Project 3
Finite Context Models

Diogo Mendes[88801], Lúcia Sousa[93086], Rodrigo Martins[93264]

Universidade de Aveiro
DETI - Departamento de Electrónica, Telecomunicações e Informática
Information and Coding

Abstract. This paper consists on a report of Project#3. Every exercise is briefly explained with the key points being explained. Each exercise also as a subsection with some results obtained.

Keywords: Similarity · Language · Context.

1 User guide

All source code can be found in the github: https://github.com/rodrigo740/IC-Project-3

All programs explained below include the commands that must be ran to execute them individually. However, in the main directory the user can find the makefile to compile all files.

2 Part A

2.1 Class Filter

This class is used to filter texts into a more usable state for the Fcm class. This is achieved through converting every character to lowercase and removing all numbers from the text.
To test this class a test program was developed named *testFilter.cpp*, this program receives 2 arguments, the first one being the input file and the second one the output file, to compile and run the test the following commands must be executed:

```
$ g++ -g -Wall testFilter.cpp -o testFilter.o
$ ./testFilter.o texts/sample_texts/lusiadas.txt
      texts/lusiadas_filtered.txt
```

2.2 Class Fcm

This is the main class of this project, responsible for collecting statistical information about texts, using finite-context models.

The class constructor method needs to receive the name of text file (that we want to collect information about), the order of the model and the smoothing parameter. The class will automatically generate a map with a string and a value attached to it. The string corresponds to k characthers (known as context) plus one (known as event), which is the next characther to the k ones, and the value corresponds to the number of times this next characther appears after that specific context, or in other words, the number of times a certain event occurs after a certain context.

To calculate the entropy of the whole text, first it is necessary to calculate the entropy of each event knowing that a certain context occured. And to calculate this specific entropy it is necessary to first calculate the probability of happening a certain context c (P(c)). Approaching this step by step.

$$P(e|c) = \frac{n_{(e|c)} + \alpha}{n_c + \alpha * |A|} \tag{1}$$

: Probability of e knowing that c happened.

Where $n_{(e|c)}$ corresponds to the number of times e happened after context c. n_c corresponds to the number of times that context c happened. And α is the smoothing parameter

$$H_{(c)} = \sum_{n=1}^{\infty} -P_{(e|c)} log_2(P_{(e|c)}) \tag{2}$$

: Entrophy of context c.

$$H = \sum_{n=1}^{\infty} H(i) * P(i) \tag{3}$$

: Entrophy the whole text.

Where H(i) is the entropy of the whole context i and P(i) is the probability of that context. This probability is different from 1. This one is calculated with the following equation.

$$P(i) = \frac{\sum_{n=1}^{\infty} n_{(i|c)}}{n_t} \tag{4}$$

: Probability of context i.

Where n_t is the total number of contexts and events on the whole text and $n_{(i|c)}$ is the number of times the event i happened after context c.

3 Part B

3.1 Class Lang

The Lang Class was developed to accept a text representing a language, a text to analyse the language, the order of the context model and the parameter α of the probability estimator.

```
lang(string model, string text, int order, double alpha)
```

This class uses the Fcm Class. First an Fcm object will be created, with the order of the model pretended, the smoothing parameter and the text representing the language. The entropy of the language is calculated usin the doFCM function. At last the function auxFcm, with the input of the text to analyse is called returning the estimated number of bits required to compress the text using the text representing the language.

```
Fcm fcm_lang = Fcm(k,alpha,model);
float num_bits = fcm_lang.doFCM();
num_bits = fcm_lang.auxFcm(text);
```

To test this class it is necessary to run the following commands:

```
$ g++ -g -Wall test_lang.cpp -o test_lang.o
  ./test_lang.o <modelLanguage> <text> <k> <alpha>
$ ./test_lang.o lang/portuguese.txt
      texts/sample_texts/lusiadas_filtered.txt 3 0.1
```

3.2 Findlang

Findlang is a program developed to, from a set of model languages, guess the language of a text introduced as an argument. This program supports two modes of operation, filtered and not filtered, this means that if the user chooses the filtered mode the program will filter the *input_text* and each model before trying to guess the language and if the user chooses not filtered the program won't filter any of the files used.

To test this program it is necessary to run the following commands:

```
$ g++ -g -Wall findlang.cpp -o findlang.o
  ./findlang.o <input_text> <k> <alfa> <mode>, mode= -f/-nf
$ ./findlang.o texts/lusiadas.txt 3 0.1 -f
```

The texts that constitute the set of model languages are inside the *lang_models* folder, this texts were choosen because they are representative of the languages they are written in, this comes from the fact that this texts are modern and big enough to have many different contexts written.

The program will, for each model text and the *input_text*, call the function *doLang()* and register the number of bits per symbol necessary to encode the language present in the *input_text* in the language of the model.
Finally the program will return the language associated with the minimum bits per symbol as the language that the *input_text* was written in.

4 Results

The *findLang* program was subject to tests for each file in the *sample_texts* folder, using the filtered and not filtered modes.
The results of a test run, using the filtered mode, order = 3 and $\alpha = 0.1$, can be seen in the table below:

Model Text	Input Text	Bits per Symbol	Input Text	Bits per Symbol	Input Text	Bits per Symbol
latvian	lusiadas	6.52371	french_wiki	5.82544	harry_potter	6.71237
italian		6.00272		5.44704		6.59701
czech		6.57483		6.0072		6.69802
swedish		6.47267		5.53875		6.44962
portuguese		4.76373		5.47578		6.65899
hungarian		6.66649		6.00739		6.71179
bulgarian		6.69823		5.93873		6.66512
romanian		6.32206		5.42904		6.63900
spanish		5.51958		5.28086		6.55061
greek		6.73795		6.01325		6.71870
slovenian		6.44559		5.95695		6.6053
french		6.22986		3.14525		6.51597
polish		6.66791		5.96797		6.72308
danish		6.52303		5.36986		6.36299
estonian		6.53380		5.80832		6.53662
english		6.24994		5.01223		4.79053
german		6.61341		5.47727		6.44064
croatian		6.54275		5.92693		6.50506
finnish		6.62918		6.06609		6.73577
maltese		6.64041		5.82752		6.71769
dutch		6.55713		5.42789		6.38432
slovak		6.58126		6.01440		6.69287

Input Text	Processing Time (s)
lusiadas	24
french_wiki	10
harry_potter	27

After analysing the results it is noticeable that the findlang found the correct language for each input text. For the text lusiadas, which is writen in portuguese, it returned 4,76 bits per symbol, as it is a latin language, similar to italian, spanish, the bits per symbol calculated for those languages were the closest to the portuguese one, 6,00 and 5,52 respectively, while for the remaining languages higher values were obtained.

The obtained results a test run, using the not filtered mode, order = 3 and $\alpha = 0.1$, can be seen in the table below:

Model Text	Input Text	Bits per Symbol	Input Text	Bits per Symbol	Input Text	Bits per Symbol
latvian	lusiadas	7.06116	french_wiki	6.37108	harry_potter	7.20273
italian		6.44705		5.83425		7.03315
czech		7.12185		6.51419		7.19958
swedish		6.98987		5.98404		6.86422
portuguese		5.2103		5.8967		7.12273
hungarian		7.20319		6.48174		7.19137
bulgarian		7.43787		6.64389		7.29331
romanian		6.81295		5.81655		7.09649
spanish		5.96205		5.66989		7.00416
greek		7.47543		6.72366		7.37514
slovenian		6.99261		6.42267		7.08905
french		6.68696		3.42472		6.94037
polish		7.20816		6.49295		7.19582
danish		7.05011		5.81176		6.79511
estonian		7.08386		6.30075		7.03076
english		6.65751		5.38924		5.10348
german		7.17559		6.05500		6.93058
croatian		7.1882		6.55058		7.07848
finnish		7.16234		6.52859		7.20323
maltese		7.15347		6.36415		7.18042
dutch		7.05889		5.89434		6.75096
slovak		7.1163		6.51042		7.18444

Input Text	Processing Time (s)
lusiadas	25
french_wiki	12
harry_potter	28

For the unfiltered mode, just as expected, the number of bits per symbol increased. Since the unfiltered mode contains numbers and uppercase letters, there will be a wider variety of contexts, and more bits will be necessary to compress the input text.

To test the impact of the α value more tests were done where $\alpha \in [0.1;1]$, order = 3 and mode = filtered. The results of these are shown below:

Input Text	**Alpha**	**Order**	**Bits/symbol of solution**	**Avg. bits/symbols of other models**
lusiadas	0.1	3	4.76373	6.46345
	0.2		4.76669	6.39509
	0.4		4.82948	6.3891
	0.6		4.8969	6.41113
	0.8		4.95801	6.43649
	1.0		5.013	6.46139

From this table we concluded that the higher the α the higher the number of bits per symbol. As α is used to calculate the probability of a character appearing knowing the context, it will increase the probability of a character appearing and therefore increase the number of bits per symbol needed.

To test the impact of varying the value of the order more tests were done. Where order $\in [3,8]$, $\alpha = 0.1$ and mode = filtered. The results of these are shown below:

Input Text	**Alpha**	**Order**	**Bits/symbol of solution**	**Avg. bits/symbols of other models**
lusiadas	0.1	3	4.76373	6.46345
		4	5.50663	6.93818
		5	6.18079	7.10179
		6	6.6534	7.15891
		7	6.92305	7.18044
		8	6.99845	7.19241

It is possible to infer that as the order increases the number of bits per symbol also increases. Because there are going to be more different contexts.

We have tested 10 texts, written in different languages, and it can be seen in the table below the results. And concluded that for each text *findLang* found the right language for each one.

Input Text	**Detected Language**	**Bits/symbol**
lusiadas	portuguese	4.76373
french_wiki	french	3.14525
harry_potter	english	4.79053
italian_wiki	italian	3.48849
finnish_wiki	finnish	4.09798
spanish_sinopse	spanish	3.92263
polishA1	polish	4.59235
german_wiki	german	3.60215
dutch_wiki	dutch	3.32218
latvian_wiki	latvian	3.96617

5 Conclusion

Throughout this project, it was concluded that the Finite Context Model is a successful and effective model to save the number of occurrences of events after certain contexts and to allow the calculation of the probabilities of these events happening. This lead to the verification that it also allows the identification of the language of a certain text using model texts for different languages. It was also confirmed that adjusting the smoothing factor affects the minimum number of bits to compress calcultated, since the larger the value of α, the bigger the values obtained.

Moving on to the findlang module, all languages tested were identified correctly. It was verified that as the order, k, of the FCM model decreases, the more generic the k-sized contexts will be, and therefore they will be more frequent in the input text. This means that the estimated number of bits to compress will be smaller.

However, if the value of k is too large, the context used to build the FCM will be too specific and harder to find in the text, which will translate into requiring more bits to represent the same text, so higher values will be calculated.
It was also observed that languages with the same origins are more similar, so it is more difficult to distinguish between them when analyzing a text.