Universidade de Aveiro

School Year 2021/2022

TAI - Practical Assignment 1

Diogo Bento 93391

Pedro Laranjinha 93179

Leandro Kiame 78177

Professors Armando J. Pinho & Diogo Pratas

Introduction

For this project it was requested that we develop a program that utilizes **finite-context models** to collect and report statistical information from texts, namely an estimation of the text's entropy based on the model developed through analysis.

Additionally, an additional piece of software that takes those same finite-context models and uses them to **procedurally generate text** was created.

We chose to implement our solutions in *Python* due to its high convenience factor, and use *bash* scripts to perform entropy calculation for a variety of parameters and record the results of said process.

Code Steps

fcm.py

The program developed to return an estimation of a text's entropy is ***fcm.py***.

This program has 3 arguments:

- **--order:** Sets the order of the model, the default value is **2**
- **--smoothing:** Sets the smoothing value, the default value is **1**
- **--source:** The name of the file that will be read to generate the model, the default value is **../example/example.txt**, and is sensitive to the current folder context

From inside the /bin/ folder, one can run the code by using *python3* as follows:

python3 fcm.py <args>

This program functions as a pipeline of functions:

1. We use the order of the model and the input file to return a map that contains how many times a symbol occurs as follow-up to a context, a dictionary with how many times the context appears in the text, and the alphabet of the text.
2. Then we use the symbol follow-up map, the alphabet and the smoothing parameter to return a probability table which contains the probability of any character in the alphabet occurring in each context.
3. Finally we use the probability table, the dictionary with how many times the context appears in the text and the length of the alphabet to return the estimation of the text's entropy according to the formula we were provided.

generator.py

The other program which was developed to procedurally generate text is ***generator.py***.

This program has 6 arguments, the first 3 being the same as *fcm.py*:

- **--order:** Sets the order of the model, the default value is **2**
- **--smoothing:** Sets the smoothing value, the default value is **0.00001**
- **--source:** The name of the file that will be read to generate the model, the default value is **../example/example.txt**, and is sensitive to the current folder context
- **--output:** The name of the file that will be used to output the generated text, the default value is **output.txt**, and is sensitive to the current folder context
- **--length:** The length of the text to be generated, the default value is **1000**
- **--start:** The starting sentence so the program knows which context to use first, if no value is provided one is randomly selected from the provided text, with weights decided by how frequently a string occurs in the text

From inside the /bin/ folder, one can run the code in a similar way to ***fcm.py***:

```
python3 generator.py <args>
```

This program also functions as a pipeline of functions with the first 2 stages being the same as in ***fcm.py*** and the last one being:

- We use the probability table, alphabet, desired length of the generated text and the starting sentence to procedurally generate the text, which is then written into the output file.

Design Decisions

We read the input file in its entirety rather than in chunks and then analyze the extracted string as doing so simplifies the code.

Instead of using a table with rows for all context permutations and columns for all symbols to represent a model, we use a map with only the existing contexts and only the symbols which occur in each one.

The probability table generated in the second phase of the pipeline is actually a map which has a ***default*** key related to the probability value for all possible follow-up symbols that have not been observed in a given context. This allows it to function as a matrix while occupying less memory.

When calculating entropy via ***fcm.py*** the **smoothing factor cannot be 0** as this causes problems in the entropy calculation. This **does not apply** to ***generator.py***.

If an unknown k-length string is generated the program defaults to choosing a random symbol from the observed alphabet, with every symbol having an equal probability of being chosen.

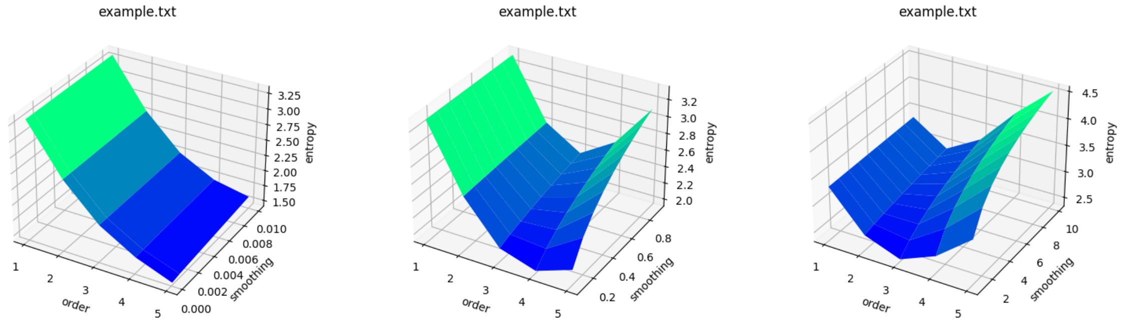
Results

We tested ***fcm.py*** using 7 files, 6 different languages, 5 different order values and 22 different smoothing values and recorded the output of said process. We then used ***surface_plot.py*** to create plots of the output for easier analysis.

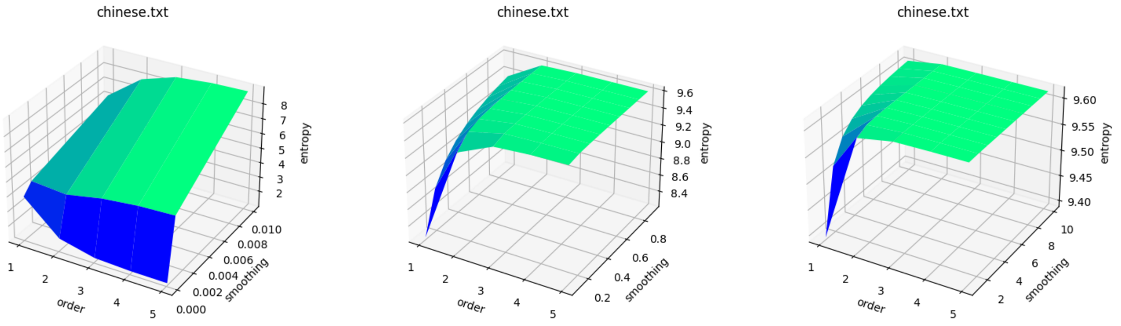
We found that, in all files we used, a bigger smoothing parameter always corresponded to a bigger entropy and that unless the smoothing parameter was very small (*0.0001 or 0.001*) the same could be found with the order.

These findings were only contradicted in ***example.txt***, the file provided by the teacher and the biggest file we tested, in which the entropy value always decreased for orders with values from 1 to 3, followed by an increase in values 4 and 5.

The text files used can be found in the ***example*** folder.

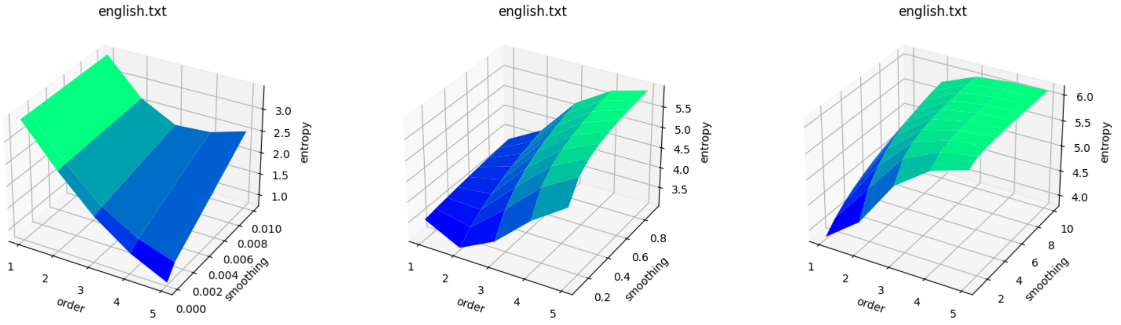


Entropy of ***example.txt*** when using various order and smoothing values



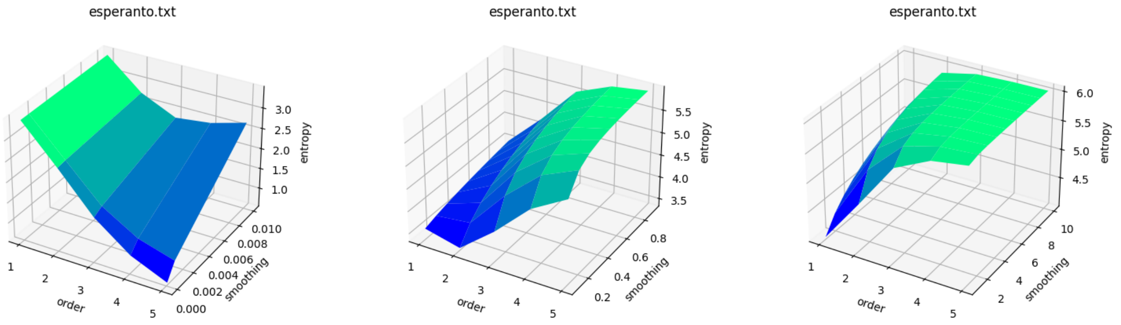
chinese.txt chinese.txt chinese.txt

Entropy of ***chinese.txt*** when using various order and smoothing values



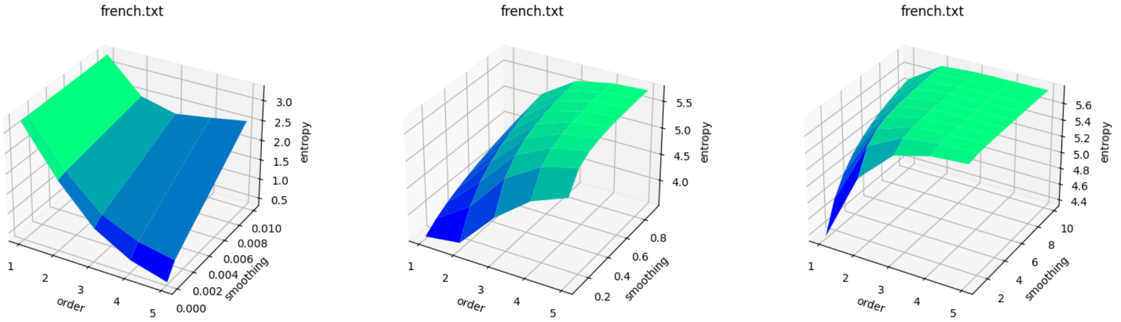
english.txt english.txt english.txt

Entropy of ***english.txt*** when using various order and smoothing values

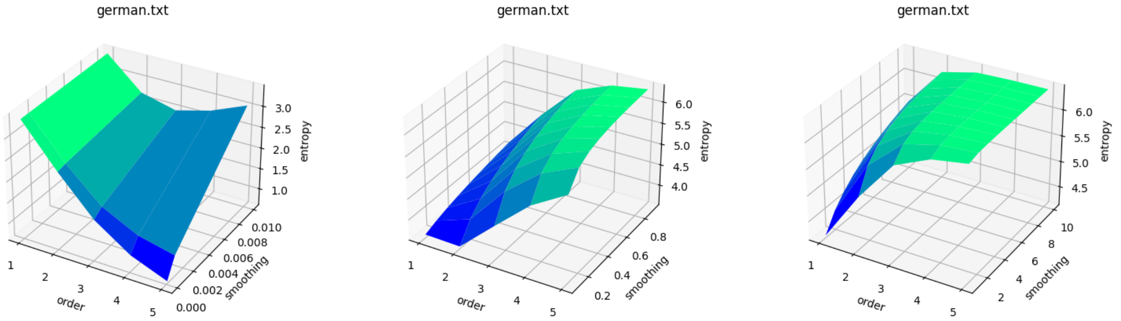


esperanto.txt esperanto.txt esperanto.txt

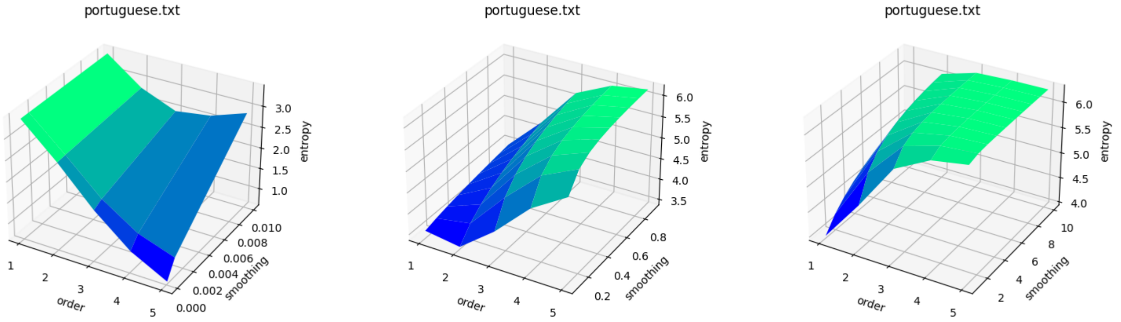
Entropy of ***esperanto.txt*** when using various order and smoothing values



Entropy of ***french.txt*** when using various order and smoothing values



Entropy of ***german.txt*** when using various order and smoothing values



Entropy of ***portuguese.txt*** when using various order and smoothing values

Due to it being a project specification we are also including a table of the values for a text file that was provided to us by the professors as an annex, and the full output file can be found in our repository as */report/result.txt*.

We also found that runtime is only affected by order and file length, both for input and output, as a bigger order and length cause slower runtimes. Even then, ***fcm.py*** using ***example.txt*** and order 1000 and ***generator.py*** using the previous arguments and length of 1000000 both take under 20 seconds to complete on our hardware.

When it comes to optimizing the output of ***generator.py*** we concluded that higher orders should be matched by lower smoothing in order to make better looking text. This parameter combination also causes generated text to have closer similarity to the text analyzed for model development.

For example, while order 1 works with smoothing larger than 1, order 4 barely works with smoothing 1 and works best at around smoothing 0.001, and order 50 only starts working at smoothing 0.000001.

Annex A: example.txt values

Order	Smoothing	Entropy
1	0.0001	3,31872
1	0.001	3,31873
1	0.01	3,31880
1	0.1	3,31943
1	0.2	3,32005
1	0.3	3,32065
1	0.4	3,32122
1	0.5	3,32178
1	0.6	3,32232
1	0.7	3,32286
1	0.8	3,32338
1	0.9	3,32390
1	1	3,32441
1	2	3,32922
1	3	3,33370
1	4	3,33795
1	5	3,34204
1	6	3,34598
1	7	3,34981
1	8	3,35353
1	9	3,35717
1	10	3,36073
2	0.0001	2,54760
2	0.001	2,54778

2	0.01	2,54917
2	0.1	2,55888
2	0.2	2,56756
2	0.3	2,57532
2	0.4	2,58252
2	0.5	2,58930
2	0.6	2,59576
2	0.7	2,60195
2	0.8	2,60793
2	0.9	2,61370
2	1	2,61931
2	2	2,66892
2	3	2,71109
2	4	2,74864
2	5	2,78287
2	6	2,81452
2	7	2,84409
2	8	2,87191
2	9	2,89824
2	10	2,92328
3	0.0001	1,99709
3	0.001	1,99855
3	0.01	2,00932
3	0.1	2,07412
3	0.2	2,12513
3	0.3	2,16763
3	0.4	2,20502

3	0.5	2,23881
3	0.6	2,26987
3	0.7	2,29875
3	0.8	2,32583
3	0.9	2,35138
3	1	2,37562
3	2	2,57064
3	3	2,71585
3	4	2,83355
3	5	2,93318
3	6	3,01983
3	7	3,09663
3	8	3,16565
3	9	3,22835
3	10	3,28578
4	0.0001	1,66264
4	0.001	1,66960
4	0.01	1,71676
4	0.1	1,94097
4	0.2	2,08665
4	0.3	2,19688
4	0.4	2,28766
4	0.5	2,36566
4	0.6	2,43446
4	0.7	2,49621
4	0.8	2,55237
4	0.9	2,60395

4	1	2,65170
4	2	3,00175
4	3	3,23177
4	4	3,40349
4	5	3,54015
4	6	3,65329
4	7	3,74952
4	8	3,83301
4	9	3,90655
4	10	3,97211
5	0.0001	1,44383
5	0.001	1,46509
5	0.01	1,59826
5	0.1	2,09290
5	0.2	2,35388
5	0.3	2,53352
5	0.4	2,67270
5	0.5	2,78698
5	0.6	2,88419
5	0.7	2,96888
5	0.8	3,04395
5	0.9	3,11139
5	1	3,17260
5	2	3,59062
5	3	3,84079
5	4	4,01703
5	5	4,15151

5	6	4,25926
5	7	4,34848
5	8	4,42415
5	9	4,48952
5	10	4,54681