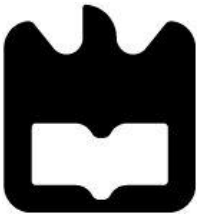
IC - Assignment III

Universidade de Aveiro

Departamento de Eletrónica, Telecomunicações e Informática

Mestrado em Engenharia Informática



Diogo Carvalho nº92969 1/3

Hugo Ferreira nº93093 1/3

Rafael Baptista nº93367 1/3

06 de Janeiro de 2023

Índice

1. **Introdução** 3
2. **Metodologia** 4
3. **Implementação e Modo Utilização Programas** 5
4. **Análise de resultados** 16
5. **Conclusão** 30
6. **Links** 30

1. Introdução

O presente relatório pretende descrever o trabalho realizado na execução do terceiro projeto proposto na unidade curricular de Informação e Codificação, cuja estrutura do enunciado se encontrava dividida em 3 partes distintas.

A primeira fase do projeto era constituída por um exercício único relacionado com a construção de um programa que permita recolher informação estatística de uma fonte de texto através do uso de um modelo de contexto finito, atribuindo probabilidade estimada para a ocorrência dos diferentes símbolos do alfabeto tendo em conta um contexto (sequência de símbolos) prévio.

Na segunda parte, o enunciado é formado por três exercícios obrigatórios e um adicional.

No primeiro é solicitado o desenvolvimento de um programa 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 construído a partir de um texto referência. Este programa irá utilizar o modelo de contexto finito desenvolvido na primeira fase do projeto.

No segundo exercício foi proposta a construção de um segundo programa que, com base no primeiro programa, 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.

No terceiro exercício, foi proposto o desenvolvimento de um terceiro programa 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.

Para complementar, o enunciado continha um exercício extra onde fomos desafiados a utilizar múltiplos modelos FCM para representar cada língua.

Por fim, a última parte do projeto consiste na escrita deste mesmo documento cujo objetivo é explicar as decisões efetuadas e resultados obtidos ao longo do projeto.

2. Metodologia

Para a realização deste projeto, utilizamos a plataforma Github como repositório de código compartilhado. A estrutura do nosso repositório é constituída por duas pastas distintas. A pasta "src" contém todo o código desenvolvido ao longo do projeto, uma pasta “examples” que contém os ficheiros de texto utilizados para testar os diferentes programas que foram desenvolvidos.

No que diz respeito à comunicação entre os elementos do grupo, utilizamos as plataformas Discord e Messenger 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 linguagem de programação C++. A estrutura do código é composta por:

- Programa *fcm.cpp* que utiliza a classe FCM definida no ficheiro *fcm.h* para calcular a entropia de um ficheiro de texto.
- Programa *lang.cpp* que faz uso da classe FCM definida no ficheiro *fcm.h* e de funções definidas no ficheiro *lang.h* para calcular o número de bits necessários para comprimir um texto alvo, a partir de um texto de referência.
- Programa *findlang.cpp* que, contendo um conjunto de vários textos de referência de diferentes línguas, faz uso das funções definidas no ficheiro *lang.h* para conseguir efetuar uma previsão da língua utilizada para escrever o texto alvo a ser analisado.
- Programa *locatelang.cpp* que, contendo um conjunto de vários textos de referência de diferentes línguas, recebe um texto alvo escrito em várias línguas e vai fazer uso das funções definidas no ficheiro *lang.h* 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 aspetos extra na realização deste projeto. Primeiramente, no que diz respeito ao programa *locatelang.cpp*, 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.cpp* e *findlang.cpp*, 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.cpp*.

3. Implementação e Modo Utilização Programas

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.cpp*. 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. fcm.cpp e fcm.h

Vamos começar por descrever o programa desenvolvido na primeira fase do projeto, *fcm.cpp*. O mesmo tem como objetivo efetuar o cálculo das probabilidades de ocorrência dos diferentes símbolos que compõem o nosso alfabeto de símbolos após um determinado contexto que possui comprimento *k* (parâmetro definido pelo utilizador). Em adição, o programa está implementado para executar também o cálculo da entropia do texto fornecido.

Para executar este programa é necessário especificar o caminho para o ficheiro de texto que contém o texto para analisar e ainda é possível customizar o seguinte conjunto de parâmetros:

- -k : Define o número de caracteres dos contextos;
- -a : Define o smoothing parameter utilizado no cálculo das probabilidades;

```
                          Usage:
./fcm <input_text_file> [-k context_length (def 3)]
                        [-a alpha_value (def 0.5)]
                  Usage Examples:
      ./fcm ../examples/example.txt -k 3 -a 0.4
```

Ao executar o nosso programa, o primeiro passo a ser efetuado é a verificação dos parâmetros fornecidos pelo utilizador. No que diz respeito ao parâmetro relacionado com o texto, não existe nenhuma verificação direta, apenas na leitura do ficheiro fornecido, caso o mesmo não exista, o utilizador é informado. O parâmetro k deve ser um número inteiro positivo, 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 se encontra no intervalo]0,1].

Por razões de qualidade de código, de forma a garantir uma fácil interpretação e manutenção do mesmo, decidimos criar uma classe Fcm que é utilizada para a implementação do Finite Context Model. O próximo passo do nosso programa consiste então

na criação de um objeto desta classe, onde são necessários 3 argumentos para a criação do mesmo:

- nome do ficheiro a analisar
- o valor de k que corresponde ao tamanho dos contextos;
- o valor do smoothing parameter (α).

O objeto ao ser criado, vai guardar algumas variáveis extra para além dos 3 argumentos que recebe, que vão ser necessárias no cálculo da entropia. Mais concretamente as variáveis são:

- alphabet: set que contém todos os símbolos do texto;
- number_of_states: corresponde ao número total de contextos que existem no texto;
- model: o modelo que é inicializado como um dicionário vazio;
- state_probabilities: as probabilidades associadas aos caracteres que surgem após cada contexto que é inicializado também como um dicionário vazio;
- context_probabilities: as probabilidades de ocorrência de cada um dos contextos existentes no texto a analisar.

Na criação do objeto da classe Fcm, a função create_fcm_model() que pertence à classe Fcm é invocada. Esta função tem como objetivo ler o texto e percorrer o mesmo enquanto efetua o preenchimento do finite context model. Como referido em cima, o modelo é guardado na variável model, a qual é uma instância de um dicionário que vai ter como chaves os contextos, e como valores associados um outro dicionário que, por sua vez, possui como chaves os caracteres, e como valores o número de ocorrências desses caracteres após o contexto correspondente. É possível entender melhor o armazenamento explicado acima através do seguinte exemplo:

Para o seguinte texto "AABAABB", com k = 2:

- model = { "AA" : {"B": 2 }, "AB": {"A": 1, “B”: 1}, "BA": {"A": 1} }

Por fim, é efetuada a invocação da função calculate_entropy() que também pertence à classe Fcm. Esta função 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 (*a*) definido pelo utilizador. Após isso, calcula a entropia dentro de cada contexto e no fim efetua a soma dessa mesma entropia multiplicada pela probabilidade dos diferentes contextos por forma a obter a entropia final do texto.

Todos estes cálculos foram efetuados seguindo as fórmulas numéricas discutidas durante as aulas da unidade curricular.

3.2. lang.cpp e lang.h

Nesta secção vamos descrever a implementação do programa *lang.cpp* que foi desenvolvido já durante a segunda fase do projeto, e que 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 algumas funcionalidades que este programa contém têm propósito de servir os programas *findlang.cpp* e *locatelang.cpp,* pelo que, apesar de estarem relacionadas com estes dois últimos programas, como foram inseridas no ficheiro lang.h por forma a serem utilizadas por esses mesmos programas, vão ser explicadas já nesta secção.

Para executar este programa é necessário especificar o caminho para o ficheiro de texto que contém o texto que deve ser utilizado para criar o modelo de contexto finito, que iremos designar de ficheiro de referência, o caminho para o ficheiro de texto a ser analisado, que iremos designar de ficheiro alvo, e é ainda possível customizar o seguinte conjunto de parâmetros:

-k : Define o tamanho (número de caracteres) dos contextos do finite contextmodel;
-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.

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 relacionados com os textos, 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 pertencente ao intervalo]0,1].

Após a verificação dos parâmetros, é efetuada a chamada à função langCalculation() presente no ficheiro lang.h que será responsável por efetuar os cálculos e que foi inserida neste ficheiro para que possa ser reutilizada por outros programas.

Dentro desta função, é efetuada a leitura do ficheiro alvo 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.

Ainda no interior da função langCalculation(), , no caso do valor da “flag multiplemodels” ser falso, consiste em fazer a geração do modelo de contexto finito usando o ficheiro de referência. 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 de referência, o valor de *k* e o valor de α.

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. 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.cpp 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.cpp) é igual.

3.2.1. Criação das palavras presentes no texto alvo

A primeira solução que desenvolvemos para a detecção 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 detecção de secções de diferentes línguas no texto) que no caso do programa *lang.cpp* irá ter sempre o valor de falso.

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 target 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.cpp*. Este valor vai ser utilizado 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:

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

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:

$$Número\ de\ bits = -log_2(prob.evento)$$

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:

$$\frac{1}{Número\ de\ caracteres\ diferentes}$$

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:

$$\frac{alpha}{alpha * tamanho\ do\ alfabeto}$$

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.

3.2.2. 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) que no caso do programa *lang.cpp* irá ter sempre o valor de falso.

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 target 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:

$$threshold = \frac{1}{2} * \ log_2(|\varepsilon|)$$

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.cpp*, 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.cpp* 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:

$$threshold = \frac{3}{4} * \ log_2(|\varepsilon|)$$

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.

3.2.3. Utilização de Múltiplos Modelos

Recuperando agora o início da execução do programa, mais concretamente o passo seguinte à leitura do ficheiro target 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 target 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.

3.3. findlang.cpp

Relativamente ao programa *findlang.cpp*, o mesmo tem como objetivo identificar em qual língua foi escrito um determinado texto alvo e. para conseguir esse mesmo objetivo, utiliza como base funções descritas no ficheiro *lang.h* 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 permite comprimir o texto com menor número de bits terá, em princípio, a mesma língua do texto alvo.

Para executar este programa é necessário especificar o caminho para o ficheiro de texto que contém o texto a ser analisado, e é ainda possível customizar o seguinte conjunto de parâmetros:

-k : Define o tamanho (número de caracteres) dos contextos do finite contextmodel;
-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.

```
                        Usage:
./findlang <target_file_name> [-k context_length (def 3)]
                              [-a alpha_value (def 0.5)]
                              [--multiplemodels]
                    Usage Example:
./findlang ../examples/example.txt -k 3 -a 0.4
```

Inicialmente, o primeiro passo a ser efetuado é a verificação dos parâmetros fornecidos pelo utilizador. No que diz respeito ao ficheiro de texto, 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 α, é verificado que o mesmo é um float e que é um valor que se encontra no intervalo]0,1].

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 identificadoras 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 utilizada função presente no ficheiro *lang.h* para calcular 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.

3.4. locatelang.cpp

Relativamente ao programa *locatelang.cpp*, 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 diferente língua e para cada uma destas secções identificar onde a mesma inicia e qual a língua correspondente.

Para executar este programa é necessário especificar o caminho para o ficheiro de texto que contém o texto a ser analisado, e é ainda possível customizar o seguinte conjunto de parâmetros:

-k : Define o tamanho (número de caracteres) dos contextos do finite contextmodel;
-a : Define o smoothing parameter utilizado no cálculo das probabilidades;

```
                                Usage:
./locatelang <target_file_name> [-k context_length (def 3)]
                                [-a alpha_value (def 0.5)]

                            Usage Example:
./locatelang ../examples/multilang1-PT-ENG-PT.txt -k 3 -a 0.4
```

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 ficheiro de texto, 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 α, é verificado que o mesmo é um float e que é um valor também pertence ao intervalo]0,1].

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 identificadoras 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 desse 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:

3.4.1. 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 utilizada uma função definida no programa *lang.h* 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.cpp*. Após isto, existe a truncação 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.

3.4.2. 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.h* 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;
- 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 o objetivo de comprimir ao máximo o número de secções da seguinte forma:
Nota: neste exemplo, o tamanho da janela deslizante é de 5 caracteres.

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 não 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 tamanho da janela deslizante é de 5 caracteres.

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.

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 seção do relatório iremos apresentar algumas experiências efetuadas com os programas desenvolvidos e analisar os resultados obtidos.

4.1. fcm.cpp

Relativamente ao programa fcm.cpp fizemos algumas experiências relacionadas com a variação dos parâmetros k e α e o cálculo da entropia. Por fim efetuamos também alguns testes de performance.

Começando pela análise da influência dos parâmetros k e α, na tabela abaixo podemos verificar os resultados obtidos para o valor da entropia ao variar o α entre 0.1 e 1 com intervalos de 0.1 e com valores do parâmetro k entre 1 e 6. Os resultados foram obtidos através do texto presente no repositório examples/POR/portuguese-big.utf8.

Tabela 1: Variação da entropia em função de k e a

	k = 1	k = 2	k = 3	k = 4	k = 5	k = 6
α = 0.1	3.33	2.67	2.19	2.14	2.44	2.96
α = 0.2	3.33	2.70	2.31	2.40	2.87	3.51
α = 0.3	3.33	2.72	2.40	2.60	3.16	3.86
α = 0.4	3.34	2.74	2.48	2.76	3.39	4.11
α = 0.5	3.34	2.76	2.55	2.90	3.57	4.31
α = 0.6	3.34	2.78	2.61	3.02	3.73	4.48
α = 0.7	3.34	2.80	2.67	3.13	3.87	4.62
α = 0.8	3.35	2.82	2.73	3.22	3.99	4.74
α = 0.9	3.35	2.83	2.78	3.31	4.10	4.85
α = 1.0	3.35	2.85	2.83	3.40	4.20	4.94

Para uma melhor análise dos dados obtidos, apresentamos os dados no gráfico abaixo:

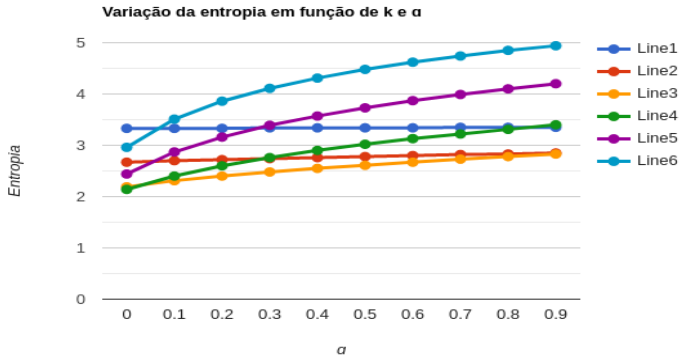


Figura 1: Variação da entropia em função dos parâmetros k e a

No eixo das abscissas do gráfico temos os valores de α e no das ordenadas os valores da entropia. Cada linha representa um valor de k diferente. Da análise dos resultados, podemos verificar que para valores de k inferiores existe uma menor variação da entropia ao aumentar o parâmetro α. À medida que o valor de k atinge valores mais altos (4, 5 e 6), a entropia tem uma maior variação. Isto pode estar relacionado com o facto de, em contextos mais pequenos (exemplo k = 1), existem ocorrências de determinados caracteres após outros que são bastante comuns (por exemplo o símbolo “h” surgir a seguir ao “t”). Ao aumentar o k, aumentamos também a incerteza do símbolo seguinte e por isso existe uma maior variação da entropia final.

No que diz respeito ao α, podemos também verificar que o valor de entropia é superior quando o valor de α é ele também mais elevado. Isto deve-se ao facto de que, com α mais elevados, existe uma maior probabilidade de ocorrência dos símbolos que nunca ocorreram após determinado contexto, aumentando a incerteza da fonte de informação e por sua vez a quantidade de informação é também superior.

Nas duas tabelas seguintes, está representada exatamente a mesma experiência, mas desta vez para duas fontes de texto diferentes. Um texto de tamanho médio (examples/POR/portuguese-medium.txt) e outro de tamanho mais pequeno (examples/POR/portuguese-small.utf8).

Texto tamanho médio:

Tabela 2: Variação da entropia em função de k e a

	k = 1	k = 2	k = 3	k = 4	k = 5	k = 6
α = 0.1	3.36	2.97	3.05	3.69	4.55	5.25
α = 0.2	3.38	3.09	3.41	4.26	5.14	5.74
α = 0.3	3.40	3.19	3.66	4.61	5.45	5.95
α = 0.4	3.42	3.27	3.85	4.84	5.64	6.08
α = 0.5	3.43	3.34	4.01	5.02	5.77	6.16

α = 0.6	3.45	3.40	4.14	5.16	5.87	6.22
α = 0.7	3.46	3.46	4.25	5.28	5.95	6.26
α = 0.8	3.47	3.51	4.36	5.38	6.01	6.30
α = 0.9	3.48	3.56	4.45	5.46	6.06	6.33
α = 1.0	3.50	3.61	4.53	5.53	6.10	6.35

Texto tamanho pequeno:

Tabela 3: Variação da entropia em função de k e a

	k = 1	k = 2	k = 3	k = 4	k = 5	k = 6
α = 0.1	3.36	2.94	3.12	3.82	4.55	5.12
α = 0.2	3.40	3.13	3.62	4.48	5.20	5.68
α = 0.3	3.43	3.28	3.94	4.85	5.53	5.93
α = 0.4	3.45	3.41	4.19	5.11	5.73	6.08
α = 0.5	3.48	3.52	4.38	5.29	5.87	6.18
α = 0.6	3.50	3.61	4.54	5.44	5.98	6.25
α = 0.7	3.53	3.70	4.68	5.55	6.06	6.31
α = 0.8	3.55	3.77	4.79	5.65	6.12	6.35
α = 0.9	3.57	3.85	4.89	5.73	6.17	6.38
α = 1.0	3.59	3.91	4.98	5.79	6.21	6.41

Podemos verificar pela análise das tabelas que as conclusões retiradas acima para o texto portuguese-big.utf8, se repetem agora para fontes de texto de tamanhos diferentes.

De seguida, procurámos avaliar a performance do nosso programa analisando a influência do parâmetro k na velocidade de cálculo da entropia para dois ficheiros de dimensões diferentes. Para ambos os ficheiros verificamos que quanto maior o valor de k maior o tempo que o programa demora a executar, e consequentemente, mais demorado é o cálculo da entropia. Esse tempo cresce de forma exponencial sendo que para o ficheiro médio com um k = 1 o tempo é de 0,10 segundos e para um k = 25 atinge os 6,36 segundos. Numa fase inicial, para valores de k menores, a variação é muito grande até ao valor de k = 12. A partir desta fase, o tempo de execução tem tendência a convergir para um valor de aproximadamente 6,20 segundos. No texto de tamanho mais reduzido podemos verificar que a curva é idêntica, apenas os valores do tempo alteram.

Texto: portuguese-medium.txt

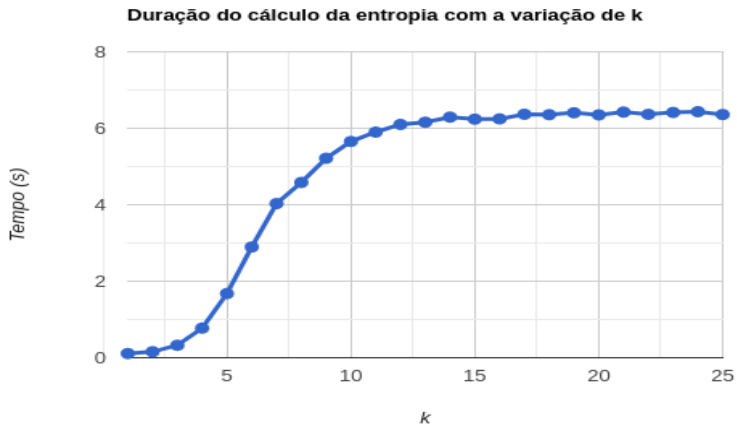


Figura 2: Duração do cálculo da entropia com a variação de k

Texto: portuguese-small.utf8

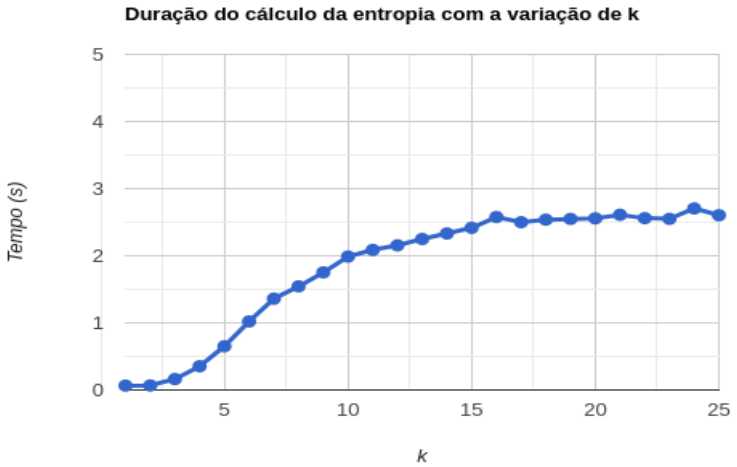


Figura 3: Duração do cálculo da entropia com a variação de k

Por último, procurámos avaliar o gasto de memória que o modelo de contexto finito introduziu no nosso programa, bem como a poupança que obtivemos ao criar o modelo de contexto finito apenas com os contextos existentes no texto a analisar, invés de todos os contextos possíveis. Posto isto, contando com um valor de bit counters = 16, seguimos a seguinte fórmula:

$$Memória\ (MB) = número\ Contextos * len(alfabeto) * 16 / 8 / 1024 / 1024$$

Considerando o cenário 1 ser o programa guardar todos os contextos possíveis no modelo de contexto finito, e o cenário 2 ser o programa guardar apenas os contextos que efetivamente ocorrem pelo menos uma vez no texto (que é o que está implementado), e considerando o valor de k = 2, k = 3 e k = 4:

Texto: portuguese-big.utf8

Tabela 4: Memória ocupada pelo modelo de contexto finito

	Cenário 1	Cenário 2

k = 2	Memória = 4.387 MB	Memória = 0.639 MB
k = 3	Memória = 579.063 MB	Memória = 3.743 MB
k = 4	Memória = 76436.314 MB	Memória = 12.636 MB

Como podemos observar pelos resultados demonstrados na tabela acima, os ganhos de memória são enormes e crescem de forma exponencial à medida que o valor de k aumenta também. Mesmo para valores baixos de k já existe uma grande diferença.

De realçar que mesmo o cenário 2 é um limite máximo da memória que o nosso modelo pode gastar, visto que para cada contexto nós guardamos apenas o número de ocorrências para os símbolos que efetivamente ocorrem após aquele contexto ao invés de guardar todos os símbolos para cada contexto existente, que é aquilo que os cálculos consideram. Posto isto, o valor de memória gasto pelo nosso modelo é, geralmente, inferior ao do cenário 2.

4.2. lang.cpp

Relativamente ao programa lang.cpp fizemos algumas experiências relacionadas com a variação dos parâmetros k e α, fixando o ficheiro de referência e o ficheiro alvo utilizados.

Começando pela análise da influência dos parâmetros k e α, na tabela abaixo podemos verificar os resultados obtidos relativos ao número de bits necessários para comprimir um texto ao variar o α com valores entre 0.1 e 0.6 e ao variar o parâmetro k entre 1 e 5. 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).

Tabela 5: Variação do número de bits necessários para comprimir um texto em função de k e a

	k = 1	k = 2	k = 3	k = 4	k = 5
α = 0.1	13,058,672	10,278,176	7,958,871	6,626,456	6,195,773
α = 0.2	13,059,670	10,293,472	8,031,984	6,840,927	6,643,598
α = 0.3	13,060,599	10,306,596	8,093,624	7,015,520	6,991,780
α = 0.4	13,061,487	10,318,431	8,148,666	7,167,328	7,284,589
α = 0.5	13,062,345	10,329,377	8,199,196	7,303,641	7,540,611
α = 0.6	13,063,181	10,339,657	8,246,354	7,428,439	7,769,841

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 (3.0MB).

Tabela 6: Variação do número de bits necessários para comprimir um texto em função de k e do α e com

Reference		k = 1	k = 2	k = 3	k = 4	k = 5
FRA	α = 0.1	13,058,672	10,278,176	7,958,871	6,626,456	6,195,773
GER	α = 0.1	19,367,923	19,640,675	22,279,828	23,644,113	24,858,472
RUS	α = 0.1	23,270,642	25,328,715	30,311,696	31,836,611	31,953,542
FRA	α = 0.3	13,060,599	10,306,596	8,093,624	7,015,520	6,991,780
GER	α = 0.3	19,223,601	19,402,768	21,818,210	23,404,117	24,891,392
RUS	α = 0.3	24,136,746	26,946,954	30,896,631	31,932,070	31,541,366
FRA	α = 0.6	13,063,181	10,339,657	8,246,354	7,428,439	7,769,841
GER	α = 0.6	19,150,396	19,303,204	21,675,928	23,453,514	25,028,431
RUS	α = 0.6	25,082,476	28,166,716	31,253,906	31,988,803	32,014,585

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.cpp 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 α é sempre 0.1 e o valor de k varia entre 1 e 6.

Tabela 7: Diferença entre o número de bits necessários para comprimir um texto ao usar múltiplos modelos

	Bits	Bits (multiplemodels)	Diferença
k = 1	1241813	1230833	-10,980

k = 2	1212625	1230833	+18,208
k = 3	1333779	1230833	-102,946
k = 4	1542219	1230833	-311,386
k = 5	1759613	1230833	-528,780

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.

4.3. findlang.cpp

Para colocarmos à prova o programa findlang.cpp, 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 19 textos, cada um deles pertence a uma língua diferente. Os parâmetros k e α foram fixados com o valor de 3 e 0.1, respetivamente.

Tabela 8: Previsões efetuadas pelo programa findlang.cpp

Ficheiro Alvo	Língua	Língua prevista	Língua prevista (multiplemodels)
afghanistan-medium.utf8 (118kB)	Afegão	Afegão	Afegão
afrikaans-big.utf8 (3737kB)	Africanês	Africanês	Africanês
arabic-big.utf8 (7009kB)	Árabe	Árabe	Árabe
bulgarian-medium.utf8 (6770kB)	Búlgaro	Búlgaro	Búlgaro
croatian-medium.utf8 (966kB)	Croata	Croata	Croata
danish-medium.utf8 (3966kB)	Dinamarquês	Dinamarquês	Dinamarquês
english-big.utf8 (8364kB)	Inglês	Inglês	Inglês
spanish-medium.utf8 (319kB)	Espanhol	Espanhol	Espanhol
finnish-medium.utf8 (3701kB)	Finlandês	Finlandês	Finlandês

french-medium.utf8 (3977kB)	Francês	Francês	Francês
german-medium.utf8 (3983kB)	Alemão	Alemão	Alemão
greece-medium.utf8 (1860kB)	Grego	Grego	Grego
hungarian-medium.utf8 (3748kB)	Húngaro	Húngaro	Húngaro
icelandic-medium.utf8 (3740kB)	Islandês	Islandês	Islandês
italian-medium.utf8 (3961kB)	Italiano	Italiano	Italiano
polish-medium.utf8 (3972kB)	Polaco	Polaco	Polaco
portuguese-medium.txt (325kB)	Português	Português	Português
russian-medium.utf8 (2976kB)	Russo	Russo	Russo
ukrainian-medium.utf8 (3325kB)	Ucraniano	Ucraniano	Ucraniano

Como podemos ver segundo os resultados obtidos na tabela acima, podemos concluir, que o programa findlang.cpp para os 19 textos testados, consegue prever de maneira correta a língua presente em cada um deles. A accuracy do programa pode ser calculada da seguinte forma:

$$Acc = \frac{número\ de\ previsões\ corretas}{número\ total\ de\ previsões} * 100 = \frac{19}{19} * 100 = 100\%$$

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.

4.4. locatelang.cpp

Por último, realizamos um conjunto de testes relativos ao programa locatelang.cpp 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.cpp 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 detecção 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 α igual a 0.5 e um valor de k igual a 4.

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

solucao:

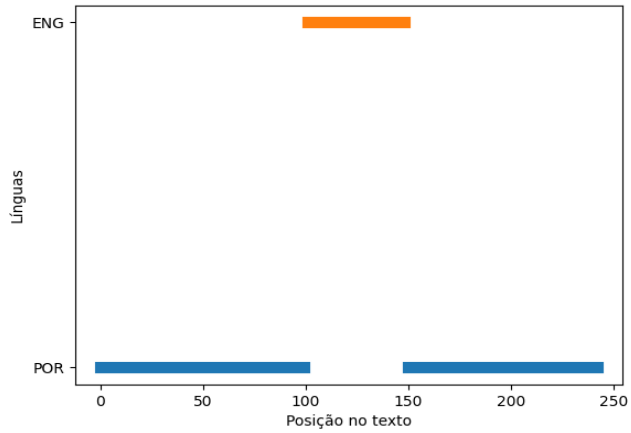


Figura 4: Solução ótima de classificação

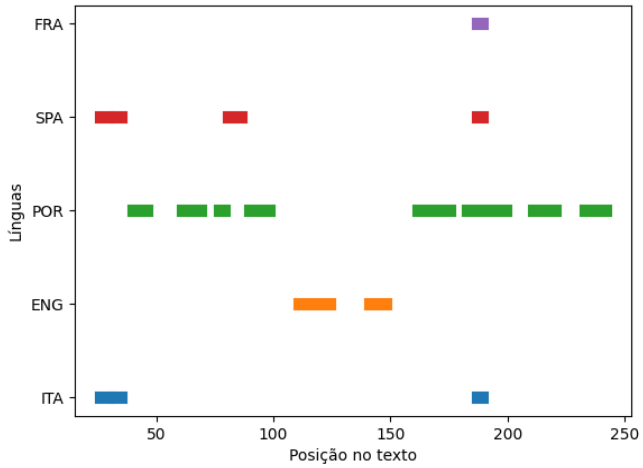


Figura 5: Resultado obtido com a implementação 1

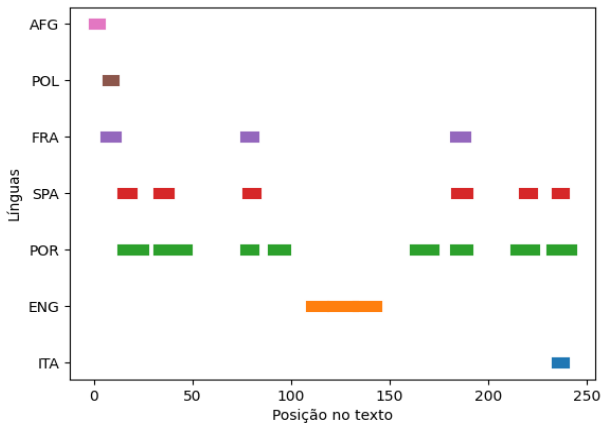


Figura 6: Resultado obtido pela implementação 2

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

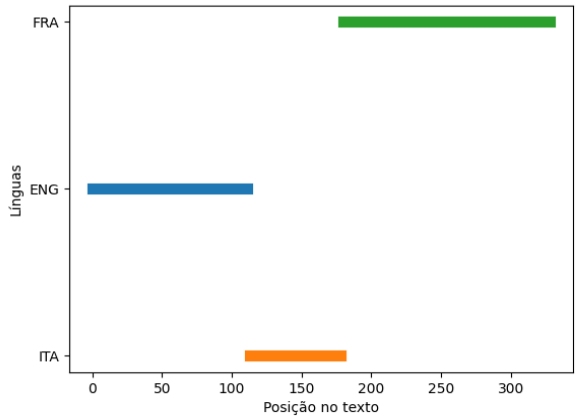


Figura 7: Solução ótima de classificação

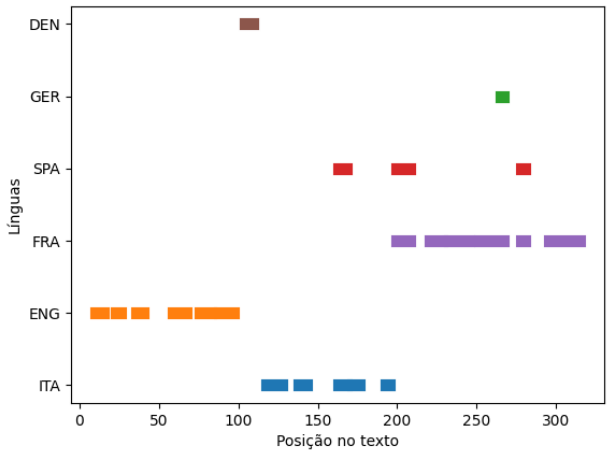


Figura 8: Resultado obtido pela implementação 1

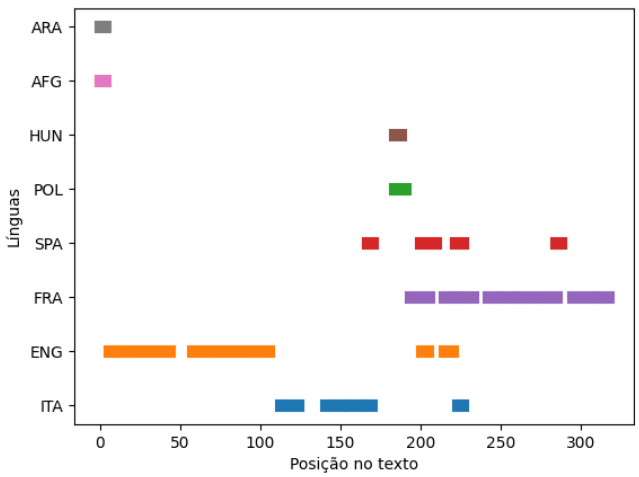


Figura 9: Resultado obtido pela implementação 2

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 detecção

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 detecta 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 efectuámos está relacionada com a influência do valor do parâmetro k nas secções que são detectadas pelo programa. Posto isto, para o ficheiro alvo mullang1-PT-ENG-PT.txt, fixámos o valor de α 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):

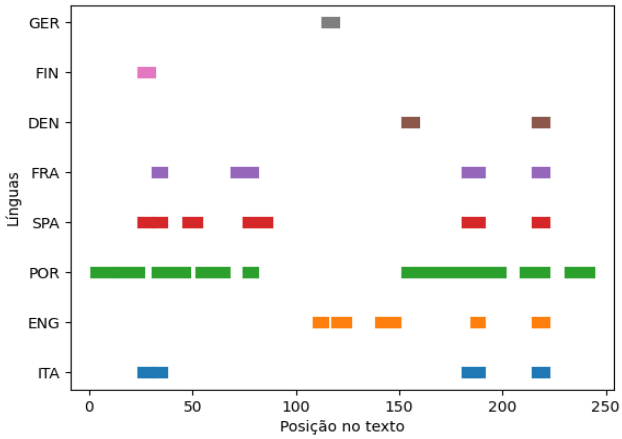


Figura 10: Resultado obtido para a primeira implementação com k = 3

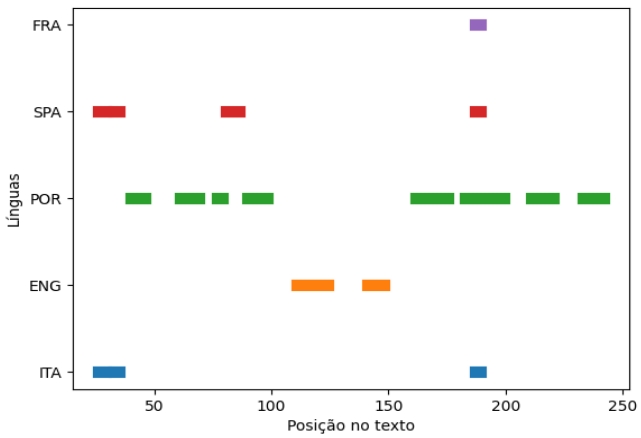


Figura 11: Resultado obtido com a primeira implementação para k = 4

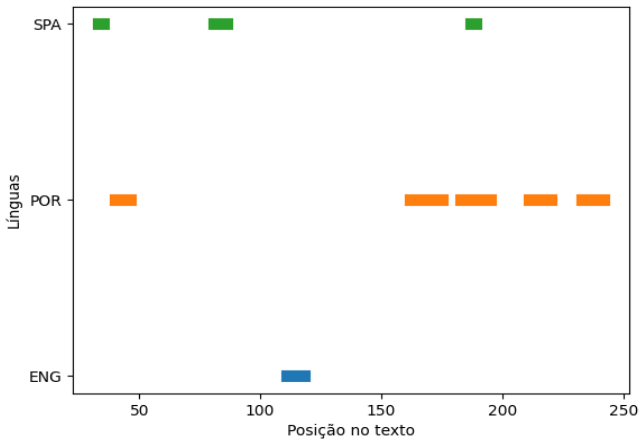


Figura 12: Resultado obtido com a primeira implementação com k = 5

- Resultados relativos à segunda implementação (Janela Deslizante):

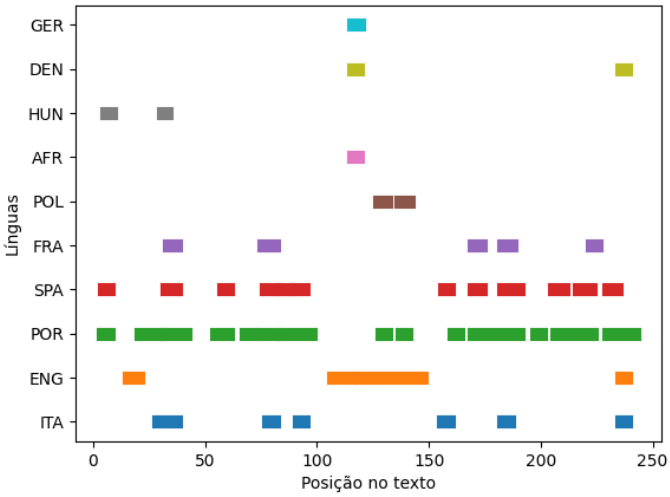


Figura 13: Resultado obtido com segunda implementação para k = 3

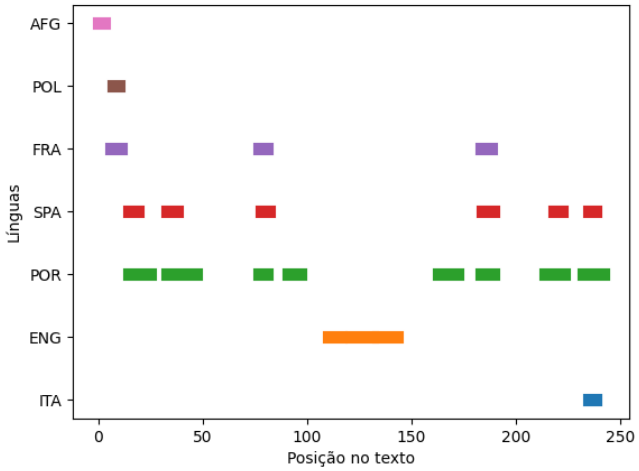


Figura 14: Resultado obtido com segunda implementação para k = 4

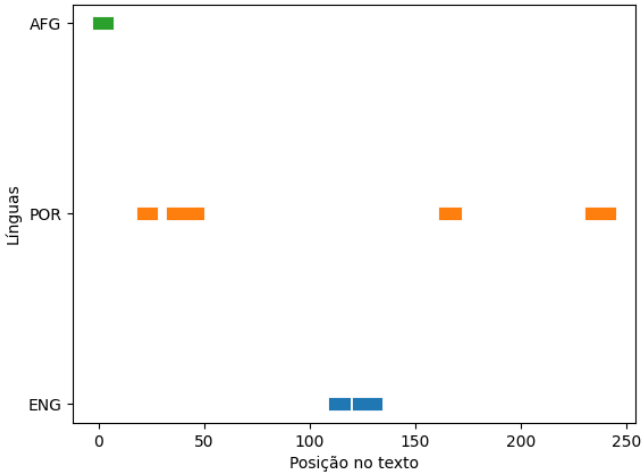


Figura 15: Resultado obtido com segunda implementação para k = 5

A partir dos gráficos apresentados acima, podemos afirmar que ambas as implementações do programa locatelang.cpp 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.cpp* 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 α igual a 0.5.

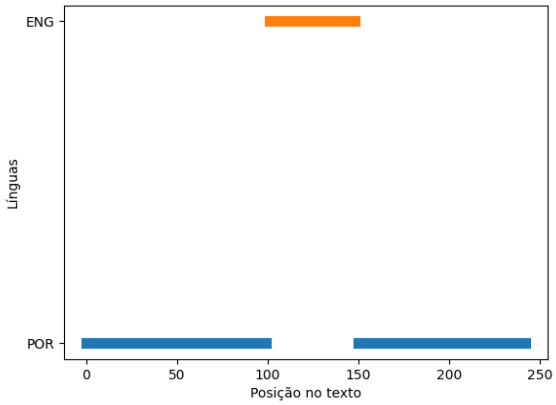


Figura 16: Solução ótima de classificação

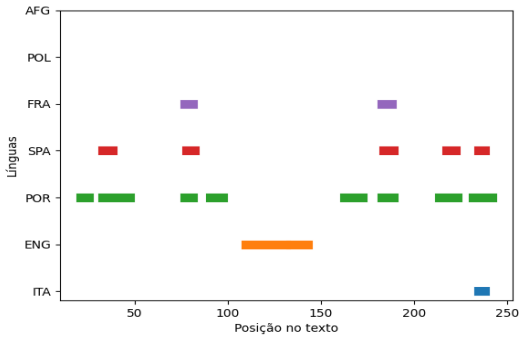


Figura 17: Secções classificadas pelo primeiro threshold

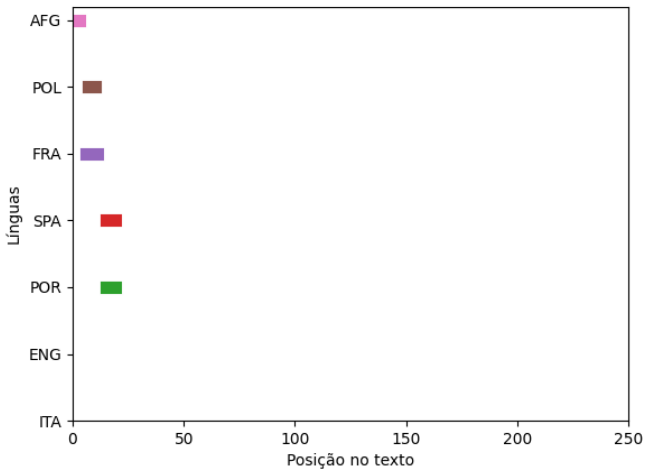


Figura 18: Secções classificadas pelo segundo threshold

Como podemos ver na análise dos gráficos, o segundo threshold torna possível a classificação de algumas secções extra, pelo que é uma mais-valia para o programa.

5. Conclusão

Com a realização deste projeto conseguimos aprofundar e pôr em prática os nossos conhecimentos relativos à implementação e construção de um Finite Context Model, bem como efetuar a utilização do mesmo para calcular probabilidades de ocorrência dos diferentes símbolos do alfabeto.

Para além disso, fomos capazes de interligar este modelo com uma aplicação útil para o mesmo, neste caso, deteção de 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 os resultados obtidos vão de acordo com os esperados, e que, conseguimos de uma forma geral desempenhar a tarefa que nos foi proposta com êxito.

6. Links

Github: https://github.com/rafaelbaptista13/IC/tree/main/Assignment3

Video Promocional:
https://www.youtube.com/watch?v=VUOnhJJPmgk&ab_channel=RafaelBaptista