universidade de aveiro
deti departamento de electrónica, telecomunicações e informática

Report lab work 2

Algorithmic Information Theory

Pedro Figueiredo - 97487
Renato Dias - 98380

Index

1. Introduction

The objective of this work was to build 3 programs in C++:

- A program with objective is to report the estimated number of bits required to compress a certain text under analysis, *t*, using the model computed from another text that represents the class *ri* (for example, representing a certain language);
- A language recognition system, based on the first program, that, from a set of examples from several languages (the ri), provides a guess for the language of a text t;
- An application that can process text containing segments written in different languages. The goal is to return the character position at which each segment starts, as well as the language in which the segment is written.

The language recognition was trained with 17 languages models (we added some extra languages in testing, in order to test the outcome when languages who were not included in the training are used):

- Catalan
- Dutch
- English
- French
- German
- Greek
- Hungarian
- Italian
- Norwegian
- Polish
- Portuguese
- Romanian
- Russian
- Serbian
- Slovak
- Spanish
- Ukrainian

2. Compression Model

As first program we implemented a program that estimates the number of bits required to compress a text file, *filename_t*, using a model computed from another text file, *filename_ri*. We implemented a compression algorithm based on word-level modeling and probability estimation. It uses a sliding window approach to extract words of a specified length, *word_length*, from the reference text, *riFile*, and builds a model based on the frequency and probability of occurrence of each word. In other words we use our copy model from the first lab to build the model.

2.1 Command Line Arguments

The program takes in 7 command line arguments (explained in order): The **name of the text file with a text representing the class ri** (in our case represents a certain language), the **name of the text file with the text under analysis**, the **length of a word** to consider, a **threshold value, α**, used to calculate the probability associated to each word ($P(hit) \approx (Nh + \alpha)/(Nh + Nf + 2\alpha)$), **a minimum probability value** and a **maximum number of consecutive failures**, 2 values that serve as failsafes (when they are hit, the next symbol in the word structure is considered obsolete and replaced by a new symbol) and finally the **cost of encoding a symbol literally**. This value will be added to the total number of bits if a certain word does not exist in the word Map.

2.2 Data Structures

Talking about the data structures we used, in this program we just used one: *WordData*, a data structure that contains data about a word (number of hits, number of fails, estimated probability (calculated with the formula $P(hit) \approx (Nh + \alpha)/(Nh + Nf + 2\alpha)$), the next symbol that occurs after the word and the number of consecutive fails).

2.3 *calculateCompressionBits* Function

This function is the core of the program and has the following workflow: open the reference text file and target text file and read the contents of both files into strings. Then iterates over the sequence of characters in the input file, taking each substring of a length defined as a command line argument. For each substring, the program checks if the substring is in the word map. If it is not, the program initializes a new entry in the map with a hit count and a fail count of 0, an estimated probability of 0.5 (P(hit) ≈ α/2α), and the next symbol that occurs after the substring. If the substring is already in the map, the program updates the hit and fail counts based on whether the next symbol after the substring matches the expected next symbol. The program then checks if the next symbol needs to be updated, by checking the number of consecutive failures or if the estimated probability is below the minimum threshold. The program then updates the estimated probability of the word using the hit count, fail count, and smoothing factor, and updates the next symbol that occurs after the word. Passing to target text, it iterates through each character of the target text and extracts words of length *wordLength* and checks if the word exists in the *wordMap*. If exists, adds the negative logarithm of the word's probability to total bits variable and if the word does not exist in the *wordMap*, it adds the *literalCost* to variable *totalBits*.

2.4 Results

The next results were obtained using the file “portuguese.txt” as a text representing the class *ri* and the file “por.txt” as text under analysis.

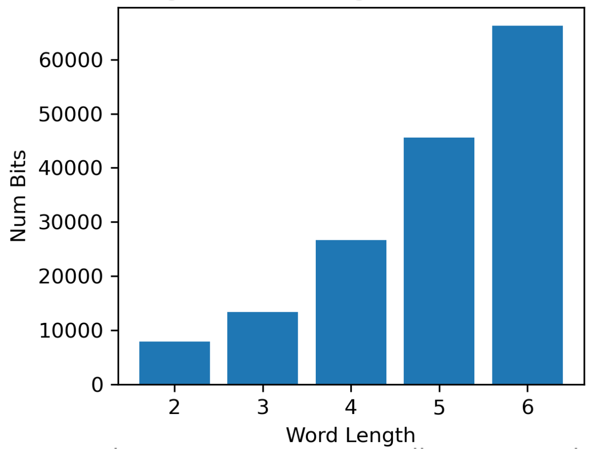


Fig. 1 - Word Length vs. Num Bits

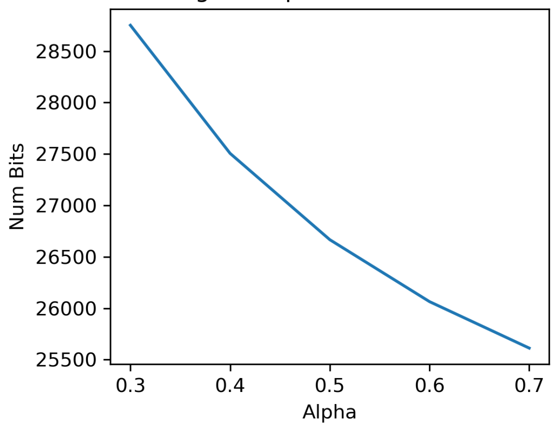


Fig. 2 - Alpha vs. Num Bits

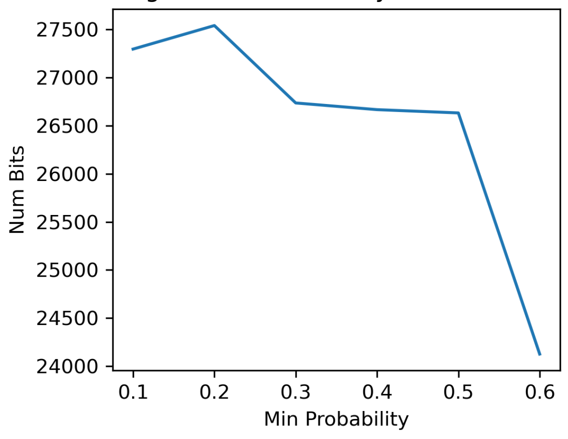


Fig. 3 - Min Probability vs. Num Bits

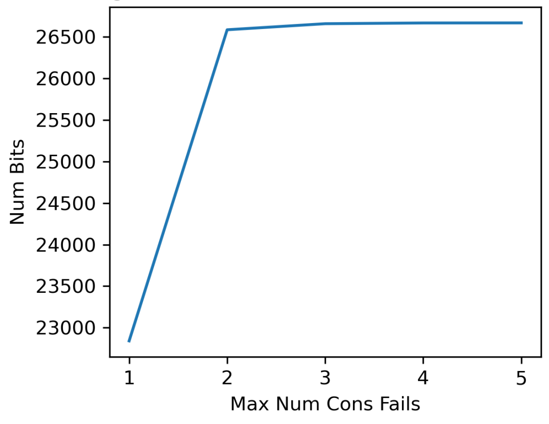


Fig. 4 - Max Num Cons Fails vs. Num Bits

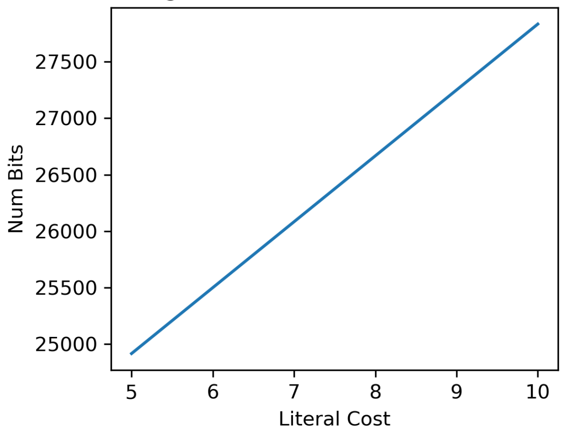


Fig. 5 - Literal Cost vs. Num Bits

In the figures above, we can see the changes that different values and different parameters can cause to the compression model.

Most parameters, from alpha to literal cost (figures 2 through 5) do not cause massive swings in the number of bits used, with the number always being in the region of 25000 bits.

When we look at word_length, this is the parameter that offers by far the largest swing in results. The extra compression of word_length = 2, when compared to word_length = 6, is in the order of 50000 bits.

We can justify this discrepancy as the smaller word_length correlates to higher chances of repetition, which in a copy model based system translates to larger compression levels.

3. Language Recognition System

This program aims to identify the language of a given text file based on a set of language models built from example texts. The program uses statistical language modeling techniques to estimate the likelihood of a given text belonging to a particular language.

3.1 Command Line Arguments

In terms of command line arguments, these are almost the same as the compression model except the name of the text file with a text representing the class ri. In this case we pass as command line argument the **path to the directory containing the language examples**. The rest of the command line arguments remain as above.

3.2 Data Structures

Besides the data structure explained above, we use another data structure called *LanguageModel*, containing the language name, a mapping of words to *WordData* objects representing the language model and the total number of bits required to encode the language model.

3.3 *buildLanguageModel* Function

This function constructs a language model from a given reference text file, *riFile*. It takes the reference file path, the language name (extracted from a file containing all languages available), word length, alpha value, minimum probability, and maximum number of consecutive fails as input parameters. The function reads the reference text file, iterates through the text to process each word, and updates the word map of the language model accordingly. It calculates the probability of each word based on the hits, fails, and other parameters like we do in the compression model program. Finally, it returns the constructed language model. This language model is a data structure containing the name of the language, the wordMap that contains the words and data associated with each word like hits, fails and so on and also the total number of bits.

3.4 *recognizeLanguage* Function

The recognizeLanguage function performs language recognition based on the given target text file, tFile, and a vector of language models. The function reads the target text file, iterates through it using a sliding window approach, and calculates the total number of bits required to compress the text using each language model. It then compares the scores of each language and determines the recognized language as the one with the minimum score (total bits).

3.5 Results

The results above were obtained using this parameters values: word_length=4, alpha=0.5, min_prob=0.4, max_num_cons_fails=3 and literal_cost=8.0

Example Language	Language Guess
Brazilian	Russian
Catalan	Catalan
Danish	Russian
German	Russian
English	Russian
French	Russian
Galician	Portuguese
Greek	Greek
Hungarian	Russian
Italian	Russian
Dutch	Russian
Polish	Russian
Portuguese	Portuguese
Romanian	Russian
Slovenian	Russian
Spanish	Spanish
Serbian	Russian
Slovak	Russian
Ukranian	Russian

Now we will test the program for different values of word_length because as we can see for the first program, this parameter is the one that has the most influence in the number of bits.

For word_length=2:

Example Language	Language Guess
Brazilian	Portuguese
Catalan	Catalan
Danish	Norwegian
German	Dutch
English	English
French	French
Galician	Spanish
Greek	Greek
Hungarian	Hungarian
Italian	Dutch
Dutch	Dutch
Polish	Polish
Portuguese	Portuguese
Romanian	Romanian
Slovenian	Slovak
Spanish	Spanish
Serbian	Russian
Slovak	Slovak
Ukranian	Russian

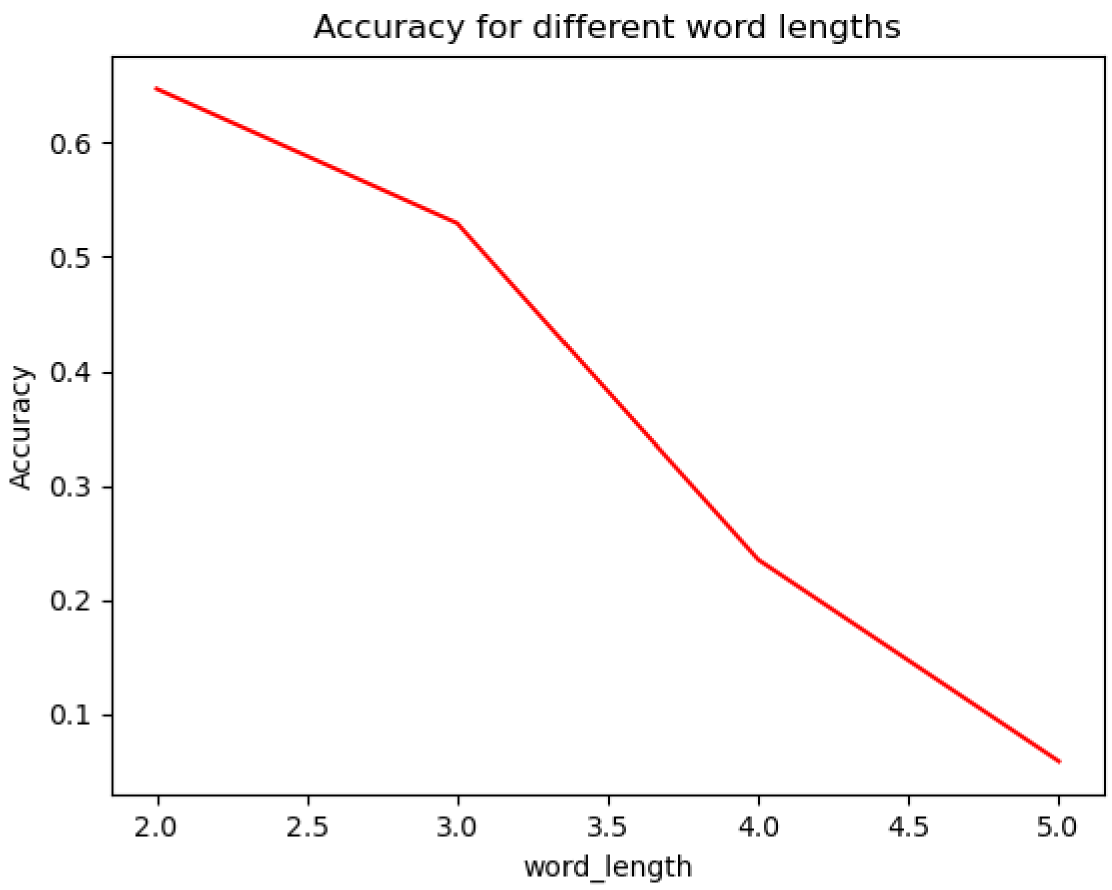
For word_length=3:

Example Language	Language Guess
Brazilian	Portuguese
Catalan	Catalan
Danish	Dutch
German	Dutch
English	English
French	Portuguese
Galician	Portuguese
Greek	Greek
Hungarian	Hungarian
Italian	Portuguese
Luxembourgish	German
Dutch	Dutch
Polish	Polish
Portuguese	Portuguese
Romanian	Romanian
Slovenian	Slovak
Spanish	Spanish
Serbian	Russian
Slovak	Russian
Ukranian	Russian

And finally for word_length=5:

Example Language	Language Guess
Brazilian	Russian
Catalan	Russian
Danish	Russian
German	Russian
English	Russian
French	Russian
Galician	Russian
Greek	Greek
Hungarian	Russian
Italian	Russian
Luxembourgish	German
Dutch	Russian
Polish	Russian
Portuguese	Russian
Romanian	Russian
Slovenian	Russian
Spanish	Russian
Serbian	Russian
Slovak	Russian
Ukranian	Russian

The accuracy results for the different values of word_length, can be seen in the graph below (accuracy in the y-axis and word_length in the x-axis:



We can conclude that the accuracy values have a tendency to decrease as word_length increases. This is due to the fact a larger sliding window means we are likely to find combinations we've never seen before, creating more errors.

4. Segmented Language Recognition System

The last program is a language recognition tool that is designed to identify the language of different segments. It uses the same language model built from a directory of language examples we used in the second program to perform the recognition process. The command line arguments are the same as for the second program.

4.1 Data Structures

For this program, we use 3 data structures: *wordData*, explained in the first program section, *LanguageModel*, explained in the second program section and a new one called *Segment*, that represents a language segment within the text file and contains the starting position of the segment in the text file, the ending position of the segment in the text file and also the recognized language of the segment.

4.2 *buildLanguageModel* Function

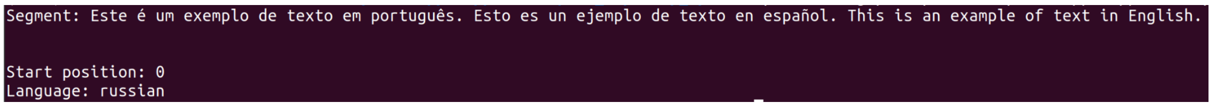
This function is similar as we do for the second program and the objective is to build the language model from a given language example file, *riFile*. The function reads the content of the language example file and constructs the language model using the provided parameters like *wordLength, alpha, minProb and maxNumConsFails*. It returns the constructed *LanguageModel* data structures.

4.3 *processTextSegments* Function

The *processTextSegments* function is responsible for dividing the input text into segments and determining the language for each segment. The function reads the content of the text file, tokenizes it into segments, and calculates the scores for each language model. Based on the scores, it determines the recognized language for each segment and returns a vector of *Segment* structures.

4.4 Results

We could not obtain any results for this program due to a little error concerning segments identification as we can see in the next figure. However we believe our language recognition tool demonstrates the process of building a language model from a directory of language examples and utilizing that model to identify the languages of different segments within a text file.



```
Segment: Este é um exemplo de texto em português. Esto es un ejemplo de texto en español. This is an example of text in English.


Start position: 0
Language: russian
```

Fig.7 - Bad segment identification issue

4.5 Use of Finite Context Models

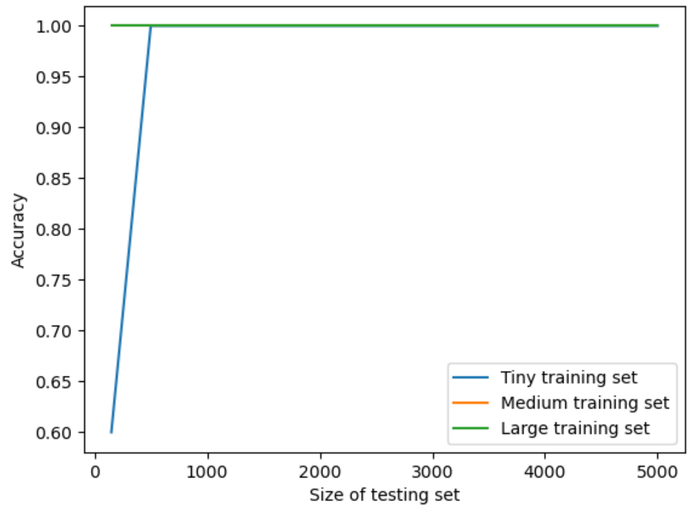
Even though we didn't end using them, we explored wayd to incorporate finite-context models into the program and provide probabilities for non-hit symbols:

- Modify the *WordData* data structure to include an *unordered_map<char, int>* to store the count of symbols encountered after each word. This map will represent the finite-context model for each word;
- Incorporate the finite-context model in the function *buildLanguageModel* by after reading each word from the input text, update the symbol count in the finite-context model for that word;
- Update *processTextSegments* function in order to use the finite-context model and calculate probabilities for non-hit symbols (incorporate the finite-context model by checking the symbol count for each word, this inside the loop where the word probabilities are calculated)

5. Accuracy as function of text size

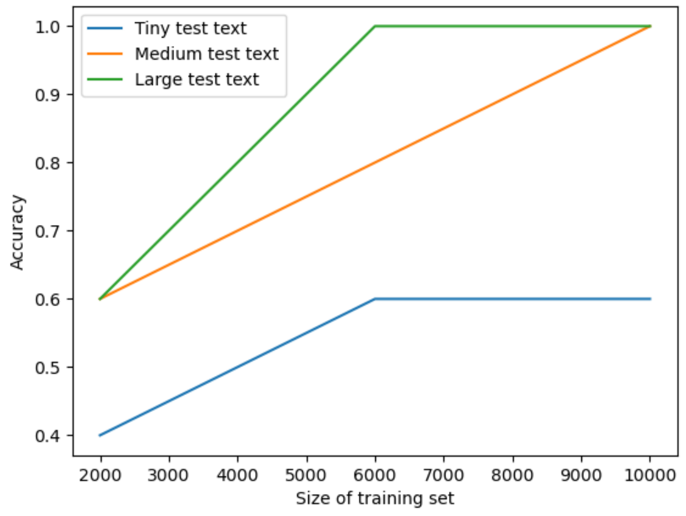
In order to perform the computations for this section, we created 3 text sizes, both for training the model and for testing it (tiny, medium and large).
Due to time constraints,we only performed these tests for 5 languages:

Fig 8 - Accuracy for different sized training sets



We can see that only the tiny test set has errors. That can be justified as the smaller may have words or references that the training set may have missed. The large training set was used for the results shown above.

Fig 9 - Accuracy for different sized training sets



There is a lot more variance in the results for different sized training. The noticeable trend here is that as the training sets grow larger, the accuracy improves, as expected.

6. Conclusion

As the work described in this report was split in 3 major parts, so too will we split our conclusion.

In regards to the compression model, we believe we achieve a solid level of efficiency, and it represents a good progress on top of the work previously developed in the context of the class.

As for the language recognizer, we believe we achieved a somewhat decent level of results, considering we didn't use a huge amount of training data for the language models we used. Still, an accuracy level of around 35% makes it clear there is much room for improvement here.

The progress of the segment language recognizer was curtailed by time constraints and the errors found in it's respective section. The section could also have made use of finite context models. However, we ended up running out of time to use them. As such, we did not make use of the model.