Finite-context-model and text generator

Diogo Mendonça 93002
Leandro Silva 93446
Margarida Martins 93169

Index

1. Overview

In this work, our goals were to develop 2 programs, *fcm* and *generator*. The first one aimed to collect statistical information about texts, using a type of markov model, the finite-context model. The second aims to generate text that follows the statistical model developed in *fcm*.

The chosen programming language is C++, as it allows for greater control on the data structures and execution of the code.

2. Data structures

For the 2 programs, it is necessary to construct a type of table to store the number of times that each symbol occurs in each context of size k.

For this we could use a 2D array, saving the value of each character in the alphabet for the corresponding context, leading to a table of size of $A^{k+1}C$ bytes, with A being the size of the alphabet and C the size in bytes of the counter (we used unsigned integers - 2 bytes).

The problem with this approach is that, for example, with A equal to 4 and k equal to 20, we would have a table of size 8GB. For table sizes larger than 256MB, we decided to use a hash table in order to save space, as this data structure doesn't save untrained contexts and non-visited members of contexts (initialized to 0 in the 2D array). The hash table has binary trees associated with it, in order to store the number of times that each encountered symbol occurs in the given context.

Using a hash table also allows for greater program performance, as the smaller number of entries leads to a decreased complexity in the transversal of the table, needed to calculate the entropy of the model, in the transversal of the number of times that each symbol occurs in the given context, needed for the generator. A downside of this choice is the slightly higher insertion and lookup time, due to the higher overhead of the data structure and the possible existence of collisions in the hash table.

File *fcm.hpp*

Defines the classes *FCM*, *Table*, *TableArr* (derived class form *Table*) and *TableHash* (derived class from *Table*).

Class *Table*

This class has as attributes the order of the model, a map which stores the frequency (value) of a symbol (key) and the total number of characters a file has. The order of the model and the symbols and corresponding frequency are passed in the constructor of the class. *Table's* derived classes methods will be explained below.

Class *TableArr*

This class the information of the model is stored in a two dimensional array (*table*) where $table_{ij}$ represents how many times the symbol *j* appears immediately after context *i*. This class also has a map which stores the ID (value) of a character (key), allowing the easy indexing of each context to the table (using the function *get_index* explained below).

Class *TableHash*

In this class the storage structure is an unordered_map (map using a hash table) with the context as key and a *struct state* as value. The *state* stores an integer which represents the number of times a given context appeared, as well as a map that for each symbol (key) gives the amount of times it appeared immediately after the context.

Class FCM

This class saves a pointer to the table, and it determines the alphabet size and which type of table is generated (explained in more detail below).

3. Program options

fcm: For this program to work it is required that the user introduces the name of the input file, the size of the context (k) and the value of the smoothing parameter (α), in this order. The user can also utilize the flag -i to plot additional statistical information about the model.

generator: This program has the same requirements as *fcm*, but offers more options to the user, namely:

- -p or --prior: The argument of this option will be given to the program as the prior for the text generation (a random prior will be used if not specified).
- -s or --text-size: The argument of this option will determine the number of characters generated by this program (1000 as default).
- -t or --threshold: The argument of this option will determine the maximum table size in MB to use an array based model rather than hash based (default is 256).
- --show-random: If this flag is set, symbols generated randomly, for not having a trained context, will be highlighted.
- --relative-random: If this flag is set, when the context isn't trained, the program will generate symbols according to the character frequency in the input text.

4. Functions

train

Method *train* receives a file pointer as argument. This function will read all contents of the file and store the number times a symbol follows a certain context in a structure *table*.

In *TableArr* class this information is stored in a two dimensional array (*table*) where $table_{ij}$ represents how many times the symbol *j* appears immediately after context *i*.

In *TableHash* the storage structure is a map with the context as key and a *struct state* as value. The *state* stores an integer which represents the number of times a given context appeared, as well as a map that for each symbol (key) gives the amount of times it appeared immediately after the context.

get_index

Function used by *TableArr* objects in order to get the index of a certain context passed as argument, through the transformation of the context (string) into a number (index), based on the IDs of the characters, the size of the context *k* and the alphabet size A (for k=3, index = $ID3*A^2 + ID2 * A + ID1$). This allows for a fast lookup in the matrix.

get_entropy

Method *get_entropy* receives a smoothing parameter as an argument. This function calculates the overall entropy of the model. When the smoothing parameter is greater than 0 it takes into account the contexts not present in the file as well as the letters which do not occur after a certain context.

generate_text

Method *generate_text* receives as arguments a smoothing parameter, a prior, a text size and two boolean flags.The purpose of this method is to generate a text using the symbol probabilities given a certain context.

First, a value between 0 and 1 is randomly generated (*random*). Next, the function iterates over each symbol and computes the probability of its occurrence given the context using the smoothing parameter and the *table* structure. The iteration continues until the cumulative probability is equal or greater than *random*. The new context will be the last k-1 characters of the context and the symbol where the iteration stopped. This process repeats until the text size has been reached.

If there is a context that never appeared in the file, the method behaves in two different ways depending on the flag *relative_random* passed as argument. If *relative_random* is true, the next character will be generated considering the probability of its occurrence in the file, else the next character is randomly generated with all the symbols having the same probability.

If an empty prior was passed, the program will choose one context (present in the file) randomly.

print

Method print will print in a more legible way the structure built in the *train* method. With this method we can visualize the frequencies of the symbols for each context.

5. Execution times

In order to test the efficiency of our fcm program we measured the time it takes to train our model according to the size of the context. For this measure, the files *example.txt (4,4MB)* and *quim.txt (7,5KB)* were used.

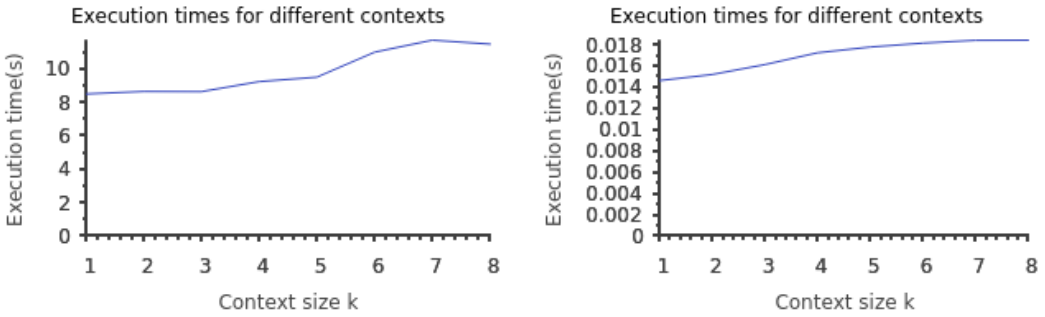

Figure 1 - Execution time for file *example.txt* (left) and *quim.txt* (right)

As we can observe, the execution time increases with bigger context sizes. However, this increase is not significant in order to be an issue.

For the *example.txt* file the program executes in about 10s while in *quim.txt* it takes only about 0.017s, approximately 600 times faster. This value comes from the fact that *example.txt* size is approximately 600 times bigger than *quim.txt.*

6. Results

By creating multiple models with different k-orders and smoothing parameters, but trained with the same source, the *fcm* was able to make the graphs shown in Figure 2 and 3.

In Figure 2, we can see that the optimal context size is between 2 and 4. After this interval, the higher the context size, the higher the entropy. A size of 1 is too low for the model to learn. Having a low entropy means that the model is stricter to a pattern found in the source, so, we can say that for this data, the optimal parameters are $k = 3$, with $a = 0.001$.

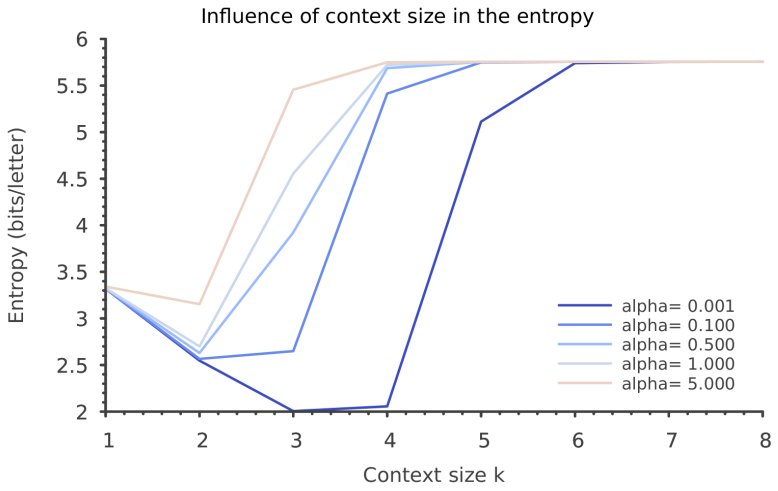

Figure 2 - Influence of *k* in the entropy for the *example.txt* file

In Figure 3, we can see more clearly how the alpha affects the entropy of the model. We can be assured that for all *k* values, the greater the alpha, the greater the entropy. This is because the alpha makes the model more uniform, allowing it to generate text more independently of the source and leading to a more random output.

Comparing Figure 2 to 3, we can also observe that the smoothing parameter has less impact on the model entropy than the context size.

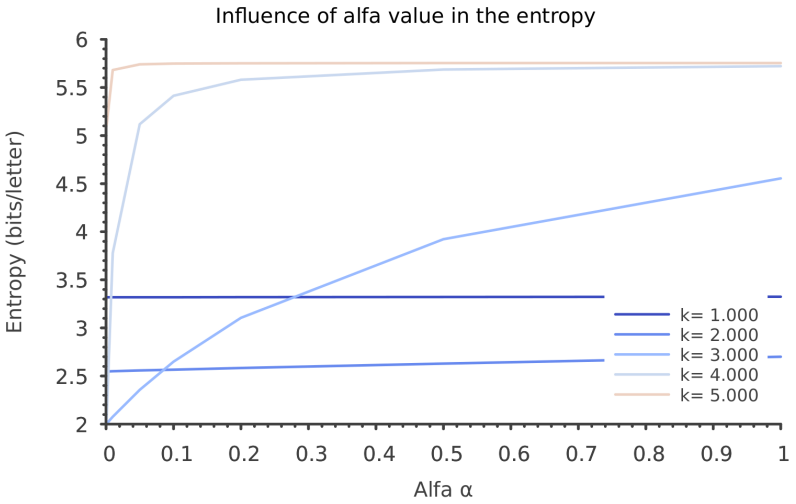

Figure 3 - Influence of *a* in the entropy for the *example.txt* file

From this point on, we will present some examples of the outputs of our generator. In the results obtained, we often used alpha equal to zero to obtain a text truthful to the origin source. This has no harmful implications for cases when the context obtained has no occurrences, as the algorithm will simply generate an equiprobable symbol from the alphabet. The only problem with doing this is that the entropy calculated is undefined, so to fix this, we calculated the entropy with a very low value, in the order of 10^{-10}.

From the results shown in the examples 6.1 and 6.2, where $k = 1$ and $k = 2$, respectively, we can see that the generator has some trouble in forming correct words. Example 6.2, was able to output more correct words, at least the smaller ones with 2-3 letters, which is quite normal for a context size of 2.

Example 6.1 (k = 1, a = 0, text_size = 400)

```
mollougas, me, worid,


pan azanorand wereres, to wh cknd an ewoulimingh juthbrgar knof thevensaverit
```

Entropy ≈ 3.31872

Example 6.2 (k = 2, a = 0, text_size = 200)

```
ommand: and not wore o hat i by oplem, weren theructied und
makinighteoplanswong, of the they spes: ho
```

Entropy ≈ 2.54758

For higher values of k, the generator starts to receive contexts never seen more often, meaning that it has no information about the probabilities of each letter that it should follow. As it was said before, when this happens, the model yields an equiprobable symbol from the alphabet. The symbols which were generated like this are colourized in **blue**.

In the example 6.3.1, as the prior is not recognized by the model, it generates ‘2’ has the next symbol. Returning a uniformly random symbol makes it less probable to obtain an existent context, which will result in a chain of characters that do not follow the source.

Example 6.3.2 fixes this by returning a random symbol according to its frequency. Thus, despite having a non-optimal prior, the model returns symbols closer to the source, one ‘s’ and one space, which will rapidly end up in a context that the model recognizes. Thus, it can continue on with more accurate results.

Example 6.3.1 (k = 4, a = 0, text_size = 400) with equiprobable symbols

```
ndom2;;.jd($
,?f)vj4."y2@bd!ua?@wv@fhlb*@1w0:q.t                                    o4si-as7k4!!
n91v--/y7%bk"z-i-"-jftgw0882)*dgbsb;; pk1*a;!g$
rilb%t'b2k?78nu7w8a('%?.h,k1wu%shza6lt814p6:k'v:c!sl?%1c;/!(3:q8gn;6i)*;6nk!lsja3q
5mydv5!126(up:

12:32 for as leperselves, that down in his man honourage, 35:25 and that god,
and the time fourt offering unto they hearth: for the might.

2:2 declare lend, the walled to meat? know white, so to
```

Entropy ≈ 1.66164

Example 6.3.2 (k = 4, a = 0, text_size = 200) without equiprobable symbols

```
ndoms of god of joyful make a cave, i sent times iniquities a lamps, and that
where seen i wilt they took up to rest he shall mizraim, and the reven from that
came to
cyprus wait, and in the seven them; s
```

We then felt curious to check how the generator would perform with different kinds of alphabets. **Morse code** was a good candidate, as it was considerably different from having an alphabet of letters, where each letter corresponds to only one symbol. In Morse code, one letter could have a maximum of 6 symbols (if using an extended Morse code version with punctuation). For that matter, we weren't expecting to get as good results, since the context would end up 'tearing' letters apart.

The file *morse_code.cpp* in the folder *extras* has a simple converter from text to Morse code, which we used to transform the *example.txt*.

After all was set up, we attempted first to generate text with a model that was trained with a context size of 6. The example 6.4 shows the result of the generation of 500 characters, and its decoded result.

Example 6.4.1 (k = 6, a = 0) Raw format

```
.----..-. --- ..- ... / . --. . -. ... / ..-. / .- .. .- -.. --- -... // .... .- .-.. . /
.... .. ... -.. / --- ..- -. -.- .... .- - / -. . / .-- ....- / -- ---... .. . / .- / -
.... . / ..-. / . ...-- / ... .. .-. . .- .-.. .-. .-. ..- -.. / - .... .---- ..-. / .--
... - .... . -. / - .... . / .. ....- . -.. / .- -. -.. . .- - / ...- . .-- --- .-.. -..
..--- --- ..-. / - .... . ... .. -- .. - .... . / --. --- .-. -.-- / -.-. --- .-- . ... .-
-- -- . -. / .... . / ... ...- --. ... - .... ... / --
```

Entropy ≈ 0.98271

Example 6.4.2 (k = 6, a = 0) Decoded format

```
1fous egens f aiadob
hale hisd ounkhat ne w4 m:ie a the f e3 sirealrrud th1f wsthen the i4ed andeat
vewold2of thesimithe gory cowesammen he svgsths m
```

Looking at the decoded format, it doesn't appear completely randomized. The model attempt to follow its source is clear, but not enough. We could say that this result is similar to the example 6.1, where *k = 1*. This is probably because the average length of a Morse code letter is on average 3,91, with 6 for the maximum length, meaning that the context is quite sufficient to represent 1 letter.

By increasing the context size, the decoded format of the text generated became more and more accurate to the source (examples 6.5 and 6.6). However, values much higher than *k = 18* result in very complex contexts that the generator fails to recognize, like the generator from the text *example.txt* for contexts size higher than 5.

Example 6.5 (k = 12, a = 0) Decoded format

```
1:4 and the land in their led# fore that i man whing thy they fuld was in all be camely as
peak yet him shed becrelnated thousand, when ward.

thou s
```

Entropy ≈ 0.59069

Example 6.6 (k = 18, a = 0) Decoded format

```
1:17 and the land above all that i drank in the philistines will sit at my lord.

8:12 and he beginning, and reuben,
and none effection for them the
```

Entropy ≈ 0.42700

---

7. Conclusion

With this work we can conclude that the context size is an important parameter in order to generate good texts. A small context will produce texts that seem random. On the other hand, with large contexts the text produced is comprehensible but the model entropy is higher.

A good choice for the alpha parameter is also important. It allows new contexts to appear while keeping the text comprehensible.

Our fcm program can adapt to different alphabets creating suitable models.