Text Similarity

Mestrado em Engenharia Informática

Universidade de Aveiro

DETI - Aveiro, Portugal

UC 40752 - Teoria Algorítmica da Informação
2022/2023

José Trigo 98597
Pedro Monteiro 97484
Eduardo Fernandes 98512

Text Similarity

May 30, 2023

Contents

1 Introduction

Algorithmic Information Theory (AIT) is a field of study that focuses on the representation, communication, and processing of information in efficient and concise ways. Data compression, a cornerstone of AIT, aims to minimize the size of data while preserving its original structure and meaning. This reduction in size is crucial for efficient storage, transmission, and processing of data in various applications such as multimedia, data communications, and data mining. One promising approach to achieve data compression is through the use of copy models.

Copy models for data compression capitalize on the inherent self-similarities within data sources. By identifying and exploiting these similarities, we can efficiently represent data through references to previously occurring parts with minor modifications.

We will discuss the estimation of probabilities involved in this process, addressing the challenge of assigning probabilities to previously unseen events using a smoothing parameter. The report will also touch upon how these probabilities can be utilized to estimate the amount of information required to represent a new symbol, ultimately driving the effectiveness of data compression.

Furthermore, we will create a language recognition system, based on copy models, and an application that will be able to process text containing segments written in different languages.

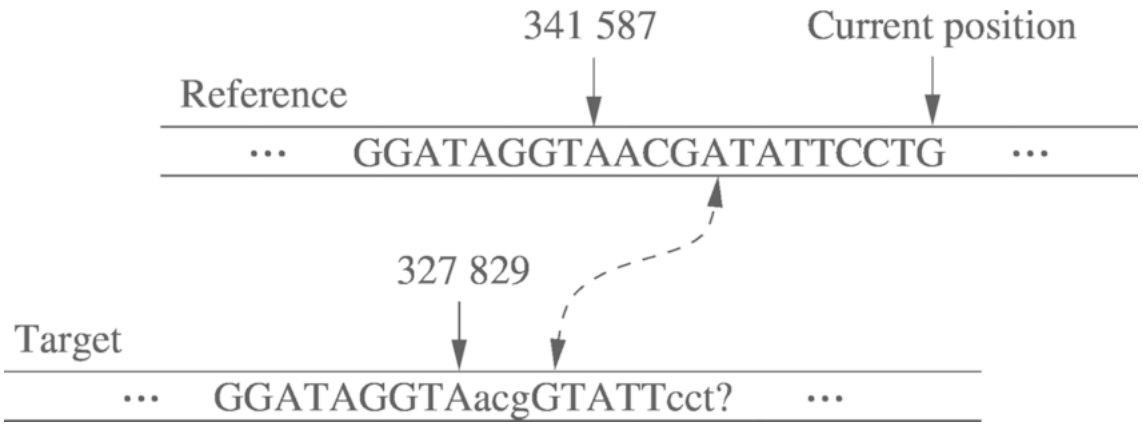

Figure 1: Copy Model Example.

2 Objectives

This assignment focuses on the similarities between texts, the first objective being to build a *lang* program that will accept two files: the first with a text representing a certain language, and the second a text that will be analyzed. This whole process will be done using the copy models used in the first assignment.

Next, based on the *lang* program, the *findlang* program was implemented, which works as a language recognition system, that is, given a set of texts, it tries to predict the language of the text under analysis. For the implementation of this project, and later for its study, about 20 different languages were used.

In addition to these two programs, an application was also built, *locatelang*, which is capable of processing text segments written in different languages. This application should return the character position at which each segment starts, as well as the language in which the segment is written.

3 Understanding the Copy Model

3.1 What is a Copy Model?

A copy model is a method used to predict the next character in a sequence based on the previously observed sub-sequences of a certain length, referred to as k-strings. By analyzing the frequency and positions of these k-strings, the copy model attempts to identify patterns that can be used to estimate the probability of a specific character following a given k-string. In the context of this project, the copy model is employed to predict the next characters in a DNA sequence.

3.2 Factors Affecting Copy Model Performance

The performance of a copy model depends on several factors, including the choice of k, the fail threshold, and the input data. A larger value of k may result in a higher accuracy of predictions, as it allows the model to capture more information about the patterns in the sequence. However, a larger k may also increase the computational complexity and memory requirements of the model.

The fail threshold is another important parameter that can impact the performance of the copy model. A lower fail threshold may improve the accuracy of predictions by ensuring that positions with a high number of failures are quickly removed from consideration. However, a lower threshold may also increase the likelihood of removing potentially useful positions.

Finally, the nature of the input data can also affect the performance of the copy model. If the input data contains clear and consistent patterns, the model is more likely to make accurate predictions. In contrast, if the input data is highly variable or lacks discernible patterns, the model may struggle to make accurate predictions.

4 Implementation

In this section, we will address how we perform our implementation as well as organize the code and exercise files. The functions, structures used, and the strategies applied for each of the algorithms will also be discussed.

4.1 Organization

The image below shows the structure of the project. In the **src** folder are all the files related to the developed code, the *lang* program, the language recognition system, *findlang*, and the *locatelang* application.
In the **examples** folder it is possible to find several text files that were used to carry out tests.
Furthermore, **README.md** describes how to run the program and finally, in the **report** folder there is this report that describes what was done during the development of the entire project.

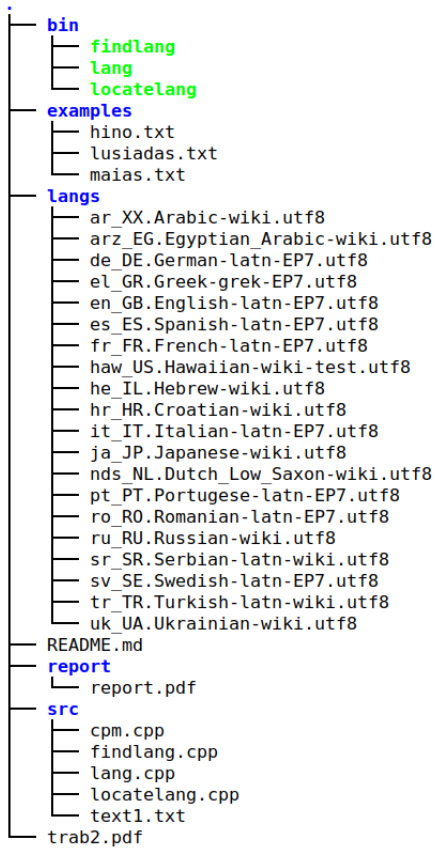

Figure 2: Project Repository Structure.

4.2 Developed programs

For this project, as previously mentioned, three programs were developed. The first one, *lang.cpp*, uses a copy model to build the model from the reference text and to analyse the target text . This program estimates the number of bits required to compress the target text, using the model computed from the reference text.
The second, *findlang.cpp* is a program designed to build a language recognition system. It takes a set of examples from different languages (ri) as input and uses this information to guess the language in which a given text (t) was written.

Finally, the *locatelang* program is designed to process text segments written in different languages and provide information about those segments. Specifically, it returns the character position at which each segment starts and identifies the language in which each segment is written. In the next sections we will address each of the problems separately.

4.3 Text compression(*lang*)

4.3.1 Data structures

To achieve the goals of the lang program we kept the structure of the copy model from the previous assignment that uses the following structures:

- *chunks_positions*: map of k-strings to their positions in the file;
- *chunks_vector*: vector of chunks of length k;
- *positions_count*: map containing the number of fails for each position;

4.3.2 Strategy addressed

The following list explicit the strategy addressed in the copy model:

1. Get the inline command arguments for the **reference file**, the **target file**, the size of the k-string, **k**, the value of **alpha** and the **fail threshold** (reference_file, target_file, k, alpha, fail_threshold respectively);
2. Read the content of the reference and target files;
3. Count the number of distinct characters in the reference data;
4. Iterate the data from reference file and add the current k-string to the *chunks_vector* as well as the position in the *chunks_positions* map;
5. Use the model from reference text to analyse the target text;
6. Get the predicted character by getting the first position in the vector of positions for that chunk, going to that position in the *chunks_vector* and retrieving the character immediately after the k-string;
7. Verify if its a hit or fail:
    - If its a hit, increment the hits counter;
    - If its a fail, increment the fails counter and deletes the current position if the fail counter is equal or bigger than *fail_threshold*;
8. Lastly, calculate the probability of a correct prediction and update the estimated total number of bits.

One thing to notice is that this model only starts to increase the hits/fails counters when the first repeated k-string appears.

4.3.3 Read the text file

In order to read the text file we created a function called *read_file()* that reads the text in the file and store it in a string.

4.3.4 Count the number of distinct characters

Similar to the copy model of the previous assignment, which was used to compress DNA, where there were 4 distinct symbols, the same technique will be used here to estimate the number of bits needed. For this, instead of $\log_2 4$, the different characters of the used alphabet, N, were calculated. This value is then used for $\log_2$ in the calculation of the estimated number of bits needed.

The following code snippet demonstrates the function that retrieves the distinct characters from a file. It utilizes a set data structure to store unique symbols, and finally returns the size of the set

```
// count the number of distinct characters in a string
int count_distinct_chars(const string &str)
{
    std::set<char> distinct_chars(str.begin(), str.end());
    return distinct_chars.size();
}
```

4.3.5 Store the k-string's in the data structures

In this program, the model for the reference text, chunks_positions and chunks_vector, will be created, which will be later used to analyze the target text. This logic is done inside the *main* function of the program. It iterates each k-string and adds each one to the map with the current position. If the chunk is already in the map, only adds the position to the vector associated.

In this step, it's very important to pay attention to the estimation of the total number of bits, since if only new sequences are found, that is, that are not yet in the map, it is necessary to add $\log_2 N$ to the estimated number of bits, where N is the number of different characters in the alphabet, as already mentioned. This decision was made to adapt in the presence of symbols not seen in the reference because it worked well in the previous copying model, having obtained good results, but other ways could be considered such as renormalizing the alphabet by reducing or augmenting.

Also stores the chunks in the *chunks_vector* so that, in the future, it can get the next character of a chunk for a certain position.

The code for the section addressed is below.

```
for (int i = 0; i < data.size() - k; ++i){

        // extract k length chunk from the file and add it to the chunks vector
        std::string current_chunk = data.substr(i, k);
        chunks_vector.push_back(current_chunk);
        //cout << "Current chunk: " << current_chunk << endl;

        // if the current chunk is not in the map add it to the map
        if (chunk_positions.find(current_chunk) == chunk_positions.end())
        {
            chunk_positions[current_chunk] = std::vector<int>{i};
            estimated_bits += log2(N);
            continue;
        }

        // add the position of the current chunk to the map
        chunk_positions[current_chunk].push_back(i);
}
```

4.3.6 Get the predicted character and modify the counters

As already mentioned before, we get the predicted character by getting the first position in the vector of positions for that chunk, going to that position in the *chunks_vector* and retrieving the character immediately after the chunk.
To get the actual character, we go to the position of the current chunk in the full text string, returning the character immediately after the current k-string.

After getting this two characters, we compare them in order to know if we should increase the hits counter or the fails counter.

In order to improve the results, we have implemented a *fail_threshold* which restricts the number of allowed failures at each position. Specifically, if a position associated with a certain k-string has a number of failures equal to or exceeding the *fail_threshold*, that position is removed and replaced by another position which becomes the first one in the vector of positions for that k-string.

4.3.7 Calculate the probabilities

The calculation of the probabilities happens every time the hits/fails counters are updated. The formulas that we used are the ones provided by the teachers in assignment.

```
// calculate the probability of a correct prediction and
// update the estimated total number of bits
prob = (hits + alpha) / (hits + fails + 2 * alpha);
estimated_bits += -log2(prob);
```

4.4 Language recognition system (*findlang*)

4.4.1 Data structures

As *findlang* is based on the code of the previous program *lang*, it maintains its data structures. This meaning that it has the structures below:

- *chunks_positions*: map of k-strings to their positions in the file;
- *chunks_vector*: vector of chunks of length k;
- *positions_count*: map containing the number of fails for each position;

4.4.2 Strategy addressed

As it was said before, the *findlang* is based on the last program. The main change in this version lies in its inclusion of multiple texts, which function as models for the target text. The following list explicit the strategy addressed in this exercise:

1. Get the inline command arguments for the **reference directory**, the **target file**, the size of the k-string, **k**, the value of **alpha** and the **fail threshold** (reference_directory, target_file, k, alpha, fail_threshold respectively);
2. Read the content of target file;
3. Iterate through each file of the directory;
4. Read the content for each file and count the number of distinct characters in the reference data;

5. Iterate the data from reference file and add the current k-string to the *chunks_vector* as well as the position in the *chunks_positions* map;
6. Use the model from reference text to analyse the target text;
7. Get the predicted character by getting the first position in the vector of positions for that chunk, going to that position in the *chunks_vector* and retrieving the character immediately after the k-string;
8. Verify if its a hit or fail;
9. Calculate the probability of a correct prediction and update the estimated total number of bits;
10. Repeat the process for all files in the directory provided.

In the end we should obtain the estimated number of bits for the target data in each language. The language with the lowest estimated bit count is likely the one in which the target text was written.

4.4.3 Iterate through the files directory

As the purpose of this model is to calculate the estimated number of bits for each of the reference models, the *main* function contains a while loop that allows iterating over each of the files in the given directory. Thus, the model is generated for each language, and consequently, the number of bits is calculated.

4.5 Text processing (*locatelang*)

4.5.1 Data structures

Some of the previous structures were maintained in this program. For this reason, just new ones are explained below:

- *CopyModel*: this struct is a language model based on copying chunks of text. It includes a map of chunk positions, a vector of chunks, and the number 'N' of distinct characters;
- *FiniteContextModel*: this struct is a language model based on counting occurrences of chunks and their following characters. It includes two maps for chunk counts and symbol counts, and the number 'N' of distinct characters;
- *chunk_counts*: map where the outer map's keys are string chunks of fixed length from the text, and the values are inner maps. These inner maps store each unique character that follows the chunk as the key, and the frequency of each character as the value. So, it essentially keeps track of how often each character follows each context;
- *symbol_counts*: map that uses chunks of fixed length as the key and the frequency of their occurrence as the value. It records the total count of each chunk in the text, which is used to calculate the probability of the next character following a specific chunk;
- *language_models*: map that associates each reference file name to a corresponding CopyModel;
- *finite_context_models*: map that associates each reference file name to a corresponding FiniteContextModel;

4.5.2 Strategy addressed

The strategy of the program is to identify the language of segments of text based on reference files by measuring and comparing the 'cost' of encoding the segment with each reference file. The program uses either a *Copy Model* or a *Finite Context Model* for this, which are language models built from the reference files.
The program steps are as follows:

1. Read command-line arguments to determine the model type (*-c* or *-f*), reference directory, target filename, segment size, **k** (chunk size), **alpha** (smoothing factor), and **fail_threshold** (for Copy Model);
2. The program reads the entire content of the target file into a string for further processing;
3. The program iterates over each file in the reference directory. For each file, it reads the content and builds either a CopyModel or a FiniteContextModel, depending on the specified model type:
    - If a *CopyModel* is requested, it divides the reference data into chunks of size **k** and stores each chunk in a vector, and also stores the positions of each distinct chunk in a map;
    - If a *FiniteContextModel* is requested, it divides the reference data into chunks of size **k**, and for each chunk, it counts the occurrences of each character following that chunk, and also counts the total number of times the chunk occurs;
4. For each segment of the target data (where segments overlap and move one character at a time), the program estimates the number of bits required to encode the segment using each reference language model;
5. For each segment, the program selects the language model that provides the minimum estimated bits;

4.5.3 CopyModel and FiniteContextModel structs

- *CopyModel* struct: This structure consists of a map that stores each distinct chunk of size k and the positions of its occurrences in the reference text. It also stores a vector of all chunks of the reference text in their original order, and N which is the number of distinct characters in the reference text.
- *FiniteContextModel* struct: This structure contains two maps - one for counting each next character after a chunk, and another for counting occurrences of each chunk. It also stores k - the number of distinct characters in the reference text.

4.5.4 Finite Context Model and related functions

The Finite Context Model is a language model that, for each chunk of k characters, stores a count of each character that follows the chunk, and a count of the total number of times that chunk appears. This model can be used to predict the character that will follow a given chunk.

The *predict_char_fcm* function uses the *Finite Context Model* to predict the next character after a given chunk. It calculates the probability of each potential next character (based on the count of that character following the chunk and the total count of the chunk), and returns the character with

the highest probability.

The *estimate_bits_fcm* function estimates the number of bits required to encode a given target data string using a *Finite Context Model*. It does this by iterating over the target data, for each chunk calculating the prediction and checking against the actual next character in the target data. It then calculates the probability of the prediction being correct, and adds the negative log base 2 of this probability to a running total of the estimated bits. The function returns this total.

5 Results

The same computer was used for all tests, consisting of an Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz with 6 cores and 1 socket. The results presented were also obtained by taking the average value of several tests.

5.1 Lang Results

For testing purposes for this program, the famous Os Maias by Eça de Queiroz (1) was used. As a text to analyze, a text written in Portuguese found in the langs (2) referred to in the project statement.

The Eça de Queiroz's text can be found in the examples folder and the other examples in the langs folder.

Also, was used $\alpha = 1$ and a fail threshold of 3.

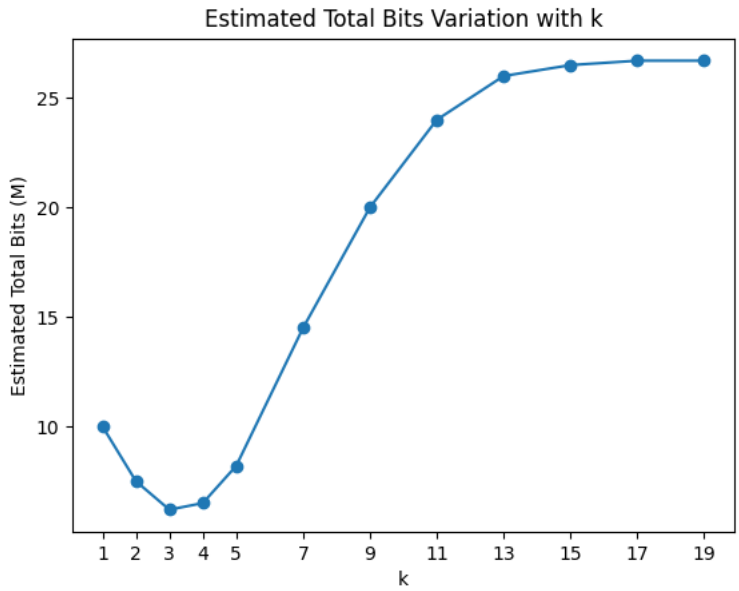

Figure 3: Estimated Total Bits Variation with K values.

Looking at the results above, where different values for the sequence size, k, and the respective values of the estimated total number of bits are represented it's easily noticed that **k=3** is the ideal size to compress the data according to the implemented copy model and for these two texts. From k=3 the total number of bits increases, stabilizing when **k=13**.

This indicates that the copy model behaves better in sequences of smaller size, and that with the increase of the value of k, the sequences that appear are practically all new, being the value of the total number of bits represented approximately by the log_2 of the cardinality of the alphabet.

For testing purposes, a new model was created, now with the text of the examples in Portuguese serving as a model and with the text Os Maias being analyzed. Also, a model built from a text in Croatian (text also found in the examples (2)) and once again Os Maias to be analyzed were also represented in the graph below. In the same way that the previous example was used $\alpha = 1$ and a

fail threshold of 3.

Looking at the results, and as expected, the model with the text in Portuguese is easily verified to be the best, compressing the information better, that is, using fewer bits. This is due to the fact that Os Maias was written by a Portuguese author, and therefore the model based on the Portuguese text will be better to analyze.

It is also important to note that the Croatian text model stabilizes earlier. This is due to the fact that it arrives earlier to the point where is always finding new sequences, so the result for the total number of bits will correspond to log_2 of the cardinality of the alphabet. That is, it stops counting hits and fails earlier and just adds the logarithm.

A small note too, in the case of the Portuguese model, the best value for k is 3 and in the model with the Croatian text it is 2 (finds better sequences of 2 characters).

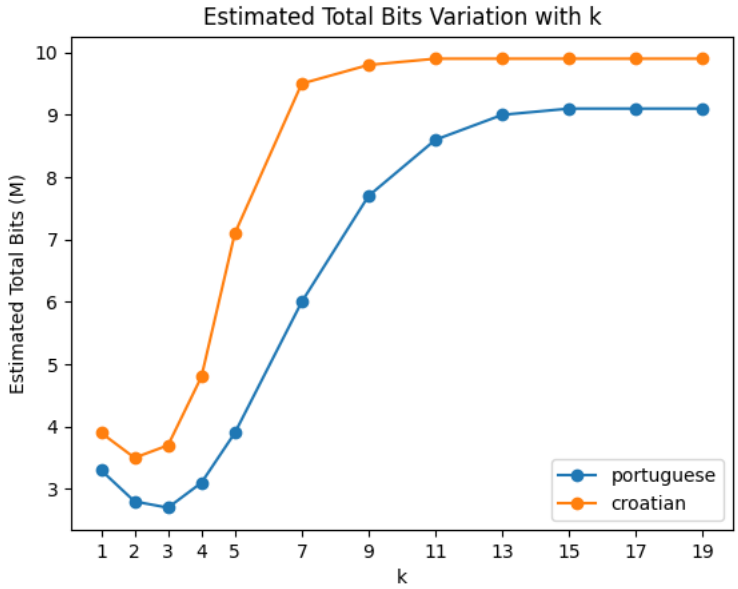

Figure 4: Estimated Total Bits Variation with K values for Different Models.

By analyzing the table below, which presents the results for the example text in Portuguese as a reference and Os Maias as the text under analysis, it is observed, as expected, a general increase in the number of bits required as the value of the fails limit increases. With the increase in the limit, a position can fail more often, so the model will normally fail more and therefore need more bits.

Also note that k=3 keeps as the best sequence size for the copy model.

k	α	**Threshold**	**Estimated Total Bits (millions)**	**Avg. Bits per Symbol**
2	1	5	2.83	2.16
2	1	10	2.84	2.17
2	1	20	2.84	2.17
3	1	5	2.73	2.09
3	1	10	2.75	2.10
3	1	20	2.77	2.12
5	1	5	3.94	3.01
5	1	10	3.96	3.02
5	1	20	3.98	3.04
7	1	5	6.04	4.61
7	1	10	6.05	4.62
7	1	20	6.06	4.63
9	1	5	7.72	5.89
9	1	10	7.72	5.90
9	1	20	7.73	5.90

Table 1: Experimental Results with Varying Values of k and Threshold

5.2 Findlang Results

In this case, to verify the behavior of the language recognition system, a new *langs* directory was created that contains at least 20 texts examples from different languages.

The main objective of this program is, given a text written in a language (target text), through the models created with the copy model, the findlang program must say which language is represented in the file.

For this, a first test was carried out, where the target text corresponds to the Portuguese national anthem (3). For the program to run, it is necessary to pass as argument the directory that has the files to build the models (langs/), the target text, the value of k, alpha and threshold. It was considered k=3, $\alpha = 1$ and threshold=3.

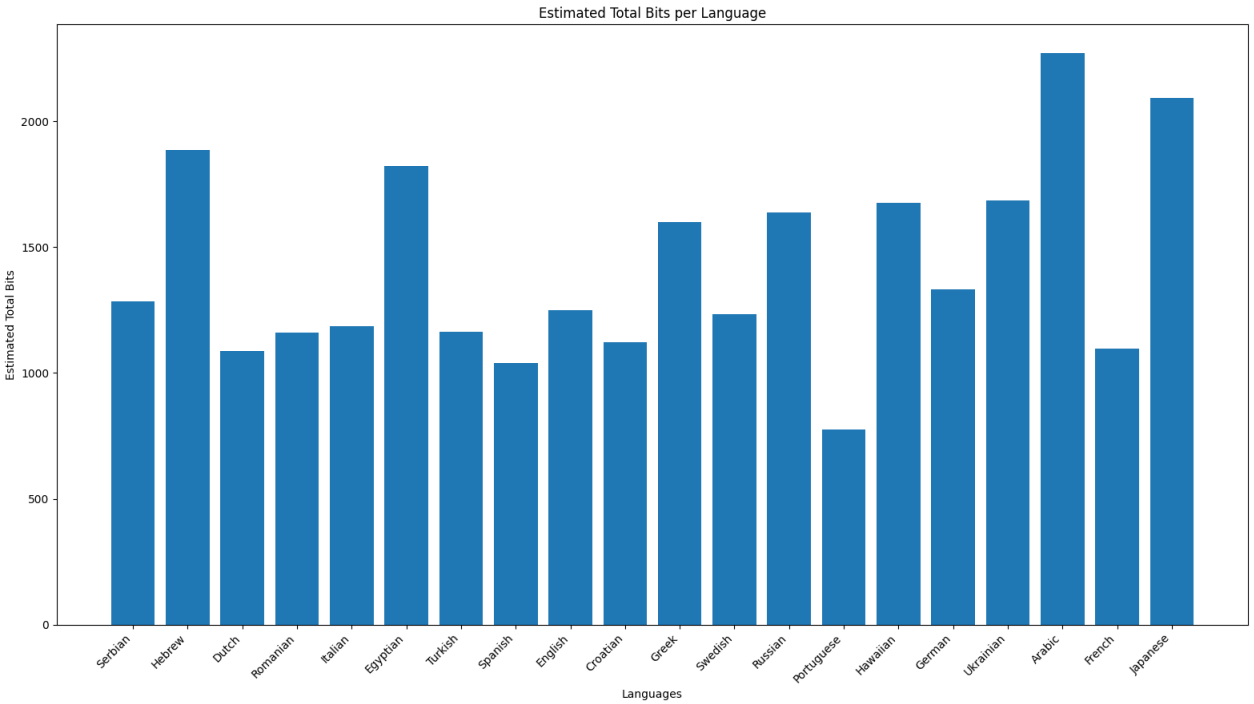

Figure 5: Estimated Total Bits per Language.

Upon analyzing the bar graph above, it is evident that the Portuguese language has the lowest estimated number of bits required for compression. That said, it turns out that the findlang correctly identified the Portuguese anthem as being effectively Portuguese.

Note the fact that the Arabic, Japanese and Hebrew languages need more than twice as many bits, which indicates that are quite different languages from the Portuguese language, which, as we know, is true.

Below is what is shown in the terminal, where the number of bits required if the text is in that language is indicated and finally which language the text is in.

Running the program with **./findlang ../langs/ ../examples/hino.txt 3 1 3**, the expected result should be:

```
Estimated total bits for sr_SR.Serbian-latn-wiki.utf8: 1286.21
Estimated total bits for he_IL.Hebrew-wiki.utf8: 1886.53
Estimated total bits for nds_NL.Dutch_Low_Saxon-wiki.utf8: 1089.5
Estimated total bits for ro_RO.Romanian-latn-EP7.utf8: 1162.85
Estimated total bits for it_IT.Italian-latn-EP7.utf8: 1187.38
Estimated total bits for arz_EG.Egyptian_Arabic-wiki.utf8: 1821.72
Estimated total bits for tr_TR.Turkish-latn-wiki.utf8: 1165.69
Estimated total bits for es_ES.Spanish-latn-EP7.utf8: 1041.31
Estimated total bits for en_GB.English-latn-EP7.utf8: 1249.04
Estimated total bits for hr_HR.Croatian-wiki.utf8: 1123.95
Estimated total bits for el_GR.Greek-grek-EP7.utf8: 1598.15
Estimated total bits for sv_SE.Swedish-latn-EP7.utf8: 1234.66
Estimated total bits for ru_RU.Russian-wiki.utf8: 1637.97
Estimated total bits for pt_PT.Portugese-latn-EP7.utf8: 776.23
Estimated total bits for haw_US.Hawaiian-wiki-test.utf8: 1675.9
Estimated total bits for de_DE.German-latn-EP7.utf8: 1332.05
Estimated total bits for uk_UA.Ukrainian-wiki.utf8: 1684.58
Estimated total bits for ar_XX.Arabic-wiki.utf8: 2272.36
Estimated total bits for fr_FR.French-latn-EP7.utf8: 1096.18
Estimated total bits for ja_JP.Japanese-wiki.utf8: 2093.43
The text is likely to be written in pt_PT.Portugese-latn-EP7.utf8: 776.23
```

5.3 Locatelang Results

To test this application, a new text file, locatelang.txt, was created and added to the examples directory. This file contains 3 different languages: Portuguese, Turkish and Hebrew.

```
A minha família é constituída por mim, pelo meu pai, a minha mãe, a minha irmã,
os meus avós maternos e o meu avô paterno.
```

```
Ailem ben, babam, annem, ızkz şkardeim, anneannem ve
büyükbabam ve baba ıtarafndan dedemden şoluuyor.
```

```
המשפחהשלימורכבתממני
, אבי, אמי, אחותי, סביוסבתימצדאמיוסבאמצדאבי .
```

The program takes seven arguments:

- model type: can be a copy model (-c) or a finite context model (-f)
- reference directory: directory with files to be used as reference
- target file: file to be analized
- segment length: substring of the target text to be analized
- k
- α
- fail threshold

Several tests were made and the program was run several times for different argument values, reaching segment_length=50, k=5, α=1 and threshold=3. These values were the most moderate we found. They are enough to determine with some precision the range of a segment and its respective language. So, the program was run with:

./locatelang -c ../langs/ ../examples/locatelang.txt 50 5 1 3

In this case was used the copy model and the result was:

```
Segment: 0-107 is likely in pt_PT.Portugese-latn-EP7.utf8 with
estimated bits: 12049
Segment: 107-212 is likely in tr_TR.Turkish-latn-wiki.utf8 with
estimated bits: 9795
Segment: 212-313 is likely in he_IL.Hebrew-wiki.utf8 with
estimated bits: 6253
```

Upon reviewing the obtained results, it is evident that the program with the copy model (-c) accurately classified the languages and properly identified the positions of each text segment.

In the same way, but using a finite context model (-f):

```
Segment: 0-105 is likely in pt_PT.Portugese-latn-EP7.utf8 with
estimated bits: 10763
Segment: 105-212 is likely in tr_TR.Turkish-latn-wiki.utf8 with
estimated bits: 10035
Segment: 212-313 is likely in he_IL.Hebrew-wiki.utf8 with
estimated bits: 5579
```

Looking at both results, it's clear that both accurately recognize the three languages present in the target file, and the finite context model presents better values in data compression, since they can efficiently provide probabilities for ”non-hit” symbols , i.e., symbols that don't directly match the reference data.

To better evaluate the application, an additional test file, mixed-locate-lange.txt, was created. This file contains a single sentence consisting of the three previous mixed languages.

```
A minha família é constituída por mim, babam için,
annem için, ıkz şkardeim, סבאוסבתאמצדאמיוסבאשלימצדאבי .
Ailem benden şoluuyor, babam için
```

The result obtained for the copy model (-c) was:

```
Segment: 0-17 is likely in pt_PT.Portugese-latn-EP7.utf8 with
estimated bits: 1830
Segment: 17-60 is likely in tr_TR.Turkish-latn-wiki.utf8 with
estimated bits: 5330
Segment: 60-121 is likely in he_IL.Hebrew-wiki.utf8 with
estimated bits: 6722
Segment: 121-132 is likely in tr_TR.Turkish-latn-wiki.utf8 with
estimated bits: 1920
```

And for the finite context model (-f):

```
Segment: 0-18 is likely in pt_PT.Portugese-latn-EP7.utf8 with
estimated bits: 1870
Segment: 18-60 is likely in tr_TR.Turkish-latn-wiki.utf8 with
estimated bits: 5075
Segment: 60-121 is likely in he_IL.Hebrew-wiki.utf8 with
estimated bits: 6329
Segment: 121-132 is likely in tr_TR.Turkish-latn-wiki.utf8 with
estimated bits: 1869
```

Once again, both approaches exhibited accurate language classification in the target file. This further strengthens the claim that the model is reliable and demonstrates the application's efficiency and accuracy in identifying and classifying text.

6 Copy Model vs FCM

Copy models are a type of modeling approach used in information theory and data compression. The idea is to create a model that can efficiently compress the reference text, making it a good representative of its class. When encountering a target text, the program estimates the number of bits required to compress it using the model computed from the reference text. The class corresponding to the model that requires fewer bits to describe the target text is then assigned to it, indicating the likely language or class of the text.

Finite-context models are another type of modeling approach commonly used in information theory and data compression. These models are designed to capture and analyze the statistical dependencies and patterns present in a sequence of symbols. The main idea is to assign probability estimates to symbols based on a conditioning context, which is computed from a finite and fixed number of past outcomes in the sequence.

In summary, copy models focus on compressing and describing reference texts efficiently to measure the similarity between files, while finite-context models aim to capture statistical dependencies and predict the next outcome in a sequence of symbols. Both modeling approaches have their applications in data compression and can be utilized in language recognition systems.

When deciding between copy models and finite-context models for language recognition and text classification tasks, it's important to consider the respective advantages and disadvantages.

Copy models offer a simple and efficient approach to language recognition by implicitly compressing the relevant features of reference texts and measuring the similarity to the target text. They are easy to implement and can be applied to various classification problems.

On the other hand, finite-context models provide a richer representation of data by capturing statistical dependencies and can offer enhanced accuracy, especially when considering non-hit symbols.

7 Conclusion

In this assignment, we tackled the problem of language recognition using an information-theoretic approach based on data compression. Instead of traditional feature extraction and selection methods, we explored the idea of using compression algorithms to measure the similarity between texts and classify them into different languages.

We developed a program called *lang* that accepts two files: one representing a reference text (ri) for a specific language and another containing the text under analysis (t). The program models the reference text using copy models and estimates the number of bits required to compress the target text using the computed model.

Building upon the *lang* program, we created a language recognition system called *findlang* that can make guesses about the language in which a given text (t) was written.

To further enhance our language recognition capabilities, we developed an application called *locatelang*. This application can process texts containing segments written in different languages and provide the character position at which each segment starts, along with the corresponding language.

In conclusion, our information-theoretic approach utilizing data compression for language recognition proved to be effective. By modeling reference texts and measuring the compressed sizes of target texts, we could accurately classify texts into different languages.

References

[1] Os Maias, Eça de Queiroz https://github.com/florist-notes/Data-Analysis/blob/master/data-analysis%20project%20-%205%20-%20Books_frequency_Eng%2CFre%2CPort%20%26%20Ru/Books/Portuguese/Queir%C2%A2s/Os%20Maias.txt

[2] Text Examples https://sourceforge.net/projects/la-strings/files/Language-Data/

[3] Hino Português https://pt.wikipedia.org/wiki/A_Portuguesa