UNIVERSIDADE DE AVEIRO

DEPARTAMENTO DE ELECTRÓNICA, TELECOMUNICAÇÕES E INFORMÁTICA

INFORMAÇÃO E CODIFICAÇÃO

---

# PROJETO 3.A

---

*Autores:*
António Ferreira (89082)
Guilherme Claro (98432)
Luís Couto (89078)

https://github.com/Toja-Ferreira/IC_G07_Proj3

2022

# Conteúdo

# Lista de Figuras

# Introdução

O presente relatório visa descrever a resolução do Projeto 3.A desenvolvido no âmbito da unidade curricular de Informação e Codificação.
Para cada problema é apresentada uma explicação teórica que fundamenta a solução implementada, seguida de uma explicação prática e finalmente apresentação de resultados.

No repositório do projeto, para além de todo o material desenvolvido, existe um ficheiro README.md, com instruções de como compilar e testar todos os programas.
O repositório contém também o vídeo pedido, que demonstra o funcionamento do trabalho desenvolvido.

# Parte I

## Modelos de Contexto Finito

Este projeto tem como foco principal o conceito de Modelos de Contexto Finito, também conhecidos como Cadeias de Markov em Tempo Discreto. Estes são descritos como uma classe de modelos estatísticos, usados para prever uma palavra ou símbolo, dado um conjunto finito de palavras ou símbolos anteriores.

Uma cadeia de Markov de ordem $k$ verifica que:

$$P(x_n|x_{n-1}...x_{n-k}) = P(x_n|x_{n-1}...x_{n-k}...)$$

onde o contexto do processo é dado por

$$\text{String c} = x_{n-1}...x_{n-k}$$

e o número de caracteres do contexto dado por

$$\text{ordem } k$$

Para uma cadeia com N contextos, a entropia H é dada pelo valor médio da entropia de cada contexto $S_i$:

$$H = \sum_{i=1}^{N} P(S_i) \times H(S_i)$$

Onde a probabilidade $P(S_i)$ de cada contexto $S_i$ é dada pela frequência relativa do número de ocorrências de caracteres nesse contexto em relação ao número total de ocorrências em todos os contextos:

$$P(S_i) = \frac{Total_{S_i}}{\sum_{k=1}^{N} Total_{S_k}}$$

E a entropia $H(S_i)$ de cada contexto $S_i$ é dada pelo somatório das probabilidades de frequência relativa $P_i$ para cada ocorrência de caracter $i$ desse contexto:

$$H(S_i) = \sum_{i=1}^{N} -P_i \times \log_2(P_i)$$

Adicionalmente, de modo a evitar o problema de "probabilidade-zero", ao estimar a probabilidade $P_i$ de um caracter $i$ aparecer após um dado contexto, é utilizada a fórmula:

$$P_i = \frac{n_i + \alpha}{n + \alpha|\Sigma|}$$

onde

$n_i$ representa o numero de ocorrências do caracter no contexto

$n$ representa o número total de ocorrências de qualquer caracter para o contexto

$\alpha$ representa o smoothing parameter

$\Sigma$ representa o tamanho do alfabeto

## Exercício 1 - fcm

Nesta exercício, foi desenvolvido um programa chamado **fcm**, com base na teoria previamente explicada, com o objetivo de reunir informação estatística sobre textos usando modelos de contexto finito.

Para compilar todos os ficheiros da Parte I:

```
# Estando no diretório base do projeto
> cd Part_I/
> cmake .
> make
```

Este programa consiste em dois ficheiros principais:

- **fcm.hpp** - contém a declaração das principais funções usadas no projeto, assim como uma descrição dos seus parâmetros, funcionamento e retorno. Contém ainda funções auxiliares (i.e ler caracteres, verificar inputs, etc), também usadas no resto do projeto.
- **fcm.cpp** - contém a implementação das funções declaradas

Adicionalmente, existe uma pasta chamada test_programs/, que contém:

- **fcmTest.cpp** - ficheiro que contém a implementação da funçao de teste
- **models/** - pasta onde modelos criados durante a execução do programa são armazenados
- **test_files/** - pasta que contém ficheiros de texto que servem para testar o programa. Neste caso, são todos ficheiros que representam cada um uma linguagem

### Construtor da Classe fcm

A classe principal é criada através da chamada ao seu construtor:

```
fcm(int k, double alfa);
```

O primeiro argumento representa a ordem $k$ do modelo, existindo a verificação se este é um número inteiro positivo maior que 0.
O segundo argumento representa o smoothing parameter $\alpha$, existindo a verificação se este é um número positivo maior que 0.

### Carregar Modelos

A função base deste programa, que é a leitura de um ficheiros, geração do correspondente modelo e cálculo da entropia, é dada pela função:

```
void loadModel(char *fName, char toSave, map<string,map<char,int>> *emptyModel=0);
```

De uma maneira resumida, a implementação pode ser descrita nos seguintes passos:

1. Abrir ficheiro de texto e verificar se este é válido (existe e não está vazio)
2. Criar estrutura de dados vazia que representa o modelo, dada como:

   ```
   map<string, map<char, int>> model;
   ```

   Onde a string representa o contexto de ordem $k$ e o mapa associado representa um caracter e o número de vezes que ele ocorre para esse contexto
3. Ler ficheiro de texto, caracter a caracter, e preencher o modelo de acordo com a ordem $k$ escolhida (ocorrências de newlines, tabs e EOF são excluídas)
4. Caso o argumento emptyModel tenha o valor default (0), a função está a ser chamada pelo programa fcm e portanto os passos seguintes são:
   (a) Escrever modelo num ficheiro de texto e salvar em memória, na pasta models, caso pedido (argumento toSave = 'Y')
   (b) Estimar a entropia do ficheiro através do modelo gerado e imprimir mensagem com o seu valor na consola
5. Caso contrário, está a ser chamada por outros programas (Lang, FindLang, LocateLang) e portanto o último passo é atribuir ao ponteiro emptyModel o modelo criado, de maneira a que estes programas tenham acesso ao modelo

### Escrita de Modelos em Memória

Para evitar a análise repetida de ficheiros de texto, é dada a possibilidade de guardar modelos gerados em memória, para posterior leitura. Esta funcionalidade é dada pela função:

```
void saveModel(char *filename, map<string, map<char, int>> model);
```

De uma maneira resumida, a implementação pode ser descrita nos seguintes passos:

1. Criar ficheiro de texto onde vai ser escrito o modelo. O nome destes ficheiros segue o seguinte padrão:

   ```
   ./models/modelOrder + k + "_" + nome de ficheiro original + ".txt"
   ```
2. Escrever conteúdo do modelo no ficheiro de texto criado
   - A primeira linha deste ficheiro é sempre escrita no formato:

     ```
     <k>,<alfa>
     ```
   - As restantes linhas são sempre escritas no formato:

     ```
     <contexto> '\t' <nextChar> '\t' <count> ....
     ```
3. Imprimir mensagem de informação na consola

Importante notar que nos restantes programas desenvolvidos neste projeto é possível usar modelos já criados anteriormente, contudo os ficheiros onde estes estão escritos têm de seguir o formato agora descrito.

### Leitura de Modelos em Memória

A leitura de modelos previamente criados é portanto também possível. Esta funcionalidade é dada pela função:

```
void readModel(char *filename, map<string, map<char, int>> &model,
               int &orderM, double &alphaM);
```

De uma maneira resumida, a implementação pode ser descrita nos seguintes passos:

1. Abrir ficheiro de texto a ler e verificar se este é válido (existe e não está vazio)
2. Ler ficheiro de texto e salvar a informação nos argumentos que foram passados por referência:
   (a) A primeira linha é lida e os correspondentes valores salvos em orderM e alphaM
   (b) As restantes linhas são lidas e os correspondes valores guardados em model

A leitura é realizada tendo como base a formatação descrita na secção anterior. Caso o ficheiro não esteja escrito com esse formato, é atirada uma exceção.

### Calcular Entropia de Modelo

O valor de entropia de um modelo é dado pela função

```
calculateModelEntropy(map<string, map<char, int>> model);
```

Os cálculos aqui realizados seguem as fórmulas descritas no inicio deste capítulo, que foram implementadas.

### Testes

Para testar o programa desenvolvido (no terminal):

```
# Estando no diretório IC_G07_Proj3/Part_I/test_programs
> ./fcmTest test_files/pt_PT.Portugese-latn-EP7-train.utf8 5 0.1

# Seguidamente aparecerá o seguinte menu:
----------------------------------TEST FCM----------------------------------
Order k = 5
Smoothing parameter alpha = 0.1

Please choose an option:
[1]: analyse file and save model in memory
[2]: analyse file without saving model
[q]: quit

# Selecionando a opção [1]:
>>> 1

# Resultados Obtidos:
--------------------------------------------------------------------------------
Analysing file pt_PT.Portugese-latn-EP7-train.utf8 and generating model

Model has been saved in: ./models/modelOrder5_pt_PT.Portugese-latn-EP7-train.utf8.txt

Estimated entropy of pt_PT.Portugese-latn-EP7-train.utf8 is 1.33781
--------------------------------------------------------------------------------
```

O utilizador tem então a opção de apenas analisar o ficheiro ou então analisar e também salvar o modelo em memória, para uso futuro. Na seguinte imagem é possível ver o ficheiro de texto com o modelo gerado a partir do teste apresentado.


Figura 1: Exemplo de Modelo Guardado em Memória

### Resultados

Para efeitos de análise, foram realizados testes no mesmo ficheiro, de maneira a observar o impacto da ordem do modelo nos modelos gerados.
Para tal, foram usados os seguintes parâmetros no programa:

```
Ficheiro a analisar      > pt_PT.Portugese-latn-EP7-train.utf8
Order de modelo k        > 1 até 12
Smoothing parameter alfa > 0.1
```

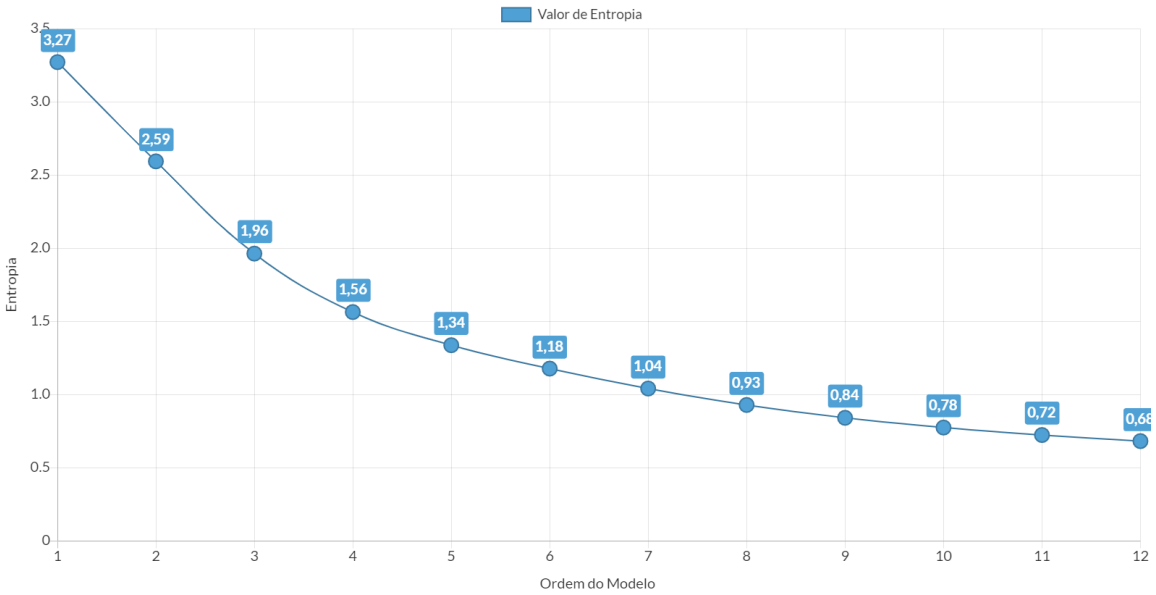


Figura 2: Evolução da entropia do modelo de acordo com a ordem

Ao observar o gráfico anterior, é possível verificar que quanto maior é o valor da ordem do modelo, menor será a entropia deste.

Considerando que a ordem do modelo é o que define o tamanho do contexto, é fácil de perceber esta evolução, pois ao aumentar o tamanho do contexto, ou seja, da palavra, diminui o número de diferentes caracteres possíveis que podem aparecer após esta.
Isto parte da razão de que em qualquer linguagem, quanto mais caracteres são adicionados a uma palavra, maior é a probabilidade desta ser uma palavra existente e com significado.
Um exemplo fácil de perceber, em português, seria:

- **ordem 6 ⇒ gelado** - neste caso, o número de caracteres que pode aparecer é mínimo, na maioria das vezes 's' ou um espaço
- **ordem 3 ⇒ pat** - neste caso, o número de caracteres que pode aparecer é mais elevado, pois o número de palavras que podem ser formadas a partir desta é maior. Exemplos de caracteres possíveis seriam 'a', 'e', 'i', 'o', 'r', etc..

É concluído então que o aumento da ordem de um modelo reduz a entropia, pois diminui o número possível de palavras que se podem formar a partir do contexto já existente, o que por sua vez reduz as probabilidades de ocorrência de um novo caracter e consequentemente diminui a entropia.

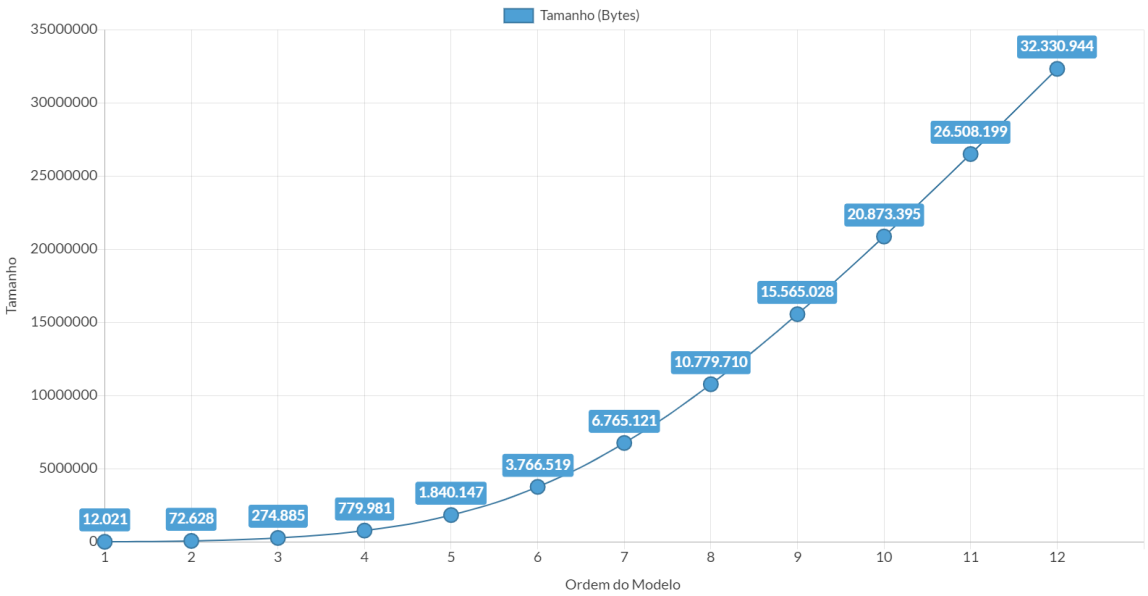


Figura 3: Evolução do tamanho do modelo de acordo com a ordem

Ao observar o gráfico anterior, é possível verificar que quanto maior é o valor da ordem do modelo, maior será o tamanho dos modelos.

Considerando que a ordem do modelo é o que define o tamanho do contexto, é fácil de perceber esta evolução, pois ao aumentar o tamanho do contexto, também aumenta o número de contextos únicos que podem existir.
Um exemplo fácil de perceber, em português, seria:

- **ordem 1** - neste caso, o número de contextos únicos que podem existir dependem do número de caracteres únicos da linguagem, pois cada contexto tem apenas 1 elemento. No caso de português, podem existir cerca de 127
- **ordem 9** - neste caso, o número de contextos únicos que podem existir é bastante mais elevado, pois cada contexto pode ter 9 elementos, o que permite maior combinação

# Parte II

Nesta parte, foram desenvolvidos três programas distintos, tendo todos como base o programa fcm desenvolvido na parte anterior:

1. **lang** recebe um ficheiro de texto e um ficheiro que representa uma linguagem/modelo e devolve o número estimado de bits requeridos para comprimir o texto
2. **findlang** é baseado no *lang*, recebe um ficheiro de texto e um conjunto de exemplos de várias linguagens/modelos e devolve um guess de que linguagem é que o ficheiro de texto foi escrito
3. **locatelang** recebe um texto com segmentos em diferentes linguagens e deteta que linguagens contém, assim como em que caracter cada uma começa

Para compilar todos os ficheiros da Parte II:

```
# Estando no diretório base do projeto
> cd Part_II/
> cmake .
> make
```

## Exercício 2 - lang

### Construtor da Classe lang

A classe principal é criada através da chamada ao seu construtor:

```
lang(int k, double alpha);
```

O primeiro argumento representa a ordem $k$ do modelo, existindo a verificação se este é um número inteiro positivo maior que 0.
O segundo argumento representa o smoothing parameter $\alpha$, existindo a verificação se este é um número positivo maior que 0.
Ao chamar este construtor, é também criado dentro da classe lang um objeto da classe fcm, que vai ser usado para analisar os ficheiros:

```
this->fcmObj = fcm(k, alpha);
```

### Estimar Bits Necessários para Comprimir um Texto

Toda a funcionalidade desta classe é dada pela função:

```
double estimateBits(char *filename, char *classFilename, char typeOfComp);
```

De uma maneira resumida, a implementação pode ser descrita nos seguintes passos:

1. Criar estrutura de dados vazia que representa o modelo, dada como:

```
map<string, map<char, int>> model;
```

2. Caso o parâmetro typeOfComp seja igual a 'F', é uma comparação entre texto e ficheiro que representa uma linguagem, pelo que são seguidos os seguintes passos:
   (a) Chamar a função loadModel, através do objeto fcm criado, que vai carregar a estrutura criada no passo 1 com os dados do novo modelo gerado
   (b) Abrir ficheiro de texto a analisar e verificar se este é válido (existe e não está vazio)
   (c) Ler o ficheiro de texto e comparar com o modelo gerado, estimando assim o número de bits necessários para comprimir o texto usando o modelo gerado
   (d) Devolver o número de bits
3. Caso contrário, é uma comparação entre texto e ficheiro que representa um modelo existente, pelo que são seguidos os seguintes passos:
   (a) Chamar a função readModel, através do objeto fcm criado, que vai carregar a estrutura criada no passo 1 com os dados do modelo que está guardado no ficheiro de modelo
   (b) Abrir ficheiro de texto a analisar e verificar se este é válido (existe e não está vazio)
   (c) Ler o ficheiro de texto e comparar com o modelo gerado, estimando assim o número de bits necessários para comprimir o texto usando o modelo lido
   (d) Devolver o número de bits

### Testes

Para testar o programa desenvolvido, comparando com um ficheiro que representa uma linguagem (no terminal):

```
# Estando no diretório IC_G07_Proj3/Part_II/test_programs
> ./langTest test_files/brothers_karamazov.txt
                language_files/pt_PT.Portugese-latn-EP7-train.utf8

# Seguidamente aparecerá o seguinte menu:
Which type of file are you using to represent the language?
[1] text file
[2] model file
[q] quit

# Selecionando a opção [1] (opção tem de ser a correta):
>>> 1

# Terá de inserir os valores desejados para gerar o modelo:
Value of model order: 5

Value of smoothing parameter: 0.1
```

```
# Resultados Obtidos:
--------------------------------------------------------------------------------
Analysing file pt_PT.Portugese-latn-EP7-train.utf8 and generating model

Comparing the text with the obtained language model

Estimated number of bits to compress text: 1.55695e+07
Estimated number of bits per character: 4.75505
--------------------------------------------------------------------------------
```

Para testar o programa desenvolvido, comparando com um ficheiro que representa um modelo (no terminal):

```
# Estando no diretório IC_G07_Proj3/Part_II/test_programs
> ./langTest test_files/brothers_karamazov.txt
                models/modelOrder5_pt_PT.Portugese-latn-EP7-train.utf8.txt

# Seguidamente aparecerá o seguinte menu:
Which type of file are you using to represent the language?
[1] text file
[2] model file
[q] quit

# Selecionando a opção [2] (opção tem de ser a correta):
>>> 2

# Resultados Obtidos:
--------------------------------------------------------------------------------
Comparing the text with the obtained language model

Estimated number of bits to compress text: 1.55695e+07
Estimated number of bits per character: 4.75505
--------------------------------------------------------------------------------
```

É então possível analisar com ficheiros que representam linguagens ou com ficheiros que representam modelos de linguagens.
Nos dois exemplos agora apresentados, o ficheiro com modelo no 1º exemplo contém o modelo gerado a partir do mesmo ficheiro de linguagem com os parâmetros iguais ao usado no 2º exemplo.
Como tal, os valores obtidos foram iguais, pois no final o modelo usado foi o mesmo.

## Exercício 3 - findlang

### Construtor da Classe findlang

A classe principal é criada através da chamada ao seu construtor:

```
findlang(int k, double alpha);
```

O primeiro argumento representa a ordem $k$ do modelo, existindo a verificação se este é um número inteiro positivo maior que 0.
O segundo argumento representa o smoothing parameter $\alpha$, existindo a verificação se este é um número positivo maior que 0.
Ao chamar este construtor, é também criado dentro da classe findlang um objeto da classe lang, que vai ser usado para analisar os ficheiros:

```
this->langObj = lang(k, alpha);
```

### Estimar Linguagem que Representa Texto

Toda a funcionalidade desta classe é dada pela função:

```
string estimateLanguage(char *filename, vector<string> languages, char typeOfComp);
```

De uma maneira resumida, a implementação pode ser descrita nos seguintes passos:

1. Para cada ficheiro passado no conjunto de ficheiros (mínimo 1 ficheiro):
   (a) Chamar a função estimateBits, através do objeto lang criado, que vai estimar o número de bits necessários para comprimir o texto usando a linguagem/modelo representada no ficheiro
   (b) Guardar numa variável o número mínimo de bits e noutra o nome do ficheiro que contém a melhor linguagem/modelo encontrado até ao momento
2. Devolver nome do ficheiro que contém a linguagem/modelo que melhor representa o texto

À medida que esta função executa, são imprimidos na consola todos os valores calculados, caso o utilizador queira fazer uma análise mais detalhada.

### Testes

Para testar o programa desenvolvido, comparando com um conjunto de ficheiros onde cada um representa uma linguagem (no terminal):

```
# Estando no diretório IC_G07_Proj3/Part_II/test_programs
> ./findlangTest test_files/brothers_karamazov.txt
                    language_files/alb_AL.Albanian latn-wiki-test.utf8
                    language_files/ar_XX.Arabic-wiki-train.utf8
                    language_files/de_DE.German-latn-EP7-train.utf8
                    language_files/en_GB.English-latn-EP7-train.utf8
                    language_files/es_ES.Spanish-latn-EP7-train.utf8
                    language_files/fr_FR.French-latn-EP7-train.utf8
                    language_files/it_IT.Italian-latn-EP7-train.utf8
                    language_files/ja_JP.Japanese-wiki-train.utf8
                    language_files/pt_PT.Portugese-latn-EP7-train.utf8
                    language_files/ru_RU.Russian-wiki-train.utf8
                    language_files/sv_SE.Swedish-latn-EP7-train.utf8
                    language_files/tr_TR.Turkish-latn-wiki-train.utf8

# Seguidamente aparecerá o seguinte menu:
Which type of files are you using to represent the languages?
[1] text file
[2] model file
[q] quit

# Selecionando a opção [1] (opção tem de ser a correta):
>>> 1

# Terá de inserir os valores desejados para gerar o modelo:
Value of model order: 5

Value of smoothing parameter: 0.1

# Resultados Obtidos (Apenas indicado aqui o resultado final,
# contudo na consola aparecem todos os resultados intermédios):
*********************************************************************************

 Closest language to text is: ru_RU.Russian-wiki-train.utf8

*********************************************************************************
```

Para testar o programa desenvolvido, comparando com um conjunto de ficheiros onde cada um representa um modelo (no terminal):

```
# Estando no diretório IC_G07_Proj3/Part_II/test_programs
> ./findlangTest test_files/brothers_karamazov.txt
                    models/modelOrder5_alb_AL.Albanian-latn-wiki-test.utf8.txt
                    models/modelOrder5_ar_XX.Arabic-wiki-train.utf8.txt
                    models/modelOrder5_de_DE.German-latn-EP7-train.utf8.txt
                    models/modelOrder5_en_GB.English-latn-EP7-train.utf8.txt
                    models/modelOrder5_es_ES.Spanish-latn-EP7-train.utf8.txt
                    models/modelOrder5_fr_FR.French-latn-EP7-train.utf8.txt
                    models/modelOrder5_it_IT.Italian-latn-EP7-train.utf8.txt
                    models/modelOrder5_ja_JP.Japanese-wiki-train.utf8.txt
                    models/modelOrder5_pt_PT.Portugese-latn-EP7-train.utf8.txt
                    models/modelOrder5_ru_RU.Russian-wiki-train.utf8.txt
                    models/modelOrder5_sv_SE.Swedish-latn-EP7-train.utf8.txt
                    models/modelOrder5_tr_TR.Turkish-latn-wiki-train.utf8.txt

# Seguidamente aparecerá o seguinte menu:
Which type of files are you using to represent the languages?
[1] text file
[2] model file
[q] quit

# Selecionando a opção [2] (opção tem de ser a correta):
>>> 2

# Resultados Obtidos (Apenas indicado aqui o resultado final,
# contudo na consola aparecem todos os resultados intermédios):
********************************************************************************

 Closest language to text is: modelOrder5_ru_RU.Russian-wiki-train.utf8.txt

********************************************************************************
```

É então possível comparar um texto com um conjunto de ficheiros que representam linguagens ou que representam modelos de linguagens e estimar qual o ficheiro que melhor representa o texto.
Nos exemplos anteriores foi analisado o mesmo ficheiro de texto com 12 ficheiros que representam linguagens/modelos diferentes, com os seguintes parâmetros:

```
Ficheiro a analisar      > brothers_karamazov.txt (escrito em russo)
Order de modelo k        > 5
Smoothing parameter alfa > 0.1
```

Para ambos os casos, o resultado obtido foi o correto (ficheiros que representam russo), pelo que se conclui que o programa está implementado corretamente.

## Exercicio 4 - locatelang

No exercício **locatelang** conforme se vai percorrendo o ficheiro de texto, cada modelo vai produzir um certo número de bits igual a *log2(p)* em que é probabilidade atribuída pelo modelo ao símbolo que está a ser processado.
Portanto nas zonas que contem texto de uma certa linguagem, o número de bits produzido pelo modelo dessa linguagem deve ser, em media, inferior ao produzido pelos os outros modelos. Ou seja, a diferença é que, em vez de simplesmente se acumular o numero total de bits e comparar no final, vai ser necessário monitorizar em tempo real esse valor ao longo do texto, para se descobrir as linguagems que linguagems estão a ser escritas por partes.
Em relação á combinação de modelos, a ideia era combinar as probabilidades de vários modelos em simultâneo com uso das medias

Infelizmente o grupo não consegui implementar este exercício.

# Contribuição dos autores

A percentagem de cada membro no desenvolvimento do projeto fica:

- António Ferreira - 33.33%
- Guilherme Claro - 28.33%
- Luís Couto - 38.34%