Information Models for Prediction

Alexandre Ribeiro
Student 108122, DETI
Aveiro University, Portugal
alexandrepribeiro@ua.pt

Maria Sardinha
Student 108756, DETI
Aveiro University, Portugal
mariasardinha@ua.pt

Miguel Pinto
Student 107449, DETI
Aveiro University, Portugal
miguel.silva48@ua.pt

***Abstract*—This report presents the implementation and evaluation of a Finite-Context Models (FCMs) for text analysis and predictive text generation, developed as part of the Algorithmic Theory of Information course [1]. The primary objective is to construct a probabilistic model that estimates the symbol distribution in a text based on fixed-length preceding contexts, enabling both statistical analysis of the textual data provided and predictive text generation. The main parts of the project were developed in C++ (and Python was also used for graph plotting on generated test results) and the previously mentioned tasks were divided into two distinct programs: `fcm` for the statistical analysis and `generator` for the text generation. The effectiveness of the models developed is examined by varying its context order (`k`) and smoothing parameter (`a`), with the results compared across multiple text datasets. Additionally two formats for model import/export were implemented and also compared.**

***Index Terms*—Finite-Context Model, Markov Chain, Entropy, Laplace Smoothing, Predictive Text, Information Theory, Probabilistic Modeling.**

I. Introduction

Finite-Context Models (FCMs) are discrete-time Markov chains that predict the probability of a symbol based on a fixed-length history of preceding symbols. They take upon Shannon's [2] work on entropy and information theory, to estimate the statistical properties of textual data by counting symbol frequencies within defined contexts. This approach is widely used not only in data compression techniques, but also in predictive text generation (similarly to our approach).

In this project, two main `C++` programs were developed:

- **fcm:** This tool reads input text, constructs an FCM by updating frequency tables, computes symbol probabilities using Laplace smoothing (to address zero-probability issues), and calculates the Average Information Content of the dataset.
- **generator:** This tool generates new text based on a trained model by taking an initial context and predicting subsequent symbols using weighted random selection. The generator supports both importing an existing model and creating a model on-the-fly.

To further improve usability, an interactive editor was developed to serve as a user-friendly alternative to command-line operations. The editor displays key model properties (such as model order, smoothing parameter, alphabet size, and context summaries), and supports syntactic analysis of generated text using a supplied word list.

The remainder of this report details the implementation of the FCM and its recursive extension, the evaluation of model performance under varying parameters, and a discussion of experimental results obtained from different text datasets.

II. Theoretical Background

This section reviews the fundamental concepts behind the modeling of information sources using finite-context models. These models, originally motivated by Shannon's work on information theory [2], provide a framework for predicting future symbols in a sequence by considering a fixed-length history. The following subsections detail the structure of finite-context models, techniques for probability estimation with smoothing, and the concept of Average Information Content as a measure of uncertainty.

A. Finite-Context Models

A Finite-Context Model (FCM) is a type of a Markov model, where each symbol in a sequence is assumed to be dependent on a fixed-length history of previous symbols.

Given a sequence of symbols $x_1, x_2, ..., x_n$, a k-order Markov model assumes:

$$P(x_n \mid x_{n-1}, ..., x_{n-k}) \approx \frac{N(x_n \mid c)}{\sum_{s \in \Sigma} N(s \mid c)},$$

where:

- c is the preceding context of length k.
- $N(s \mid c)$ represents the count of the symbol s following the context c.

B. Probability Estimation and Smoothing

A common issue with raw frequency-based probability estimation is that unseen events receive a probability of zero, which affects negatively the prediction. To address this, a smoothing parameter α is introduced:

$$P(x_n \mid x_{n-1}, ..., x_{n-k}) \approx \frac{N(x_n \mid c) + \alpha}{\sum_{s \in \Sigma} N(s \mid c) + \alpha |\sum|},$$

This adjustment ensures that every symbol, even those not observed during training, is assigned a nonzero probability. The choice of α can significantly affect the performance of the model.

C. Average Information Content (Shannon Entropy)

The Average Information Content, measured in bits per symbol (bps), quantifies the uncertainty or predictability of a sequence. It is computed as:

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

A lower value of H_n indicates that the model is more accurate and better trained, as it implies reduced uncertainty in predicting the next symbol.

III. Program Structure and Workflow

The project is organized into several components to support model training, prediction, and auxiliary functions. The main parts are:

A. Core Model Files

- **`FCMModel.h` and `RFCMModel.h`:** Define the Finite-Context Model classes and their recursive extensions, declaring methods for learning, prediction, entropy computation, probability estimation, and model import/export.
- **`FCMModel.cpp` and `RFCMModel.cpp`:** Provide the implementations of the methods declared in the header files.

These are further described in Section IV, but an overview of their dependency can be seen in Figure 1.

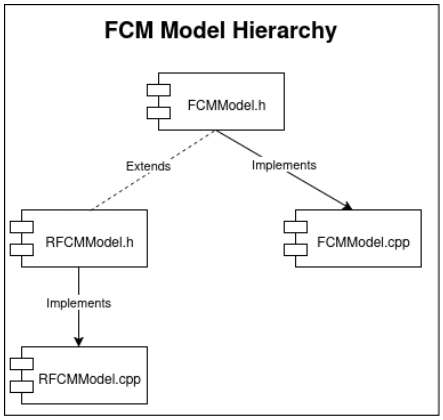

Fig. 1. Developed FCM Models Hierarchy.

B. Other Executables

- **`fcm.cpp` (Training and Analysis):**
  - Reads input text from a file.
  - Constructs a model based on user-defined parameters (context size `k` and smoothing parameter `a`).
  - Learns symbol probabilities by updating frequency tables and computing probability distributions.
  - Computes the Average Information Content of the dataset.
  - Exports the trained model as a JSON (or binary) file for later use.
- **`generator.cpp` (Text Generation):**
  - Optionally, if not given a model file, does all the operations mentioned for the `fcm.cpp` file, skipping the model loading part.
  - Loads a pre-trained model (in JSON or binary format).
  - Uses an initial context to generate a sequence of symbols based on the learned probabilities.
  - Applies selection based on the model chosen to predict the next symbol and outputs the generated text.
- **`editor.cpp` (User Interface):**
  - Provides a text-based interactive interface as an alternative to command-line operations.
  - Allows basic operations such as model creation, importing, learning from text and prediction.
  - Expands the usage of the models with advanced operations such as batch learning *(learning from multiple files)*, syntactic analysis, model lock/unlocking and access to the recursive Markov model.
- **`tests.cpp` (Automated Testing):** Contains automated tests that execute the main functionalities of both the `fcm` and `generator` executables, ensuring correctness and stability.

Figure 2 provides a high-level view of how the various executables (`fcm.cpp`, `generator.cpp`, `editor.cpp`, and `tests.cpp`) interact with each other and with the user.

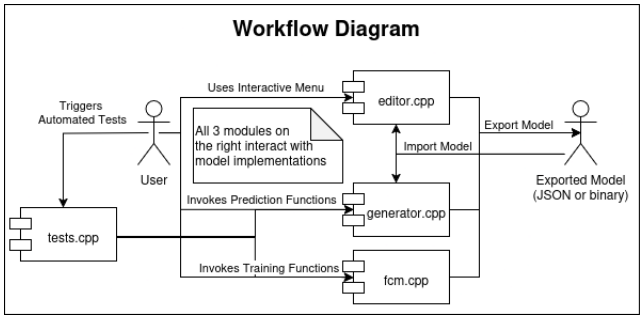

Fig. 2. Overall Project Workflow Diagram.

There are also folders containing sequences used in tests, as well as test results, and auxiliary code for plotting graphics. Notably, all sequences are text files and the graph plots were made with Python code, in opposition to the rest of the project made in C++.

IV. Model Implementation

A. Data Structures Used

The developed FCMs are implemented using hash tables (`unordered_map`) and a set to efficiently store and retrieve context-based frequency and probability distributions.

We have split the implementation of our simpler model with a fixed context or k size from a recursive model with all context sizes from k down to one.

In both cases there are some common data structures, such as:

- **Frequency table:** Stores the count of occurrences of each symbol, after a given context, where the first key is the context and the second key is the following symbol mapped to its count (number of appearances.
- **Context count table:** Tracks the total number of occurrences of each context.
- **Probability table:** Stores the computed probabilities of each symbol's appearance given a context. It is calculated based of the previous tables using smoothing techniques to avoid zero-divisions.
- **Alphabet set:** Stores all the unique symbols encountered in the training text.

By utilizing hash tables, the lookup time for contexts and symbols average at: $O(1)$ complexity, ensuring fast training and predictions even for large datasets, as the learning, storage, and retrieval of the statistical properties of the text data runs more efficiently.

B. Common Implementation Steps

This subsection describes the main stages common to both model's implementation: learning from input text, computing the average information content, and generating new text using probabilistic sampling. The implementation leverages several important design decisions that balance efficiency, accuracy, and practical usability.

1) Learning Phase: The training phase is implemented by the `learn()` method and consists of the following steps:

- **UTF-8 Splitting:** The input text is split into individual UTF-8 characters to correctly handle non-ASCII symbols. This ensures that multi-byte characters are treated as single units, preserving their integrity.
- **Sliding Window Processing:** A sliding window of size k is used to extract each context and the subsequent symbol. For every position in the text (with at least $k+1$ characters remaining), the window forms a context, and the following symbol is considered the prediction target.
- **Alphabet Updating:** Each unique symbol encountered during the parsing is dynamically inserted into the alphabet set. This set is later used to determine the total number of symbols for normalization and smoothing purposes.
- **Context Frequency Update:** Two hash tables are maintained: one for counting the frequency of each symbol after a given context (`frequencyTable`) and another for counting the total occurrences of each context (`contextCount`). These structures are updated for each sliding window, ensuring that the model accurately captures the statistical properties of the text.
- **Probability Computation:** After the learning phase, the `generateProbabilityTable()` method applies Laplace smoothing to compute the probability of each symbol following a given context:

$$P(s \mid c) = \frac{N(s \mid c) + \alpha}{N(c) + \alpha \cdot |\Sigma|}$$

where $N(s \mid c)$ is the frequency of symbol s after context c, $N(c)$ is the total count for context c, α is the smoothing parameter, and $|\Sigma|$ is the size of the alphabet. This ensures that even symbols not observed in the training data are assigned a nonzero probability.

2) Average Information Content Calculation: To quantify the quality of the trained model, the `computeAverageInformationContent()` method calculates the average information content (Shannon entropy) of the text:

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

This metric, measured in bits per symbol, reflects the model's uncertainty in predicting the next symbol. A lower H_n indicates a better-trained model with more precise probability estimates. The implementation iterates through the text (after splitting into UTF-8 characters) and accumulates the information content for each prediction based on the computed probabilities.

3) Text Generation: Text generation is accomplished using the `predict()` method, which operates in two stages:

1) **Context-Based Sampling:** Given an initial context, the method retrieves the corresponding probability distribution from the probability table. If the context is not present, a random symbol is selected from the alphabet.
2) **Cumulative Distribution and Random Sampling:** A cumulative probability distribution is constructed for the symbols following the context. A random number is then generated to select the next symbol based on the cumulative probabilities, ensuring that symbols with higher probabilities are more likely to be chosen.
3) **Rolling Context Update:** The newly predicted symbol is appended to the context while the oldest symbol is removed, forming a new rolling context for the next prediction. This process is repeated for n steps to generate a sequence that replicates the statistical properties of the original text.

Overall, the design supports both batch-mode text generation and interactive usage. The modular structure of the code (with dedicated helper functions for tasks such as UTF-8 splitting and probability computation) facilitates maintenance and future extensions.

C. Finite-Context Model (FCM) Implementation

The Finite-Context Model (FCM) serves as the baseline for text analysis and prediction in our project. It relies on fixed-length contexts to estimate the probability distribution of symbols based on their immediate history. The implementation of FCM includes the following key components and design decisions:

- **Learning Phase:** The `learn()` method processes the input text by using a sliding window of fixed size k. For each position in the text (provided that at least $k+1$ characters remain), the model extracts a context of k consecutive characters and records the subsequent

symbol. *(Explanation: This fixed-length context allows the model to capture immediate statistical dependencies, providing a straightforward yet effective means of estimating symbol probabilities.)*

- **Data Structures:** The FCM employs dedicated hash tables (`unordered_map`) to store the frequency of each symbol following a given context (`frequencyTable`) and to track the total occurrences of each context (`contextCount`). A dynamic set (`alphabet`) is also maintained to record all unique symbols observed in the text. *(Explanation: These structures ensure efficient lookup and updating, with average $O(1)$ complexity, which is critical for handling large datasets.)*
- **Probability Computation:** Once the frequency counts are obtained, the `generateProbabilityTable()` method applies Laplace smoothing to compute the probability of each symbol given a context:

$$P(s \mid c) = \frac{N(s \mid c) + \alpha}{N(c) + \alpha \cdot |\Sigma|}$$

  where $N(s \mid c)$ is the frequency of symbol s after context c, $N(c)$ is the total frequency for context c, α is the smoothing parameter, and $|\Sigma|$ represents the size of the alphabet. *(Explanation: Smoothing ensures that even symbols not observed in a particular context are given a nonzero probability, which is essential for robust prediction.)*
- **Average Information Content Calculation:** The method `computeAverageInformationContent()` evaluates the quality of the trained model by computing the Shannon entropy (average information content) of the text:

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

  (Explanation: A lower H_n indicates a model that better fits the data, as it reflects reduced uncertainty in predicting the next symbol.)
- **Text Generation:** Text is generated via the `predict()` method, which performs probabilistic sampling using the context-based probability table. Starting from an initial context, the method:
  1) Retrieves the probability distribution for the current context.
  2) Constructs a cumulative probability distribution.
  3) Uses weighted random sampling to select the next symbol.
  4) Updates the context in a rolling manner and repeats for n iterations.

  (Explanation: This process replicates the statistical properties of the original text, enabling the generation of new, contextually consistent text.)

Overall, the FCM provides a simple yet effective framework for text modeling, relying on fixed-length context windows and efficient data structures to achieve fast training and accurate prediction.

D. Recursive Finite-Context Model (RFCM) Implementation

The Recursive Finite-Context Model (RFCM) extends the basic FCM by incorporating multiple context lengths. In the RFCM, the learning and probability estimation are performed recursively for contexts of length k down to 1. The following key differences and enhancements were implemented:

- **Recursive Learning:** In the RFCM, the `learn()` method processes the input text by extracting contexts for all lengths from the maximum order k down to 1. This modification allows the model to capture statistical dependencies at different granularities. *(Explanation: This additional loop over context lengths enables a fallback mechanism when a high-order context is not available, thus improving robustness.)*
- **Separate Frequency and Context Counts:** The RFCM maintains recursive frequency tables (`rFrequencyTable`) and context counts (`rContextCount`) indexed by the context length. This differs from the FCM, which only considers contexts of fixed length k. *(Explanation: By storing counts for varying context lengths, the model can interpolate or back off to lower-order statistics when necessary.)*
- **Recursive Probability Table Generation:** A dedicated method, `generateProbabilityTables()`, is used to compute probability tables for each context length separately. This ensures that the smoothing formula is applied correctly at each level, accommodating differences in data sparsity. *(Explanation: This step is crucial for applying Laplace smoothing appropriately at each recursive level, ensuring a reliable fallback probability when higher-order contexts are unavailable.)*
- **Modified Probability Retrieval:** In the `getProbability()` method, the model attempts to retrieve the probability for the given symbol starting from the highest-order context available, and falls back recursively to lower orders if the context or symbol is not found. *(Explanation: This recursive retrieval mechanism improves prediction accuracy by using the most specific available statistics and gracefully degrading to more general statistics when necessary.)*
- **Export and Import Enhancements:** The RFCM includes extended export and import functions to handle recursive tables. The JSON/BSON serialization now saves separate tables for each context length, ensuring that the entire multi-level model can be reconstructed accurately. *(Explanation: These changes are necessary to persist the richer structure of the recursive model, allowing for later re-use and analysis without losing the multi-level information.)*
- **Model Summary Adaptation:** The `printModelSummary()` function in RFCM has been updated to print statistics for each context length, including unique contexts and total transitions at each level. *(Explanation: This provides a more detailed*

view of the model's internal state, highlighting the contributions of different context lengths to the overall performance.)

Overall, the RFCM theoretically improves the basic FCM through recursive processing to capture statistical dependencies at multiple granularities. These improvements should lead to a more robust model that can better handle cases where high-order contexts are sparse, but that was not the case, as discussed later in this report.

V. Model Exportation and Importation

The exportation and importation functionality allows trained models to be shared and reused in different text generation experiments. Initially, the `fcm` executable gathered text statistics and generated a model that was later imported by the generator for text prediction. In subsequent versions, the generator was enhanced with full model-building capabilities, while still retaining the ability to import previously exported models.

A. Export Formats

In our implementation, model serialization is supported in two distinct formats, that include the same essential parameters and data structures:

- Context size (`k`).
- Smoothing parameter (`a`).
- Alphabet set.
- Frequency and probability tables.
- Context occurrence counts.
- Model lock status.

The difference lies solely in the serialization method used from the `nlohmann::json` library.

1) JSON Format: This format produces a human-readable file that is easy to inspect and manipulate. However, the resulting file size tends to be larger due to its text-based nature.

2) Binary Format: The binary format, on the other hand, uses BSON serialization. It produces a more compact file, which is beneficial in terms of storage and potentially faster I/O operations, but the file is not human-readable.

B. Comparison: JSON vs Binary

Both formats encapsulate identical model parameters and structures. The choice between them depends on the intended use:

- **JSON:** Preferred for debugging, manual inspection, or scenarios where readability is critical.
- **Binary:** Ideal for production environments where efficiency and reduced storage space are prioritized. In the image below (1), a comparison in terms of model size between JSON and binary can be seen.

This dual-format approach provides flexibility, allowing users to select the most appropriate format based on their specific requirements.

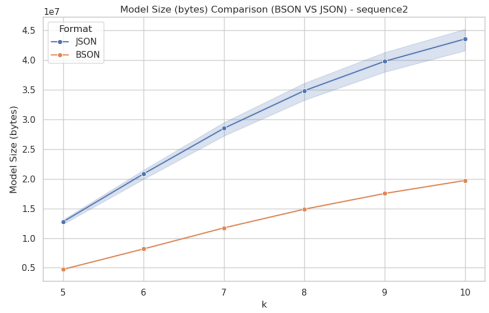

Fig. 3. Model Size Comparison (JSON vs Binary).

VI. Editor and Auxiliary Functionalities

To enhance usability beyond the command-line interface, an interactive text-based editor was developed. The editor enables users to create, import, and manipulate models with ease, and offers the following features:

- **Interactive Menu:** Provides options to create a new model, import an existing one, learn from text (via direct input or from files/directories), generate predictions, and obtain a syntactic analysis (as described in Section VI-A).
- **Model Properties Display:** Shows key parameters—such as model name, order k, smoothing parameter, alphabet size, and context summaries. Notably, when a model is imported in binary format, the editor re-establishes its readability for further manipulation.
- **Continuous Use:** Keeps the user-created model active, enabling iterative refinement through multiple learning sessions. Users can test the model at their convenience and seamlessly import or export it at any stage.

A. Syntactic Analysis via Word List

An auxiliary feature performs a high-level syntactic evaluation of generated text by comparing each word against a supplied word list (one valid word per line). It calculates a validity score—the percentage of words in the prediction that are found in the word list—offering a quick measure of syntactic coherence.

Overall, the editor and its auxiliary functionalities significantly improve the user experience by offering a graphical-like interface for model manipulation, quick visual feedback on model properties, and a simple yet effective mechanism for syntactic evaluation of the generated text.

VII. Experimental Results

To evaluate the performance of both developed models, we conducted experiments using various text datasets and different parameters: k (context size, with the option to set k based on the text size), α (smoothing factor), number of recursive steps, text generation size, and output format (JSON or binary).

This Section has the goal of demonstrating said results and discussing the trade-offs. In order to avoid showing all generated plots in this report we will use the original text from *sequence2.txt* file [3].

A. Generated Text Samples

Before analyzing the impact of any parameters, let's start by showcasing some generated text samples, visible in Figures 4, 5, and **??**). Where we used the `editor` program for easier visualization of the steps using different model parameters. The original text was taken from *sequence2.txt* [3] as mentioned before, and the syntactic analysis shown comes from a dictionary of Portuguese words [4].

```
Enter the context to predict from: por
Enter the number of symbols to predict: 200
Starting prediction with context: 'por'
Context has 3 characters, model k=3
Using rolling context: 'por'
Prediction:
 com da de o Rei serve está da mos:
Que hometeu Rei Joando,
A gerado já dos em génie os tratando,
E co'o prova tinha.

51
Istos,
A Rei deiralma nua falto da Euros
Do Reino Hebre o Céu, vai a quatre em

============================================
Do you want to perform a syntactic analysis? (y/n): y
Enter the filename to load the word list from: syntactic_analysis/ptWords.txt
Read 13929038 characters from syntactic_analysis/ptWords.txt
Syntactic Analysis Results:
Total words in prediction: 43
Valid words in prediction: 32
Percentage of valid words: 74.4186%
```

Fig. 4. Sample Text 1 ($k = 3$, $\alpha = 0.1$).

```
Enter the context to predict from: porta
Enter the number of symbols to predict: 200
Starting prediction with context: 'porta'
Context has 5 characters, model k=5
Using rolling context: 'porta'
Prediction:
 o mar aparece toda Frandecerá desbarata
Inda não cego enganosas se parece que esse a tanto menos de Anfitrite deleito,
Pelo neto de tão dirá que tenro a espanto
Que tão fizerem ao mesta, por que tant

============================================
Do you want to perform a syntactic analysis? (y/n): y
Enter the filename to load the word list from: syntactic_analysis/ptWords.txt
Read 13929038 characters from syntactic_analysis/ptWords.txt
Syntactic Analysis Results:
Total words in prediction: 37
Valid words in prediction: 28
Percentage of valid words: 75.6757%
```

Fig. 5. Sample 2 ($k = 5$, $\alpha = 0.5$)).

The first two examples in Figures 4 and 5 are using the simpler FCM Model (not the recursive one) clearly show that the model generates somewhat readable text and even has the capability of recreating new chapters with around 75% of the words being recognized as syntactically correct.

This last sample from our recursive models shows a slight improvement of around 5% in syntactically correct words, but it is word noting it does take longer to train, as all context sizes from k all the way to 1 are saved and made predictions upon. Furthermore, through logging in development we can say that most of the time, the model is using context of size k, it just reduces context when absolutely necessary for prediction at hand.

Overall, the text predictions shown, demonstrate how models maintain the coherence depending on context size and smoothing parameter.

B. Effect of Context Size on Information Content

In order to measure how the context size impacts the predictability of generated text, we performed the computation of

```
Enter the context to predict from: temos
Enter the number of symbols to predict: 300
Starting prediction with context: 'temos'
Context has 5 characters, model k=5
Using rolling context: 'temos'
Prediction:
 se vai da largo Oceano Índia, e a majestade.

14
"Vê-los soberanos;
Mas um Cochim, e o Céu justa com ânimo edifico.

41
"Não no reinos lhe corte do foi a tornava!
Que, rodeado
De que para que nunca escudo às aves, cuja voz grandes não, mas não te estreito a dividiu
A nossos foram, que buscar o furo

============================================
Do you want to perform a syntactic analysis? (y/n): y
Enter the filename to load the word list from: syntactic_analysis/ptWords.txt
Read 13929038 characters from syntactic_analysis/ptWords.txt
Syntactic Analysis Results:
Total words in prediction: 59
Valid words in prediction: 48
Percentage of valid words: 81.3559%
```

Fig. 6. Sample 3 (Recursive Model, $k = 5$), $\alpha = 0.1$.

the Average Information Content across multiple values of k. The results are summarized in Table I based on *sequence2.txt* [3].

Context Size k	**Average Information Content (bits per symbol)**
1	3.34668
3	2.4112
5	2.41512
10	3.14763
30	3.24091

TABLE I
AVERAGE INFORMATION CONTENT BASED ON k

As expected, increasing k results in a more accurate model up to a certain point, after which diminishing returns and data sparsity issues are raised.

C. Impact of Smoothing Parameter on Information Content

The smoothing parameter α helps in the handling of unseen contexts. To analyze its effect on the average information content, we tested our model with different values, while keeping k constant during each test. In the end, we created a visualization of all information content values for various k values. The results can be seen below in Figure 7.

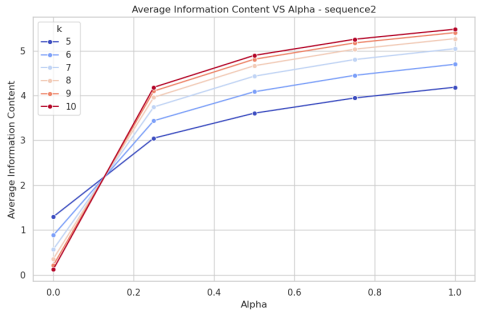

Fig. 7. Smoothing Parameter Impact on Information Content.

So, we can conclude that $alpha$ must be chosen carefully to balance over-smoothing and under-smoothing. A very

low $alpha$ can underestimate unseen symbols (leading to an overly "peaked" distribution with lower measured information content), while a large $alpha$ can flatten the distribution excessively (raising the overall entropy). In practice, moderate values (e.g., around 0.1) should result in more balanced and robust models.

D. Complexity Profile

To evaluate how well the model captures the statistical properties of the training text, we computed complexity profiles for different datasets. Figures 8 and 9 show the results for *sequence2.txt* and *sequence1.txt*, respectively.

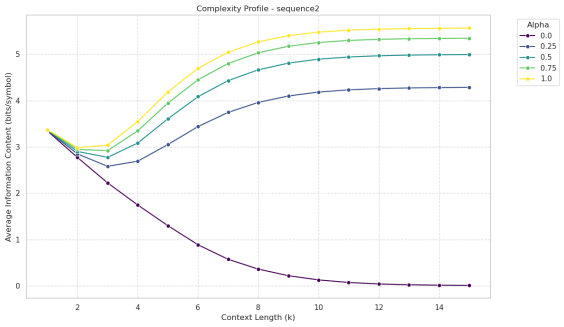

Fig. 8. Complexity Profile Evolution.

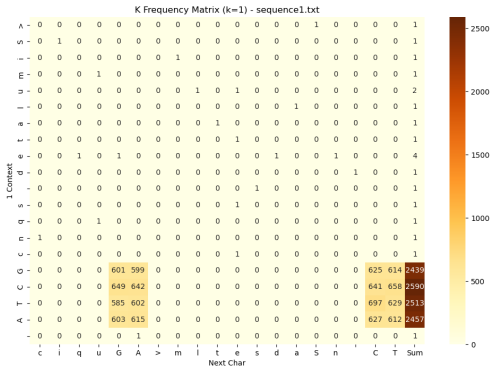

Fig. 9. Complexity Profile Matrix (k=1).

- For low α values, such as $\alpha = 0$, the complexity decreases sharply, indicating highly structured and predictable sequences. In contrast, for higher α values, the complexity initially increases before stabilizing, suggesting more randomness and presence for long-range dependencies. In this case 8, at k=2 all curves align before diverging, indicating that short-range correlations are well captured. As k increases, the complexity profile reflects how additional context influences predictability, with lower k values contributing more significantly to information gain. However, at higher k, the profile gradually stabilizes, reaching a point where further increasing k has minimal impact on the information content.
- The matrix analysis reveals that lower-order contexts contribute more significantly to the accuracy of the predictions, supporting the benefits of recursive models. Here 9, we can observe the distribution of information content across different symbol transitions. The darker spots indicate transitions with high predictability, while bright regions represent more uncertain predictions.

VIII. Exploratory Implementations

This Section details experimental features developed or thought about to add functionality to the code produced that were not fully included in the outcome due to time constraints. Nonetheless, they can be considered as areas for improvement.

A. Dictionary-Based Correction

The output generated by the models when processing works from authors often contained strings of characters that resembled words but were not actual words. To address this, a feature was developed that allowed users to provide a file containing a list of valid words, serving as a language dictionary (which ended up being simplified to the feature mentioned in Section VI-A). The goal was to improve the coherence of the output by integrating this dictionary into the prediction process. The initial implementation involved feeding the model with the dictionary, word by word, with spaces before and after each word.

However, the results were far from acceptable, leading to outputs that were nearly random strings of characters. A thorough analysis of the implementation revealed that the issue likely stemmed from how the model internalized the dictionary. Specifically, feeding the model an extensive list of words caused the frequency table to become increasingly ambiguous due to the sheer volume of entries, disrupting meaningful word associations within the actual text to analyze and predict upon.

A more effective approach would involve storing the language dictionary separately and using it as a post-processing step rather than directly embedding it in the training data. After generating predictions based on the model's learned frequency table, words could be mapped to the closest valid dictionary entries using techniques such as the Levenshtein distance [5] or Word2Vec embeddings [6].

B. Dynamic Weight Shift in Context Selection

During our implementation of the Recursive Finite-Context Model (RCFM), we identified a potential enhancement that could significantly improve prediction quality. The current RCFM implementation assigns a fixed importance to each context size all the way from k down to 1, being less important the smaller the context, meaning that the smaller context sizes are only there as a fallback when the context of size k is not found, so the model searches on lower sizes but it does not adapt.

This flat weighting scheme fails to account the changes in reliability of different context lengths, and a more sophisticated approach would involve implementing a dynamic

weighting mechanism that would adjust the contribution of each context length based on comparing several context sizes in each prediction done.

We recognized that this extension of functionality would require significant additional development, as it would have to rely on prediction accuracy across various datasets and parameter configurations to calibrate the weight adjustment mechanism.

IX. Conclusion

In this Section we will go through the findings we consider relevant while developing and analyzing the results of the problem at hand, as well as what could be improved in future instances of a similar work to ours.

A. Main Findings

Our work demonstrated that Finite-Context Models (FCM) and their recursive extension (RFCM) can provide effective frameworks for probabilistic text analysis and generation. Through all our experimentation with various parameters and training data, we have established some findings we would like to discuss:

- **Context Size Dynamics:** The context size parameter k significantly impacts model performance, with an optimal range observed between 3-5 for the tested datasets. As shown in our experiments, increasing k beyond this range often leads to diminishing returns and eventually degraded performance due to data sparsity issues. This is reflected in the Average Information Content measurements, which reached minimum values around $k = 3$ to $k = 5$ (approximately 2.41 bits per symbol) before rising again at higher context lengths.
- **Smoothing Parameter Selection:** The Laplace smoothing parameter $alpha$ plays a crucial role in balancing the model's handling of unseen events. Our experiments demonstrated that moderate values (around 0.1) generally produce more reliable models, while extreme values tend to either underestimate rare events (very low $alpha$) or excessively flatten probability distributions (high $alpha$).
- **Export Format Efficiency:** Our implementation of binary (BSON) serialization significantly reduced storage requirements compared to JSON format, with size reductions of approximately 40-60% observed across various model configurations.
- **Model Architecture Comparison:** The standard FCM demonstrated great text generation capabilities, producing some syntactically valid output (around 75%) with recognizable stylistic elements. Whereas the RFCM, conceptually more "sophisticated", showed a slight improvement with an increase of around 5% in the output validity.
- **Text Generation Quality:** Our syntactic analysis revealed that the models could generate text with a moderate to high quantity of words matching a Portuguese dictionary [4], when using optimal parameters. Additionally, we can say the text exhibited some stylistic characteristics of the source material.

B. Future Work

We were also left with some promising directions for future research and development, namely:

- **Adaptive Context Selection:** Implementing a dynamic mechanism that selectively varies the context size based on the availability of matching patterns in the training data could optimize the trade-off between specificity and generalization. This could potentially be achieved through weighted interpolation of predictions from different context lengths.
- **Enhanced Dictionary Integration:** Concluding the idea of a text post-processing pipeline (similar to an auto-correct in today's world) that leverages techniques such as Levenshtein distance [5] or word embeddings (e.g. Word2Vec [6]) could improve the linguistic coherence of generated text without disrupting the underlying statistical model.
- **Interactive Text Generation:** Expanding the interactive editor to support real-time suggestions and completions could transform the system into a practical writing assistant tool that leverages the predictive capabilities of the trained models.

References

[1] T.A.I. 1st Project Guidelines - Elearning. Retrieved from: https://uapt33090-my.sharepoint.com/personal/an_ua_pt/Documents/DETI/TAI%202024-2025/Pub/Project%20%2301/TAI_WORK_1_2024_2025_RC.pdf?CT=1740591621382&OR=ItemsView

[2] Shannon, C. E. (1951). Prediction and Entropy of Printed English. Bell System Technical Journal, 30(1), 50-64.

[3] Camões, L. Vaz de. *Os Lusíadas*. First published in 1572. Public domain text available via multiple sources (e.g., Project Gutenberg).

[4] jfoclpf. *words-pt: Portuguese Words List*. GitHub repository, 2020–2024. Available at: https://github.com/jfoclpf/words-pt (accessed on 1st March 2025).

[5] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," *Soviet Physics Doklady*, vol. 10, no. 8, pp. 707–710, 1966.

[6] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, "Distributed Representations of Words and Phrases and their Compositionality," in *Advances in Neural Information Processing Systems*, 2013, pp. 3111–3119.