TAI Report 1 - Markov Models

Diogo M. Marto NºMec: 108298[a,*], Diogo Pinto NºMec: 110341[a,*],
Miguel Vieira, NºMec: 85095[a,*] and Ilker Atik, NºMec: 123947[a,*]

[a]*DETI, University of Aveiro, Portugal*

1. Introduction

In this lab work, we explore the application of finite-context models in measuring the information content of text sequences. Our primary objective is to develop a program that computes the Average Information Content of a given text using a Markov model of a specified order. Additionally, we implement an automatic text generator that produces new text based on a learned model, allowing us to assess the model's effectiveness in capturing the statistical properties of the sequence.

The lab work involves implementing two main programs: **FCM**, which calculates the information content using finite-context models, and the **Generator**, which generates text based on a learned model. We also investigate how various parameters, such as the order of the model (k) and the smoothing parameter (α), affect the model and its performance in generating realistic text.

In the following sections, we describe the methodology used to develop the programs, discuss the results obtained from different texts, and analyze the impact of parameter variations on the average information content. This report also provides a comprehensive overview of the approach and insights gained during the development process.

2. Finite Context Model (FCM)

The objective is to implement a Finite Context Model to approximate the entropy of a text using a k-th order Markov chain and additive smoothing. The application reads a text file in its entirety and stores all unique characters in a set, which defines the alphabet of symbols. It then iterates over each position in the text (up to its length minus k) to extract a substring of length k, known as the context, and identifies the immediate next character that follows this context. These contexts and their respective next characters are recorded in a map, which holds, for each context, another map of character counts that indicates how many times each symbol appears after that context. At each step of processing, the class updates this nested map by incrementing the occurrence count of the newly observed character, thereby capturing context dependencies. To address sparse data issues, an additive smoothing technique is employed using the parameter α; effectively, a small positive quantity is added to the counts so that rare contexts or characters do not result in zero

*Corresponding author. E-mail address: diogo.marto@ua.pt,pintodiogo@ua.pt,miguelmvieira@ua.pt,ilker@ua.pt

probabilities. With each new symbol, the model calculates the log probability of that symbol given the current context, summing these log probabilities in a running total. Once all positions are processed, the final entropy is computed by the average negative log probability, producing an estimate of the average information content per symbol.

2.1. Data Structure

In the development of the model, we explored 2 different data structures, namely the already mentioned *Map<String,Map<Char,Int>>* which maps a context to a map of characters that contains the counts. The other data structure uses two maps: one that maps the context and the next character to a count where one map maps the context plus the next character to a count ($N(e|c)$), and another that maps the context to the total count of that context ($\sum_{s \in \Sigma} N(s|c)$). We ran some experiments and found that the *Map<String,Map<Char,Int>>* implementation was faster by around 10%, this was against our initial hypothesis since we expected the cost of allocating small maps would outweigh the cost of inserting in 2 distinct maps, but in practice we found the opposite.

2.2. Comparison of Different Languages

We implemented the FCM program in several programming languages with the interest of seeing the difference in performance across them. Our test consists of computing the entropy in sequence 4 at different context lengths and with $\alpha = 1$, our environment consisted of a desktop with an AMD Ryzen 7 7800X3D, 32GB RAM GGDR5 6000MT/s running WSL (Windows Subsystem for Linux).

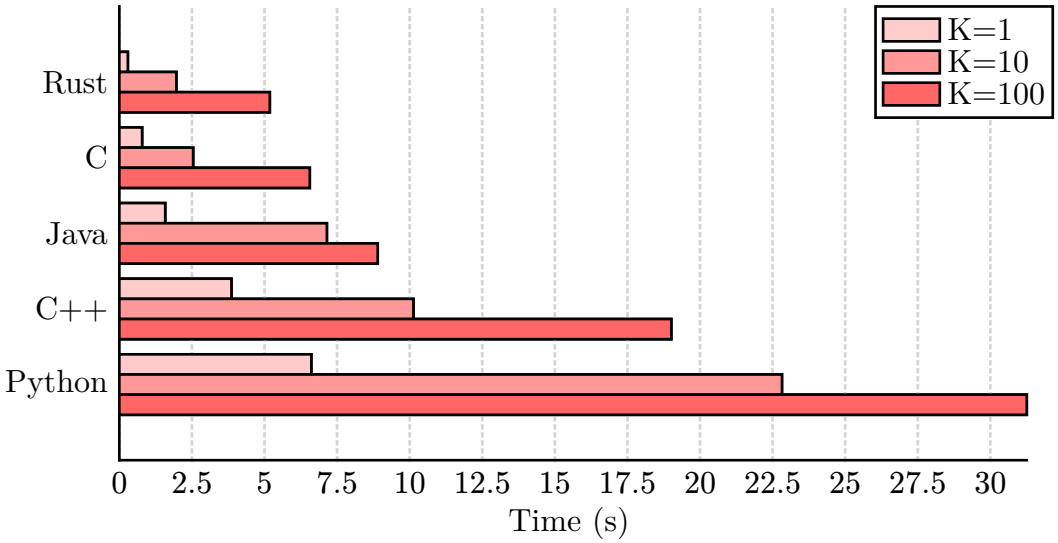

Figure 1: Comparison of implementations in different programming languages on time for sequence 4 on context size 1,10 and 100. The results for the Rust and C version at K=1 are 293ms and 801ms, respectively

As we can see from Fig. 1, it is noteworthy that the Rust implementation performed the best despite our limited experience with the language. On K=100, the C version was 26.5% slower, the Java version was 71.7% slower, the C++ version was 266.8% slower, and the Python version was 503.1% slower.

To note, all implementations use a similar data structure. The C and Rust implementations also have a custom hash function, the other implementations use the hash function with the builtin implementations of the hash table. The C++ version was developed using the provided collections such as unordered_map and string_view, otherwise, the code would be equal to C. These results are very biased to our coding ability.

2.3. Results

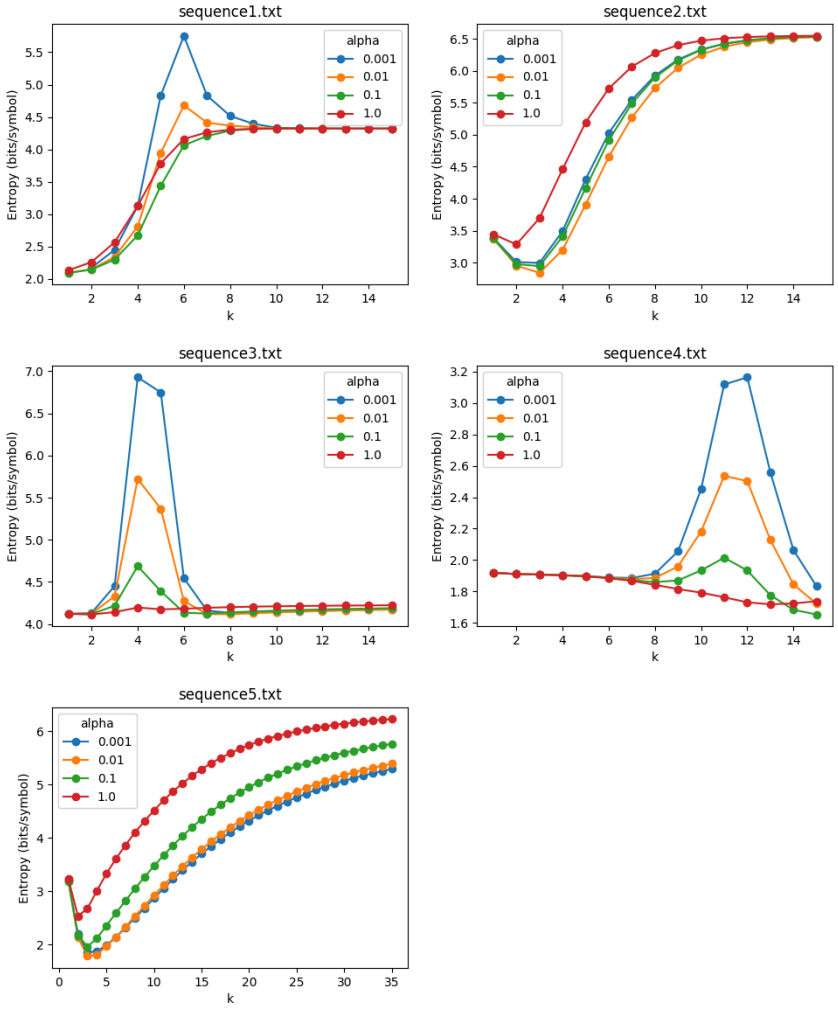

Figure 2: Context Width (k) vs Entropy

From the results, we can see that some sequences have similar behaviours, namely sequence 2 and sequence 5, which are both texts, one being "Os Lusiadas" and the other C code, respectively. They show similar results where they the best entropy with a context of 2 for an $\alpha = 1$ with results between 2.5 and 3 and slowly converge to a similar number, although sequence 5 converges to a lower value (since it has a shorter alphabet).

Sequences 1 and 4 are DNA sequences, but their graphs are different. This is due to sequence 1 containing the text `>Simulated sequence` at the beginning, which alters the alphabet size and makes it so that the entropy converges to a higher value with higher

values of the context. But if we focus on small context sizes, the sequences have similar values of entropy near 2 since the alphabet size is 4.

Sequence 5 appears to be a protein sequence, and its entropy is slightly above 4 for reasonable values of α, which is very close to $\log_2(19) \approx 4.25$, where 19 is the alphabet size.

2.4. Smoothing parameter (α)

- If α is very small (near zero), we rely almost entirely on raw counts. Very rare events (symbol x after context c that never occurred in the data) will get almost zero probability.
- If α is large, it "flattens" or "uniformizes" the probabilities, making them less sensitive to the data's actual frequencies. Entropy tends to increase as α grows because forced uniformity raises uncertainty.
- As α goes from near 0 toward larger values (e.g., 1.0 and beyond), it pushes every conditional probability $P(x|c)$ closer to a uniform distribution.
- Once α is significantly large relative to the total counts, further increases in α won't change things much: the conditional probabilities are already dominated by the smoothing term, so entropy levels will be near to the maximum possible for the data.
- We observed that as the α grows, the entropy slowly converges to log2 of alphabet size; this is the result of the probability of each symbol getting closer to 1/alphabet size. If α is too big, each symbol has an equal chance to be the succeeding character after any context.

2.5. Context Width (k)

- For small k (e.g., $k = 1$), we only look at a 1-symbol context. If the data has strong correlations extending over many symbols, we might miss them, resulting in relatively large (or inaccurate) entropy estimates.
- As k increases, we capture more complex contextual dependencies. In truly correlated data, the next symbol becomes more "predictable," which can reduce the measured entropy.
- Eventually, when k is so large that it spans or exceeds the natural correlation length of the data, increasing k further will have little effect: the entropy "saturates." That is, once

we've captured all relevant correlations, more context doesn't reduce (or dramatically change) the conditional uncertainty.

2.6. Memory usage

- Memory usage of FCM observed on the Java program.
- As the k grows the memory usage also tends to grows, the growth is highly dependent on the randomness and diversity of the text, and after a certain point all sequences become unique so the map, for each context, will be filled with one symbol that follows the context and it will have count 1, resulting in high number of inner maps holding one key and value 1.
- The sequence1 which is a small file, when running FCM, the memory usage was similar in all k and alpha values on the other side sequence4 required above 4 GB RAM to calculate $k = 15$, whereas the rest of the sequences were using 2 GB RAM at maximum, therefore the file size has a big impact on memory usage.

2.7. Time

- FCM runtime depends on the file size, α value has no impact on the time, higher k values usually increases the time consumption especially for big files, since the hash table will be much bigger due to uniqueness of contexts.

2.8. Complexity over file

In the FCM program, we also implemented the option to view the information content of the symbols of a sequence (only in C and Java). With the example sequence `ATGTGATTTTAATAGCTTCTTAGGAGAATGACAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA` we get the graph in Fig. 3 and we can see that the tail of the sequence which contains only `A`'s correspond to a drop in the graph.

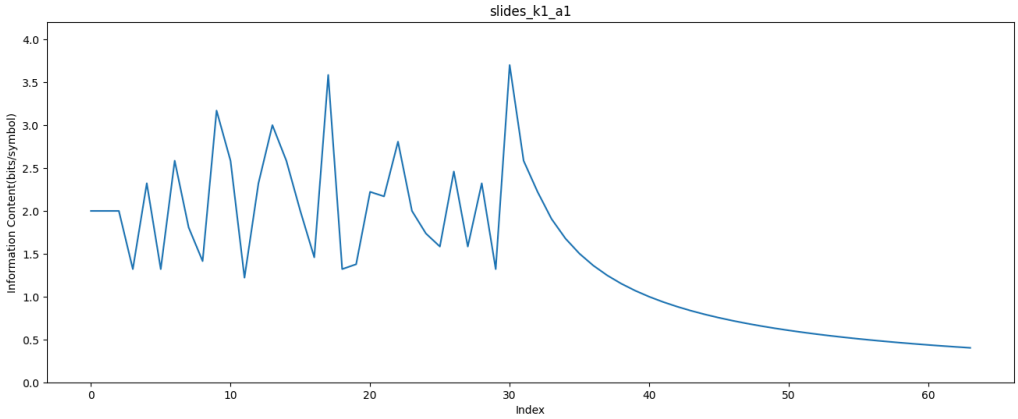

Figure 3: Information Content (bits/symbol) vs Index of symbol. For sequence1 K=1 and $\alpha = 1$.

For Fig. 4, we can see that there are parts of the sequence where the information content drops significantly, hinting at an underlying structure of the sequence where certain parts have different information contents.

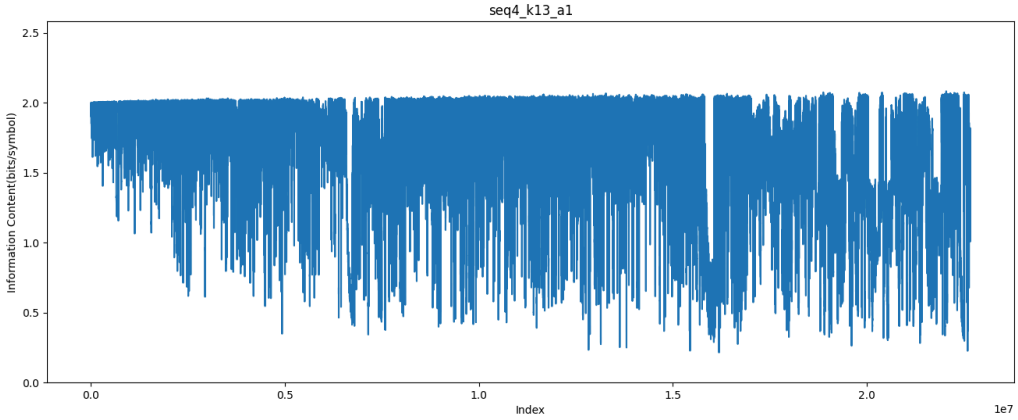

Figure 4: Information Content (bits/symbol) vs Index of symbol. For sequence4 K=13 and $\alpha = 1$ with a low pass filter.

For Fig. 5, we can see that the graph is very spiky. This is due to the smaller context and the fact that the underlying sequence is C code where there are parts such as languages construct (for, if, while) which are repeated a lot and parts such as variable names which aren't as repeated, so there are a lot of fluctuations.

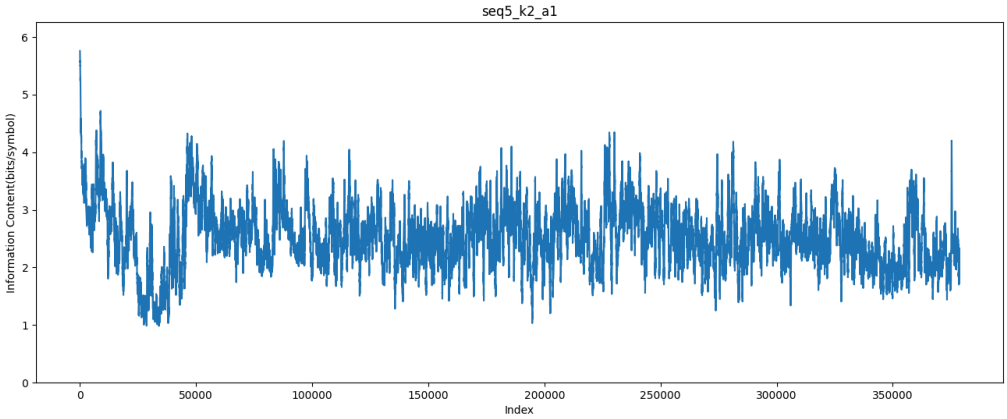

Figure 5: Information Content (bits/symbol) vs Index of symbol. For sequence5 K=2 and $\alpha = 1$ with a low pass filter.

We were interested in how the information content would look like on images. So for the same image of a purple flower, we computed the information content for an uncompressed ppm image (Fig. 7), a compressed png image (Fig. 8), and an uncompressed image where we applied the JPEG-LS predictor described in Eq. (1) (Fig. 9), considering each byte has a symbol.

Figure 6: Image of a purple flower.

In the uncompressed image graph Fig. 7, we can see portions where the entropy drops significantly due to redundant patches that contain repeated colors (probably purple). The average entropy is around 7,36.

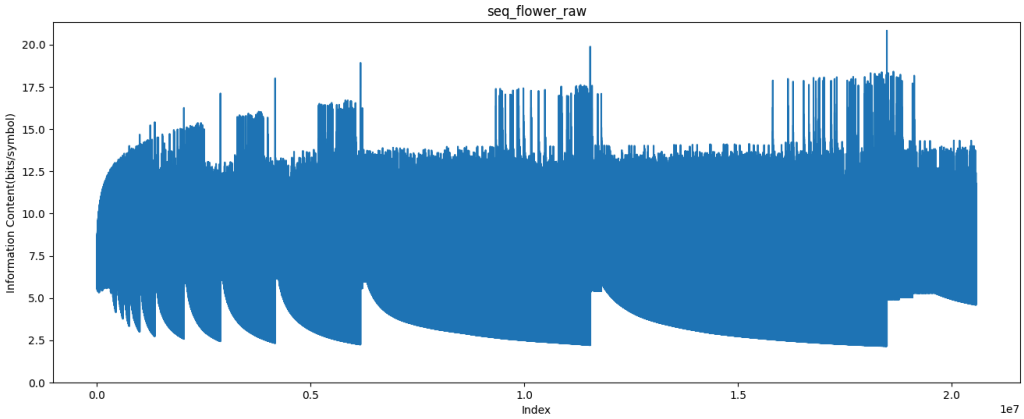

Figure 7: Information Content (bits/symbol) vs Index of symbol. For purple flower (uncompressed) K=1 and $\alpha = 1$

In the compressed image graph Fig. 8, we can see that it oscillates around 8 bits, which is expected since on the compressed image, we expect the compressor to have removed most redundancy. The average entropy is around 8.01, which means that each byte holds all the information it could.

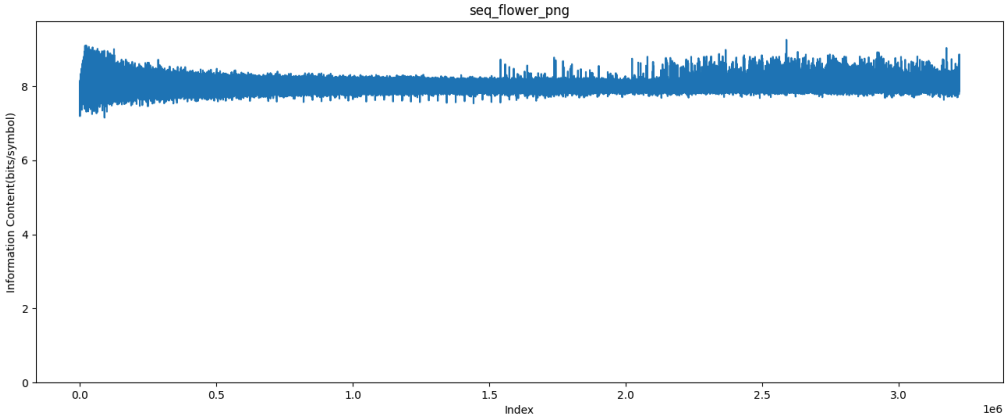

Figure 8: Information Content (bits/symbol) vs Index of symbol. For purple flower (compressed) K=1 and $\alpha = 1$

In the image with the JPEG-LS predictor applied Fig. 9, we can see similar behaviours as in Fig. 7 where we have patches where the entropy drops significantly, but the values of the entropy are significantly lower being on average 1.664. This implies that using a

predictor made the image much more compressible ($\frac{7.35}{1.664} \approx 4.42$ less information; the PNG compression was 6.37, which is slightly better).

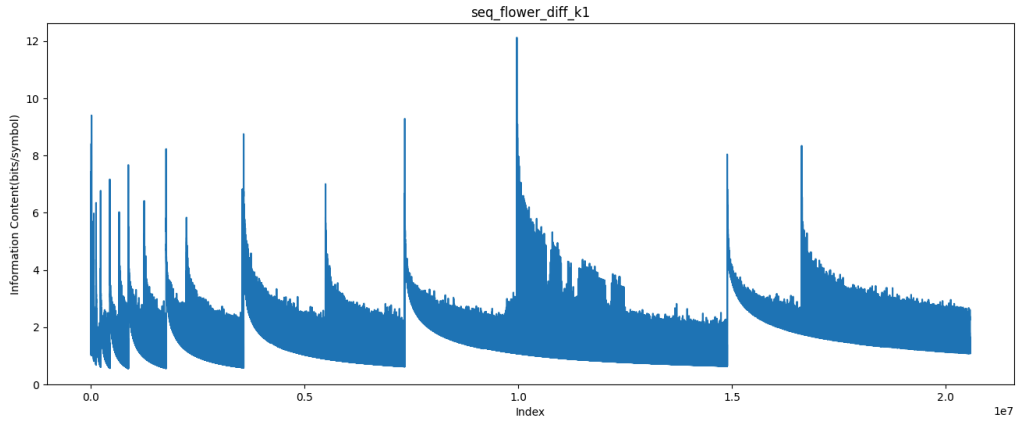

Figure 9: Information Content (bits/symbol) vs Index of symbol. For purple flower (predictor) K=1 and $\alpha = 1$ with a low pass filter.

The formula for the JPEG-LS predictor is the following:

$$X = \begin{cases} \min(A, B), \text{if } C \geq \max(A, B) \\ \max(A, B), \text{if } C \leq \min(A, B) \\ A + B - C, \text{otherwise} \end{cases} \tag{1}$$

3. Generator

In this section, we discuss the development of the text generator program. We begin by detailing the first implementation, addressing its flaws with examples to illustrate the challenges encountered. Then, we explore how a stochastic generator works by utilizing probabilities to model a random process based on the learned context. And experimented with the use corrected probabilities with α.

We also consider context reduction as a technique for handling cases where context is insufficient and where smaller contexts can be used by ignoring further characters. This technique has a lower overhead when processing but can have a bigger impact on the runtime of the generation.

Finally, we present the final implementation, highlighting its features and command-line arguments, and provide examples using different training files to demonstrate its functionality.

3.1. ***Java Implementation***

To implement the generator, we defined several execution phases:

1. Initialize Values
2. File Reading (Sequences)
3. Processing Component Initialization
4. Data Transformation into Frequency Tables
5. Character Generation Component Initialization
6. Final Output Generation

1. Initialize Values

In this phase, we receive the required parameters, allowing the user to specify the inputs for the model.

Mandatory Parameters

- -p –prior → Refers to the initial context or sequence of symbols provided to the model before it begins generating new symbols. This must have a length equal to K, and it must exist as a context in the frequency table, or the *priorFixing* option must be enabled.
- -rl –responseLength → Specifies the length of the output.
- -f –file → Defines the path to the desired training sequence file.

Optional Parameters

- -k –contextWidth → Specifies the size of the contexts analyzed. The default value is K=5.
- -a –alpha → The smoothing parameter, relevant only if a component uses adjusted probabilities with alpha. If not specified, alpha = 0.01.
- -m –mode → Strategy for selecting the next character.
    - MAX: Selects the most frequent character given the previous context.
    - PROBABILITY: Selects a character randomly based on a probability distribution derived from the frequency table.
    - PROBABILITYALPHA: Selects a character randomly based on a probability distribution adjusted by alpha.

- -sm –searchMode → Strategy for selecting a context when the current one does not exist in the frequency table.
    - RANDOM: Selects an existing context randomly.
    - CUTFIRSTCHAR: Reduces the context size by removing the first character and searches for contexts with the same suffix, selecting the most popular one.
- -s –seeding → For modules using random components, a seed can be defined. By default, no seed is applied.
- -pf –priorFix → Enables or disables the adjustment of the before ensure its length matches K. By default, this option is disabled.
- -nc –noChar → Changes how the output is returned, either character by character or as a full string.

Mandatory values are passed to the constructor, while optional parameters must be set before calling the init() function.

2. File Reading (Sequences)

When reading the file, we take the path provided by the user, search for the respective file in the resources folder, and load its entire content into memory as a string. While this approach may not be ideal for large files, given the size of the provided sequences, we deemed it appropriate for this assignment.

3. Processing Component Initialization

At this stage, we instantiate the **Processor class**, which handles data transformation from raw text into structured frequency tables. This instantiation includes all parameter values and modes, ensuring the **Processor** is fully prepared for data processing.

4. Data Transformation into Frequency Tables

The content of the file is converted into a frequency table structured as `"Map<String, Map<Character, Integer>>"` where the key is the current context (String with k-length), and the value is another Map with the key being the character that proceeds and the value the frequency that each character for that given context in the intire text.

The table is generated by scanning the sequence using a sliding window of size K, incrementing frequencies accordingly. Characters that do not appear in a specific context are omitted from the table to optimize memory efficiency.

For modes requiring probability tables adjusted with alpha, an additional table is generated during this process.

5. Character Generation Component Initialization

Like the processing component initialization, we instantiate the **Predictor** class, which manages the generation of new characters. The Predictor is fully configured based on the provided parameters and modes.

6. Final Output Generation

Finally, the generation of each character is performed by taking the prior and storing it in a variable that will hold the final response. From this evolving response, the next character is selected based on the frequency table and the chosen mode.

The selected character is then appended to the response, and the process continues by extracting the last K characters to serve as the new context. This cycle repeats until the desired output length is reached. The final response is then returned to the user as the output.

Modes

Given a context, how do we exactly select the next character?

To address this question, we implemented the following modes that determine which character to choose:

- Max: Selects the most frequent character given the previous context.
- Probability: Selects a character randomly based on a probability distribution derived from the frequency table.
- ProbabilityAlpha: Selects a character randomly based on a probability distribution adjusted by the alpha parameter.

Search Modes:

What happens if the generated characters create new contexts that have not been seen before?

This issue is particularly relevant for modes that introduce randomness since they can generate more contexts not seen in the training sequence.

To handle this, we implemented the following search modes that determine which context to use when the current one is not found in the frequency table:

- Random: Selects an existing context at random.
- CutFirstChar: Reduces the context size by removing the first character, then searches for contexts with the same suffix, selecting the most frequent one.

3.2. ***Generator Results:***

The generator was tested under different configurations in the GeneratorTest class, where we aimed to find optimal values for k, alpha, and to determine which modes and searchModes performed best. The analysis was done intuitively and qualitatively by examining the generated sequences and assessing their coherence. Only sequences 2 and 5 were analyzed in this way.

Overall Parameters for All Tests:

- Size of each generated sequence: 10000;
- The prior given to the model is sampled from the source sequence at position: 50;
- When randomness is required, the following seed is used: 85095;

Default values (if no other parameters are specified):

- Mode = "PROBABILITY";
- SearchMode = "CUTFIRSTCHAR";
- k = 6 (context size);
- $\alpha = 0.004096$;

3.2.1. ***Testing Different Values of k***

The test class starts by generating sequences to find an appropriate k. We observed the following behaviors:

- For $k < 3$:

  The generated text barely forms recognizable words, giving the impression of a different language. Example for sequence 2:

  - $k = 1 \rightarrow$ `Ebes quengointeco`
  - $k = 2 \rightarrow$ `Fambor entracegurava.`

For coding sequences, the output appears heavily corrupted. Example:

- $k = 1 \rightarrow$ `++ whtff(AX_ssisa[ize  0),  Y_1;  it_dy));`
- $k = 2 \rightarrow$ `i = push(r->pace? 1]) || (istar += brend der of (st(fp)); valargv[12;`

- For $3 \leq k < 5$: Words are formed, but sentence structure and cohesion is lost. Example:
  - $k = 3 \rightarrow$ `Era que pai Bande toda a púreas forço,`
  - $k = 4 \rightarrow$ `Pelo bom elemente noites passantas criação alegrina`

  In coding sequences, it seems as though the model recognizes parts of previously seen code but lacks proper structure. It also introduces occasional random characters. Example:
  - $k = 3 \rightarrow$ `return 0; i < vc ? 1 : node \*eigs\][0];`
  - $k = 4 \rightarrow$ `int exists[i \* alpha_min_count = pred_error("Unable \*y, dy); }`
- For $5 \leq k < 7$: Sentences start making sense but resemble a "broken dialect". Example:
  - $k = 5 \rightarrow$ `Numa altas figura templos queimando a esculpado o Capitão disse, é estarte o primeira e esforço e ardente;`
  - $k = 6 \rightarrow$ `Isto fazem que a agasalhou o ilustre e claro Tejo choro piedosa humano:`

  Coding sequences resemble normal syntax but lack logical coherence. Example:
  - $k = 5 \rightarrow$ `scanf(fp, "%.5g, %.5g", x[i];\n char \*beads = (double X[][5] = {3, -1, -3, 0}};`
  - $k = 6 \rightarrow$ `val = valueOcurranceCArray(CArray(CArray \*aarray, min, i);`
- For $k \geq 7$: The generated sequences appear to be direct copies of fragments from the source text. Example:
  - $k = 7 \rightarrow$ `Como dama que falta minha gentil donzela`
  - $k = 8 \rightarrow$ `A calma, ao frio, ao qual os seus, que do Tejo e do Mondego,`
  - $k = 9 \rightarrow$ `Veja agora o Império, e as terras se passasse d'Alentejo,`
  - $k = 10 \rightarrow$ `Mas o Gama, que também com falsa conta e nua,\nÀ nobre terra alheia chamam sua.`

In coding sequences, even at higher k values, the generated output is inconsistent with programming logic. Syntax errors are mostly resolved, but logical errors persist. Example:

- k = 7 → `if (sum == N ? 0 : t2->left);\n return EXIT_SUCCESS;`
- k = 8 → `for (i = 0; i < graph->edge[4].dest = 1;`
- k = 9 → `void duplicateZeros(int \*arr, int size = 6;`
- k = 10 → `unsigned long i = 1, num_digits > 0; num_digits \* sizeof(double \*)malloc(sizeof(CArray));`

3.2.2. ***Testing Different Values of α***

To find a fitting alpha, we applied the same approach while keeping k constant at 6 and using the PROBABILITYALPHA mode. The alpha values were varied for the values 0.004096, 0.01024, 0.0256, 0.064, 0.16, and 0.4.

The generated text appeared corrupted, with lower alphas showing less severe corruption, allowing for some words or even occasional sentences to be deciphered. Higher alphas produced completely unrecognizable sequences.

Examples:

- $\alpha = 0.004096$ → `Düa estava aPara lhe a garHierosólima celebrado, às terra os hu:--QLusitano,`
- $\alpha = 0.01024$ → `Se não pesaôídos;ço cr[s]ude cz pesa?Ó PTob35`
- $\alpha = 0.0256$ → `JMouro ao Ba;Que, ilfÚngaro eLusitana., reino deiJáá)Men Castela,`
- $\alpha = 0.064$ → `QuÁtéz:Mouros caÁídosçÚ?úBemLu7É?31ÁDÁm nãém aoDe99gos;`
- $\alpha = 0.16$ → `á sSó infForçês,1['oãMòZQu95õeGaCOriGófúComõço c9:`
- $\alpha = 0.4$ → `AOZè-üF\]áÀÀBEÁ85ÁÔõezaÉOÍá sabi3ü`

In coding sequences, every α value had a negative effect, completely destroying any attempt at meaningful code generation. Examples:

- $\alpha = 0.004096$ → `while\t\tdelta: %.4g millisecond = temp->next && head = (struct ListNode *)malloc(sizeof(struct ListNode *root = parent->lJewelsrc = 0;`
- $\alpha = 0.0256$ → `#include \<GL/---------- %d f? n? 0"+)`
- $\alpha = 0.4$ → `printf("1KC;V;eturn kySf("Te[G?while (td#0; i < n;I];`

3.2.3. ***Comparison of Character Selection Modes***

To compare different modes of character selection, we tested each mode while keeping k and α fixed at 0.004096 and 6, respectively. The tested modes were: PROBABILITY, PROBABILITYALPHA, and MAX. We found that:

- PROBABILITY Mode produced the best results for both text and code generation.
- MAX Mode got stuck in cyclical generation, repeating the same fragments in both cases.
- PROBABILITYALPHA Mode generated sequences that seemed like distorted versions of a meaningful text, as if passed through a filter that randomly replaced some characters. Additionally, the frequency of uncommon characters was noticeably higher.

Examples:

- PROBABILITY Mode:
  - `Como Nero, que a mais se os passavam,.`
  - `printf("Found at: %d\n", shift = 0;`
- PROBABILITYALPHA Mode:
  - `Pois que tor constante, e dá-lheHiero\[panhadores, que desce;`
  - `int pindex = 0, hig| end_num < 0Q)\_t start_time = (t2 - t1) / CLOCKS_PER_SEC);`
- MAX Mode:
  - `99\nO mesmo ofício,\nOs vosso.\n`
  - `{\n int y, int z)\n`

3.2.4. Comparison of Search Modes for Missing Context

The comparison of SearchModes for contexts (when the full context is missing) was conducted similarly to the previous tests. We used the same fixed values as described earlier and set the mode to PROBABILITY. The available SearchModes were CUTFIRSTCHAR and RANDOM. We found that:

- RANDOM Search Mode was a temporary and simplistic approach to handling missing context. However, it failed completely, producing unrecognizable results. The problem arises because when the first few contexts are not available in the countTable. This leads to a cycle where each generated character comes

from a random, unseen context, which in turn generates another invalid context, compounding the issue.

- CUTFIRSTCHAR Search Mode was the only viable search mode we implemented. However, it is not perfect. This search method is computationally expensive, especially in settings with high randomness and for large training sequences. Although it could be precomputed, it would lead to an increase in the memory consumption of the model.

4. Some examples for generator command-line calls:

- Basic call (file under resources):

```
$ java -jar target/tai-1.0-SNAPSHOT.jar g -f sequence1.txt -rl 500 -p "ATGAA"
```

- Recommended call:

```
$ java -jar target/tai-1.0-SNAPSHOT.jar g -f sequence1.txt -rl 500 -p "ATGAATGAAT" -pf -k 5 -s 85095
```

- Extensive calls:

```
$ java -jar target/tai-1.0-SNAPSHOT.jar g -f sequence1.txt -rl 500 -p "ATGAATGAAT" -pf -k 5 -s 85095 -m PROBABILITYALPHA -a 0.01 -nc
$ java -jar target/tai-1.0-SNAPSHOT.jar g -f sequence1.txt -rl 500 -p "ATGAATGAAT" -pf -k 5 -s 85095 -m MAX -sm RANDOM -nc
```

4.1. ***Conclusion:***

For natural language generation, the ideal k should be around 5–6 to balance coherence and novelty, while for code generation, k should be larger (typically $k > 10$) to minimize logic errors, though other approaches may yield better results.

Higher alphas break the generation cycle too often, preventing any coherence in the output. Lower alphas (below 0.064), while not completely disrupting the generation, introduce random characters and appear to yield worse results than using the standard probability-based mode without alpha adjustments.

PROBABILITY Mode is the superior mode, as it generates the most coherent sequences. MAX Mode can escape cyclical generation by increasing k, but this causes it to start reciting the source text, effectively becoming an overcomplicated copy-paste. PROBABILITYALPHA Mode is a worse version of the PROBABILITY mode. While it

doesn't get stuck in a cyclical generation, it introduces random uncommon characters in inconvenient places, disrupting the sequence. RANDOM Search Mode is not a valid solution, as it leads to chaotic and meaningless outputs. CUTFIRSTCHAR Search Mode is functional but comes with a high computational cost, particularly when randomness is increased.

5. Future Improvements

5.1. Better Hashing

In practice, we found that changing the hash function in the FCM program greatly affected performance, so exploring more hash algorithms could speed up the program. In this case, the 2 *Map<String,Int>* structure could beat out our current structure.

5.2. Combining multiple context sizes

We could use a combination of several models with different context sizes and use weights to combine them and obtain a new probability as long as $\sum_k w_{k,n} = 1$. This could be done, for example, by using a neural network to find the weights or by a probabilistic approach where we consider the probability that model k has generated the sequence until that point and use that probability to assign weights.

5.3. Automated analysis of the generator

For future work, an in-depth analysis using automated processing of the generated sequences using the generator and quantitative comparisons is recommended. This could be done by using measures like perplexity, overlap, N-gram Matching, and cosine similarity and even critiques using LLMs.

5.4. Potential Improvements for PROBABILITYALPHA Mode:

If the generation of very uncommon characters could be eliminated—either by removing characters outside the normal alphabet, by adjusting alpha to match the general character distribution, or by disallowing characters below a certain threshold of frequency. This mode might become more useful.

5.5. Alternatives to CUTFIRSTCHAR Mode:

Other alternatives to this mode could be explored since in this mode, when the full context is not present, the mode will ignore further characters, and may not be desired, some other

approaches can perform better, like Backoff Strategy, Edit Distance Matching or N-gram Approximation.

5.6. Alternative Smoothing Methods for Improved Entropy Estimation

Our current approach uses additive smoothing, but more advanced smoothing techniques could enhance entropy estimation accuracy, particularly for handling rare events. Some promising alternatives include:

- Kneser-Ney Smoothing: Unlike simple additive smoothing, Kneser-Ney adjusts probabilities based on the number of unique contexts in which a word appears, improving predictions for rare words. Implementing this would require modifying the frequency tables to account for word diversity rather than raw counts.
- Good-Turing Smoothing: This method estimates the probability of unseen events by redistributing probabilities from low-frequency occurrences. Integrating Good-Turing would involve computing adjusted frequency estimates and applying them dynamically during entropy calculation.