

Teoria da informação trabalho prático nº1

Entropia, Redundância e Informação Mútua

2 de novembro de 2021

Grupo 5

Eduardo Nunes

Diogo Tavares

André Moreira

# Introdução

O objetivo deste trabalho é adquirir conhecimentos fundamentais da teoria da informação, nomeada, a informação, redundância, entropia e informação mútua. Para isto, resolvemos as questões apresentadas na ficha recorrendo à linguagem *Python* que nos permite criar gráficos e extrair dados a partir das fontes fornecidas (imagens, áudios, texto).

Para resolver as questões colocadas, usamos conceitos lecionados nas aulas teóricas tais como:

* Entropia (limite mínimo teórico para o número médio de bits por símbolo), calculada pela definição de média e através da sua fórmula:

\*

* Codificação de Huffman;
* Taxa de Compressão:
* Informação mútua, calculada através da sua fórmula:

# Exercícios 1, 2 e 3

Uma imagem com texto

Descrição gerada automaticamente Estes exercícios tinham como objetivo calcular a entropia de uma fonte dada juntamente com um histograma de ocorrências de símbolos dessa fonte.

Para isso, determinamos o **alfabeto** da fonte ao lermos o tipo de ficheiro dado, como podemos ver na função *lerficheiro* e, seguidamente, limpamos a fonte de acordo com alfabeto gerado.

Uma imagem com texto

Descrição gerada automaticamente Com isto, tendo o **alfabeto** e a **data** que corresponde ao mesmo, contamos as ocorrências de cada elemento do alfabeto na nossa fonte, criando assim, uma lista de ocorrências que tem a mesma ordem dos elementos do alfabeto.

*Função lerficheiro*

Limpar a fonte lida e criar a lista contagens

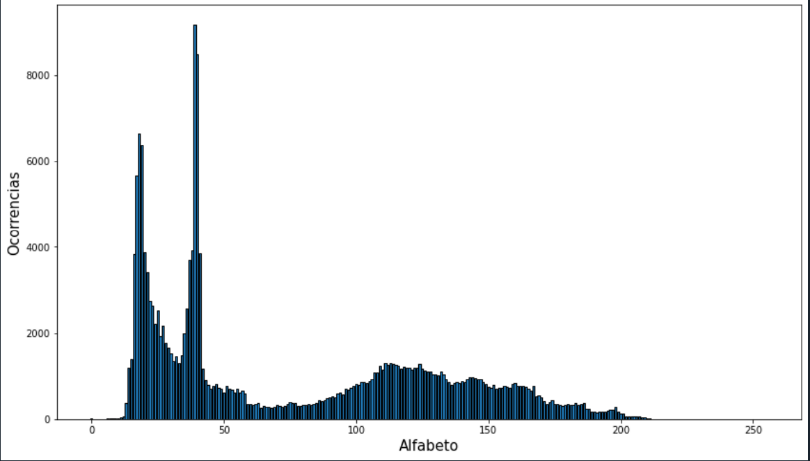
Seguidamente, tendo a **data** e o **alfabeto** usamos a função *entropia* que passadas as ocorrências, devolve a sua entropia calculada a partir da definição acima mostrada com o auxílio da biblioteca *numpy*. Deve referir-se que para evitar erros, só usámos probabilidades maiores que 0.



Finalmente, para criar o seu histograma usámos a função *criarhist* onde passamos as **ocorrências** e o **alfabeto** para fazer ambos os eixos *x* e *y*.

Uma imagem com texto

Descrição gerada automaticamente

Apresentamos agora os resultados obtidos:

Para esta fonte o alfabeto consiste em valores de 0 a 255 visto que é uma imagem a preto e branco com diferentes tonalidades. Uma vez que a imagem apresenta diversos tons de preto e branco, a entropia vai ser maior e, portanto, o histograma vai ser mais disperso.

Fig1. Histograma “*kid.bmp*”, Entropia = 6.95414

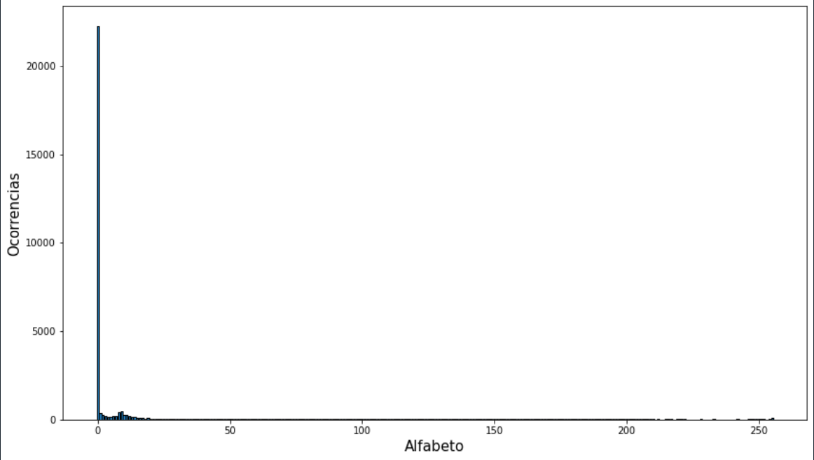
Para esta fonte, a dispersão apresenta um grande número de zeros e valores próximos dele porque a imagem tem muitos pixéis escuros e pouca variação de tons. Uma consequência disto é a redução da sua entropia em relação às outras imagens.

Fig2. Histograma “*homer.bmp”*, Entropia = 3.4658

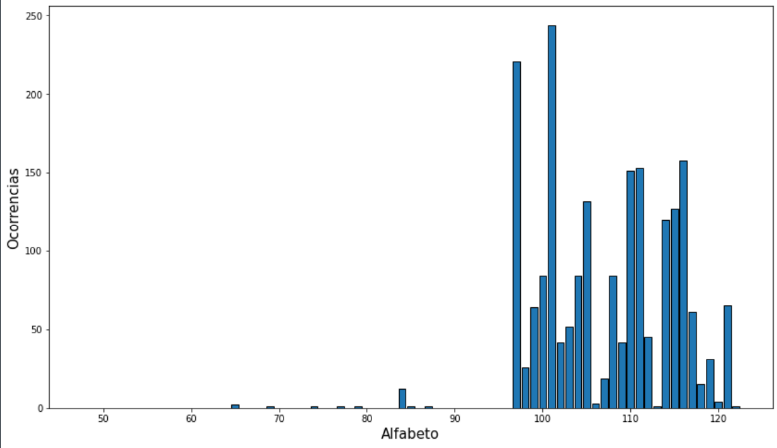
Para esta fonte, o histograma apresentado tem como alfabeto símbolos que convertidos para a tabela ASCII são letras regulares do alfabeto latino, é de notar que as vogais tendem a surgir mais vezes. Podemos verificar alguma dispersão.

Fig3. Histograma “*texto.txt*”, Entropia = 4.22797

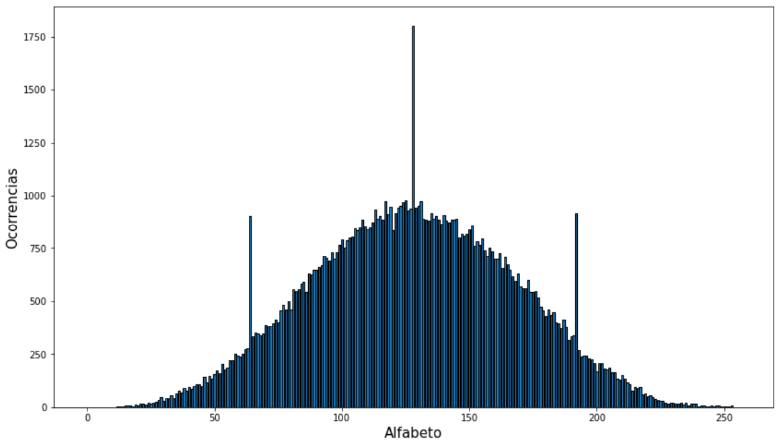
Para esta fonte, o histograma apresentado está em forma de “distribuição normal” porque este foi normalizado para estar inserido entre valores entre 0 e 255. Como está tão disperso, este é uma das fontes com maior entropia.

Fig4. Histograma “g*uitarSolo.wav*”, Entropia = 7.32920

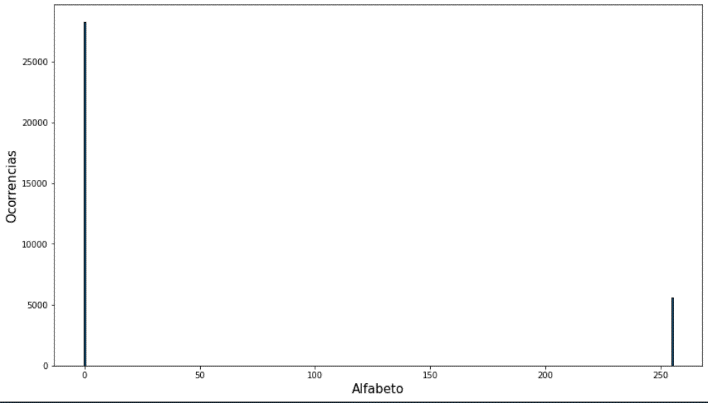
Para esta fonte, o histograma aparece com apenas duas barras porque é uma imagem em binário, ou seja, é composta apenas por pixéis com valores a 0 (pretos) ou 255 (brancos). Esta também é a fonte com menor entropia visto que é só preciso 1 bit para codificar estes dois valores, uma vez que tem só dois tons.

Fig5. Histograma “*homerbin.bmp*”, Entropia = 0.64478

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

O princípio da compressão consiste em reduzir o espaço ocupado por dados num dispositivo.

Para que haja compressão não destrutiva, é necessário eliminar dados repetidos de modo que haja uma reconstrução exata da fonte original sem haver data a ser repetida. “*Se a informação já está alocada porquê estar a guardá-la outra vez?”* Teoricamente, uma compressão ideal seria uma fonte em que não existia redundâncias (um bom exemplo seria a codificação de Huffman).

Passando agora às nossas fontes, para as “comprimir de forma não destrutiva” elas teriam de ter, como foi dito anteriormente, redundâncias que possam ser eliminadas sem deixar de ser igual à fonte original. Só com esta condição é que podemos comprimir de forma não destrutiva.

Como a entropia (limite mínimo teórico para o número médio de bits por símbolo) é inferior em todas as amostras aos bits usados para guardar a informação de cada amostra, podemos comprimir a amostra de forma não destrutiva.

**Qual a compressão máxima que conseguimos alcançar?**

A Taxa de Compressão é dada por:

Sendo

(X tem #X elementos)

Fazendo uma tabela com os valores obtidos usando a fórmula da Taxa de Compressão:

|  |  |  |
| --- | --- | --- |
| **FONTE** | **Entropia (bits/símbolo)** | **Taxa de Compressão (%)** |
| kid.bmp | 6.95414 | 13.07 (#X = 256) |
| homer.bmp | 3.46587 | 56.87 (#X = 256) |
| english.txt | 4.22797 | 28.99 (#X = 62) |
| guitarSolo.wav | 7.32920 | 8.385 (#X = 256) |
| homerbin.bmp | 0.64478 | 91.94 (#X = 256) |

# Exercício 4

Neste exercício usamos a biblioteca fornecida pelo professor na função *huffmancodec* para codificar a nossa fonte usando o **algoritmo de Huffman** retornando os símbolos e o seu respetivo tamanho, obtendo depois com estes, a **entropia** (o número médio de bits por símbolo) e a sua **variância** através da função *entropiaHuffman*:

Uma imagem com texto

Descrição gerada automaticamente

A entropia é calculada através da definição de média, contamos as ocorrências de cada símbolo no alfabeto da fonte e depois dividimos pela soma das ocorrências de todos os símbolos no total.

E a variância usamos a seguinte formula:

Onde a variável e .

Daqui gerámos a seguinte tabela:

|  |  |  |  |
| --- | --- | --- | --- |
| **Ficheiro** | **Entropia**  **(bits/símbolo)** | **Huffman**  **(bits/símbolo)** | **Variância** |
| kid.bmp | 6.95414 | 6.98322 | 2.09930 |
| homer.bmp | 3.46587 | 3.54832 | 13.19684 |
| english.txt | 4.22797 | 4.25183 | 1.19085 |
| guitarSolo.wav | 7.32920 | 7.35016 | 0.72743 |
| homerbin.bmp | 0.64478 | 1.00000 | 0.00000 |

Observamos, através da tabela que a entropia dos códigos de Huffman é ligeiramente superior à entropia calculada anteriormente. Concluímos, que os códigos de Huffman não são assim tão ideais visto que estas ainda são superiores ao limite mínimo teórico de bits por símbolo.

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

Sim, é possível reduzir a variância porque nos pode dar comprimentos mais homogéneos ao usarmos a codificação de Huffman.

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

Reduzir a variância não implica alterar o comprimento médio do código, portanto estamos a trabalhar com a mesma data de qualquer maneira. Criar uma árvore de Huffman com menor profundidade, ou seja, aproveitar mais os ramos de cima diminui a variância da data, visto que esta está comprimida em valores mais próximos e, portanto, o comprimento dos símbolos é mais uniforme. Desta maneira os bits produzem-se de forma mais regular e reduz-se a possibilidade de encontrar erros na descodificação dela, tornando, assim, o código mais eficaz.

# Exercício 5

No exercício 5 é pedido para repetir o exercício 3 mas agrupando os símbolos da fonte e também com um alfabeto agrupado. Ou seja, fazendo com que cada símbolo seja uma sequência de dois símbolos contíguos.

Para tal, usamos a função *agrupar* que percorre metade da data (tirando o último elemento se esta for ímpar) e a cada elemento agrupa-o com o próximo numa só lista.

Uma imagem com texto

Descrição gerada automaticamente

Podemos assim, com os valores obtidos, verificar que ao agrupar o alfabeto há uma otimização do código visto que a entropia agrupada é menor que a entropia anterior!

|  |  |  |
| --- | --- | --- |
| **FONTE** | **Entropia (bits/símbolo)** | **ENTROPIA AGRUPADA (bits/ símbolo)** |
| kid.bmp | 6.95414 | 4.90910 |
| homer.bmp | 3.46587 | 2.41273 |
| english.txt | 4.22797 | 3.65215 |
| guitarSolo.wav | 7.32920 | 5.75438 |
| homerbin.bmp | 0.64478 | 0.39782 |

# Exercício 6

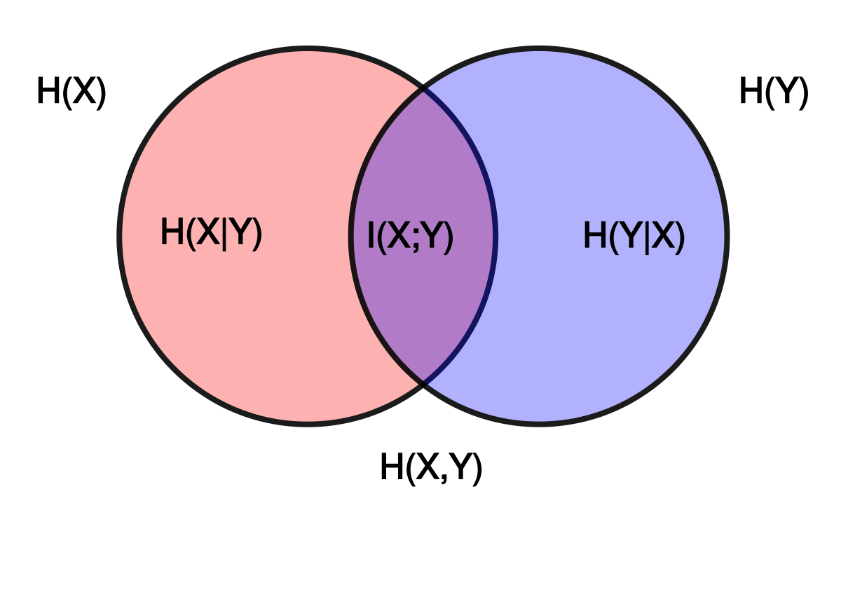
Para fazer o exercício 6 é preciso a partir de um *query*, um *target* e um *passo*,uma função que devolve uma lista de valores de informação mútua em cada passo do *query* para o *target*.

Para isto, usamos a função *InfMut* que a partir destes valores como parâmetro devolve a lista pretendida, esta percorre o target a partir de um determinado *query*, este query é calculado fora da função *calcinfmut* visto que é único em cada passo.

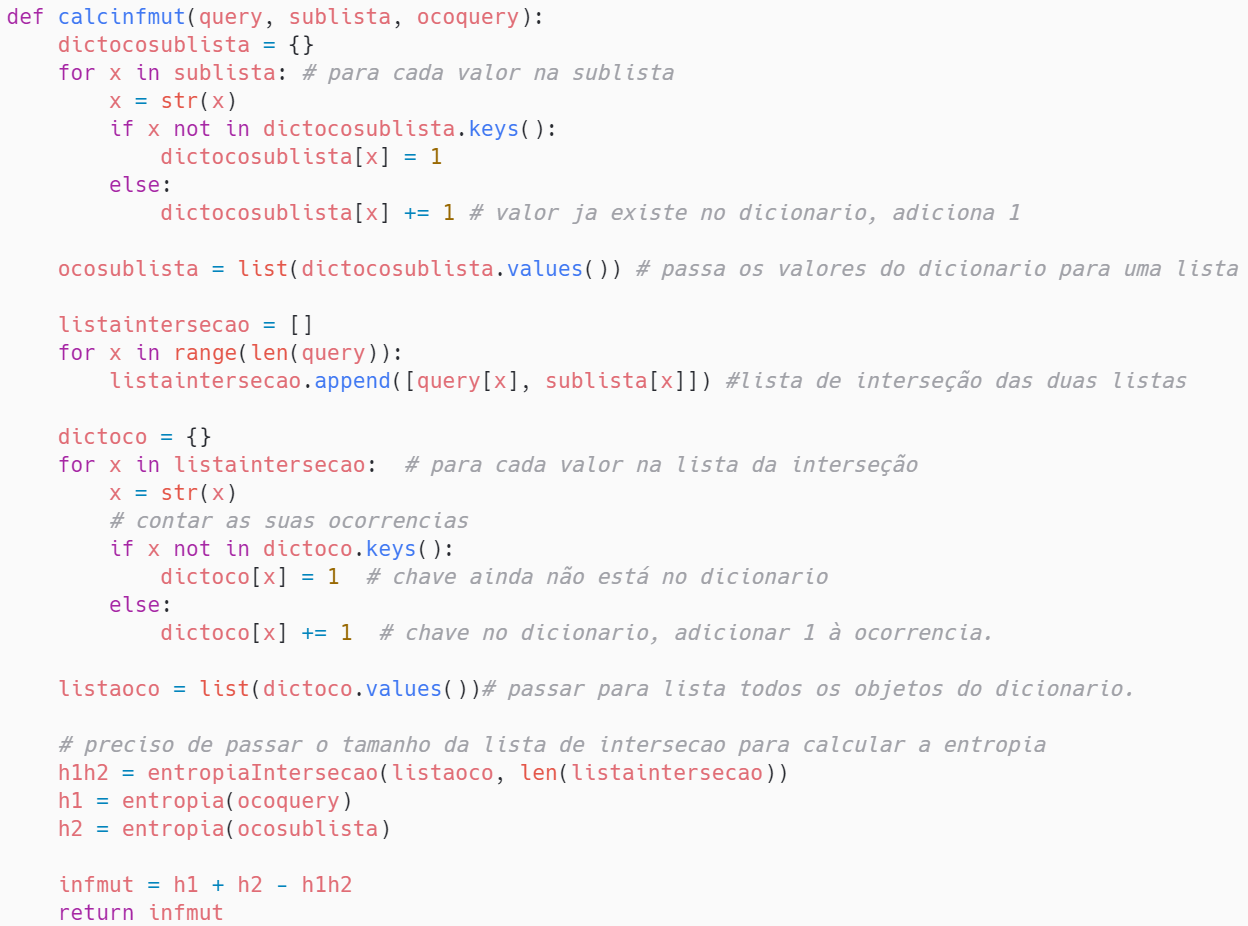


Agora que já temos a função que percorre toda a sublista de passo a passo, preciso de saber calcular a informação mútua entre duas listas. Para isso, usámos a função *calcinfmut*, ela calcula a **Informação Mútua** através das entropias dos conjuntos, mais especificamente:

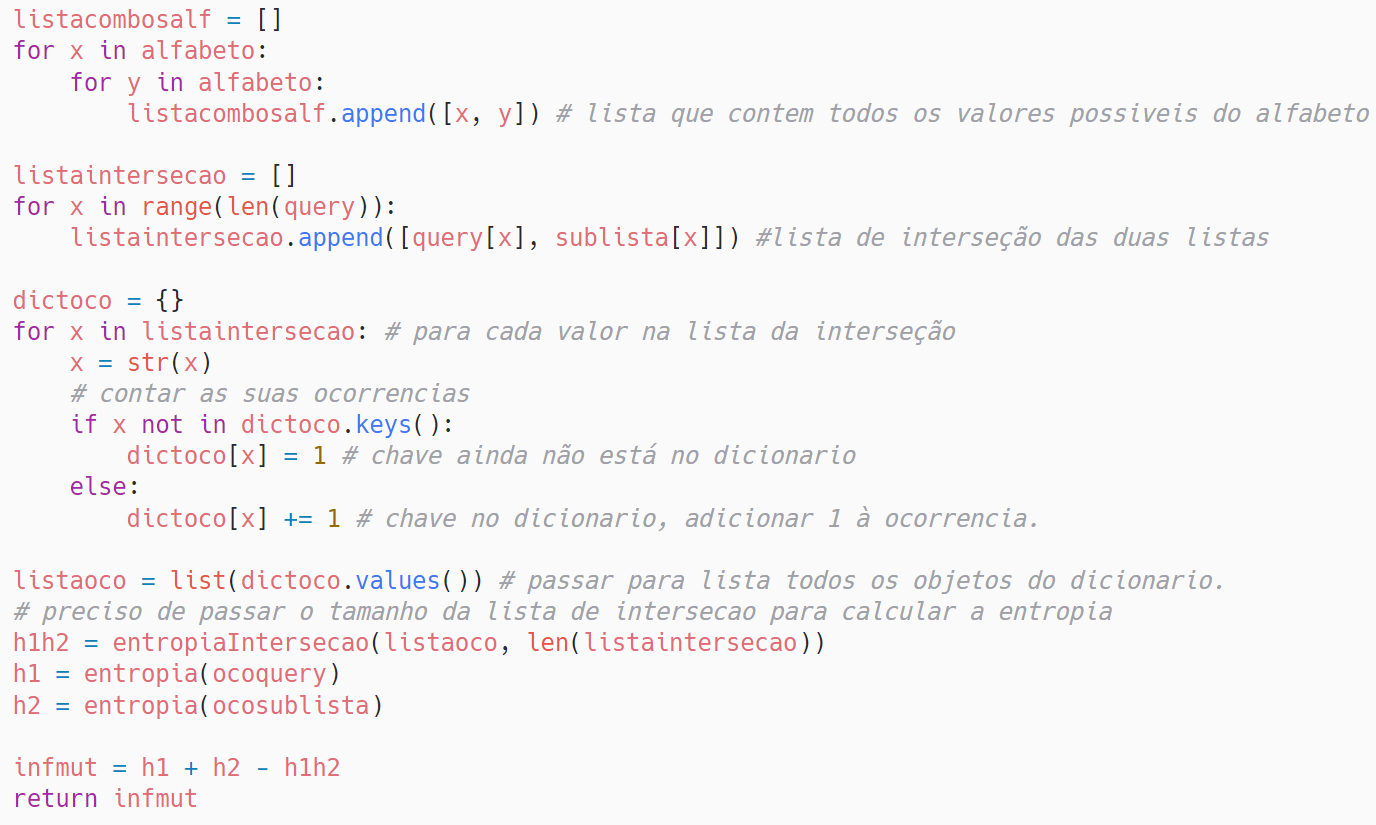
*Visto que…*



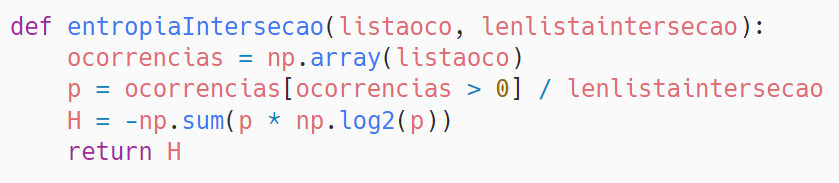
Para codificar esta fórmula em Python precisamos das ocorrências de cada valor no *query*, o query (para calcular a lista de interseção) e as ocorrências de cada valor no pedaço do target que estamos a verificar, portanto, na sublista.



Agora que tenho as ocorrências das listas, preciso ainda de calcular a **Entropia de Interseção**, para isso, preciso de fazer uma lista de interseção com as duas listas em questão e o “alfabeto de interseção” que vai conter todas as interseções teoricamente possíveis.



Depois disso basta contar as ocorrencias da lista de interseção e calcular a entropia das três listas de ocorrencias obtidas. Usando a função da entropia já criada anteriormente para calcular as ocorrencias do query e da sublista e a função entropiaIntersecao para calcular a entropia de uma lista com listas de dois termos.



Finalmente, usamos a defenição referida acima para calcular a informação mútua.

Na alínea seguinte, precisamos de aplicar a função criada anteriormente, usando, agora, fontes previamente selecionadas como *query* e *target* obtendo os seguintes resultados:

Valor 1:

**“guitarSolo.wav” e “target01 – repeat.wav”**

**Informações Mútuas:**

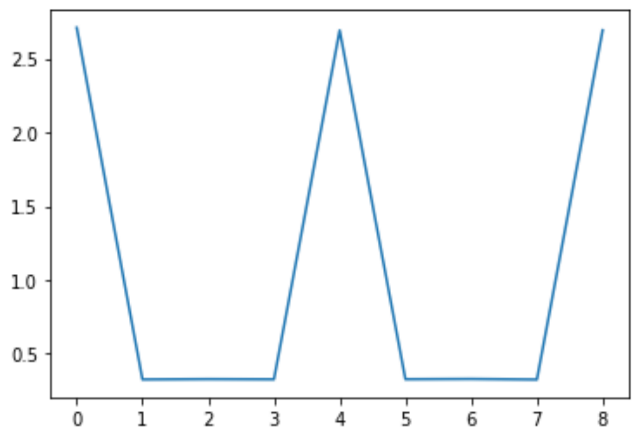
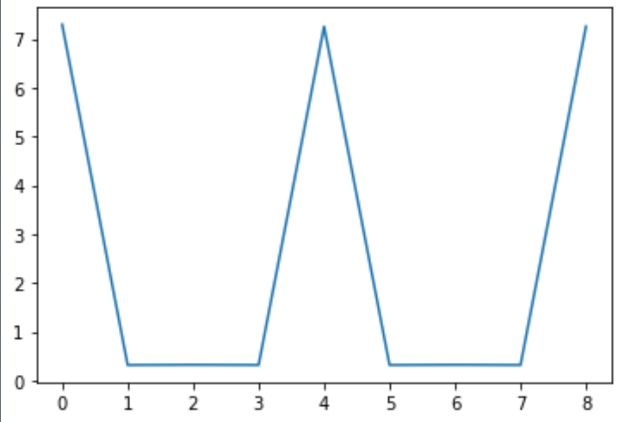
[2.7091, 0.3261, 0.3283, 0.3269, 2.6909, 0.3276, 0.3294, 0.3254, 2.6919]

Valor 2:

**"guitarSolo.wav" e "target02 - repeatNoise.wav"**

**Informações Mútuas:**

[2.7091, 2.6919, 2.6909, 0.3294, 0.3283, 0.3276, 0.3269, 0.3261, 0.3254]

Aplicando ambas as informações mútuas ao longo do passo em dois *plots*:

Informação mútua

“guitarSolo.wav” – “Target02 – reapeatNoise.wav”

Informação mútua

“guitarSolo.wav” – “Target01 – reapeat.wav”

Concluímos, assim, que há mais Informação mútua entre “guitarSolo.wav” – “Target01 – reapeat.wav” do que “guitarSolo.wav” – “Target02 – reapeatNoise.wav” isto pode dever-se ao ruido presente no áudio “Target02 – reapeatNoise.wav”, o que o faz afastar mais do áudio original encontrando, portanto, menos informação em comum entre os dois.

Finalmente, para terminar os exercícios feitos, usamos os diferentes ficheiros “Song\*.wav” como target e o “guitarSolo.wav” como fonte para determinar a evolução da Informação Mútua entre os dois e a respetiva Informação Mútua máxima por ordem:

Os dados obtidos foram os seguintes:

|  |  |
| --- | --- |
| **Ficheiro WAV** | **Informação Mútua máxima** |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

Com esta informação podemos deduzir os seguintes gráficos: