Lab Work 1 Report

Guilherme Amorim, 107162
José Gameiro, 108840
Tomás Victal, 10918

University of Aveiro - TAI

Abstract
This project consists on the development of two main components:

- **fcm**: a program that measures the information content of text provided using a learned finite-context model
- **generator**: a text generator that creates text following depending on a model created

Both programs read a text file and train a finite-context model before execution of their main roles.

This report details all the steps and decisions taken during the process.

Contents

1 Introduction

The goal of this project is to develop a program capable of building a model that learns from a given text, as well as measuring the entropy of the respective text according to the created model. Besides that, other objective of this work is to generate a new text following the learned model. We developed our project using Rust [5] [1] [2] [4] [3]

2 Implementation

2.1 File Reader

The first step of each program consists of reading a text file and for this, ***file reader*** is used.

This module provides, there are three reading functions and an extra one to open a file:

- **read char:** reads a single character in UTF-8;
- **read buff:** reads bytes into a buffer, convert them to String and return the string length.
- **read word:** reads a line from the file and stores in a internal buffer every time that it is called it returns an word until the buffer is empty reading another line.

2.2 Fcm

To implement a ***finite context model*** we created a structure that contains 5 elements:

- **K value**
- **Alpha value**
- **Current context**
- **Symbols**: store all the distinct symbols present in the provided sequence
- **Counts**: A *HashMap* where the key is a context (a *sub-string* of length k), and the value is another *HashMap*. In this inner *HashMap*, each key represents a character that appears after the given context, and the corresponding value indicates the number of times that character occurs following the context.

The implementation of the ***fcm using words*** is exactly like the one explained before but instead of using char it uses Strings.

2.2.1 Train a model

The training process of the finite context model involves analyzing an input sequence and updating the frequency table ***counts*** that records the occurrences of characters following specific **k-length** contexts. This is implemented in the ***train char*** function, which processes characters sequentially while maintaining a sliding window of context.

Training Process

1. *Tracking unique symbols:* when a new character is encountered, it is added to the ***symbols*** vector if it is not already present.
2. *Updating the frequency table*: If the length of the **current context** has reached **k**, the function treats it as a valid context and retrieves (or initializes) the corresponding entry in the ***counts*** HashMap. Finally, the frequency of the current character following the context is increments.
3. **Sliding window mechanism**: To maintain a **k-length** context window, the oldest character is removed before appending the newly processed character.

2.2.2 Calculate the Average Information Content

The process to calculate the Average Information Context is done in two steps, first estimate the probability for a symbol and second calculating the Information Content Based on these probabilities.

The function ***compute probability*** estimates the probability of a character **symbol** occuring after a given context using **Laplace Smoothing** to prevent zero probabilities:

- It retrieves the stored occurrences of characters that have appeared after the given context;
- If the context has not been encountered before, an empty frequency map is used;
- The probability of the symbol is com-

puted as:

$$P(e \mid c) \approx \frac{N(e \mid c) + \alpha}{\sum_{s \in \Sigma} N(s \mid c) + \alpha|\Sigma|} \quad (1)$$

Where $N(e \mid c)$ is the count of the symbol e following c, $\sum$ represents the set of all possible symbols, and α is the smoothing parameter.

The function ***calculate information content*** iterates over a given text, extracting **k-length contexts** and the subsequent characters. For each context-symbol pair, the probability is computed using the function described above. Finally the total information content is summed across all characters in the sequence, and to obtain the average information content this sum is divided by the length of the sequence, has the following formula states.

$$H_n = -\frac{1}{n} \sum_{i=1}^{n} \log_2 P(x_i|c)$$

2.3 Generator

The first part of the module ***generator*** is quite similar to the ***fcm***: it begins by processing a given file and subsequently training a finite-context model. However, in this case, training can be performed either character by character or word by word. Additionally, the module simultaneously trains two distinct models: one where $\boldsymbol{k}$ is set to 1, and another where $\boldsymbol{k}$ is assigned the value specified by the user. This approach ensures that text generation remains possible even when the seed length is smaller than $\boldsymbol{k}$. Once this step is complete, the program is prepared to accomplish its primary objective: generating text based on the learned models.

2.3.1 Generate Text

The generator supports two modes:

- **Character-based mode**: Uses sequences of characters as contexts.
- **Word-based mode**: Uses sequences of words as contexts.

The program determines whether the **character-based** or **word-based** model should be used based on the option ***-m*** defined by the user.

Beyond the mode chose by the user, it is also necessary to indicate the context size $\boldsymbol{k}$, the prior of the sequence (the first characters/words) and the length of it

The process of generating text follows these steps:

1. **Choose the correct model**: if the seed length is smaller than k then use the model trained with k=1, when there is enough symbols for context where the length is k then switch for the other model.
2. **Initialize the context**: The last $\boldsymbol{k}$ elements of the prior are used to predict the next element
3. **Generate the sequence**: A random number generator is initialized to introduce randomness in the selection process. The frequency of the characters that appear after the given context is obtained. After this, the probabilities for each character are computed and if there are no recorded occurrences a space is given. Then a random threshold is obtained from a uniform distribution between 0 and 1, each character frequency count is converted in to a probability and a cumulative probability sum is maintained, so that once this value is exceeded the corresponding character is returned.

2.4 Chart Generator

To evaluate the sequence's entropy profile, another class was developed to generate a chart that represented this profile in a scatter plot.

For this we followed the following formulas:

$$P(e|c) \approx \frac{N(e|c) + \alpha}{\sum_{s \in \Sigma} N(s|c) + \alpha|\Sigma|}$$

$$-\log P(e|c)$$

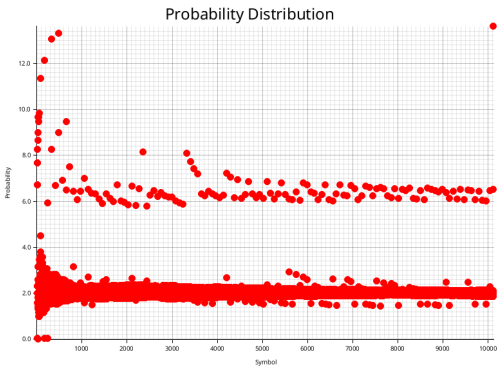
C

And created the ***ChartGenerator*** class, it estimates symbol transition probabilities using a smoothed frequency-based approach and then applies a logarithmic transformation to measure information content. These probabilities are stored later plotted on a scatter chart to provide insight into the sequence's entropy distribution.

3 Experiments

Comparative analysis to evaluate our implementation

In this section we present a comparative analysis to evaluate our implementation. We decided to perform multiple tests by executing our finite context model with multiple sequences (provided by the class coordinators), changing the values of **k** and **alpha** and training a model with sequences that our generator created.

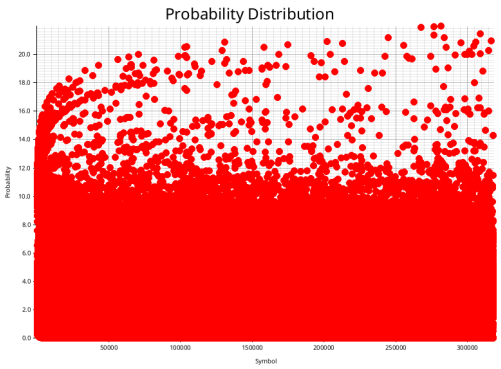


3.1 Multiple Sequences

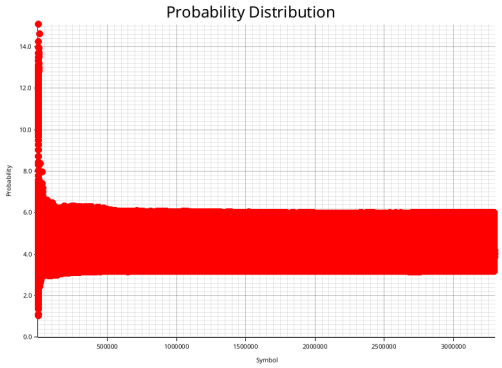
As stated before, we used multiple sequences to test our finite context model, where one of them included code written in **C** (sequence 5), another is a book from "*Os Lusíadas*" (sequence 2), and the rest of them included random characters (sequence 1 with 10124 characters, sequence 3 with 3295751 characters and sequence 4 has even more characters, but they are displayed in one line). We set the values of **k** and **alpha** to 3 and 0.01, respectively.

We obtained the following results for the average information content and generated a scatter plot that describes the sequence entropy profile:

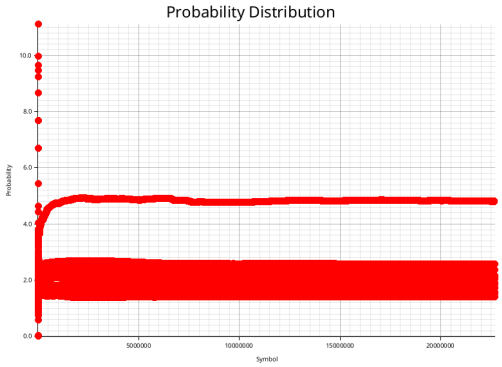
- **Sequence 1:** 2.0473 bits;
- **Sequence 2:** 2.2099 bits;



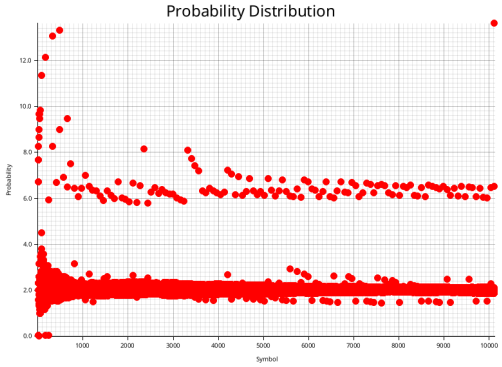
- **Sequence 3:** 3.9966 bits;



- **Sequence 4:** 1.8793 bits;



- **Sequence 5:** 1.3215 bits;



From the obtained results, we can observe that the average information content varies significantly across different sequences, reflecting the nature and structure of the data they contain.

The sequence with the lowest average information content is **Sequence 5 (C code)**, with **1.3215 bits**, indicating that programming languages tend to have a more structured and predictable syntax, leading to a lower entropy. Similarly, **Sequence 4** despite being composed of random characters, has a relatively low value (**18793 bits**), likely due to the natura of its characters distribution.

On the other hand, **Sequence 3** exhibits the highest information content (**3.9966 bits**), suggesting a higher level of randomness and unpredictability. **Sequence 1** and **Sequence 2** show intermediate values, with the latter (an excerpt from "*Os Lusíadas*)" having a slightly higher information content (**2.2099 bits**) than the former (**2.0473 bits**), which is expected as natural language carries more structure and redundancy than pure randomness.

3.2 Experimenting with different values of k and alpha

To analyze the impact of **k** and **alpha** on the average information content, we used **Sequence 1** and varied k in steps of 2, starting from 2 up to 10. and adjusted alpha by multiplying it by 10, ranging from 0.0001 to 10.

When varying the k value while keeping alpha fixed at 0.01, we obtained the following results:

K	Average Information Content
2	2.0528
4	2.0409
6	1,9257
8	1.7540
10	1.7267

When varying the alpha value while keeping k fixed at 3, we obtained the following results:

Alpha	Average Information Content
0.0001	2.0108
0.001	2.0143
0.01	2.0473
0.1	2.2908
1	3.4798
10	5.8183

From the results, we observe that increasing **k** leads to a decrease in the average information content. This suggests that a larger context

allows the model to capture more structure in the sequence, making predictions more certain and reducing entropy.

On the other hand increasing **alpha** has the opposite effect, increasing the average information content. This is expected, as higher **alpha** values introduces more smoothing, reducing the model's confidence in frequent patterns and making the predictions more uncertain. Notably, for **alpha = 10** the information content is significantly higher.

3.3 Text Generated

As previously mentioned, we use two methods for text generation: one treats characters as tokens during training and generation, while the other uses words as tokens using spaces as separators. The second approach improves the quality of generated text in the case of sequence 2, as it aligns with natural language patterns and ensures that real words are produced in the text generated.

Example of text generated with characters, k=2:

> Qualescerguantos e gerrando inadadeparendos toda pro tam,
> Que peça,
> Não Eôo porfeinhega,
> No pelo aquecinas reito do sena frio de.

Example of text generated with words, k=2:

> armas Nas brandamente,
> conduzidos adereça.
> arremessa, apelida
> resistirei Tomai consolar-te!
> Oceano;
> deixamos

However, if ***k*** is equal to 5, the character-based generator also produced quite good results, making the word-based generator only more viable in cases where ***k*** is small, as the training takes mutch more time for the words.

Example of text generated with characters, k=5:

> armas se no bélica elegantes,
> Por que saíam,
> Onde a pintura pôr a armas, porque a tua frio,
> A prata um remédio cerca areno,
> Para ajudasse
> De áspero, sábio e forte Artabro, e aspereza;

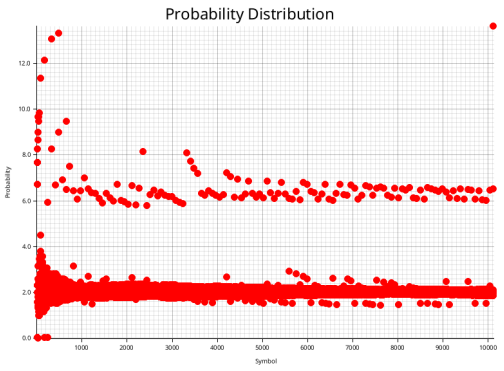
Example of text generated with words, k=5:

> armas em hospitais, nos Baco na Massília touro,
> esquecerão Abrantes,
> troque Umas, enseada
> 26
> sagaz "Porém Canace, penhor Gueos causaram adornado,

3.4 Train a model with Text Generated

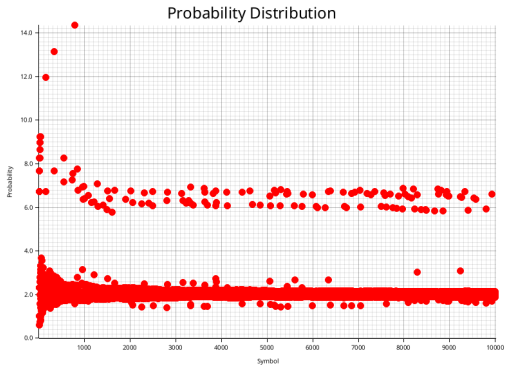
Another experiment we conducted involved training a model using a sequence generated by our own generated. For this, we use **Sequence 1**, setting **k = 4** and **alpha = 0.1**.

Firstly we created a scatter plot to represent sequence entropy profile of the original sequence and obtained the following image:

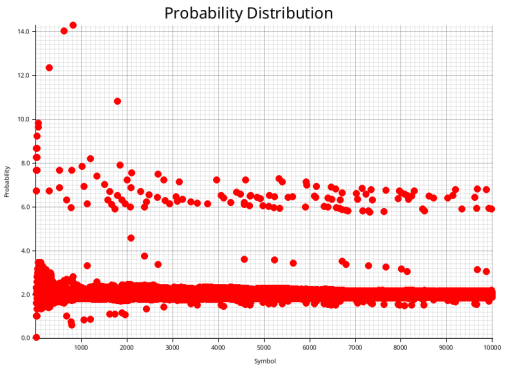


Then, we trained a model on the original sequence and obtained an average information content of **2.7458 bits**. Using this trained

model, we generated a new sequence of **10000 characters**, starting with the initial sequence *ACTG*. We also created a scatter plot for this new sequence and obtained the following:



Next, we trained a second model using the newly generated sequence and recalculated the average information content, obtaining **2,6582 bits**. Repeating this process, we generated another sequence while keeping the same parameter values, trained a third model, found that the average information content decreased to **2.5703 bits** and created another scatter plot for this new sequence.



The results indicate a progressive reduction in the average information content as the model is trained on its own generated sequences. This suggests that each iteration introduces more structure and predictability, likely due to the model reinforcing its learned patterns. However, this could also point to a loss of diversity in the generated sequences over multiple iterations.

4 Instructions

How to install dependencies, compile, and run the programs

4.1 Dependencies

Before running the programs, *Rust* and *Cargo* need to be installed. Additionally we used the following packages from rust:

- ***argparse***, version **0.2.2**
- ***plotters***, version **0.3.7**
- ***rand***, version **0.9.0**

To install Rust and Cargo, run the following command (if on Linux or macOS systems):

```
curl https://sh.rustup.rs -sSf | sh
```

On Windows download and run this file rustup-init.exe.

4.2 Compiling the Project

To compile the project, navigate to the root directory and run:

```
cargo build
```

After compilation, executables will be available in the ‘target/debug‘ folder.

4.3 Running fcm

To train the finite-context model and compute the average information content, run:

```
target/debug/fcm {file} -k {k} -a {a}
```

Where:

- **file**: Path to the ‘.txt‘ file with the data to train the model.

- **k**: Context size, i.e., the number of characters considered before the current character.
- **a**: Smoothing parameter to avoid zero probabilities.

4.4 Running generator

To generate text based on a trained model, run:

```
./target/debug/generator {file}
-k {k} -a {a} -p {p} -s {s} -m {mode}
```

Where:

- **file**: Path to the '.txt' file with the data to train the model.
- **k**: Context size, i.e., the number of characters considered before the current character.
- **a**: Smoothing parameter to avoid zero probabilities.
- **p**: Initial sequence of characters for text generation.
- **s**: Number of characters to generate.
- **m**: The mode that generator uses. The default is **normal** that uses *chars* as tokens the other mode is **words** that use *words* as tokens

4.5 Running chart generator

To generate a chart for a given sequence, execute the following command:

```
./target/debug/charts {sequence_file_path}
-a {a} -o {output_file}
```

Where:

- **sequence file path**: Path to the text file with the sequence data.
- **a**: Smoothing parameter to avoid zero probabilities.
- **o**: Name of the file that will contain the image generated with the chart it must end with *.png*

4.6 Examples

Some predefined bash scripts are available in the 'examples' folder to facilitate execution.

5 Conclusions

In this project, we had successfully implemented a finite-context model that was able to estimate the information content of text and generate output sequences that corresponded to the learned model.

The experiments made were also important to get some conclusions about the programs developed:

- Increasing the context size (k) reduces the average information content, making predictions more certain.
- Higher values of the smoothing parameter ($alpha$) values increase uncertainty in text prediction, leading to greater entropy.
- Word-based models can produce more coherent text when applied to natural language sequences.
- Training a model on generated sequences results in a progressive reduction of entropy, indicating an increase in predictability over multiple iterations.

Despite the encouraging performance, the algorithms could be optimized: with better smoothing schemes, improved training of the models and other techniques, more precision and diversity of output sequences could be achieved.

References

[1] Plotters Developers. Plotters developer's guide. https://plotters-rs.github.io/book/.

[2] The Rand Project Developers. Rand: A rust library for random number generation. https://docs.rs/rand/0.9.0/rand/.

[3] The Rust Project Developers. Rust by example. https://doc.rust-lang.org/rust-by-example/.

[4] Steve Klabnik and Carol Nichols. The rust programming language. https://doc.rust-lang.org/book/.

[5] Kevin K. Knapp. Argparse-rust: A command line argument parser for rust. https://docs.rs/argparse-rs/latest/argparse_rs/.