Lab Work 1

André Pedrosa [85098], Filipe Pires [85122], João Alegria [85048]

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

October 7, 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 (***?***), 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 command to be executed in order to generate a finite-context model, given one information source. 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, and calculate the text entropy, as estimated by this instance. The `fcm` command has the following format:

```
$ ./fcm.cpp [-h] k alpha fileName
```

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 `fileName` is, as the name indicates, the name of the file that contains 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 open the file 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(fstream &reader) {
  char letter;
  string context;

  set<char> oldAbc(abc);
  set<char> newAbc;

  while (reader.get(letter)) {
      abc.insert(letter);
      newAbc.insert(letter);

      if (context.length() >= ctxLen) {
          statsTable[context].nextCharStats[letter].count++;
          statsTable[context].stats.count++;
          totalContextsCount++;
          context = context.substr(1);
      }

      context += letter;
  }

  set<char> lettersNotChanged;
  set_difference(
      oldAbc.begin(), oldAbc.end(),
      newAbc.begin(), newAbc.end(),
      inserter(lettersNotChanged, lettersNotChanged.begin())
  );

  calcProbabilitiesAndEntropy(lettersNotChanged);
}
```

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()*.

.... explain Stats, nextCharStats and contextStats ....

The mathematical equations required for the calculation of the entropy are also available in the document that describes the assignment. The actual work was to implement these formulas and apply them.

.... explain entropy calculation .... how do we retrieve the probabilities to apply the formulas ? ... how do we iterate over the tables ?

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

```
double Model::getModelEntropy() const{
  string context;
  map<char, int> occurMap;
  int curOccur;
  int contextCountTotal = 0;
  map<string, int> contextCount;
  double conditionalProb;
  double H = 0.0;
  double Hc = 0.0;

  for (auto &it : occurTable) {
    context = it.first;
    occurMap = it.second;
    for (auto &it2 : occurMap) {
      contextCount[context] += it2.second;
      contextCountTotal += it2.second;
    }
  }

  for (auto &it : occurTable) {
    context = it.first;
    occurMap = it.second;
    for (auto &it2 : occurMap) {
      curOccur = it2.second;
      conditionalProb = (curOccur + alpha) /
                        (contextCount[context] + alpha * abc.size());
      Hc += (conditionalProb * log2(conditionalProb));
    }
    H += -(((double)contextCount[context] / contextCountTotal) * Hc);
    Hc = 0.0;
  }

  return H;
}
```

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 `fcm.cpp` as a starting point and developed `generator.cpp`. This program, similarly to `fcm`, works as a command when executed. `Generator` ....

....

The `generator` command has the following format:

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

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 `trainFile` is, as the name suggests, the name of the file to be processed by the model and used as training. The `outputFile` is where the generated text will be written to. The `startSequence` 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. Finally we have the `numChars` argument, that tells the program how many characters are to be output.

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.

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

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 **??** shows the resulting plot.

[width=]

Figure 1: ....

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

....

3.2. Text Comparison

Lorem ipsum ...

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.