Lab Work 1

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

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

September 24, 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. Architecture

The file `fcm.cpp` serves as the command to be executed in order to generate a finite-context model, given one information source. It is the *main()* function. We decided to organize the program by several files, for readability purposes and to allow future modifications without the need of much refactoring by following a modular architecture. So, by executing the `fcm` command, it calls a set of functions from the remaining files and creates an instance of the Markov model. It then provides the entropy of the information source, as estimated by the instance.

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`. 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. `alpha` stands for the value of the 'smoothing' parameter for estimating the probabilities of events. And `fileName` is, as the name indicates, the name of the file to be processed by the model.

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.
... explain the steps and decisions taken ...

```
void Model::fileParser(){
  map<string, map<char, int>> relFreq;
  char letter;
  string context;

  while(reader.get(letter)){
    if(abc.find(letter) == abc.end()){
      abc.insert(letter);
    }
    if(context.length()>=ctxLen){
      if(relFreq.find(context) != relFreq.end()){
        map<char,int> nextChar = {{letter,1}};
        relFreq.insert(pair<string,map<char,int>>(context,nextChar));
      }else{
        if(relFreq[context].find(letter) == relFreq[context].end()){
          relFreq[context].insert(pair<char,int>(letter,1));
        } else{
          relFreq[context][letter]=relFreq[context][letter]+1;
        }
      }
      context=context.substr(1,2);
      context+=letter;
    }else{
      context+=letter;
    }
  }
  for(auto &pair: relFreq){
    if(pair.second.empty()){
      relFreq.erase(pair.first);
    }
  }
  occurTable=relFreq;
}
```

1.2. Entropy

Finally, as the file is parsed and the data is structured in a table, `fcm` then calculates the estimated value for the entropy. The mathematical equations required for this calculation are also available in the document that describes the assignment. The actual work was to implement these formulas and apply them. Our solution was an iterative function that would calculate the probabilities of each context and of each event and applying the equations on them and summing the results. This function is presented below 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. Generator

Lorem ipsum ...

3. Results

Lorem ipsum ...

3.1. Parameter Variation

Lorem ipsum ...

3.2. Text Comparison

Lorem ipsum ...

Conclusions

Lorem ipsum ...

References

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