Title: TAI - Lab work n° 1

Author: Bruno Bastos, Eduardo Santos, Pedro Bastos

Date: 02/11/2021

Contents

1. Introductory Note

The objective of this work was to build a *Finite-Context-Model* that reads the text from a file and stores the number of the times a character appears after a sequence of characters. It then can use the stored information to generate new text or to calculate the entropy of the text. The program makes use of several parameters that can influence the obtained results.

- **k** - parameter responsible for the size of the sequences.
- α- parameter responsible for the uniform probability.

In this work, we will also describe the main decisions taken during the implementation of the problem, comparing not only the different results from the variation of certain parameters, but also the how the entropy varies on different texts.

2. Steps and Decisions During the Implementation

2.1. Alphabet and data structures

The program starts by reading the provided text and making a list of the distinct symbols in that text, denoted as the alphabet (Σ). After that, using the parameters, it chooses between 2 data structures to store the occurrences of the characters that are read from the text file.

It decides which data structure to use by calculating the necessary memory to store the information. If the required memory is less than a defined threshold, the data structure used is a 2D array, else it uses an hashtable.

The required memory is calculated using the following formula:

$$mem = |\Sigma|^k * |\Sigma| * \frac{s}{(8 * 2^{20})} \tag{1}$$

where $|\Sigma|$ is the length of the alphabet and s is the size in bits of the stored information. The result is obtained in MB, assuming that the values are stored using a 16(s) bit integer.

2.1.1. 2D array

The 2D array has size $|\Sigma|^k$x$|\Sigma| + 1$, where each entry will store the number of occurrences of a given character of the alphabet appearing after a certain sequence of size k. An extra row is allocated to keep the sum of the values in the row.

The index allocated for a specific sequence of k characters is calculated using the following formula:

$$sequence\_index = \sum_{i=0}^{k-1} |\Sigma|^i * char\_index_i \tag{2}$$

where, $char\_index_i$ is the index of the char i in the alphabet list.

Using this formula, it is possible to calculate the row allocated for that sequence of characters.

Although accessing and changing the value of the each entry is fast, $O(1)$, the table does not scale because it has many sequence of characters that were never seen in the provided text, leaving many entries at 0 that are using a large portion of the memory.

2.1.2. Hashtable

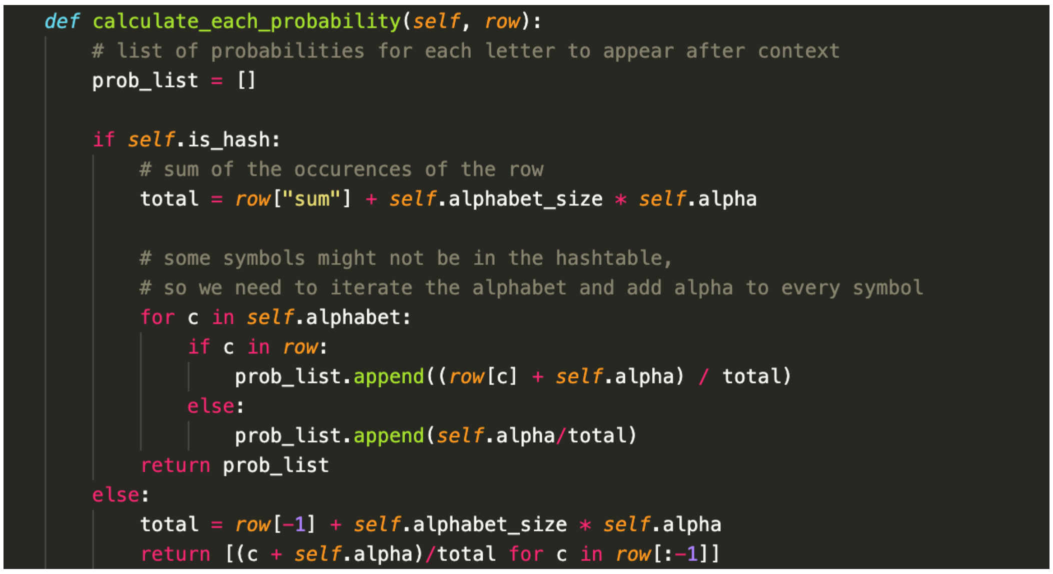
Another approach, when the number of rows and columns is huge, is to use an Hashtable to store only the values that are seen in the text. Hashtables have fixed size, but can be re-sizable, and the index of the sequence is calculated using an hash that returns a value from 0 to the length of the hashtable. Each entry in the table is also another hashtable, so that only the characters that appeared after that sequence are stored. This saves a lot of space as there will not be any entry with value 0. Also, for each row, there is a key in the hashtable called "sum" that tracks the sum of the values in that hashtable.

2.2. Entropy

After the table is filled, we then proceed to calculate the entropy of the text. Firstly, we need to calculate the probabilities of each symbol given a certain context(c). As we know, this is achieved by the following formula:

$$P(e|c) \approx \frac{N(e|c) + \alpha}{\sum_{s \in \Sigma} N(s|c) + \alpha|\Sigma|} \tag{3}$$

The α is used to prevent 0 probability to events that didn't occur. We calculate the probabilities for each row of the table, i.e, for each context(c), we generate a list that has the probabilities of each symbol after that context:

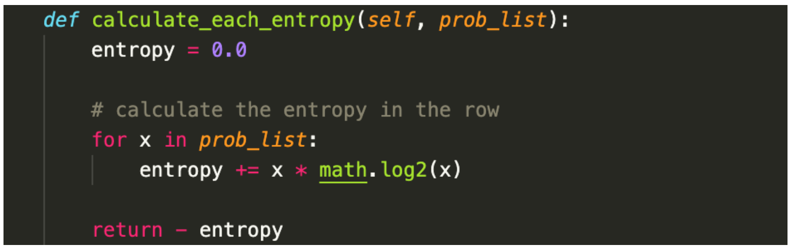


```python
def calculate_each_probability(self, row):
    # list of probabilities for each letter to appear after context
    prob_list = []

    if self.is_hash:
        # sum of the occurences of the row
        total = row["sum"] + self.alphabet_size * self.alpha

        # some symbols might not be in the hashtable,
        # so we need to iterate the alphabet and add alpha to every symbol
        for c in self.alphabet:
            if c in row:
                prob_list.append((row[c] + self.alpha) / total)
            else:
                prob_list.append(self.alpha/total)
        return prob_list
    else:
        total = row[-1] + self.alphabet_size * self.alpha
        return [(c + self.alpha)/total for c in row[:-1]]
```

We can now get the entropy of each row, given by the formula:

$$Hc = -\sum_{s \in \Sigma} P(s|c) log P(s|c) \tag{4}$$

This is calculated in the following function:



```python
def calculate_each_entropy(self, prob_list):
    entropy = 0.0

    # calculate the entropy in the row
    for x in prob_list:
        entropy += x * math.log2(x)

    return - entropy
```

Now, we have everything needed to calculate the final entropy of the text:

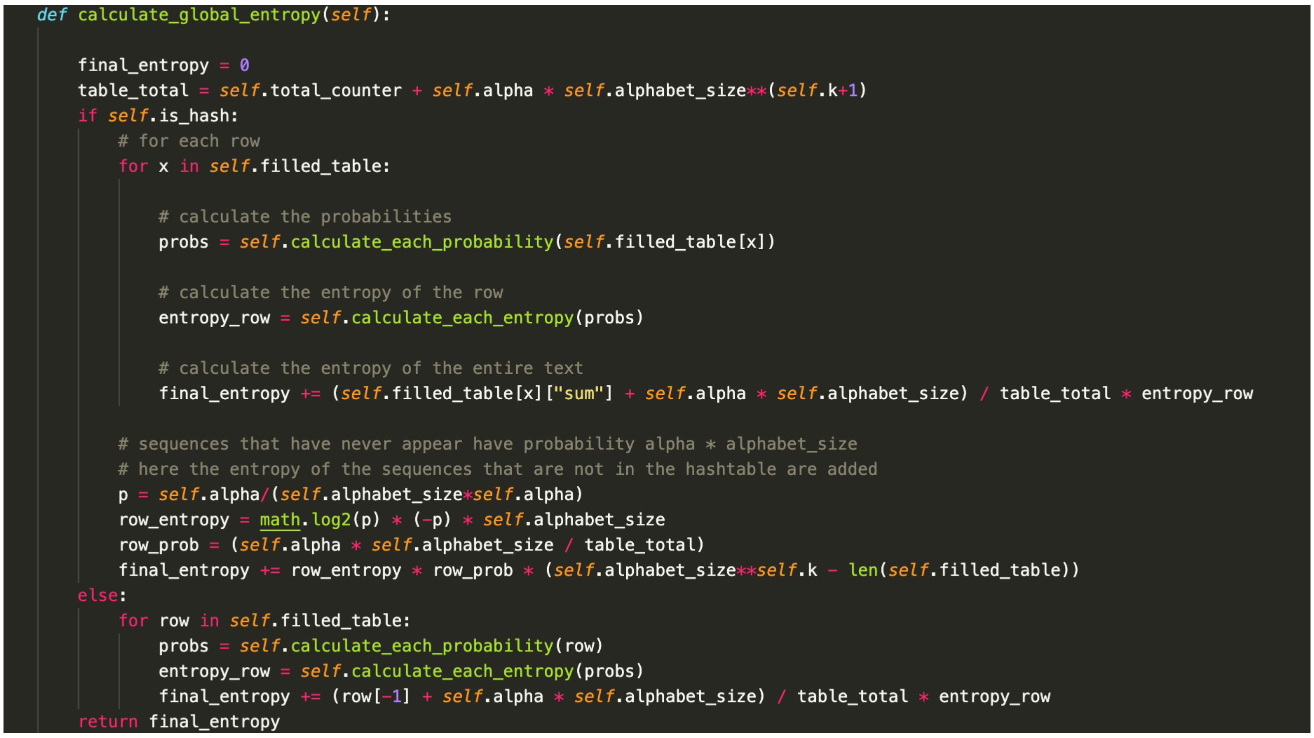
$$H = \sum_{c} P(c)Hc \tag{5}$$

The probability of a context, just like the probability of a character, also depends on the a, and can be calculated using the following formula:

$$P(c) = \frac{\sum_{s\in\Sigma} N(s|c) + \alpha|\Sigma|}{total + \alpha|\Sigma|^{(}k+1)} \tag{6}$$

where $total$ is the total number of characters in the text. This was done this way because it follows the same principal as before, if an event has never occurred before it does not mean that its probability is 0. So if a context has never appeared, its probability should be the sum of the probabilities of its entry in the table, divided by the total sum of the table, after adding α to each entry.

This is done in the calculate_global_entropy function:



```python
def calculate_global_entropy(self):

    final_entropy = 0
    table_total = self.total_counter + self.alpha * self.alphabet_size**(self.k+1)
    if self.is_hash:
        # for each row
        for x in self.filled_table:

            # calculate the probabilities
            probs = self.calculate_each_probability(self.filled_table[x])

            # calculate the entropy of the row
            entropy_row = self.calculate_each_entropy(probs)

            # calculate the entropy of the entire text
            final_entropy += (self.filled_table[x]["sum"] + self.alpha * self.alphabet_size) / table_total * entropy_row

        # sequences that have never appear have probability alpha * alphabet_size
        # here the entropy of the sequences that are not in the hashtable are added
        p = self.alpha/(self.alphabet_size*self.alpha)
        row_entropy = math.log2(p) * (-p) * self.alphabet_size
        row_prob = (self.alpha * self.alphabet_size / table_total)
        final_entropy += row_entropy * row_prob * (self.alphabet_size**self.k - len(self.filled_table))
    else:
        for row in self.filled_table:
            probs = self.calculate_each_probability(row)
            entropy_row = self.calculate_each_entropy(probs)
            final_entropy += (row[-1] + self.alpha * self.alphabet_size) / table_total * entropy_row
    return final_entropy
```

Therefore we only need to iterate over the table and, for each row, calculate the entropy (using the probabilities). Is important to notice that the implementations are different depending on the type of table being used. If it is an hash table, we need to do the additional step that adds the sequences that never appear in the text.

2.3. Generator

With the help of the *FCM*[1] it is possible to generate texts with variable size. The program takes the size of the generated text as an argument and using the *FCM* it creates a text based on another text.

It starts by generating an initial sequence of size k randomly or the sequence can be provided by the user. Then by checking the table entry for that sequence, it chooses a random next character using the probability of that character in the table and adding the value of α. As the parameter α increases the more uniformly distributed the text will be.

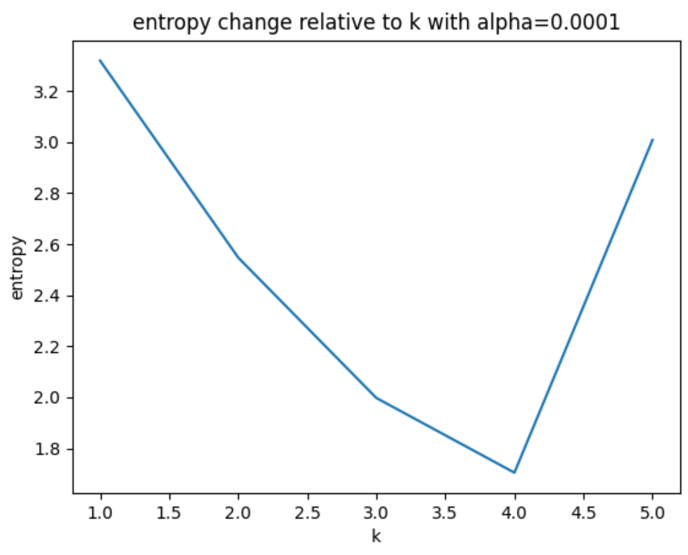
3. Results and Analysis

The purpose of this section is to show the results of the work and analyse the variation of different parameters and its meaning, as well as the entropy of the text. Different example texts and parameters were used to be able to properly analyse the results.

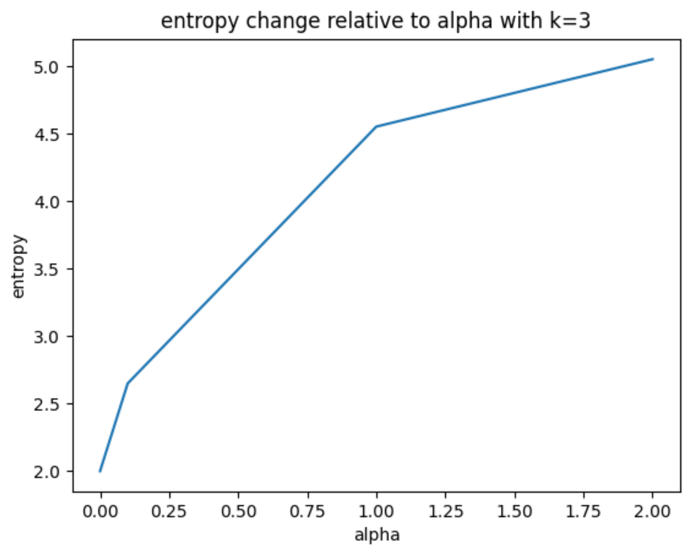
[1] Finite-Context-Model

3.1. Parameters variation

Both the values of the *k* and α were changed to understand its impact on the generated text. In this first analysis the default example text ('example.txt') was used.

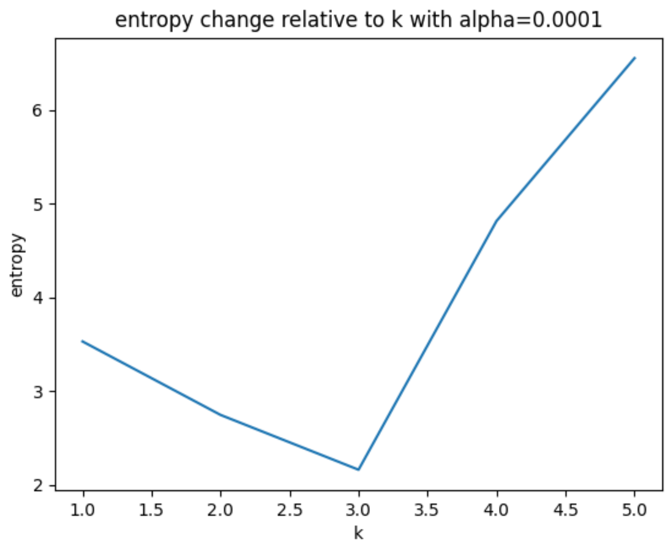


As we can see, using a fixed value for the α, the entropy of the text varies with different values of the k. It decreases until k=4 is reached, and then increases again.

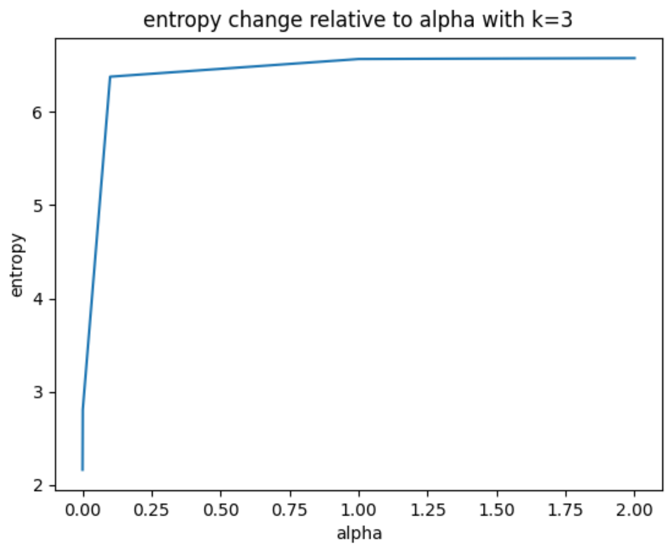


On the other hand, if we fix the k and increase the α, we get an increase of the entropy in an almost proportional way.

To confirm this, we made the same tests to another text example ('example1.txt') and we got, for the variation of the k:

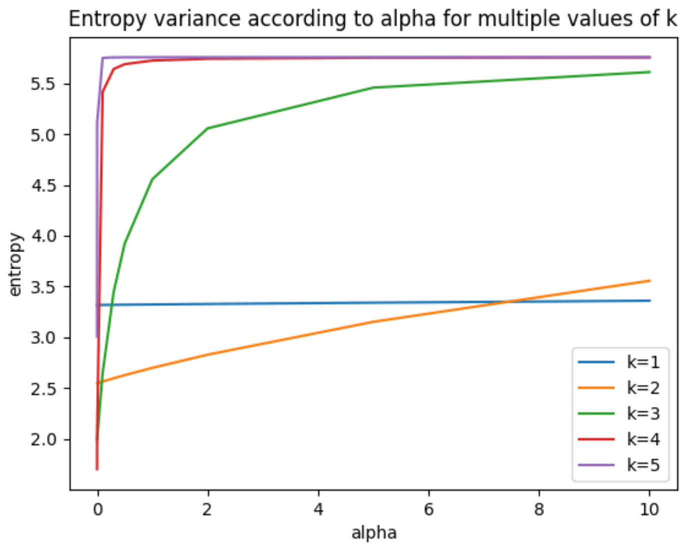


And for the variation of the α:



As we can see, the graphics have a few differences from the other example, but with the same global hypothesis. With that in mind, it is possible to conclude that the entropy decreases with k until a certain value of k and then increases afterwards. As α increases the entropy also increases, this is due to the fact that the higher the α the more uniform the text will be, meaning that the probability of each entry in the table will be closer.

Using the variance of the entropy, but this time plotting multiple lines where each line represents a value of k, it is possible to see how the combination of the parameters can influence the value of the entropy.



The value of the entropy seems the converge to value as the value of the parameter α increases. The higher the value the faster it converges. This happens because for higher values of k, the number of times of a sequence of characters appears is smaller than when the k is small. Thus, the value of α will have a greater impact in the probability of the next character, making the it more uniformly distributed. The entropy converges to the value of the entropy if the text had the same probability for each sequence of characters.

3.2. Generated text

Using the text ”example.txt” with parameters k=3 and α=0.001 to generate a text of size 1000 characters for example, the result obtained is:

* * *

r3@-spq!g5e.i)*daughtly fell that thoul in lifts a make unto die, sone eliansweepeth of out given also is hastumber in shall: 10:24 thou areaces shou had can by the eath the eat reace owledomisses, o lord in them, they nothissesself, ance from thee.

25:15 the him shall man all evernaclean in thanded hat ephunning, and them again hosed the lay but withe comercy shee.

22:7 gospear not head uncless the of his ves for word.

9:3 thousalves' cons.

3:22 and a bread said.

32:8 and thoughost, and the sperstfruitfulfil the lordaiah a werefore mout of the daughtenbera, as broken the lord conce: and burim the surely cometh the brought against to have shout.

36:10 mand i begunishalt it will is writ in it reth is shaldness whom beast fall ther ram is in of his the seven them, and time.

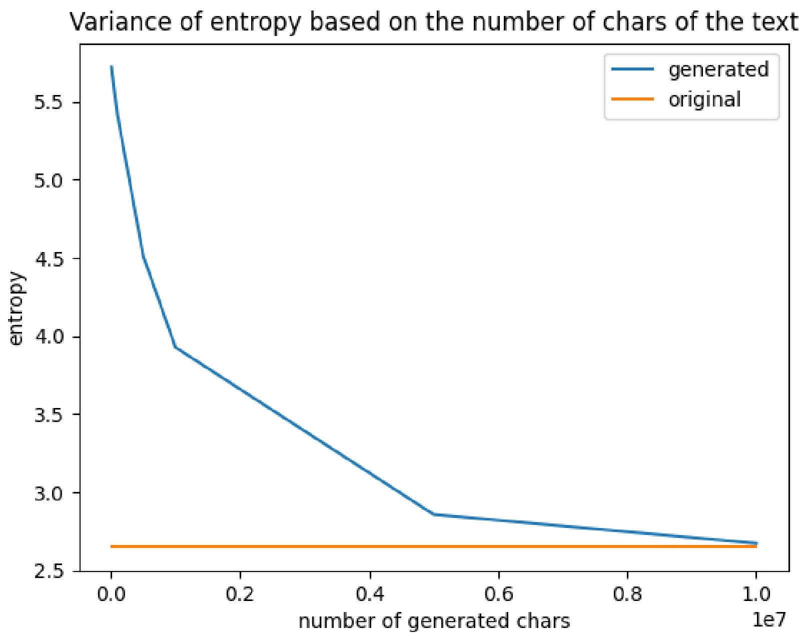
22:7 and afterefore to the let then the visin the more man; be and who seen jere it speechab, tar, temalem were i and that shall thee fell not yea, anding if alse grought thy pressove.

12:13 and he down rewised,

* * *

This result used a random initial sequence.

As it was shown, the entropy of the text can change a lot depending in the values of the k and α. The next experiment checks if the generated text has an entropy value that is close to the original text. This was done for multiple sizes of the generated text for k=3 and α=1, using the same initial sequence has the original text.



With a fixed k and α, we can see that the longer the generated text is, the more accurate is the entropy of it compared with the example text. This is due to the fact that the smaller the text is, the bigger impact a low probability character appearing will have. As the sample size increases, the probabilities for each character will get closer to the ones in the original text.

4. Conclusion

This work gave us some insights on how a probabilistic approach can be used for text generation and how the small variance of the provided parameters can have a major influence in the achieved results.This work also allowed us to really understand the finite-context models, as well as the influence of each parameter in the entropy of the text and how it varies from text to text.