, 5, 2, 7, 8 ,0 ,7 ,4 ,8 ,5 ,7 ,4 ,3 ,2 ,2
,7 ,3 ,5,2, 7, 4, 9, 9, 6],dtype=np.uint8)
    a=np.arange(0,11,dtype=np.uint8)
    print('TESTE:')
    print(getMutualInfoArray(q,t,a,1))
    print('FIM TESTE')
    '''
    step=round(len(saxriffData)/4) #for 6b) and 6c)
    #6b)
    [fs,target1Data]= loadAudioData('target01 - repeat.wav')

mutualInfo1=getMutualInfoArray(saxriffData,flattenData(target1Data[:,0
]),saxriffA,step)
    print('\n[saxriff.wav ; target01-repeat.wav]')
    print('Evolução da Inf. Mútua :',end='')
    print(mutualInfo1)
```

```
    #plot
    t=np.arange(0,(target1Data.shape[0]-
len(saxriffData)+1)/fs,step/fs) #calculate time instance array
(consider beggining of the sliding window instance)
    plotMutualInfo(t,mutualInfo1,'[saxriff.wav ; target01-
repeat.wav]')

    [fs,target2Data]= loadAudioData('target02 - repeatNoise.wav')

mutualInfo2=getMutualInfoArray(saxriffData,flattenData(target2Data[:,0
]),saxriffA,step)
    print('\n[saxriff.wav ; target02-repeatNoise.wav]')
    print('Evolução da Inf. Mútua :',end='')
    print(mutualInfo2)
    #plot
    t=np.arange(0,(target2Data.shape[0]-
len(saxriffData)+1)/fs,step/fs) #calculate time instance array
(consider beggining of the sliding window instance)
    plotMutualInfo(t,mutualInfo2,'[saxriff.wav ; target02-
repeatNoise.wav]')

    #6c)
    print(50*'-')
    musicIdentifier(saxriffData,saxriffA,step,'saxriff.wav')


if __name__=="__main__":
    main()
```