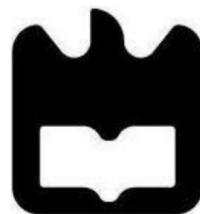


# TAI - Assignment II

Universidade de Aveiro

Departamento de Eletrónica, Telecomunicações e Informática

Mestrado em Engenharia Informática



Diogo Carvalho nº92969

Diogo Cunha nº95278

Rafael Baptista nº93367

28 de Dezembro de 2021

# Índice

<b>1. Introdução</b>	3
<b>2. Metodologia</b>	4
<b>3. Implementação</b>	5
<b>4. Análise de resultados</b>	14
<b>5. Conclusão</b>	22

# 1. Introdução

O presente relatório pretende descrever o trabalho realizado na execução do segundo projeto proposto na unidade curricular de Teoria Algorítmica da Informação.

Neste projeto, fomos desafiados a numa primeira fase desenvolver um programa inicial (*lang.py*) capaz de efetuar o cálculo do número de bits necessários para comprimir um texto alvo, com base na utilização de um modelo de contexto finito (Finite Context Model) construído a partir de um texto referência. Este programa irá utilizar o modelo de contexto finito desenvolvido no primeiro projeto desta unidade curricular.

Posteriormente, numa segunda fase foi nos proposto desenvolver um segundo programa (*findlang.py*) que, com base no primeiro programa desenvolvido (*lang.py*), funcione como um sistema de reconhecimento de línguas, onde a partir de um conjunto de texto de referências escritos em diferentes línguas, forneça uma previsão da língua utilizada para escrever o texto alvo.

Finalmente, numa terceira e última fase do projeto, foi nos proposto desenvolver um terceiro programa (*locatelang.py*) que fosse capaz de processar texto escrito em múltiplas línguas, retornando no fim os diferentes segmentos de texto e a língua associada a cada um deles.

## 2. Metodologia

Para a realização deste projeto, utilizamos a plataforma Github como repositório de código compartilhado. A estrutura do nosso repositório é constituída por uma área com o código desenvolvido, uma pasta com o relatório escrito bem como uma secção com os exemplos de texto de diferentes funcione como um sistema de reconhecimento de línguas e tamanhos que utilizamos para testar o nosso código, estrutura essa que foi recomendada pelos professores.

Os textos utilizados foram retirados do dataset LTI LangID Corpus Release 4.

No que diz respeito à comunicação entre os elementos do grupo, utilizamos a plataforma Discord para efetuar a discussão dos pontos críticos do nosso projeto, facilitando assim a cooperação à distância na realização do mesmo.

Relativamente ao código escrito, o mesmo foi desenvolvido na língua de programação Python. A estrutura do código é composta por:

- Ficheiro *fcm.py* que contém a classe Fcm utilizada para representar o modelo de contexto finito.
- Programa *lang.py* que vai utilizar a classe Fcm para calcular o número de bits necessários para comprimir um texto alvo, a partir de um texto de referência.
- Programa *findlang.py* que, contendo um conjunto de vários textos de referência de diferentes línguas, vai fazer uso do programa *lang.py* para conseguir efetuar uma previsão da língua utilizada para escrever o texto alvo a ser analisado.
- Programa *locatelang.py* que, contendo um conjunto de vários textos de referência de diferentes línguas, recebe um texto alvo escrito em várias línguas e vai fazer uso do programa *lang.py* para determinar as secções do texto alvo que foram escritas em diferentes línguas e qual a língua associada a cada uma dessas secções.

É importante referir que foram desenvolvidos alguns aspectos extra na realização deste projeto. Primeiramente, no que diz respeito ao programa *locatelang.py*, foram desenvolvidas duas implementações distintas e possíveis para a obtenção das secções de línguas distintas que irão ser descritas e comparadas mais à frente. Adicionalmente, e no que diz respeito aos programas *lang.py* e *findlang.py*, foi desenvolvida uma implementação que tira vantagem da utilização de múltiplos modelos (neste caso, 2 modelos) para calcular o número de bits necessários para comprimir o texto alvo e obter previsão para língua do texto no caso do programa *findlang.py*.

### 3. Implementação

Nesta secção do nosso relatório iremos descrever a implementação desenvolvida para cada um dos nossos programas bem como analisar e justificar as principais decisões tomadas durante esta mesma fase de implementação. Apesar de no enunciado do presente trabalho apenas pedir uma única implementação para cada programa, como referido acima, acabamos por implementar duas maneiras distintas de obter as línguas das diferentes secções no que diz respeito ao programa *locatelang.py*. Ambas as implementações estão presentes no código final do projeto, encontrando-se uma delas comentada. Ambas as implementações vão ser comparadas mais à frente na parte de análise de resultados.

#### 3.1. lang.py

Posto isto e iniciando pelo programa desenvolvido na primeira fase do projeto (*lang.py*), o mesmo tem como objetivo efetuar o cálculo do número estimado de bits necessários para comprimir um determinado ficheiro, designado texto alvo, utilizando um modelo de contexto finito gerado a partir de um outro ficheiro, designado texto referência. No entanto, é preciso ter em conta que este programa vai servir de base para os programas *findlang.py* e *locatelang.py*, pelo que algumas funções adicionais foram introduzidas e irão ser explicadas já nesta secção, apesar de serem apenas necessárias devido aos programas referidos acima.

Para correr este programa, são necessários especificar 4 argumentos:

- *-reference* : Define o caminho para o ficheiro que contém o texto que deve ser utilizado para gerar o modelo de contexto finito;
- *-ftarget* : Define o caminho para o ficheiro que contém o texto para analisar;
- *-k* : Define o tamanho (número de caracteres) dos contextos do finite contextmodel;
- *-α* : Define o smoothing parameter utilizado no cálculo das probabilidades;
- *-multiplelang* : Flag utilizada para programa efetuar os cálculos associados à detecção de secções de diferentes línguas no texto alvo.
- *-multiplemodels* : Flag utilizada para programa utilizar combinação de múltiplos modelos de contexto finito no cálculo do número de bits necessários para comprimir o texto alvo.

Ao executar o nosso programa, o primeiro passo a ser efetuado é a verificação dos parâmetros fornecidos pelo utilizador. No que diz respeito aos parâmetros *reference* e *ftarget*, não existe nenhuma verificação direta, apenas na leitura dos ficheiros fornecidos, caso o ficheiro não exista, o utilizador é informado. O parâmetro *k* deve ser um número inteiro e superior a 0, visto que o objetivo do mesmo é definir o comprimento do contexto utilizado pelo modelo de contexto finito. Por sua vez, no parâmetro *α*, é verificado que o mesmo é um float e que é um valor também ele superior a 0.

De seguida, é efetuada a leitura do ficheiro *ftarget* uma primeira vez para a construção do alfabeto de caracteres deste ficheiro. Esta variável será um set() que contém todos os símbolos diferentes presentes no ficheiro.

O próximo passo na execução do programa, no caso do valor da “flag multiplemodels” ser falso, consiste em fazer a geração do modelo de contexto finito usando o ficheiro *freference*. Para isso, tendo como base a classe criada no primeiro projeto, criamos um objeto da classe Fcm passando como argumento o caminho para o ficheiro *freference*, o valor de  $k$  e o valor de  $\alpha$ .

Após a criação deste objeto, invocamos a função `calculate_probabilities()` deste objeto, que percorre todos os contextos que foram encontrados no texto, e para todos eles, calcula e armazena as probabilidades de cada símbolo pertencente ao alfabeto ocorrer após este dado contexto tendo em conta o valor do smoothing parameter definido pelo utilizador. Todos estes cálculos foram efetuados seguindo as fórmulas numéricas encontradas no enunciado de apresentação do primeiro projeto desta unidade curricular.

A partir deste ponto foram implementadas duas formas distintas de abordar e resolver o problema relacionado com a deteção das secções de texto no caso do programa *locatelang.py* que irão ser explicadas abaixo. Importante referir que para ambas as implementações o cálculo do número de bits necessário para comprimir o texto alvo (que é o principal objetivo do programa *lang.py*) é igual.

### 3.1.1. Criação das palavras presentes no texto alvo

A primeira solução que desenvolvemos para deteção das diferentes secções de texto presentes no texto alvo foi baseada na construção das palavras que o texto alvo contém, isto é, à medida que vamos lendo carácter a carácter o texto, vamos construindo as palavras que constituem o mesmo. Dependendo depois do número de bits necessário para comprimir cada carácter desta palavra, vamos considerar a palavra como pertencente à língua do texto de referência ou não. Segue-se uma explicação mais detalhada.

Nesta implementação, depois de termos as probabilidades do modelo de contexto finito calculadas, o próximo passo é invocar a função `get_number_of_bits_required_to_compress_v1()` que tem como argumentos o modelo de contexto finito, o caminho para o ficheiro a ser analisado, o alfabeto do ficheiro a ser analisado, e uma flag designada *multiplelangflag* (utilizada para executar ou não as instruções relativas à deteção de secções de diferentes línguas no texto). Esta função tem como objetivo calcular o número de bits necessários para comprimir um ficheiro usando um modelo de contexto finito gerado a partir de um outro ficheiro de referência.

Para isso, o primeiro passo é calcular o número de caracteres presentes no ficheiro *ftarget* que não pertencem ao alfabeto do ficheiro modelo de contexto finito.

De seguida é calculado um valor de threshold que é apenas útil durante a execução do programa *locatelang.py*. Este valor vai ser usado para determinar se uma palavra está escrita na língua do texto usado para gerar o modelo de contexto finito ou não. O threshold é calculado através da seguinte fórmula:

$$\text{threshold} = -\log \left( \frac{\alpha}{\alpha |\Sigma|} \right), \text{ onde } \Sigma \text{ é o alfabeto de caracteres do texto referência.}$$

O próximo passo é escolher de forma aleatória um contexto inicial dos contextos conhecidos pelo modelo de contexto finito para ser utilizado na avaliação do número de bits necessário para comprimir o primeiro carácter do texto alvo.

Após isso o programa vai percorrer o ficheiro a comprimir, carácter a carácter, para que se calcule a quantidade de bits necessários para comprimir cada um destes, tendo em conta o contexto que o antecede. A quantidade de bits é calculada de acordo com a seguinte fórmula:

$$-\log_2(\text{probabilidade do evento})$$

A probabilidade do evento por sua vez, pode ser determinada de acordo com os seguintes 3 casos possíveis:

1. Caso o carácter e o contexto estejam no modelo de contexto finito, a probabilidade do evento é retirada diretamente do modelo de contexto finito.
2. Caso o contexto esteja no modelo de contexto finito e o carácter não:

$$\frac{1}{\text{número de caracteres diferentes}}$$

O número de caracteres diferentes refere-se ao número de caracteres presentes no ficheiro alvo que não pertencem ao alfabeto do texto que deu origem à construção do modelo de contexto finito.

3. Caso o contexto não esteja no modelo:

$$\frac{\alpha}{\alpha \times \text{tamanho do alfabeto}}$$

O tamanho do alfabeto refere-se ao número total de símbolos diferentes presentes no texto que deu origem à construção do modelo de contexto finito.

Para além de calcularmos o número de bits necessários para comprimir o ficheiro, à medida que vamos percorrendo o ficheiro carácter a carácter, vamos também criando as palavras que o ficheiro possui. Para isso verificamos se o carácter atual é um espaço “ ” ou uma quebra de linha “\n”. Quando isto acontece, começamos a criar uma nova palavra através de uma String inicializada como “ ”. Os caracteres seguintes se forem diferentes de “ ” e “\n”, são adicionados à string criada anteriormente. Este processo é repetido até que se encontre um novo carácter igual ao espaço ou quebra de linha que significa que a palavra acabou e podemos guardá-la. No processo de armazenamento da palavra, o que vamos guardar vai ser a posição inicial e também a posição final da palavra no ficheiro. Apenas as palavras em que todos os caracteres constituintes da mesma conseguem ser comprimidos com um valor de bits inferior ao valor do threshold definido em cima são armazenadas.

A variável utilizada para guardar as palavras é uma lista em que cada elemento da mesma vai ser um tuplo com a posição inicial e final da palavra no ficheiro.

No final desta função, é retornado o número total de bits necessários para comprimir o ficheiro e a lista que contém as palavras que foram classificadas como pertencentes à língua do texto referência, neste caso, a posição inicial e final das palavras no ficheiro.

### 3.1.2. Janela Deslizante

A segunda solução que desenvolvemos para deteção das diferentes secções de texto presentes no texto alvo foi baseada na utilização de uma janela deslizante que lê o texto carácter a carácter e suaviza a janela de bits atual através do cálculo do número médio de bits necessários para comprimir os caracteres pertences à janela. Dependendo depois do número médio de bits necessário para comprimir os caracteres da janela atual, vamos considerar a mesma como uma secção de texto que pertence à língua do texto de referência ou não.

Nesta implementação, depois de termos as probabilidades do modelo de contexto finito calculadas, o primeiro passo é invocar a função `get_number_of_bits_required_to_compress_v2()` que tem como argumentos o modelo de contexto finito, o caminho para o ficheiro a ser analisado, o alfabeto do ficheiro a ser analisado, o tamanho da janela deslizante, e uma flag designada `multiplelangflag` (utilizada para executar ou não as instruções relativas à deteção de secções de diferentes línguas no texto). Esta função, tal como a função da primeira implementação, tem como objetivo calcular o número de bits necessários para comprimir um ficheiro usando um modelo de contexto finito gerado a partir de um outro ficheiro de referência.

Os primeiros passos que esta função realiza são os mesmos que a implementação anterior, ou seja, vai calcular o número de caracteres presentes no ficheiro `ftarget` que não pertencem ao alfabeto do ficheiro modelo de contexto finito. É também escolhido um contexto inicial aleatório, e um valor de `threshold` da mesma forma e com o mesmo propósito que o método anterior só que neste caso este `threshold` irá ser comparado com a média do número de bits dos caracteres presentes na janela deslizante e é calculado da seguinte forma:

$$threshold = \frac{1}{2} \times \log_2(|\Sigma|)$$

Após isto, o programa irá ler o ficheiro alvo carácter a carácter, e para cada carácter irá calcular o seu número de bits dado o seu contexto precedente da mesma forma que o método anterior. À medida que estes cálculos são efetuados, é também construído de uma forma dinâmica, a janela deslizante que consiste em agrupar sequências de caracteres de tamanho definido pelo utilizador, e para cada secção de caracteres, é calculada a média do número de bits desses caracteres, e se esse valor for inferior ao valor do `threshold`, essa secção de caracteres irá ser guardada e classificada como sendo da língua em que o modelo de contexto finito foi gerado.

A estrutura de dados que é usada para guardar as secções é uma lista que contém tuplos com as posições no ficheiro alvo das secções da seguinte forma:

(`posição_inicio_secção`, `posição_fim_secção`)

Para além desta função (`get_number_of_bits_required_to_compress_v2`), também desenvolvemos uma segunda função, que irá ser utilizada no programa `locatelang.py`, designada `get_sections_from_remaining_sections()`.

Esta função tem como argumentos o modelo de contexto finito, o caminho para o ficheiro a ser analisado, o alfabeto do ficheiro a ser analisado, o tamanho da janela deslizante, e uma lista de secções e tem como objetivo ser utilizada no programa `locatelang.py` para detetar secções de texto que não foram detectadas pela primeira função. Para isso, recebe a lista de secções que não foram classificadas com nenhuma língua e efetua os mesmos cálculos que o método anterior só que com um valor de threshold superior de acordo com o seguinte cálculo:

$$threshold = \frac{3}{4} \times \log_2(|\Sigma|)$$

Para isso, o programa vai percorrer a lista de secções passadas como argumento da função, e para cada uma, vai verificar se existe um número de caracteres precedentes à posição inicial da secção que seja possível de construir um contexto inicial. Caso seja possível, o programa irá ler o contexto original. Caso não seja possível, o programa irá escolher um contexto aleatório. Para isso, o programa à medida que vai percorrendo as secções, vai também de forma dinâmica lendo as linhas do ficheiro alvo. Apenas secções do ficheiro alvo que não foram classificadas inicialmente são analisadas.

Após ter o contexto inicial, o programa vai percorrer os caracteres seguintes até à posição final da secção, calculando o número de bits de cada um, e também calculando a média dos números de bits dos caracteres presentes na janela deslizante de forma dinâmica, e caso este último valor seja inferior ao novo threshold, esta secção é adicionada à lista que vai ser retornada.

No fim é retornada a lista com as secções que foram possíveis de classificar com este novo valor de threshold superior.

### 3.1.3. Utilização Múltiplos Modelos

Recuperando agora o início da execução do programa, mais concretamente o passo seguinte à leitura do ficheiro `ftarget` para criação do alfabeto deste texto, as duas implementações descritas em cima são executadas quando a flag de utilização de múltiplos modelos está a falso. No caso da mesma possuir o valor de verdadeiro, são criados dois modelos de contexto finito distintos a partir do mesmo ficheiro de referência passado pelo utilizador, com o mesmo parâmetro alfa mas com tamanho de contexto diferentes, neste caso, um primeiro modelo com tamanho de contexto igual a 2 e um segundo modelo com um tamanho de contexto igual a 4.

De seguida, é chamada uma função designada `get_number_of_bits_required_to_compress_multiplemodel()` que recebe como argumentos os dois modelos, o caminho para o ficheiro alvo e o alfabeto do texto alvo.

Esta função tem como principal objetivo calcular o número de bits necessários para comprimir o ficheiro alvo com base em dois modelos criados a partir do ficheiro de referência. Para isso, atribui de forma dinâmica pesos diferentes a cada um dos modelos durante o

processo de compressão, de acordo com a performance destes mesmos modelos na compressão dos caracteres já lidos.

Assim sendo, a função começa por atribuir um peso igual a cada modelo (0.5) e por, de forma idêntica às funções já descritas anteriormente `get_number_of_bits_required_to_compress_v1` e `get_number_of_bits_required_to_compress_v2`, calcular o número de caracteres presentes no ficheiro `ftarget` que não pertencem ao alfabeto do ficheiro modelo de contexto finito.

De seguida, vai selecionar um contexto inicial para cada um dos modelos de entre os contextos conhecidos pelos modelos para iniciar a compressão. São usados dois contextos diferentes visto que cada modelo trabalha com um tamanho de contexto diferente. Contudo ao longo do tempo estes dois contextos são atualizados da mesma forma e diferem simplesmente no tamanho.

Prosseguindo, a função vai ler carácter a carácter o texto, e para cada um destes vai calcular o número de bits necessário para o comprimir utilizando cada um dos dois modelos. Finalmente, o número total de bits necessário para comprimir o texto inteiro é atualizado tendo em conta os pesos atuais atribuídos a cada modelo, os contextos de cada modelo são atualizados e os pesos atribuídos a cada modelo também.

### 3.2. `findlang.py`

Relativamente ao programa desenvolvido na segunda fase do projeto (`findlang.py`), o mesmo tem como objetivo identificar em qual língua foi escrito um determinado texto alvo e para conseguir esse mesmo objetivo utilizamos como base o programa `lang.py` para calcular o número de bits necessários para comprimir um determinado texto tendo em conta vários textos de referência de diferentes línguas. Assim sendo, e tendo em conta que textos semelhantes vão ser comprimidos utilizando um menor número de bits, podemos efetuar uma previsão da língua associada a um determinado texto verificando “o seu grau de semelhança” com os textos de referência. O texto a partir do qual foi criado o modelo que permitir comprimir o texto com menor número de bits terá, em princípio, a mesma língua do texto alvo.

Posto isto, para correr o programa `findlang.py`, são necessários especificar 3 argumentos:

- `-ftarget` : Define o caminho para o ficheiro que contém o texto para analisar;
- `-k` : Define o tamanho (número de caracteres) dos contextos do finite context model;
- `-a` : Define o smoothing parameter utilizado no cálculo das probabilidades;
- `-multiplemodels` : Flag utilizada para programa utilizar combinação de múltiplos modelos de contexto finito no cálculo do número de bits necessários para comprimir o texto alvo.

Inicialmente, o primeiro passo a ser efetuado é a verificação dos parâmetros fornecidos pelo utilizador. No que diz respeito ao parâmetro `ftarget`, não existe nenhuma verificação direta, apenas na leitura do ficheiro fornecido, caso o ficheiro não exista, o utilizador é informado. O parâmetro `k` deve ser um número inteiro e superior a 0, visto que o objetivo do mesmo é definir o comprimento do contexto utilizado pelo modelo de contexto

finito. Por sua vez, no parâmetro  $\alpha$ , é verificado que o mesmo é um float e que é um valor também ele superior a 0.

Posteriormente, o programa cria um dicionário onde estão presentes os caminhos a percorrer para chegar até aos ficheiros que contêm os textos de referência de cada língua utilizada. As chaves do dicionário são identificadores das línguas e os valores associados são o caminho relativo até ao ficheiro de texto.

De seguida, através de um ciclo For, vão ser percorridas todas as línguas presentes no dicionário e para cada uma delas é utilizado o programa *lang.py* para sabermos o número de bits que são necessários para comprimir o texto alvo tendo como referência o texto associado à língua em análise. À medida que as línguas vão sendo percorridas, é guardada numa variável o identificador da língua e o número de bits utilizados para a melhor solução, ou seja, a solução que precisa de um menor número de bits visto que um menor número de bits indica que existe uma maior semelhança entre os texto alvo e referência.

Após percorrer todas as línguas de referência, a língua prevista é mostrada ao utilizador através do terminal.

Necessário realçar que a flag “multiplemodels” que pode ser controlada pelo utilizador, permite ao mesmo utilizar o programa com a flag a verdadeiro efetuando assim uma previsão mais demorada visto que necessita da criação de um maior número de modelos mas também, em princípio, mais precisa ou colocar a flag a falso e efetuar uma previsão que possui na mesma um alto nível de precisão e é mais rápida.

### 3.3. locatlang.py

Relativamente ao programa desenvolvido na terceira e última fase do projeto (*locatlang.py*), o mesmo tem como objetivo, a partir de um conjunto de textos de referências de diferentes línguas, identificar num determinado texto alvo quais as secções deste mesmo texto foram escritas em diferentes língua e para cada uma destas secções identificar onde a mesma inicia e qual a língua correspondente.

Posto isto, para correr o programa *locatlang.py*, são necessários especificar 3 argumentos:

- *-ftarget* : Define o caminho para o ficheiro que contém o texto para analisar;
- *-k* : Define o tamanho (número de caracteres) dos contextos do finite context model;
- *- $\alpha$*  : Define o smoothing parameter utilizado no cálculo das probabilidades.

Inicialmente e de forma idêntica aos programas anteriores, o primeiro passo a ser efetuado é a verificação dos parâmetros fornecidos pelo utilizador. No que diz respeito ao parâmetro *ftarget*, não existe nenhuma verificação direta, apenas na leitura do ficheiro fornecido, caso o ficheiro não exista, o utilizador é informado. O parâmetro *k* deve ser um número inteiro e superior a 0, visto que o objetivo do mesmo é definir o comprimento do contexto utilizado pelo modelo de contexto finito. Por sua vez, no parâmetro  $\alpha$ , é verificado que o mesmo é um float e que é um valor também ele superior a 0.

No seguimento, o programa cria um dicionário onde estão presentes os caminhos a percorrer para chegar até aos ficheiros que contêm os textos de referência de cada língua utilizada. As chaves do dicionário são identificadores das línguas e os valores associados são o caminho relativo até ao ficheiro de texto.

De seguida, o programa irá ler o ficheiro alvo de modo a obter o alfabeto deste mesmo ficheiro, bem como o número total de caracteres que possui.

A partir deste ponto, como já foi referido anteriormente, foram desenvolvidas duas implementações que descrevemos abaixo:

### 3.3.1. Criação das palavras presentes no texto alvo

Começando com a primeira solução que desenvolvemos, o próximo passo que o programa efetua é percorrer todas as línguas presentes no dicionário e para cada uma delas é utilizado o programa *lang.py* para recolher a lista de palavras pertencentes a cada língua. Cada palavra vai ser representada por um tuplo que contém a posição inicial e final destas mesmas palavras no texto. De recordar que este processo é efetuado tendo em conta um threshold que é descrito na implementação do programa *lang.py*. Após isto, existe a truncção de cada lista de palavras de uma mesma língua, ou seja, as palavras adjacentes que foram classificadas como sendo da mesma língua são juntas numa única secção de forma que apenas fique guardada a posição inicial e final de cada secção. Este passo pode ser visualizado no seguinte exemplo:

```
lista inicial = [(14, 25), (26, 30), (40, 47), (54, 60), (61, 66), (162, 169), (170, 176), (177, 182),  
(183, 186), (187, 190), (197, 200), (211, 216), (217, 221)]  
resultado final = [(14, 30), (40, 47), (54, 66), (162, 190), (197, 200), (211, 221)]
```

Neste exemplo, considerando que a lista que está a ser processada pertence às palavras classificadas como sendo da língua portuguesa, podemos ver que no fim o resultado fica mais compacto e mais fácil de compreender as secções do texto.

### 3.3.2. Janela deslizante

Na nossa segunda implementação baseada numa janela deslizante, o passo seguinte do programa é percorrer todas as línguas, criar o modelo de contexto finito com o texto referência de cada língua, calcular as suas probabilidades através do método `calculate_probabilities()`, e invocar a função `get_number_of_bits_required_to_compress_v2()` implementada no ficheiro *lang.py* com os seguintes parâmetros:

- o modelo acabado de criar;
- o caminho para o ficheiro alvo;
- o alfabeto do ficheiro alvo;
- o tamanho da janela deslizante;
- e a `multiplelangflag` a `True` uma vez que queremos que seja calculada a lista de secções.

Com a lista de secções retornada, invocamos a função `truncate_and_merge_sections()` com os seguintes parâmetros:

- O dicionário global que vai guardar como key a secção e como valor a lista de línguas que foi capaz de comprimir a secção de uma boa forma;
- A lista de secções a adicionar ao dicionário global;
- A língua a que todas as secções da lista de secções a adicionar pertence;

O objetivo desta função é comprimir ao máximo o número de secções da seguinte forma:

Nota: neste exemplo, o tamanho da janela deslizante é de 5 carateres.

```
lista inicial = [ (3, 8), (5, 10), (12, 17), (16, 21), (20, 25), (27, 32) ]  
lista truncada = [ (3, 10), (12, 25), (27, 32) ]
```

Após percorrer todas as línguas e obter o dicionário com as secções de cada língua, o próximo passo é obter as secções do texto que não foram classificadas com nenhuma língua. Estas secções foram obtidas através de dois processos. Primeiramente obtém-se as posições que não pertencem às secções que estão guardados no dicionário global, da seguinte forma:

Nota: neste exemplo, vamos assumir que o ficheiro alvo tem 40 caracteres.

```
lista secções = [ (3, 10), (12, 25), (27, 32) ]  
lista posições = [ 1, 2, 11, 26, 33, 34, 35, 36, 37, 38, 39, 40 ]
```

Após ter a lista de posições, o passo seguinte é converter esta lista de posições para uma lista de secções, sendo que existe uma verificação do tamanho da secção, isto é, se o comprimento da secção a que ano foi atribuída uma língua for inferior ao tamanho da janela deslizante, esta secção vai ser descartada de acordo com o seguinte exemplo:

Nota: neste exemplo, o valor de k é 5.

```
lista posições = [ 1, 2, 11, 26, 33, 34, 35, 36, 37, 38, 39, 40 ]  
lista secções = [ (33, 40) ]
```

De seguida, com a lista de secções restantes, o programa vai voltar a percorrer a lista de línguas, calcular os modelos de contexto finitos, calcular as probabilidades do modelo, e invocar a função `get_sections_from_remaining_sections()` introduzindo como parâmetros o modelo de contexto finito, o caminho para o ficheiro alvo, o alfabeto do ficheiro alvo, o valor de k para o tamanho da janela deslizante, e a lista de secções que não foram classificadas com nenhuma língua de forma a fazer o mesmo processo anteriormente só que com um valor de threshold superior como foi referido na implementação deste método na secção lang.py. Depois do programa obter as novas secções agora classificadas, vai novamente invocar a função `truncate_and_merge_sections()` de forma a atualizar o dicionário de secções final.

Após este passo, o dicionário é ordenado pelo valor do começo de cada secção, e é mostrado ao utilizador as secções finais bem como as línguas associadas.

## 4. Análise de resultados

Nesta secção do relatório iremos apresentar algumas experiências efetuadas com os programas desenvolvidos e analisar os resultados obtidos a partir das mesmas.

Para realização destes testes foram recolhidos e utilizados 55 textos distribuídos por 15 línguas distintas, bem como 6 ficheiros escritos em mais do que uma língua.

### 4.1. lang.py

Relativamente ao programa *lang.py* fizemos algumas experiências relacionadas com a variação dos parâmetros  $k$  e  $\alpha$ , fixando o ficheiro de referência e o ficheiro alvo utilizados.

Começando pela análise da influência dos parâmetros  $k$  e  $\alpha$ , na tabela abaixo podemos verificar os resultados obtidos relativos ao número de bits necessários para comprimir um texto ao variar o  $\alpha$  com valores entre 0.1 e 0.6 e ao variar o parâmetro  $k$  entre 1 e 6. Para este teste o ficheiro alvo de análise é um texto de língua francesa (7.7MB) e o ficheiro de referência é também um texto de língua francesa (4.0MB).

	$k = 1$	$k = 2$	$k = 3$	$k = 4$	$k = 5$	$k = 6$
$\alpha = 0.1$	12,844,304	9,864,631	7,495,390	6,267,713	5,952,250	6,145,415
$\alpha = 0.2$	12,846,071	9,887,050	7,598,152	6,548,614	6,500,770	7,032,244
$\alpha = 0.3$	12,847,731	9,906,501	7,684,564	6,774,296	6,919,536	7,675,310
$\alpha = 0.4$	12,849,331	9,924,158	7,761,444	6,968,564	7,267,276	8,190,859
$\alpha = 0.5$	12,850,878	9,940,560	7,831,766	7,141,575	7,568,364	8,625,373
$\alpha = 0.6$	12,852,391	9,956,012	7,897,145	7,298,862	7,835,792	9,002,940

Tabela 1 Variação do número de bits necessários para comprimir um texto em função de  $k$  e do  $\alpha$

A partir dos resultados demonstrados na tabela acima, podemos concluir que à medida que o valor do parâmetro  $k$  aumenta, existe uma tendência para o número de bits diminuir. Isto deve-se ao facto de que, à medida que o contexto aumenta, temos mais informação relativamente ao texto anterior ao carácter que estamos a codificar, sendo mais fácil prever qual vai ser este mesmo carácter, reduzindo assim o número de bits necessário para o comprimir.

O próximo teste que realizamos teve como objetivo verificar que o número de bits com que o texto alvo pode ser comprimido tem dependência da semelhança entre o texto alvo e o texto de referência utilizado para criar o modelo de contexto finito.

Na seguinte tabela apresentamos os resultados obtidos ao comprimir um ficheiro alvo que contém um texto de língua francesa (4.0MB) utilizando os seguintes textos de referência:

1. texto francês (7.7MB);
2. texto alemão (4.0MB);
3. texto russo (7.5MB).

Reference		$k = 1$	$k = 2$	$k = 3$	$k = 4$	$k = 5$	$k = 6$
FRA	$\alpha = 0.1$	13,058,660	10,278,172	7,958,875	6,626,460	6,195,773	6,310,719
GER	$\alpha = 0.1$	19,367,924	19,640,678	22,279,826	23,644,114	24,858,473	25,744,548
RUS	$\alpha = 0.1$	23,270,648	25,328,715	30,311,698	31,836,611	32,064,989	32,114,495
FRA	$\alpha = 0.3$	13,060,594	10,306,590	8,093,624	7,015,520	6,991,781	7,649,200
GER	$\alpha = 0.3$	19,223,602	19,402,769	21,818,212	23,590,171	24,891,392	25,841,395
RUS	$\alpha = 0.3$	24,136,746	26,946,958	30,896,632	31,932,074	32,086,714	32,119,551
FRA	$\alpha = 0.6$	13,063,174	10,339,652	8,246,354	7,428,439	7,769,842	8,846,338
GER	$\alpha = 0.6$	19,150,390	19,303,204	21,675,925	23,453,514	25,028,431	25,957,476
RUS	$\alpha = 0.6$	25,082,476	28,166,716	31,253,908	31,988,803	32,099,262	32,122,472

Tabela 2 Variação do número de bits necessários para comprimir um texto em função de  $k$  e do  $\alpha$  e com variação do texto de referência

A partir destes resultados podemos concluir que o número de bits necessários para comprimir um determinado texto é totalmente dependente da semelhança entre o texto alvo e o texto referência. Podemos ainda concluir que esta semelhança está também muito ligada às línguas em que os textos se apresentam. Posto isto, o número de bits necessários para comprimir um texto é significativamente menor quando o texto de referência e o texto alvo pertencem à mesma língua.

Por fim, no próximo e último teste relativo a este programa, pretendemos analisar como funciona o programa lang.py quando acionamos a flag que promove a utilização de múltiplos modelos pelo programa no cálculo do número de bits e qual o ganho ou perda ao utilizar este extra.

Para isso, utilizamos um texto alvo português (4.0MB) e um texto de referência também português (325KB). O valor de  $\alpha$  é sempre 0.1 e o valor de  $k$  varia entre 1 e 6.

	Bits	Bits (multiplemodels)	Diferença
$k = 1$	13,882,781	13,184,335	-698,446
$k = 2$	12,847,543	13,184,335	+336,792
$k = 3$	13,222,256	13,184,335	-37,921
$k = 4$	14,809,399	13,184,335	-1,625,064

$k = 5$	17,643,148	13,184,335	-4,458,813
$k = 6$	20,510,892	13,184,335	-7,326,557

Tabela 3 Diferença entre o número de bits necessários para comprimir um texto ao usar múltiplos modelos

De recordar que, ao utilizar a flag que promove a utilização de múltiplos modelos, o utilizador não necessita de especificar os valores do tamanho do contexto desses modelos, visto que o próprio programa define esses valores. Este facto, permite ao utilizador não necessitar de possuir qualquer informação relativa a quais os valores de  $k$  fornecem uma melhor performance, porque o próprio programa define esses valores.

Posto isto, na tabela podemos verificar que a utilização de múltiplos modelos fornece uma melhor performance do que praticamente quase todas as utilizações de modelos simples. Isto ocorre devido ao facto de que a utilização de múltiplos modelos permite a que o programa utilize em diferentes secções do texto alvo diferentes pesos para cada modelo, variando esses pesos em função da performance dos diferentes modelos, permitindo uma melhor performance geral.

## 4.2. `findlang.py`

Para colocarmos à prova o programa `findlang.py`, realizámos alguns testes com textos de diferentes línguas.

De maneira a termos um conjunto de línguas possíveis relativamente abrangente, temos como referência um total de 15 textos, cada um deles pertence a uma língua diferente. Os parâmetros  $k$  e  $\alpha$  foram fixados com o valor de 3 e 0.1, respetivamente.

Ficheiro Alvo	Língua	Língua prevista	Língua prevista (multiplemodels)
afghanistan-small.utf8 (13kB)	Afegão	Afegão	Afegão
afrikaans-small.utf8 (135kB)	Africanês	Africanês	Africanês
arabic-small.utf8 (241kB)	Árabe	Árabe	Árabe
bulgarian-small.utf8 (246.8kB)	Búlgaro	Búlgaro	Búlgaro
croatian-small.utf8 (42.2kB)	Croata	Croata	Croata
danish-small.utf8 (260.1kB)	Dinamarquês	Dinamarquês	Dinamarquês
english-small.utf8 (256.5kB)	Inglês	Inglês	Inglês
spanish-small.utf8 (230kB)	Espanhol	Espanhol	Espanhol
finnish-small.utf8 (128.2kB)	Finlandês	Finlandês	Finlandês
french-small.utf8 (275kB)	Francês	Francês	Francês
german-small.utf8 (830.4kB)	Alemão	Alemão	Alemão
greece-small.utf8 (782.6kB)	Grego	Grego	Grego

hungarian-small.utf8 (136.8kB)	Húngaro	Húngaro	Húngaro
icelandic-small.utf8 (131.7kB)	Islandês	Islandês	Islandês
italian-small.utf8 (274kB)	Italiano	Italiano	Italiano
polish-small.utf8 (258.2kB)	Polaco	Polaco	Polaco
portuguese-small.utf8 (128kB)	Português	Português	Português
russian-small.utf8 (443kB)	Russo	Russo	Russo
ukrainian-small.utf8 (294kB)	Ucraniano	Ucraniano	Ucraniano

Tabela 4 Previsões efetuadas pelo programa *findlang.py* para 15 línguas diferentes

Como podemos ver segundo os resultados obtidos na tabela acima, podemos concluir, que o programa *findlang.py* para os 15 textos testados, consegue prever de maneira correta a língua presente em cada um deles. A accuracy do programa pode ser calculada da seguinte forma:

$$\frac{\text{número de previsões corretas}}{\text{número total de previsões}} \times 100\%$$

Podemos então observar que para estas experiências a accuracy do programa foi de 100%. Em adição, foram realizadas diversas experiências utilizando outros textos com maior e menor tamanho quer como texto referência quer como texto alvo e em todas as experiências a previsão efetuada foi a correta.

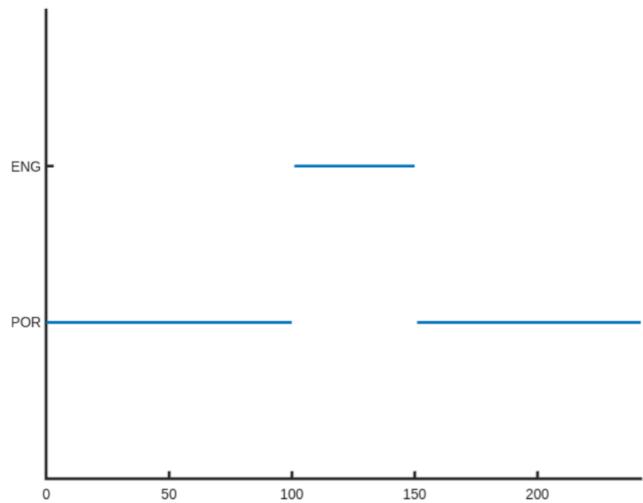
### 4.3. locatelang.py

Por último, realizamos um conjunto de testes relativos ao programa *locatelang.py* para avaliar a performance do mesmo. Relembrar que, para este programa em específico, foram desenvolvidas duas implementações possíveis. Iremos comparar os resultados obtidos com cada uma delas.

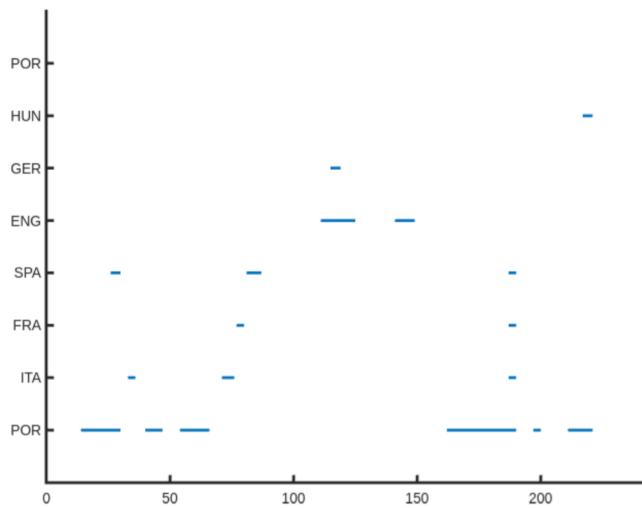
A primeira experiência realizada consistiu em executar o programa *locatelang.py* com os ficheiros multilang1-PT-ENG-PT.txt e multilang2-ENG-IT-FR.txt. Estes ficheiros encontram-se escritos em mais do que uma língua, possuindo cada um deles 3 secções distintas.

Nos gráficos seguintes, apresentamos um gráfico com a solução ótima relativamente à deteção das secções por parte do programa, e de seguida apresentamos dois gráficos, cada um com o resultado de uma das implementações que foram desenvolvidas. As experiências foram realizadas para um valor de  $\alpha$  igual a 0.5 e um valor de  $k$  igual a 4.

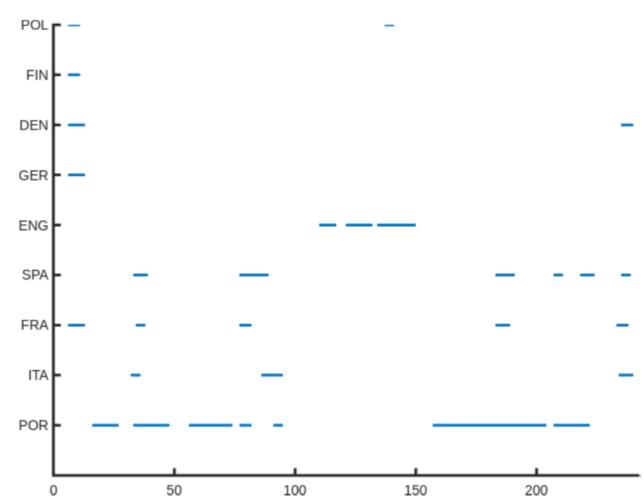
- Resultados relativos ao ficheiro multilang1-PT-ENG-PT.txt:



*Figura 1: Solução óptima de classificação*

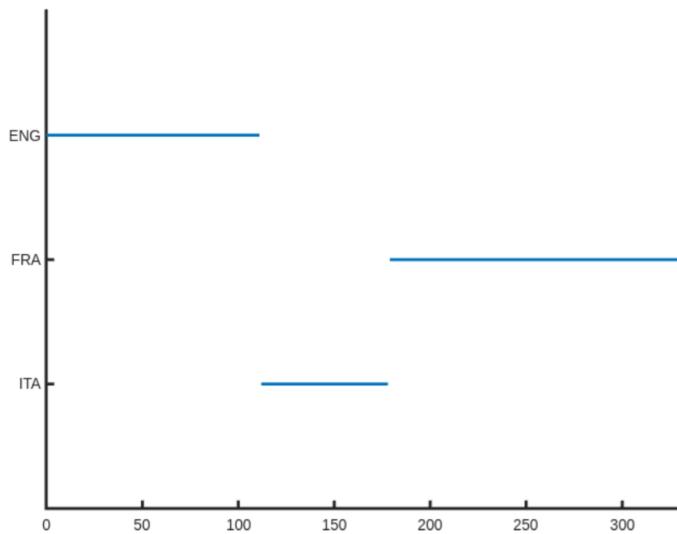


*Figura 2: Resultado obtido para implementação 1*

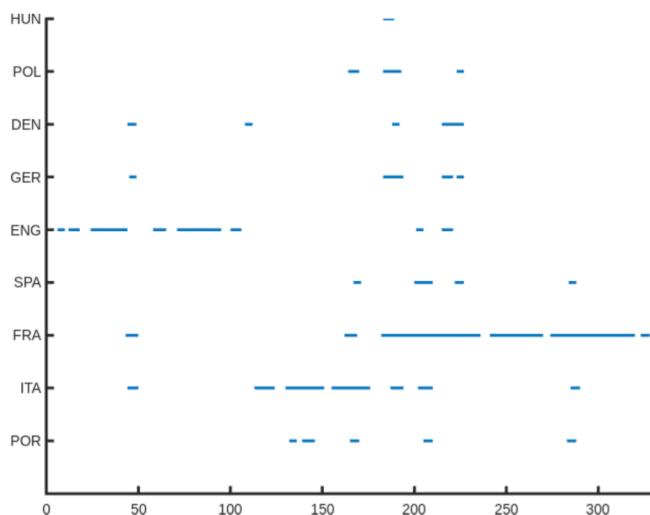


*Figura 3: Resultado obtido para implementação 2*

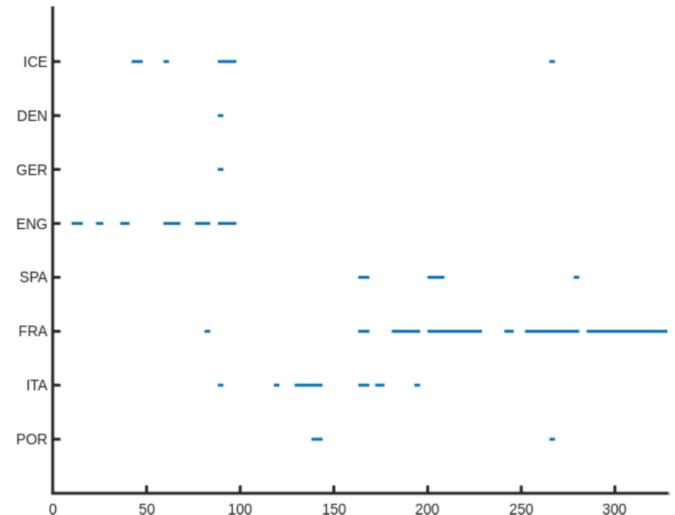
- Resultados relativos ao ficheiro multilang2-ENG-IT-FR.txt:



*Figure 4: Solução ótima de classificação*



*Figure 5: Resultado obtido para implementação 1*



*Figure 6: Resultado obtido para implementação 2*

A análise dos gráficos com os resultados referentes a ambos os ficheiros utilizados e a ambas as implementações implementadas, permite-nos aferir que de um modo geral, ambas as implementações têm um comportamento adequado e oferecem uma boa deteção das secções que um texto possui, bem como das línguas associadas a cada uma dessas secções.

Efetuando agora alguma distinção entre ambas as implementações, a segunda implementação (Janela Deslizante) consegue detetar ligeiramente melhor as secções das línguas que efetivamente existem. No entanto, a segunda implementação também deteta uma maior quantidade de secções erradas. Posto isto, a primeira implementação (Criação Palavras), fornece um resultado com um menor número de secções erradas, mas ao mesmo tempo deteta também ligeiramente menos secções verdadeiras que a segunda implementação.

A segunda experiência que efetuámos está relacionada com a influência do valor do parâmetro  $k$  nas secções que são detetadas pelo programa. Posto isto, para o ficheiro alvo mullang1-PT-ENG-PT.txt, fixámos o valor de  $\alpha$  em 0.5 e variámos o valor de  $k$ . Os resultados obtidos para ambas as implementações estão presentes nos gráficos seguintes.

- Resultados relativos à primeira implementação (Criação Palavras):

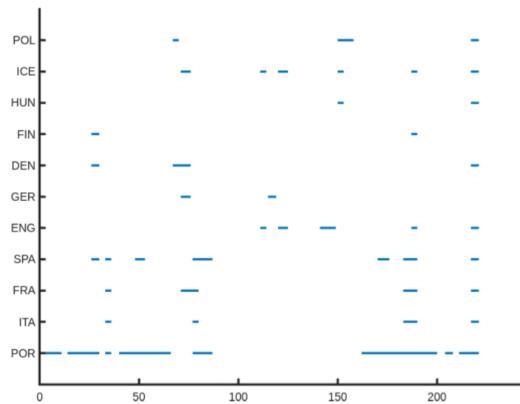


Figure 3: Resultado obtido para a primeira implementação com  $k=3$

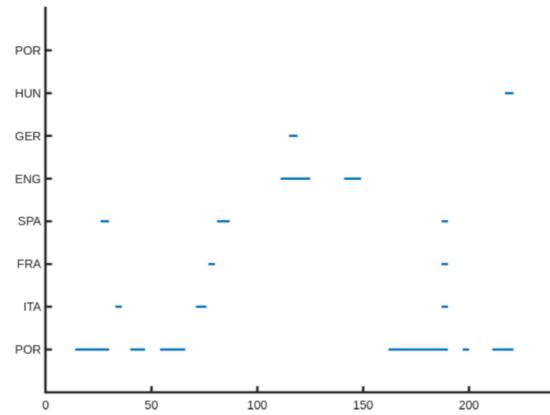


Figure 4: Resultado obtido para a primeira implementação com  $k=4$

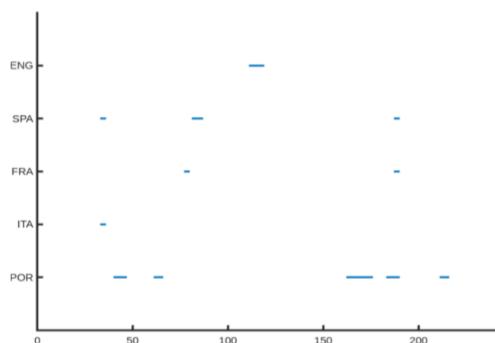


Figure 5: Resultado obtido para a primeira implementação com  $k=5$

- Resultados relativos à segunda implementação (Janela Deslizante):

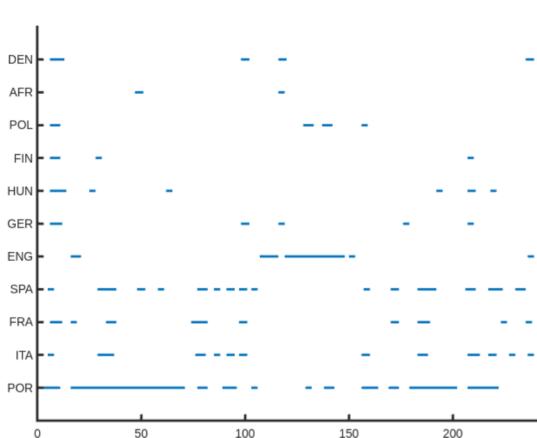


Figure 7 Resultado obtido para a segunda implementação com  $k=3$

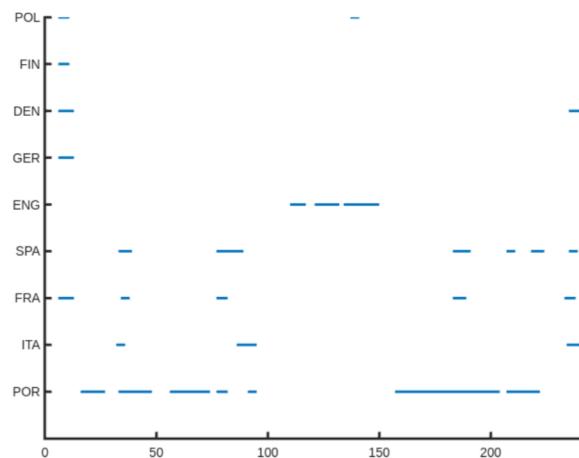


Figure 6 Resultado obtido para a segunda implementação com  $k=4$

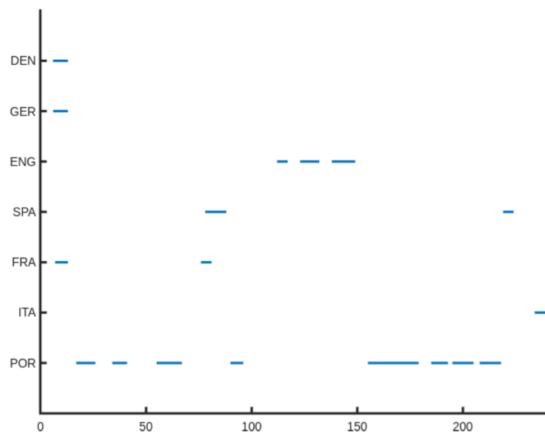


Figure 8 Resultado obtido para a segunda implementação com  $k=5$

A partir dos gráficos apresentados acima, podemos afirmar que ambas as implementações do programa *locatelang.py* têm uma melhor performance para valores de  $k$  igual a 4 e 5. No que diz respeito aos gráficos relativos ao valor de  $k$  igual a 3, existe um grande conjunto de secções que são identificadas como sendo de variadas línguas. No que diz respeito aos gráficos relativos ao valor de  $k$  igual a 5, existem muito poucas secções identificadas como sendo de uma língua que não a sua, no entanto começamos também já a perder alguma capacidade de identificação das secções que realmente existem. Posto isto, ambas as implementações funcionam bem para valores de  $k$  igual a 4 e 5 e acaba por ser a necessidade do utilizador que dita qual a melhor escolha. Se o utilizador necessitar de identificar secções com o menor número possível de secções erradas, o valor de 5 pode ser uma melhor opção. Se o utilizador estiver disponível para abdicar desse menor número de erros e estiver mais interessado em identificar o maior número de secções reais, o valor de 4 poderá ser a melhor opção.

A terceira e última experiência que realizámos teve como principal objetivo avaliar a performance do segundo threshold na execução do programa *locatelang.py* com a segunda implementação (Janela Deslizante). De recordar que, nesta implementação após a primeira classificação das secções do texto alvo, existe uma segunda avaliação das secções que não foram classificadas como pertencentes a nenhuma língua na primeira passagem. Posto isto, pretendemos avaliar a performance deste threshold dinâmico.

Assim sendo, nos seguintes 3 gráficos, apresentamos um deles com a solução ótima, um segundo gráfico com as classificações que foram efetuadas na primeira passagem e um último gráfico com as classificações efetuadas pelo segundo threshold. Os resultados foram obtidos para o ficheiro multilang1-PT-ENG-PT.txt, valor de  $k$  igual a 4 e  $\alpha$  igual a 0.5.

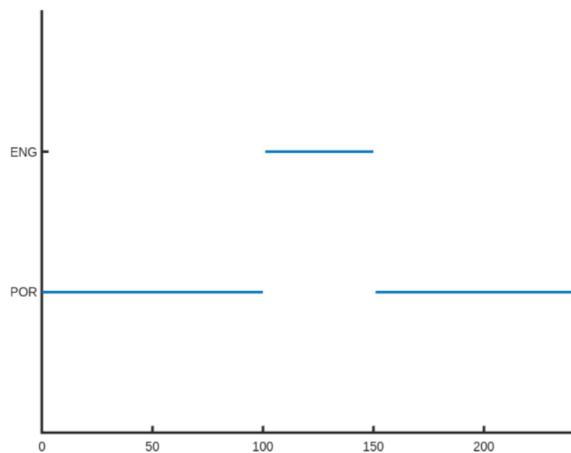


Figure 11 Solução ótima de classificação

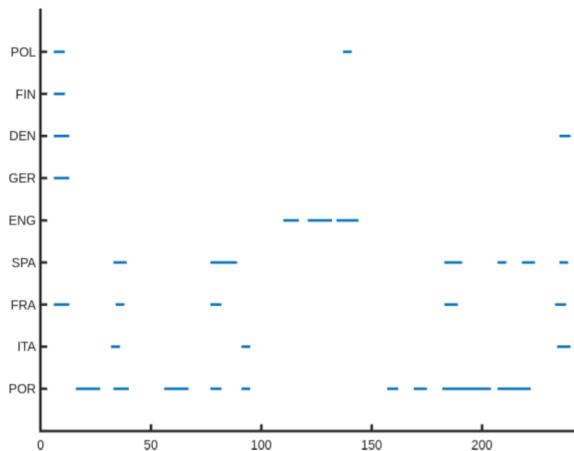


Figure 10 Secções classificadas pelo primeiro threshold

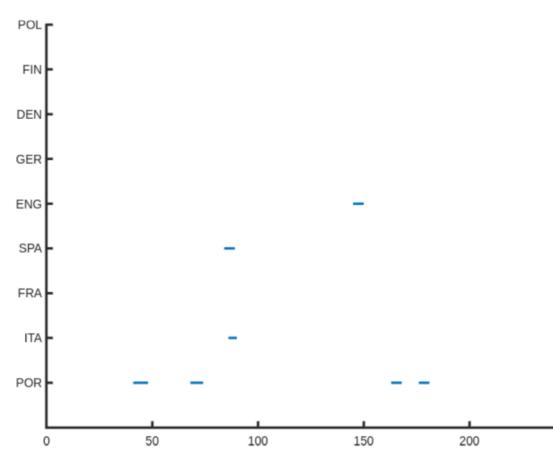


Figure 9 Secções classificadas pelo segundo threshold

Como podemos ver na análise dos gráficos, o segundo threshold torna possível a classificação de algumas secções extra, na sua grande maioria bem classificadas, pelo que se revela uma mais valia para o programa.

## 5. Conclusão

Com a realização deste projeto conseguimos interligar os conceitos estudados no primeiro projeto com uma aplicação útil para esses mesmos conceitos. Neste caso, utilizar o modelo de contexto finito para detetar semelhança entre textos.

Esta medida de semelhança que é obtida através do número de bits necessários para comprimir um texto alvo com base num texto referência permite-nos efetuar previsão da língua dos textos alvo, desde que conheçamos a língua dos nossos textos referência.

Verificámos ainda que a utilização de múltiplos modelos na obtenção destas previsões permite alcançar resultados tão bons ou melhores do que a utilização de um modelo simples, isto porque o programa consegue-se adaptar e identificar qual o melhor modelo para utilizar em cada secção do texto.

Em adição, o desenvolvimento de um programa capaz de identificar secções de texto semelhante dentro de um texto escrito em diferentes línguas demonstrou-se um desafio enriquecedor.

Um aspeto importante de realçar foi a capacidade em desenvolver duas soluções para este último problema procurando obter a melhor solução possível para o mesmo, inclusive através da execução de diversas experiências e testes relativos a algumas das decisões relacionadas com a implementação das soluções.

Por fim, consideramos que este projeto foi útil para o desenvolvimento das nossas capacidades individuais como futuros engenheiros informáticos, uma vez que se tratou de um desafio novo que conseguimos realizar, na nossa opinião, com sucesso.