Assignment 1 - Copy models for data compression

Algorithmic Information Theory (2022/23)
Universidade de Aveiro

Martinho Tavares, 98262, martinho.tavares@ua.pt
Nuno Cunha, 98124, nunocunha@ua.pt
Pedro Lima, 97860, p.lima@ua.pt

March 29, 2023

1 Introduction

One of the goals of this assignment is to implement a copy model for data compression, which is obtained through the exploration of self similarities. A copy model is only one of the ways to address the problem, and its idea is that certain data sources can be viewed as a sequence of repetitive patterns, where parts of the data have been replicated in the past, although some modifications are possible. The model consists in predicting the next symbol based on previous ones observed in the past, by keeping a pointer referring to the symbol being copied, as well as other information.

The second goal of the assignment is to implement a text generator based on the copy model, which is a way to generate new text based on a given one. The text generator receives a text as input to train the model, and then follows a similar approach to the one used in the copy model, but instead of predicting the next symbol, it uses the probability distribution to generate a random symbol based on these probabilities.

In this report we will first present how the work was organized, in the 2 section. Then, we will present the copy model in 3, how we implemented it and the results obtained, which we will compare by calculating the entropy of different text examples, like `chry.txt` given by the teachers, with different parameters we defined for the model. The next section is 4, dedicated to the text generator, where we will present the implementation and the results obtained, using different texts as input for training of the generator, as well as different starting points for the generation of text. Finally, we will conclude the report in the 5 section.

2 Work organization

To organize the work we decide to use a object oriented approach, this way it was easier to implement different parameters to test the copy model and the text generator.

First for the copy model we created a file named `main.cpp`, where we instantiate the object `CopyModel`, which is the class that implements the copy model. The `CopyModel` class has a constructor that receives the parameters that we want to test, and with these parameters we can instantiate different objects of certain classes, and test them with different files. The more important classes are the abstract classes: `CopyPointerManager`, `CopyPointerThreshold` and `BaseDistribution`. By using these base classes we could implement different strategies for the copy model, by creating child classes of these base classes, and then instantiate the `CopyModel` object with the child classes as parameters. This proved to be a very useful approach, since we could test different strategies for the copy model, and compare the results obtained to see which strategy, or combination of strategy/parameters, was the best.

For the text generator all the code is in the `cpm_gen.cpp` file, due to its more simple structure. The generator also implements some parameters and uses a similar strategy of reading patterns as the copy model.

3 Copy model

The copy model behaves in a straightforward way. At each symbol in the file, the program verifies if the pattern of k symbols, with the current symbol being the last, has already occurred in the past. That is, at position x_n, the pattern within positions x_{n-k+1} and x_n is evaluated to see if it appeared in the past.

To do that, each of these k-sized patterns at every position is saved in a hash table built incrementally while the file is being processed. The keys are the $k$mers/patterns, and the values are arrays of positions in the file where those $k$mers occurred in the past, which we call pointers. When we need to choose a pattern in the past, we look through the array of pointers for that pattern for a specific occurrence of that pattern.

Whenever the pattern is still not present in the hash table, we try to randomly guess which symbol should occur next since we can't perform a copy from the past. In these cases, the reported probabilities follow a fixed default distribution that is set beforehand.

When predictions can be made, a strategy is followed to choose the appropriate pointer for the current pattern. After a pointer is chosen, it is fixed and predictions are made sequentially from that point on. That is, if at position x_n we have the copy pointer x_m, then we will predict the character at x_{n+1} to be equal to the character at x_{m+1}, the character at x_{n+2} to be equal to the character at x_{m+2}, and so on. The predictions aren't completely certain; we assume a probability of making a correct prediction, which varies depending on how well the prediction is going.

This process is repeated until a threshold is met, which is based on the probability of making a correct prediction at the current position. When the threshold is met, we stop predicting, and keep evaluating the file's content as normal. If the current $k$mer has already occurred in the past, then we attempt a new chain of predictions. Otherwise, we perform guesses.

We need to be careful regarding the beginning of the file. For the first $k-1$ symbols, it is not possible to generate a $k$mer, but we should still be able to perform predictions starting on these $k-1$ symbols. We followed a simple strategy, which involves extending the file content backwards, generating a past of size k. The generated past is simple, it is composed of k repetitions of a single symbol of the alphabet. This symbol is the most frequent of the entire alphabet.

In order to evaluate whether the copy model can provide acceptable results, we can use a baseline below which we expect the model to report the file's entropy. We decided to use, as a baseline, the entropy considering each symbol's relative frequency in the entire file in bits, which is given by:

$$H(X) = -\sum_{x \in X} p(x) \log_2 p(x)$$

With this value in mind, we evaluated the model as a whole with different values for its parameters, on different files. The files chosen for testing are present in the repository[1], and they have the following baselines:

- `chry.txt`: 1.9652 bits
- `othello.txt`: 4.44629 bits

Throughout this section, the different program parameters are detailed, and their effect on the model's performance, both in terms of execution time and quality of the results, is studied. To analyze the effect of a parameter l, the copy model is run with different values for l while keeping all other parameters with the default values.

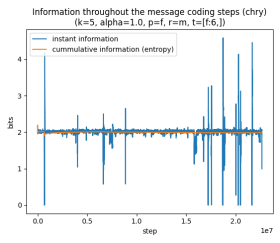
For the results regarding the information content of the symbols throughout the `chry.txt` and `othello.txt` files, the data was transformed with a low-pass filter using a moving average with window size of 10000 and 100 samples respectively, since the data exhibited a very large frequency. Additionally, the file `othello.txt` was altered to not include newlines or commas, which were respectively replaced by spaces and semicolons. These were complicating posterior data analysis due to how the model's results are saved to disk, which use the verbose machine mode described in section 3.6.

3.1 Pattern size

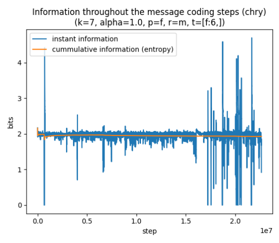
When choosing a pointer in the past from which to start copying, we need to look for an occurrence of the same k-sized pattern as the one we are currently on. Thus, k is one of the parameters that affects program performance, where k is a positive integer.

On one hand, a lower value of k means there will be more occurrences of each pattern, since there will be lesser possible k-sized patterns, which will result in

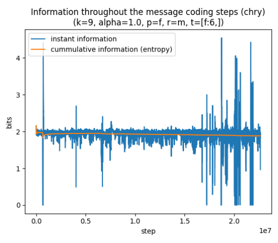
[1] `https://github.com/NMPC27/TAI-G7-Lab1`



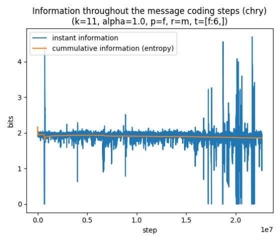
(a) $k = 5$



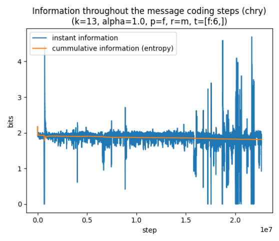
(b) $k = 7$



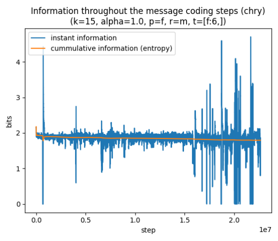
(c) $k = 9$



(d) $k = 11$



(e) $k = 13$



(f) $k = 15$

Figure 1: Information of the current symbol and entropy as the file is processed, varying the k parameter on “chry.txt”.

more oportunities to copy from the past. On the other, a larger k will cause less occurrences for each pattern, which results in more scarce predictions, and thus more overall guesses.

For a given k, the number of possible patterns M is equal to the number of permutations of the symbols of alphabet A: $M = |A|^k$. The results on changing this value will be studied taking this exponential growth into account.

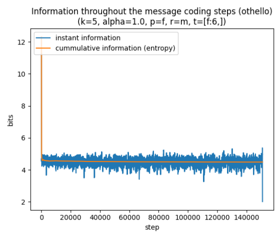
3.1.1 Results

To compare the effects of changing k, we ran the program with k from 5 to 12 while keeping all other parameters with the default values. Some of the results on the information of the current symbol in the message and the reported file entropy at each step in the coding process are present in figure 1, for file `chry.txt`. For file `othello.txt`, the results are in figure 2.

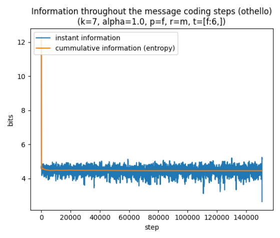
From the results obtained we can observe that the instant information of the current symbol has smaller variations when the pattern size is smaller, which is specially noticeable in the `chry.txt` file when $k = 5$, and has bigger variations when the pattern size is larger.

Other than these variations, the graphs are rather similar in behaviour, specially the big decrease in entropy at the beggining of each run for both files.

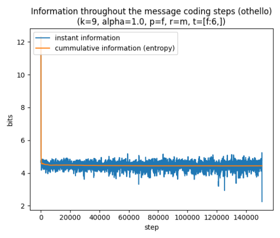
Another difference between different k values is the execution time, which is shown in tables 1 and 2. By decreasing the pattern size, the execution time



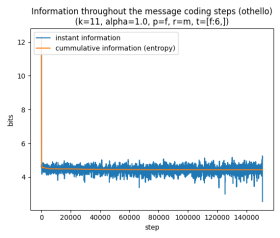
(a) $k = 5$



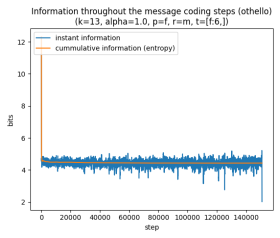
(b) $k = 7$



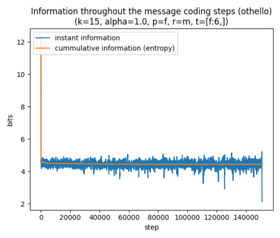
(c) $k = 9$



(d) $k = 11$



(e) $k = 13$



(f) $k = 15$

Figure 2: Information of the current symbol and entropy as the file is processed, varying the k parameter on "othello.txt".

k	α	**Dist.**	**Manager**	**Thr.**	**Entropy**	**Exec. time (s)**
5	1.0	f	m	f:6	1.99252	14201.6
6	1.0	f	m	f:6	1.97014	4314.0
7	1.0	f	m	f:6	1.94323	1427.55
8	1.0	f	m	f:6	1.91613	654.994
9	1.0	f	m	f:6	1.9051	366.689
10	1.0	f	m	f:6	1.88932	308.409
11	1.0	f	m	f:6	1.88083	275.226
12	1.0	f	m	f:6	1.86538	274.427
13	1.0	f	m	f:6	1.85025	283.567
14	1.0	f	m	f:6	1.8441	279.617
15	1.0	f	m	f:6	1.83763	321.614

Table 1: Entropy and execution time when varying k, on "chry.txt".

k	α	**Dist.**	**Manager**	**Thr.**	**Entropy**	**Exec. time (s)**
4	1.0	f	m	f:6	4.5111	11.3304
5	1.0	f	m	f:6	4.48476	9.84814
6	1.0	f	m	f:6	4.46478	9.68529
7	1.0	f	m	f:6	4.45224	9.66324
8	1.0	f	m	f:6	4.44043	9.02154
9	1.0	f	m	f:6	4.43762	8.99458
10	1.0	f	m	f:6	4.43684	8.16078
11	1.0	f	m	f:6	4.43266	8.84613
12	1.0	f	m	f:6	4.42929	8.15512
13	1.0	f	m	f:6	4.42925	9.8033
14	1.0	f	m	f:6	4.42978	7.95666
15	1.0	f	m	f:6	4.43228	7.33517

Table 2: Entropy and execution time when varying k, on “othello.txt”.

generally increases, which is expected since the number of ocurrences of each pattern increases making the values of our hash table, the arrays of pointers, much bigger in size. This is due to the fact that the copy pointer repositioning strategy used, **most common prediction** detailed in section 3.4, is has linear time complexity. With slightly bigger values of k the execution time is smaller, and for these values the entropy results are also better, if only very slightly.

3.2 Smoothing parameter alpha

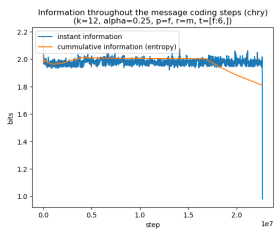
At each coding step, the probability of correctly predicting the current character is given by the formula:

$$P(\text{hit}) = \frac{N_h + \alpha}{N_h + N_f + 2\alpha}$$

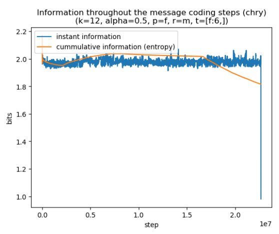
where N_h and N_f are respectively the number of hits (correct predictions) and misses (wrong predictions), and α is the smoothing parameter. The smoothing parameter is used to avoid the model having a probability of 0 when there are no hits or misses, which will happen when no prediction has been made yet, and also avoid assuming with complete certainty (probability of 1) that the prediction will be correct when no fails were made yet, which will result in infinite information in case the prediction fails.

If α is lower, we expect the reported probabilites to be more “certain” with a greater possibility of reaching high values. If α is greater, the smoothing effect should be more pronounced, producing probability values that hardly increase, which should result in relatively less information when predictions fail but also lesser returns on the correct predictions.

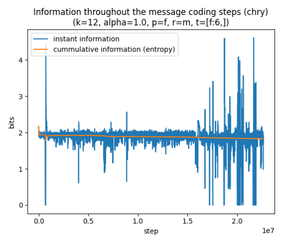
This trade-off between α and the reported information, especially regarding the information spikes that may occur, will be studied in the results.



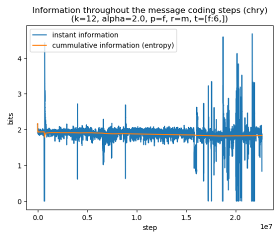
(a) $\alpha = 0.25$



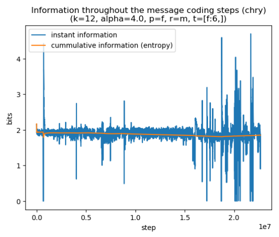
(b) $\alpha = 0.5$



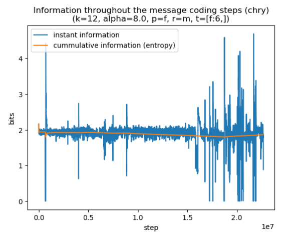
(c) $\alpha = 1.0$



(d) $\alpha = 2.0$



(e) $\alpha = 4.0$



(f) $\alpha = 8.0$

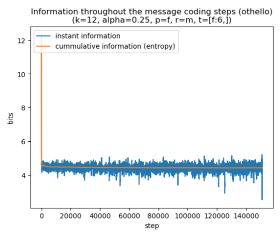
Figure 3: Information of the current symbol and entropy as the file is processed, varying the α parameter on "chry.txt".

3.2.1 Results

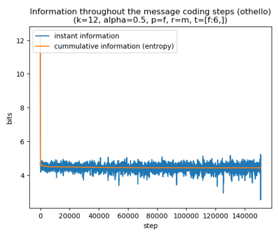
Runs were performed with $\alpha \in 0.25, 0.5, 1.0, 2.0, 4.0, 8.0$, where the next value is double the previous value. The results on the information of the current symbol in the message and the reported file entropy at each step in the coding process are present in figure 3, for file `chry.txt`. For file `othello.txt`, the results are in figure 4.

From the graphs, we can notice that for $\alpha < 1.0$ in the `chry.txt` file the behavior of the model is drastically different, in subfigures 3a and 3b. When $\alpha = 0.5$ the reported entropy in the middle of the file is considerably greater than when $\alpha = 0.25$, although the final reported entropy is the same. Additionally, the reported information at the last steps is extremely low when compared with the symbol information in previous steps, which means a very good prediction was being performed. However, we couldn't draw an explanation for why this is the case, since in the implementation the `double` storage is used for α in the whole program and the prediction probability formula doesn't hint at a major difference in reported values when $\alpha < 1.0$.

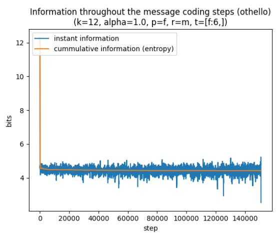
When $\alpha >= 1.0$ in `chry.txt`, the difference between runs can be seen on the reported information, especially in the middle of the processing, where the reported symbol information is lower when predicting if α is closer to 1. This makes sense, since the prediction probability is less smoothed and thus is allowed to reach higher values, which in turn results in reporting less information when



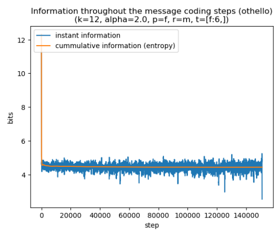
(a) $\alpha = 0.25$



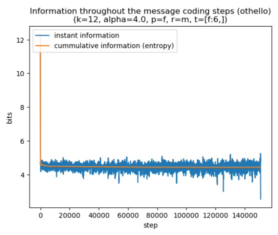
(b) $\alpha = 0.5$



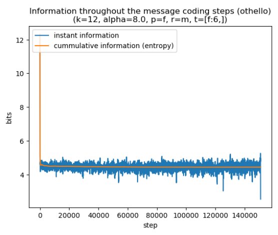
(c) $\alpha = 1.0$



(d) $\alpha = 2.0$



(e) $\alpha = 4.0$



(f) $\alpha = 8.0$

Figure 4: Information of the current symbol and entropy as the file is processed, varying the α parameter on "othello.txt".

the predictions are correct. However, it is also expected that as α increases, less information is reported when the prediction fails, but this is less clearly seen in the plots.

These latter conclusions can be drawn on the results of the `othello.txt` file too for all α. The larger the α, the "smoother" the values of the current symbol information which tend to converge towards the entropy line.

Another observation for `chry.txt` is that, as α increases, the entropy within steps 1.5×10^7 and 2.0×10^7 decreases, forming a larger valley. One possible interpretation is that in this part of the file the model was more able to perform correct predictions, which resulted in more pronounced misses, as exhibited by the larger information spikes towards the end. As such, the prediction fails would report lesser information content with larger α, which helped the model capitalize on the correct predictions better in this specific case.

Tables 3 and 4 show the reported entropies and execution times when varying alpha for both files. It can be seen that the value of α has to be chosen carefully, as both large and small values can compromise entropy, and so it's a parameter that should follow careful adjustments for better results. In terms of execution time, we can note that since α can heavily influence the reported prediction probabilities, and therefore trigger the pointer reposition thresholds at different times, the execution time can vary wildly depending on how well the predictions go through the file, since the program behavior is different depending on the prediction performance.

k	α	**Dist.**	**Manager**	**Thr.**	**Entropy**	**Exec. time (s)**
12	0.25	f	m	f:6	1.97603	317.792
12	0.5	f	m	f:6	1.97626	297.333
12	1.0	f	m	f:6	1.86538	771.466
12	2.0	f	m	f:6	1.86766	424.611
12	4.0	f	m	f:6	1.88524	477.182
12	8.0	f	m	f:6	1.8977	451.744

Table 3: Entropy and execution time when varying α, on "chry.txt".

k	α	**Dist.**	**Manager**	**Thr.**	**Entropy**	**Exec. time (s)**
12	0.25	f	m	f:6	4.43467	14.8118
12	0.5	f	m	f:6	4.43027	17.5231
12	1.0	f	m	f:6	4.42929	21.6569
12	2.0	f	m	f:6	4.43126	22.5453
12	4.0	f	m	f:6	4.43468	19.4892
12	8.0	f	m	f:6	4.43859	18.5035

Table 4: Entropy and execution time when varying α, on "othello.txt".

For instance, if the pointer has to be repositioned when using the **most common prediction** reposition strategy, which is described in section 3.4, then a relatively expensive algorithm has to be run, and so if predictions are constantly failing it's expected that the execution time worsens.

3.3 Base probability distribution

For each symbol in the file, a probability distribution has to be generated according to the prediction probability. In case a given prediction fails, we need to distribute the complement of the probability of a correct prediction $1 - P(\text{hit})$ among the other symbols of the alphabet, except the one that we predicted would occur (which has probability $P(\text{hit})$).

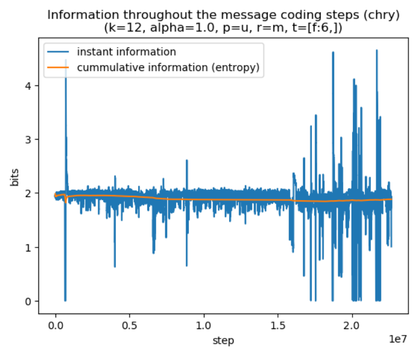
The choice of distribution also influences the guesses that are made. When a prediction can't be made, the model assumes a distribution for the entirety of the alphabet, which is the same probability distribution as the use used when distributing the complement of the prediction probability.

We developed two simple approaches to distributing the probability: a uniform distribution and a frequency distribution.

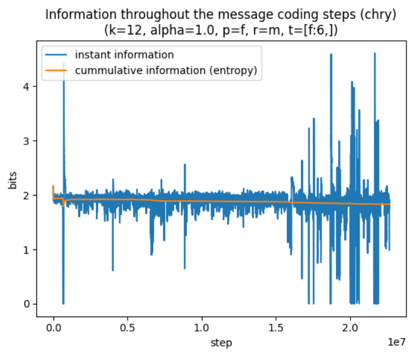
The uniform distribution assigns to each symbol a probability equal to:

$$\frac{1 - P(\text{hit})}{|A| - 1}$$

On the other hand, the frequency distribution assigns to each symbol $a \in A$



(a) Uniform distribution.



(b) Frequency distribution.

Figure 5: Information of the current symbol and entropy as the file is processed, varying the base distribution on "chry.txt".

a probability equal to:

$$(1 - P(\text{hit})) \times p(a)$$

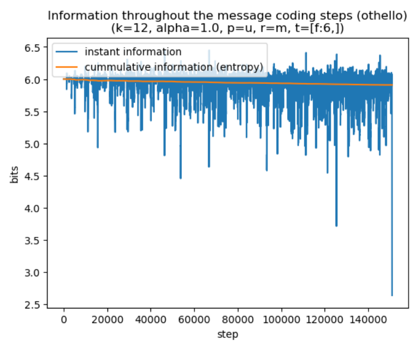
where $p(a)$ is the probability of a in the whole file.

Both distributions are compared especially in their effect on the overall reported information, when k is large and thus more guesses are made, as well as when α is large and the prediction fails should report. We assume that, since the frequency distribution models the overall distribution of the symbols, it should provide better results. We should note, however, that the file's content may have locality, i.e. select portions in the file where the symbol distribution is skewed. Therefore, we don't disregard the possibility that the assumption of the distribution being maintained throughout the file can hurt the model performance as well.

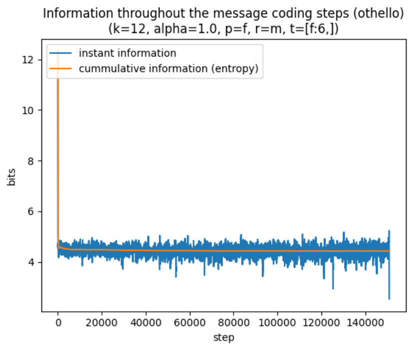
3.3.1 Results

Figures 5 and 6 show the symbol information and entropy throughout the processing of both test files, when using either the uniform or frequency distribution.

For `chry.txt`, the effect of the distributions is minimal, but noticeable. More specifically, the frequency distribution exhibited a lower and more stable entropy value throughout the file. In terms of symbol information, the runs with the uniform distribution presented very similar values of information when predictions fail (data points above 2 bits), while for those with the frequency distribution the information values on failed predictions are more irregular throughout the file. This is due to the fact that the frequency distribution assigns for each symbol of the alphabet a different probability by default, relative to the symbol's frequency in the whole file, and so different values are provided throughout the file which model the symbol's appearance in the file's contents.



(a) Uniform distribution.



(b) Frequency distribution.

Figure 6: Information of the current symbol and entropy as the file is processed, varying the base distribution on “othello.txt”.

k	α	**Dist.**	**Manager**	**Thr.**	**Entropy**	**Exec. time (s)**
12	1.0	f	m	f:6	1.86538	771.466
12	1.0	u	m	f:6	1.88618	674.502

Table 5: Entropy and execution time when varying the base distribution, on “chry.txt”.

For `othello.txt`, however, the frequency distribution provided much better entropy. The effect of this distribution is more pronounced in `othello.txt` since the size of the alphabet is larger and the probability of occurrence between symbols is greatly skewed, as in the english alphabet there are characters that appear much more often than others. The uniform distribution didn’t take this into account, and thus reported larger values for the entropy. One unexplained detail, however, is how for the run using the frequency distribution the reported entropy in the first few steps is abnormally high.

As such, the previously mentioned possibility that the frequency distribution may also hurt model performance was not found in these results.

Tables 5 and 6 present the reported entropies and execution times. For both files, the uniform distribution provided worse values, which follows what was expected. However, for `chry.txt`, using the frequency distribution took noticeably longer, when such a difference wasn’t noticed for `othello.txt`. We believe this is due to to a timing inconsistency, since the distribution is generated only once after the copy model’s first pass, and is simply reused from that point on, involving no other complex operations.

k	α	**Dist.**	**Manager**	**Thr.**	**Entropy**	**Exec. time (s)**
12	1.0	f	m	f:6	4.42929	21.6569
12	1.0	u	m	f:6	5.90869	20.7499

Table 6: Entropy and execution time when varying the base distribution, on "othello.txt".

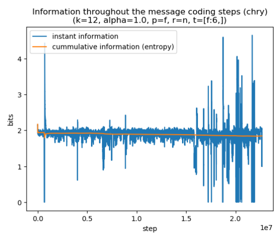
3.4 Copy pointer repositioning strategy

Different strategies can be followed when choosing a copy pointer from which to start copying. We developed 4 approaches, with the first 2 being fairly simplistic:

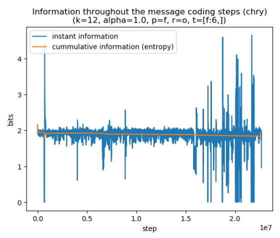
- **Oldest pointer first**: the chosen copy pointer is the oldest possible. If the current copy pointer is repositioned because the threshold was met, then that pointer will be completely discarded and the next oldest pointer is used. This is the most simple approach, which assumes similar patterns are very far in the file;
- **Newest pointer first**: the chosen copy pointer is always the newest, which assumes that similar patterns are very close in the file;
- **Most common prediction**: when a pointer has to be chosen, the predictions of all registered pointers are evaluated, and the pointer (or one of them) which presented longest and most common prediction is chosen. A more detailed description of this strategy is presented below.
- **Most common prediction bounded**: the approach is exactly the same as the **most common prediction**, with the only difference being that the registered pointers for each pattern are stored in a circular array with limited size.

The **most common prediction** strategy chooses the copy pointer in successive iterations. Initially, the next immediate character is evaluated for every registered pointer. The pointers are grouped according to that character that they predict, and the largest group is chosen, i.e. the pointers that reported the most common prediction are chosen. Afterwards, the second next character is evaluated for every pointer in that group, and then the pointers are grouped in the same manner again, with the next group being the largest at this stage. This process is repeated until all pointers at the current group report different predictions, at which point the newest pointer is chosen.

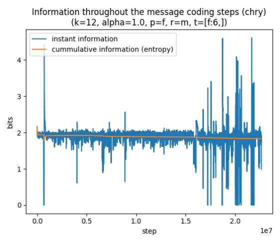
In order to obtain the most common prediction at each iteration, the Boyer-Moore majority vote algorithm was used [1]. This is a very space-efficient algorithm (complexity $O(1)$) that guarantees the majority value will be returned. However, if a majority value doesn't actually exist then the algorithm may erroneously report a value as the majority, if we only do one pass. Since in this case we don't have a specific disambiguation strategy when the pointers don't agree, the choice is arbitrary and thus this is not a problem.



(a) Oldest pointer first.

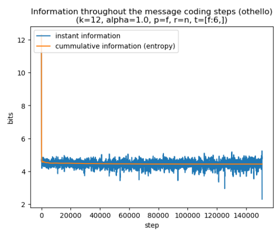


(b) Newest pointer first.

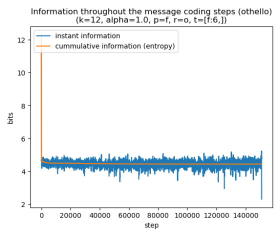


(c) Most common prediction.

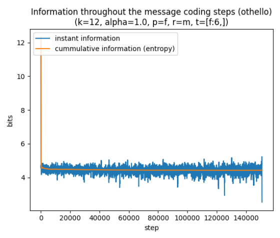
Figure 7: Information of the current symbol and entropy as the file is processed, varying the pointer reposition strategy on "chry.txt".



(a) Oldest pointer first.



(b) Newest pointer first.



(c) Most common prediction.

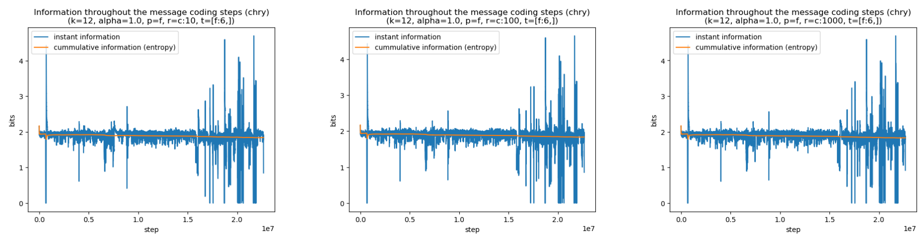
Figure 8: Information of the current symbol and entropy as the file is processed, varying the pointer reposition strategy on "othello.txt".

The bounded version of **most common prediction** uses a circular array for the set of registered pointers. The idea is for the array of registered pointers to contain only the N most recent pointers, with N being the array size. This can not only take into account the locality that the file's content may have, i.e. the repeated patterns being close to each other, but it also bounds the array size to a fixed value, which helps in performance since the reposition algorithm previously described has linear time complexity $O(N)$, when the other strategies have only $O(1)$ complexity.

3.4.1 Results

Figures 7 and 8 show the results for the symbol information and entropy throughout both files in the first three modes of execution.

There is almost no notable difference between the pointer reposition strategies for both files, specially for `othello.txt`. In the `chry.txt` file, for the **oldest** and **newest pointer first** strategies a very good copy is performed at



(a) Size 10. (b) Size 100. (c) Size 1000.

Figure 9: Information of the current symbol and entropy as the file is processed, varying the size of the circular array on the BMCP strategy on “othello.txt”.

around 1.6×10^7 steps, as evidenced by the low information per symbol that was reported, which was consistently below 2 bits. On the other hand, the **most common prediction** strategy, even though evidenced a good copy as well, the reported information values were less consistent during the copy and were closer to the desired limit of 2 bits. For the `othello.txt` file, there is no noteworthy difference between modes.

Figure 9 shows the results for the bounded version of **most common prediction** with a circular array (BMCP), for different values of the array size. These results are for the `chry.txt` file only, since for `othello.txt` there was no discernible difference between modes and so we deemed it was not worth it to present.

When the array size is 10, in figure 9a, the symbol information is consistently high at around 0.75×10^7 steps, with no good predictions, which was not verified for the other modes. Other than that, the good copy that was previously seen at around 1.6×10^7 can be verified for the modes with array size 10 and 100, but more inconsistent correct prediction values are seen for array size 1000. Overall, the BMCP strategy didn't exhibit considerable gains over the reported entropy.

The considerable benefit of the BMCP strategy comes instead in terms of execution time. Tables 7 and 8 show the entropy and execution time of running on all modes for both files. For the same reported entropy values, BMCP is considerably faster than the base **most common prediction** strategy on `chry.txt`. On the other hand, differences can't be noticed for `othello.txt`, since the pattern size k is too large considering the large alphabet size and low amount of characters in the file. This makes the bounding of the pointer array much less impactful, due to the fact that seeing a given $k$mer again is less probable as there are much more possible permutations of k-sized patterns (even though it's limited since it's english text).

3.5 Copy pointer reposition threshold

A copy pointer is used until the predictions are deemed to not be worthwhile. After that point, the copy pointer is repositioned taking the current pattern

k	α	**Dist.**	**Manager**	**Thr.**	**Entropy**	**Exec. time (s)**
12	1.0	f	o	f:6	1.86847	581.959
12	1.0	f	n	f:6	1.86829	596.317
12	1.0	f	m	f:6	1.86538	771.466
12	1.0	f	c:10	f:6	1.86738	561.659
12	1.0	f	c:100	f:6	1.86233	501.48
12	1.0	f	c:1000	f:6	1.86627	507.064

Table 7: Entropy and execution time when varying the pointer reposition strategy, on "chry.txt".

k	α	**Dist.**	**Manager**	**Thr.**	**Entropy**	**Exec. time (s)**
12	1.0	f	o	f:6	4.42905	22.0874
12	1.0	f	n	f:6	4.42996	22.4216
12	1.0	f	m	f:6	4.42929	21.6569
12	1.0	f	c:10	f:6	4.42944	22.4959
12	1.0	f	c:100	f:6	4.42929	25.9895
12	1.0	f	c:1000	f:6	4.42929	24.5401

Table 8: Entropy and execution time when varying the pointer reposition strategy, on "chry.txt".

into account. The decision of when to reposition the copy pointer can be done using different approaches.

We developed 3 strategies, all of which only factor in the prediction probability over time:

- **Static**: a static probability threshold value. If the prediction probability falls below this value, then the pointer is repositioned.
- **Successive fails**: if the number of successive prediction misses is larger than a given value, then the pointer is repositioned. In this case, the "successive fails" are evaluated using a non-negative counter of misses that is updated as predictions are made; whenever there is a miss, the counter increases, and decreases otherwise.
- **Rate of change**: if the absolute difference between the current prediction probability and the previous prediction probability is below a threshold then the pointer is repositioned.

Each strategy has a threshold value associated with them, and they are passed as parameters along with the specified strategy. These thresholds can also be combined together, indicating whether the pointer should be repositioned when at least one of the strategies reports that their threshold was surpassed.

k	α	**Dist.**	**Manager**	**Thr.**	**Entropy**	**Exec. time (s)**
12	1.0	f	m	f:3	1.86605	243.061
12	1.0	f	m	f:6	1.86538	223.103
12	1.0	f	m	f:9	1.86376	221.524
12	1.0	f	m	n:0.3	1.9815	244.255
12	1.0	f	m	n:0.4	1.97406	304.968
12	1.0	f	m	n:0.5	1.92033	313.235
12	1.0	f	m	c:0.01	1.97481	222.068
12	1.0	f	m	c:0.05	1.97643	226.548
12	1.0	f	m	c:0.1	1.97625	218.448

Table 9: Entropy and execution time when varying the threshold, on "chry.txt".

3.5.1 Results

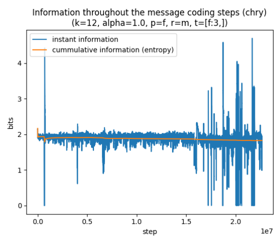
The results were initially evaluated for each strategy in isolation. The combinations of thresholds were not tested due to time constraints.

Figures 10 and 11 show the results for the symbol information and entropy throughout both files when varying the thresholds.

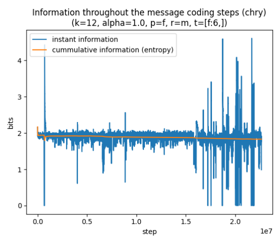
Regaring `chry.txt`, the threshold on successive fails didn't show significant differences among each other when varying the number of fails. For the threshold on static probability, as the probability threshold lowers the predictions get weaker, which suggests the model is sticking too long with a bad copy pointer. In fact, we can notice on the last 3 spikes of figure 10d that the symbol information stays above 2 bits for a considerable amount of steps, suggesting the copy pointer should have been repositioned much earlier as the predictions are failing too much. The threshold based on the rate of change presented the odd behavior that was verified in section 3.2 when $\alpha < 1.0$. Additionally, as the value of this threshold increased, the reported entropy in the middle of the file's content considerably worsens.

For the file `othello.txt`, however, there were no noticeable differences between threshold strategies.

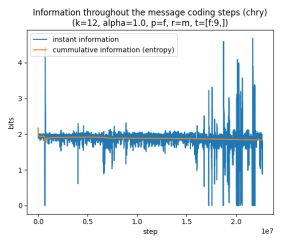
Tables 9 and 10 show the entropy and execution time of running with different thresholds for both files. For `chry.txt`, the threshold on the number of successive fails presented the best entropy values, while for `othello.txt` it was the static probability. In terms of execution time, there was a considerable difference on `chry.txt` for the different values of the static probability threshold, with higher values exhibiting a larger execution time. This is understandable, again, due to the linear time complexity of the pointer repositioning strategy used, and since this threshold is harsher the higher its value, more copy pointer repositionings are bound to happen, which will increase the execution time.



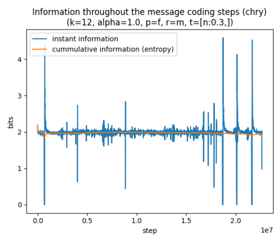
(a) Number of successive fails above 3 for chry.txt.



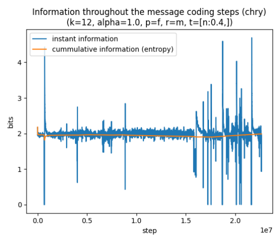
(b) Number of successive fails above 6 for chry.txt.



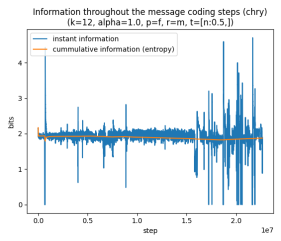
(c) Number of successive fails above 9 for chry.txt.



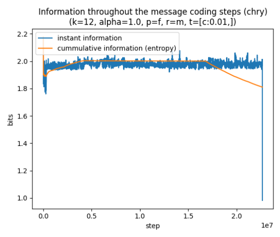
(d) Static probability below 0.3 for chry.txt.



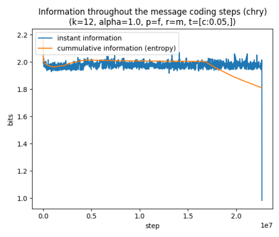
(e) Static probability below 0.4 for chry.txt.



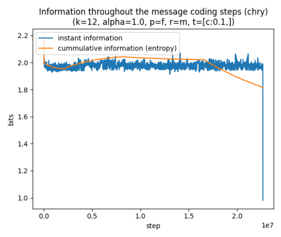
(f) Static probability below 0.5 for chry.txt.



(g) Absolute value of the negative derivative of the prediction probability above 0.01 for chry.txt.

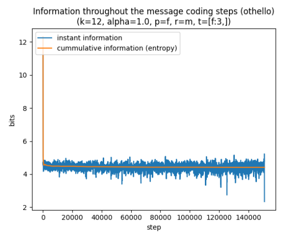


(h) Absolute value of the negative derivative of the prediction probability above 0.05 for chry.txt.

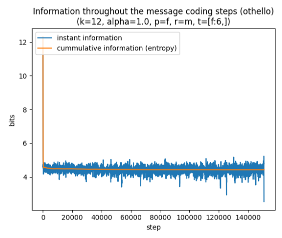


(i) Absolute value of the negative derivative of the prediction probability above 0.1 for chry.txt.

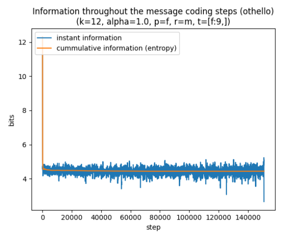
Figure 10: Information of the current symbol and entropy as the file is processed, varying the pointer reposition strategy on “chry.txt”.



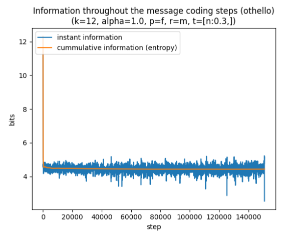
(a) Number of successive fails above 3 for othello.txt.



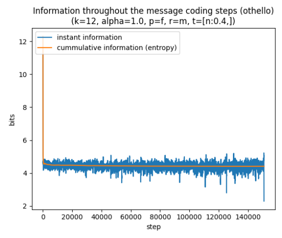
(b) Number of successive fails above 6 for othello.txt.



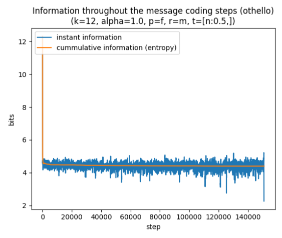
(c) Number of successive fails above 9 for othello.txt.



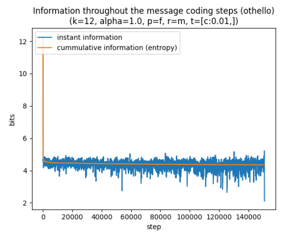
(d) Static probability below 0.3 for othello.txt.



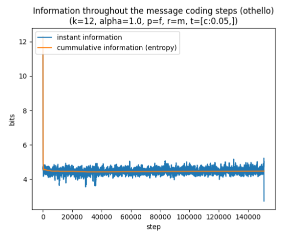
(e) Static probability below 0.4 for othello.txt.



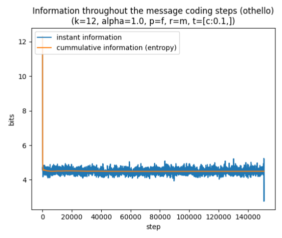
(f) Static probability below 0.5 for othello.txt.



(g) Absolute value of the negative derivative of the prediction probability above 0.01 for othello.txt.



(h) Absolute value of the negative derivative of the prediction probability above 0.05 for othello.txt.



(i) Absolute value of the negative derivative of the prediction probability above 0.1 for othello.txt.

Figure 11: Information of the current symbol and entropy as the file is processed, varying the pointer reposition strategy on “chry.txt”.

k	α	**Dist.**	**Manager**	**Thr.**	**Entropy**	**Exec. time (s)**
12	1.0	f	m	f:3	4.41545	9.55425
12	1.0	f	m	f:6	4.42929	8.15512
12	1.0	f	m	f:9	4.43929	7.45103
12	1.0	f	m	n:0.3	4.43524	7.27746
12	1.0	f	m	n:0.4	4.41081	6.96552
12	1.0	f	m	n:0.5	4.39864	9.20042
12	1.0	f	m	c:0.01	4.35129	8.39702
12	1.0	f	m	c:0.05	4.47641	10.0238
12	1.0	f	m	c:0.1	4.49229	10.8608

Table 10: Entropy and execution time when varying the threshold, on "chry.txt".

3.6 Verbose output

As a sidenote, the model offers the possibility of outputting the probability distributions and predicitons at each step, or of presenting the overall progress of processing the input file.

For the modes outputting the probabilities, the output can be in human-readable format, presenting whether the reported probabilites were from a guess or a prediction hit/miss in colored output. Alternatively, the output can be presented as comma-separated values with less extraneous characters in machine-readable format, which is useful for data analysis afterwards.

4 Text generator

The Generator module makes use of the functionalities offered by the CPM module. This module uses a Context Model calculated from a received text to calculate the next characters based on a context phrase.

It contains two main methods, the `firstPass()`, that reads the training file and makes an initial pass through the text to calculate the frequencies of the characters. This method also makes a context model based on the training file, and the size of the context is defined by the input string.

The `cpm_gen()` method uses the context model in the first pass to generate the text. When we try to generate the next symbol, we read the current context, which is the size of the input string and we check the context model for that context. If we have that context in the model, we get the list of possible next symbols, and their counts. Then, we generate a random value between 0 and the sum of the counts, and we check where that value lands between all the ranges of counts of events of that context. The symbol that is in the range of the random value is the next symbol of the text. If we don't have that context in the model, we get the list of all possible symbols in the alphabet, and their distribution, and we generate a random value between 0 and the sum of the distributions, and we check where that value lands between all the ranges of distributions of events of that context. The symbol that is in the range of the random value is the next symbol of the text.

We also have 2 other input parameters that modify the behavior of the generator. The first one is allowing the model to train itself, while generating the text. This is done by updating the context model with the generated text and the distribution of symbols in the alphabet. The second parameter is to allow all lowercase characters to be generated, this is done by converting all the characters to lowercase while reading the training file. This parameter provides a better context model without needing a big training file, but it also makes the text generated to be all lowercase.

4.1 Results

To better understand the Generator module and how different initial strings, which is our K and the parameters influence text generation, the Generator module was trained with `"othello_original.txt"` which is a very large text, varying the initial string and the parameters, but maintaining the seed value of 123, with an output limit size of 10 000 characters.

All the generated texts can be seen below.

Results for the text generator without -l or -t parameters:

Listing 1: Generator output for train file `"othello_original.txt"` with initial string "T" and without -l or -t parameters
Num guesses: 0

```
An atim Lobewast t ile: Ohincishy'diseapon'treanne s
b'tcrod (mes howintene Wim
```

Listing 2: Generator output for train file `"othello_original.txt"` with initial string "Th" and without -l or -t parameters
Num guesses: 0

```
I lour laight, lou sle orts soper shenle,
```

Listing 3: Generator output for train file `"othello_original.txt"` with initial string "The" and without -l or -t parameters
Num guesses: 0

```
She the despeake ship my
```

Listing 4: Generator output for train file `"othello_original.txt"` with initial string "The " and without -l or -t parameters
Num guesses: 0

```
The Token Wensday most the Moone you know:
```

Listing 5: Generator output for train file "othello_original.txt" with initial string ”The t” and without -l or -t parameters
Num guesses: 0

```
Enter Iago? Othe. What common his deluding her fantasie,
```

Listing 6: Generator output for train file "othello_original.txt" with initial string ”The te” and without -l or -t parameters
Num guesses: 590

```
For if such tricke vpon the error of this

fo' ofalMahetse n hrtuifo ie oeigmn   odeas' ) mHoe i i,
'lIt reodese:.se
```

Listing 7: Generator output for train file "othello_original.txt" with initial string ”The tes” and without -l or -t parameters
Num guesses: 2 604

```
all these Men, they do not, I do beseech you,

fo' ofalMahetse n hrtuifo ie oeigmn   odeas' ) mHoe i i,
'lIt reodese:.se
```

Listing 8: Generator output for train file "othello_original.txt" with initial string ”The test” and without -l or -t parameters
Num guesses: 10 000

```
doanelslhrCIh ieaY oo i yhnA:aBmygae
```

Results for the text generator with -l parameter:

Listing 1: Generator output for train file "othello_original.txt" with initial string ”T” and with -l parameter
Num guesses: 0

```
win sss'd yousd   kecer hete, abeeaurugotieuthe y gr wine
anem
```

Listing 2: Generator output for train file "othello_original.txt" with initial string "Th" and with -l parameter
Num guesses: 0

```
thery prod sh me growelfe ore toponio.
```

Listing 3: Generator output for train file "othello_original.txt" with initial string "The" and with -l parameter
Num guesses: 0

```
iago. and or pray yon. well'd be his drunke motifie sea
```

Listing 4: Generator output for train file "othello_original.txt" with initial string "The " and with -l parameter
Num guesses: 0

```
you commited, and questie too hard wise: follow high.
(prythee: an of a guilty th' confus'd,
```

Listing 5: Generator output for train file "othello_original.txt" with initial string "The t" and with -l parameter
Num guesses: 0

```
cassio. aunciant, be abus'd: exclaimed. are about me
filles vs a most graue.
 she speake heauen forget moe tapers.
```

Listing 6: Generator output for train file "othello_original.txt" with initial string "The te" and with -l parameter
Num guesses: 0

```
and made of the bride, and easie taskes.
```

Listing 7: Generator output for train file "othello_original.txt" with initial string "The tes" and with -l parameter
Num guesses: 2 191

```
make it a darling, like death, and her husband?

hr' ohdnadietse n istulho lh oelhnn  oeebt' ) naog i i,
'mat reoehsg:.sg
```

Listing 8: Generator output for train file `"othello_original.txt"` with initial string "The test" and with -l parameter
Num guesses: 10 000

```
wi:  esmlen,ntd,  bri  , so. eo e,alerc armeeiohitsdcslbo,
isodt g ittus lhitw eh are eori.h he?
```

Results for the text generator with -t parameter:

Listing 1: Generator output for train file `"othello_original.txt"` with initial string "T" and with -t parameter
Num guesses: 0

```
Geee mey atoountineenet: Hat he Tuengo. Cat agagngile
onifouph m y te, werellas te
```

Listing 2: Generator output for train file `"othello_original.txt"` with initial string "Th" and with -t parameter
Num guesses: 0

```
Othe nand he sh your Lore Beat inke dere onclonge cozent,
```

Listing 3: Generator output for train file `"othello_original.txt"` with initial string "The" and with -t parameter
Num guesses: 0

```
For lood my liue dingry. Thou haue State such my how I am
do aboue lip:
```

Listing 4: Generator output for train file `"othello_original.txt"` with initial string "The " and with -t parameter
Num guesses: 0

```
When haue was high. Haue you; a which, and adie: and sold
not man haue that forcement
that is euill she heere enemy in
```

Listing 5: Generator output for train file `"othello_original.txt"` with initial string "The t" and with -t parameter
Num guesses: 0

```
When haue was high. Haue you; a which, and adie:
 and sold not man haue that forcement that is euill she
 heere enemy in
```

Listing 6: Generator output for train file `"othello_original.txt"` with initial string ”The te” and with -t parameter
Num guesses: 319

```
Iago. You shall play the palme: I, well in Spleene,

dvdr bo'WliuaoS o( toy iyol  l dai Cdri  inc snuhmto h
```

Listing 7: Generator output for train file `"othello_original.txt"` with initial string ”The tes” and with -t parameter
Num guesses: 7 548

```
The Iustice of the place againe.

to  oe aa rhFa   ed v.otg n?— aes uu
```

Listing 8: Generator output for train file `"othello_original.txt"` with initial string ”The test” and with -t parameter
Num guesses: 10 000

```
eCLtmetakirayetotF  eelt)nhc reioneeol  s o tetmrt.dse u
```

Results for the text generator with -t and -l parameters:

Listing 1: Generator output for train file `"othello_original.txt"` with initial string ”T” and with -t and -l parameters
Num guesses: 0

```
bandont ftey wewhellllazoou t bero t t mince iseerododof
anke cor
```

Listing 2: Generator output for train file `"othello_original.txt"` with initial string ”Th” and with -t and -l parameters
Num guesses: 0

```
theerem be wit dood, why me my bassinesdeme
```

Listing 3: Generator output for train file "othello_original.txt" with initial string "The" and with -t and -l parameters
Num guesses: 0

```
iago. not isleepe to madeliuillainstrankes
```

Listing 4: Generator output for train file "othello_original.txt" with initial string "The " and with -t and -l parameters
Num guesses: 0

```
return'd againe to her grant the way. set it, and my sould
he diuell.
```

Listing 5: Generator output for train file "othello_original.txt" with initial string "The t" and with -t and -l parameters
Num guesses: 0

```
at blood beene me by and her officer iago. you shall my
husband
```

Listing 6: Generator output for train file "othello_original.txt" with initial string "The te" and with -t and -l parameters
Num guesses: 0

```
steep'd, to enflame it be true: or foredoes me without
more the strongly to you:
 for by this is othello, desdemon?
```

Listing 7: Generator output for train file "othello_original.txt" with initial string "The tes" and with -t and -l parameters
Num guesses: 7 349

```
e ,ulyirn et ai telott ,tmnhel  n uoa  oor ydo   mofe
de.bowsaege ntdhoee
```

Listing 8: Generator output for train file "othello_original.txt" with initial string "The test" and with -t and -l parameters
Num guesses: 10 000

```
easerh w : p ht tn-lyerlehm eos shn snsn ovssniro h
```

```
fanmdi'l
```

To conclude, a small test to compare the entropy of the original file and the entropy of the generated file was done. The test was done by running the copy model on the `"othello_original.txt"` file to obtain the entropy. The following listing displays the result.

```
Mean amount of information of a symbol: 4.60154 bits
 Total amount of information: 715958 bits
```

Then a generated text was produced using the seed 123 and a initial string of *the*. With the generated text, we ran the copy model again to obtain the entropy of the generated text. The following listing displays the result.

```
Mean amount of information of a symbol: 4.63313 bits
 Total amount of information: 677697 bits
```

By comparing these results we can conclude that the entropy of the generated text is smaller than the entropy of the original text, as expected, because the generated text should never produce more information than the original text.

5 Conclusion

From the development of this project, it was possible to conclude that the Copy Model, although it did produce smaller entropy than the entropy contained in the original files, it didn't demonstrate the desireable performance, even when testing various strategies and combinations. On the other hand, the generation of new text based on training examples did produce good results depending on the parameters used.

5.1 Copy Model

Even though the copy model reported lower entropy values for the `chry.txt` file, the same was not verified for `othello.txt`. We believe this is due to the greater complexity of the alphabet on `othello.txt`, which is unmetched with how small the number of characters is when compared with `chry.txt`. Even then, the entropy value is only slightly better than the baseline for `chry.txt`. Overall, by analyzing the graphs, we can note that the model can't perform long-standing predictions.

This is probably due to a factor we didn't take into account when developing the model: we have no strategy to decide the moment when we should start copying, since we copy as soon we can. This can have a great impact on the results, since the sequence we are copying may not the best one at a given time, and since we attempt predictions with a given copy pointer we miss the chance of trying another that may have provided a more successful copy.

5.2 Text generator

By looking at the results we can conclude that the initial string, which is our K, is the parameter that influences our results the most. This happens because a very small K generates words that don't make sense, and a very large K will reduce the amount of entries in the Hash Table which will lead to more guesses. The initial string should also take into account the language of the training file, and should contain normal words, not names nor a special words.

The training file should be as large as possible, to have the most accurate model possible. Because the bigger the file, the more information we have in the Hash Table to calculate the probabilities of the next symbol.

The -l parameter allows the generator to generate a text with only lower case letters, and is better overall and great for smaller input texts, although it compromises the quality of the text generated due to being only lower case letters. The reason behind this is that the Hash Table will have less entries, and the probabilities will be more accurate.

The -t parameter allows the generator to train itself, and it lowers the number of guesses, but at the cost of that guess not making sense, and then it will make the same error again. This parameter is better for larger input texts, because it will have more entries in the Hash Table, and therefore decreasing the number of guesses and increasing the accuracy of the probabilities.

Using the double parameters it behaves similarly to the -t parameter, but it will generate less guesses and have a slightly better accuracy at generating text that makes sense. Although this small increase in accuracy is only noticeable in our results in the "The te" and "The tes" initial strings. This makes it a better option than the -t option alone but it compromises the quality of the text generated due to being only lower case letters.

Also to denote that, the smaller the initial string, the more time it takes to produce an output, due to the size of each entry of the Hash Table increasing as the initial string size decreases.

6 References

References

[1] R. S. Boyer and J. S. Moore, *MJRTY—A Fast Majority Vote Algorithm.* Dordrecht: Springer Netherlands, 1991, pp. 105–117. [Online]. Available: https://doi.org/10.1007/978-94-011-3488-0_5