Lab Work 1

Andr Pedrosa [85098], Filipe Pires [85122], Joo Alegria [85048]

Algorithmic Information Theory
Department of Electronics, Telecommunications and Informatics
University of Aveiro

October 12, 2019

Introduction

This report aims to describe the work developed for the first assignment of the discipline of 'Algorithmic Information Theory', explaining all the steps and decisions taken by us, and presenting the results we considered most relevant.

The programs implemented in C++ have the purpose of collecting statistical information about texts using Markov (finite-context) models, and of automatically producing texts that follows the models built.

Along with the description of the solution, we also discuss the effects of the variation of the programs' parameters and attempt to compare different types of texts by the amount of information they hold on average.

1. Information Model

Our first goal was to be able to predict the next outcome of a text source. To do this, we needed to take in consideration the dependencies between the characters of a text. The use of Markov models for the extraction of the statistical properties of a text was due to its value as an approach to represent these data dependencies.

The specific type of model that most interests us is called discrete time Markov chain or finite-context model. This model assigns a probability estimate to the symbols of the alphabet, according to a conditioning context computed over a finite and fixed number of past outcomes. More about this is explained in the description of this work assignment (*1*), containing the mathematical equations that served as the basis for our implementation.

1.1. Collecting Data

We decided to organize the program by several files, each with a different purpose, for good readability and to allow future modifications without the need of much refactoring - so we adopted a modular architecture strategy.

The file `fcm.cpp` serves as the base code for the command to be executed in order to generate a finite-context model, given one or several information source(s). By executing this command, the program is started and begins to call a set of functions that return an instance of our implementation of the Markov's model trained from the information source(s), and calculate the text entropy, as estimated by this instance. The `fcm` command has the following format:

```
$ ./fcm [-h] k alpha trainFile [trainFile ...]
```

Here, `-h` is the option that presents the manual for the command usage. Argument `k` is the value given for the size of the context. This context corresponds to a string with `k` characters, it is based on the several contexts produced and on the single characters that follow each one of them that the model is able to calculate the text's entropy. `alpha` stands for the value of the 'smoothing' parameter for estimating the probabilities of events. These events correspond to the occurrences of a character after a given context. And `trainFile` is, as the name indicates, the name of the file(s) that contain the text to be processed by the model.

First, the command executed is pre-processed and its arguments are collected and validated by the function *parseArguments()*, implemented in `argsParsing.cpp` and defined in `argsParsing.h`. The use of a header file is to establish an interface for the possibility of creating different implementations of the parsing functions. This is also visible in the implementation of the Model class in files `model.cpp` and `model.h`.

Once the arguments are validated, the program attempts to open the file(s) for reading through function *checkAccess()* and, in case of success, reads and parses its content. The program supports any file format as long as its content is plaintext. Below we present the actual implementation in C++ of the function responsible for parsing the information source file.

Variable `abc` contains the alphabet of the input file, updated everytime a new character is found. The function *parseFile()* creates a copy of this alphabet and a new alphabet that will contain the new found characters (if any). It then iterates over the file's content letter by letter, inserts each on both alphabets (if not already in them), updates the number of occurrences of each letter after the corresponding context and updates total number of contexts in each iteration. Once the end of file (EOF) is reached, the function calls *calcProbabilitiesAndEntropy()*.

```
void Model::parseFile(list<fstream*> &input) {
  [...]
  for (auto reader : input) {
      while (reader->get(letter)) {
          abc.insert(letter);
          if (context.length() >= ctxLen) {
              statsTable[context].nextCharStats[letter].count++;
              statsTable[context].stats.count++;
              totalContextsCount++;
              context = context.substr(1);
          }
          context += letter;
      }
  }
  calcProbabilitiesAndEntropy();
}
```

This solutions makes our model prepared to accept more than one information source (i.e. several input files). Although this was not a requirement, we knew this would make the program more robust and scalable.

1.2. Training the Model and Returning Text Entropy

Finally, as the file is parsed and the alphabet is built, `fcm` then builds the information table containing the statistics of the input text and calculates the estimated value for the entropy through the function *calcProbabilitiesAndEntropy()*. To build the information table we followed a dictionary approach where we have a first level dictionary where the key is the context string and the value is a structure (ContextStatistics) that contains a structure (Statistics) with statistics (number of occurrences and probability) of that context and a dictionary. This second layer dictionary has as key a letter of the alphabet and as value the statistics structure mentioned before, but this time the data is relative to appearances of a letter after a given context.

As all context's and letter's number of occurrences are calculated on the method *parseFile()*, on the *calcProbabilitiesAndEntropy()* method we calculate their probabilities. Context's probability is given by its number of occurrences divided by the sum of appearances of all contexts, in other words, the total number of contexts in all training data. Letter's conditional probability is obtained dividing the number of occurrences of a letter after a given context by the number of occurrences of that context.

To calculate these probabilities we iterate over the two layer dictionary. On the first layer of the dictionary we calculate probabilities related to contexts ($P(c)$) and while iterating over the second layer we calculate the context entropy (H_c). For a given context not all letters of the alphabet appear after it, to fix this we did a copy of the alphabet and removed the ones that appeared and set the their count to 0 and calculated the conditional probability.

The conditional probability mentioned is affected by a smoothing parameter which allows letters that did not appear after a context to have a higher than 0 probability.

Calculating all this prevent us from having to iterate over the entire table again to calculate the model entropy or the conditional probabilities for letters after a context.

The mathematical equations required for the calculation of the entropy are available in the document that describes the assignment.

The function *calcProbabilitiesAndEntropy()* is presented next for further analysis.

```
void Model::calcProbabilitiesAndEntropy() {
  [...]
  for (auto &it : statsTable) {
      contextStats = &it.second;
      contextCount = contextStats->stats.count;
      contextStats->stats.probability =
        (double)contextCount / totalContextsCount;
      set<char> abcCopy(abc);
      for (auto &it2 : contextStats->nextCharStats) {
          letter = it2.first;
          stats = &it2.second;
```

```
            charCount = stats->count;
            conditionalProb =
              (charCount + alpha) / (contextCount + alpha * abc.size());
            stats->probability = conditionalProb;
            Hc += conditionalProb * log2(conditionalProb);
            abcCopy.erase(letter);
        }
        for (char l : abcCopy) {
            conditionalProb = alpha / (contextCount + (alpha * abc.size()));
            contextStats->nextCharStats[l] = {0, conditionalProb};
            if (conditionalProb > 0) {
                Hc += conditionalProb * log2(conditionalProb);
            }
        }
        Hc = -Hc;
        entropy += contextStats->stats.probability * Hc;
        Hc = 0.0;
    }
}
```

2. Text Generator

The second part of the assignment was to developed a program for automatic text generation that follows the statistical model learned beforehand using a training text. To do this, we use `model.cpp` as a starting point and developed `generator.cpp`. This program, similarly to `fcm`, works as a command when executed. `generator` similarly to fcm starts by passing the information source(s) to the model, that internally will construct the model by calculating the probabilities of each character of the alphabet knowing that a context appened, with the same dataflow as the fcm program when processing the file. After the model and the calculations are complete, the program begins to generate text, starting with the text passed by the user and generating as much characters as the user intended.

The `generator` command has the following format:

```
$ ./generator [-h] k alpha beginSequence numChars \\
outputFile trainFile [trainFile ...]
```

Once again we have the `-h` option that presents the manual for the command usage. Arguments `k` and `alpha` are the same as the ones on the command `fcm`. The `beginSequence` argument asks the user to give a word or character sequence for the program to start off from; this is a need instrinsic to the way the solution works and must be the same length as the context lenght. The `numChars` argument tells the program how many characters are to be outputed. The `outputFile` is where the generated text will be written to. Finally the `trainFile` is, as the name suggests, the name of the file(s) to be processed by the model and used as training.

3. Results

In this chapter we discuss the results achieved from the final version of both tasks solutions. During development, we used randomly generated texts to test our code. However, for the analysis described here, we used two text files containing *The Bible* in plaintext, one written in English (`bible_en_v1.txt`) and the other in Portuguese (`bible_pt.txt`). The reason we chose the same text source translated in different languages was to evaluate the entropy of each language and compare them in terms of average quantity of information per character of the alphabet. These files suffered a minor pre-processing in order to make their formats as similar as possible.

3.1. Parameter Variation

We defined a few assumptions after considering the problem of determining the entropy of an information model and aimed to test them out once the program was completed. In this section

we explain these hipothesis and analyse their truthfulness with the aid of a graphic plotting the evolution of the text entropy with parameters k and alpha as variables. It is important to state that our assumptions are based on the interpretation of the mathematical formulas around the model implemented and that they are supposed to apply to texts of any size.

Taking a closer look at the formula for the overal entropy of the model (equation 1), we can gather that as the context probability decreases so does the value of the entropy. But what exactly affects the probability of a context? Assuming we start from an equal probability of occurring any of the existing contexts, the more contexts there are, the less probability there is of occurring a specific one. Also, for a given text source, the longer the context is (substring of fixed size from the text source), the more possible combinations of letters there are and, consequently, the more unique contexts appear on the given text. Taking this in consideration, we are able to establish that increasing the context size results in an increase of the total number of different contexts and, consequently, in a decrease of the probability of occurring each context and finally leading to a decrease in the final value for the model's entropy (see equation 2).

$$H = \sum_{c} P(c)Hc \tag{1}$$

$$> size(c) \Rightarrow < P(c) \Rightarrow < H \tag{2}$$

Our second hypothesis regarded the 'smoothing' parameter alpha. The idea behind this parameter is to tackle the issue of constructing the model and assigning zero probability to unseen events. By adding alpha, the character probabilities is uniformized and they never actually reach zero. As we studied the effects of the variable on the formula of conditional probabilities (see equation 3), we came to the conclusion that the larger its value, the bigger will be the model's entropy.

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

This was harder to reach as, at first sight, the effect of alpha is only relevant in a binary way, i.e. it affects the result if it is bigger than zero (the same way, no matter its value), it does not if not. However, by analysing the situation more carefully, we understood that, assuming that alpha is bigger than zero, the larger its value, the smaller is the bottom parcel of equation 3 and, consequently the larger the conditional probability is. From that point on, one can understand that the larger the alpha, the larger will be the entropy.

We developed a script in `Matlab` that runs `fcm.cpp` a defined number of times for the same source of information varying the two studied parameters in several combinations. This script collects the entropy values for each combination and then plots them in a line graph. Our

next step was to run the script for the file . . . . . , varying k between ..... and ..... , and alpha between ..... and ..... . Figure 1 shows the resulting plot.

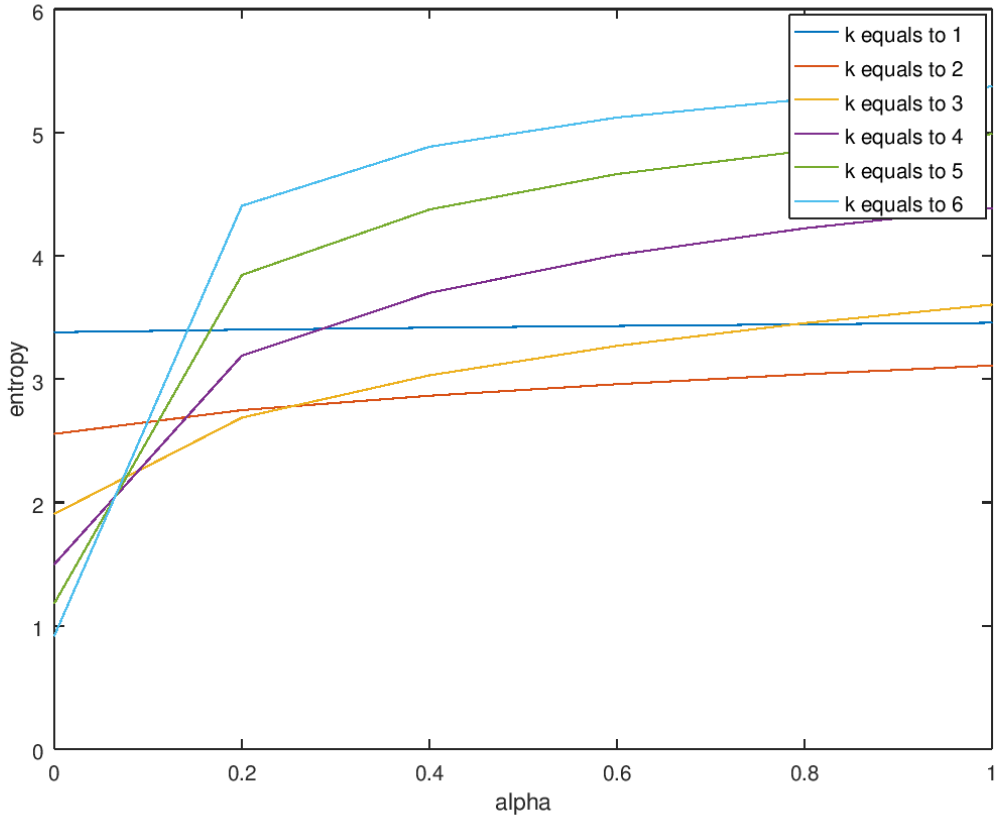

Figure 1: Entropy evolution in relation to alpha to different context sizes

grafico confirma, mas, para ks baixos, a segunda condicao nao se confirma pq, como o texto tao grande,

.....

3.2. Text Comparison

As a curiosity we thought on calculating the entropy to different languages to assert if languages differ in entropy and if so, what causes those variations. For this we used again the `bible_en_processed.txt` in addition to the `bible_pt_processed.txt` to compare the English language to the Portuguese language. We purposefully used the same document in different languages to maintain the message itself, so this way we can compare the languages entropies to a given message, normalizing the process. The results can be observed in `Figure 2` and `Figure 3`, and as expected the entropy behavior is the same as already explained in sec-

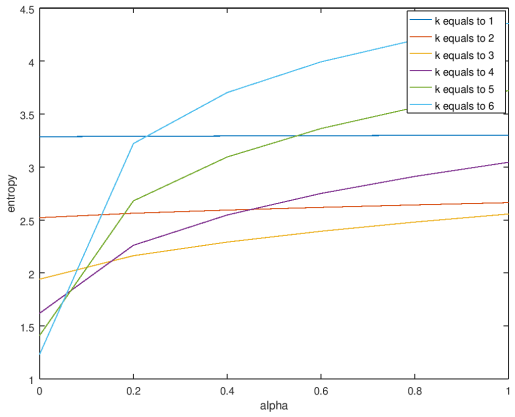

Figure 2: English Bible Entropy

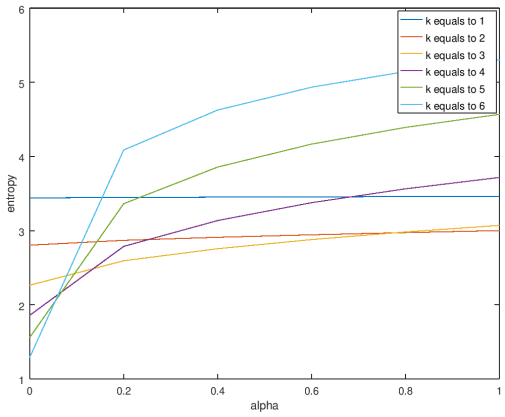

Figure 3: Portuguese Bible Entropy

tion `Parameter Variation`. Taking a closer look, observing the entropy values to alpha equal to 0, we see a small difference between the two, where the values are:

3.3. Generator's Response to Parameter Variations

Lorem ipsum ...

Conclusions

Lorem ipsum ...

References

1. Armando, *AIT: Lab Work no.1*, University of Aveiro, 2019/20.