Title: **TAI - Lab work n°2**

Author: **Bruno Bastos, Eduardo Santos, Pedro Bastos**

Date: **22/12/2021**

Contents

1. Introductory Note

Identifying the language a text is written on, is a problem that has been having attention for a while. Many approaches try to extract features from the text and then use them to identify the correspondent language. However, this problem can be seen as a special form of lossy data compression, allowing the use of compress algorithms to measure the similarity between texts.

The objective of this work was to approach the problem of determining the ”similarity” between a target text, *t*, and some reference texts, r_i.

2. Steps and Decisions During the Implementation

The project consists of three main programs: Lang, FindLang and LocateLang.

2.1. Lang

The objective of the Lang program is to provide comparative information between two files. It takes a text file as reference and, by using a Finite Context Model, calculates the number of required bits to encode the target text for that reference text. The program allows the user to specify the parameters *k* and α that are needed for the *FCM*.

2.2. FindLang

With the help of the Lang program, FindLang can predict the language of a given target text by calculating the necessary number of bits needed to encode it for each of the reference texts in the language database. The program predicts that the target text is in the same language as the reference text that produced the lowest number of bits needed to encode.

2.3. LocateLang

The previous program predicts the language of a text file, however sometimes in a file there can be more than a single language. LocateLang tries to predict the occurrences as well as the positions of a given language in the target file. It does that by calculating the number of needed bits to encode a letter for every single reference language and saves them in a stream, which was already a feature of the FindLang program to adapt to this one. It then passes through the stream with a sliding window, where the size can be passed by the user. For that window it calculates the mean of the values inside and checks if it is lower than a giving threshold. The program stores the positions in the target file where for a specific language it is below that threshold. However, a certain text excerpt can be part of many different languages, as the program does not choose the best one but all that stay below the threshold.

When developing the program, many modifications where made in order to obtain better results and give the user more freedom in the selection of the parameters.

Calculating the stream of necessary bits to encode a letter can be done utilizing multiple configurations of the *FCM*. The user can provide a list of *k* and α. The values of the stream are merged having into consideration their sizes. Then, instead of using only the mean of the sliding window values, the user can opt to use the max of the values, the entropy of the alphabet or a fixed value. It is also possible to define a noise threshold. Sometimes the values leave the threshold but come back in a few positions, meaning that the values go a bit higher than the threshold but go down in the next position. This can caused by an outlier that changes a

bit the value of that sliding window. By using the noise threshold, the user can increase the threshold by a given percentage that will only be taken into consideration when the calculated value in the sliding window is leaving the threshold.

2.4. Utils

This module is used to generate the intended graphic. If the user uses the arguments in the **LocateLang** program, it calls the utils module to plot:

- **–show_langs** - Shows, for every language found in a text with multiple languages, the positions that each language sub-string starts.

3. Results and Analysis

This section presents the results of the each one of the programs, demonstrating the benefits of using the features implemented.

3.1. Lang

Lang program, as it was previously stated, makes use of a *FCM* built using a reference text, and provides information about the number of bits necessary to encode the target text using that reference. Different values for the context and smoothing parameter will result in a different output. The output will give the user the information about:

- **Average Entropy**: corresponds to the average entropy of the *FCM* model built using the reference text.
- **Reference Text Size**: the number of symbols in the reference text.
- **Total Average Entropy**: it's the same as the Average Entropy times the Text Size.
- **Average Bits to Encode**: the average number of bits to each encode each of the symbols in the target text.
- **Target Text Size**: the number of symbols in the target file.
- **Number of Bits to Encode**: represents the number of bits needed to encode the target text using that reference.

The most important of those is the Average bits to encode the target text, because with that, the program can predict the target text language. Table 1 shows for different reference languages, the average number of bits necessary to encode a text in English.

Reference Language	Number of Bits
English	2.715
Portuguese	5.887
French	5.315
Spanish	5.903

Table 1: Average number of bits to encode target per Reference Language

The results align with what was said previously, but one important thing is that the size of the reference file and it's contents matter a lot. An test example was done by using the same reference files but this time only the first 1000 lines of them. The results are presented in Table 2 and they demonstrate this relation

between the size and the performance of the model. This occurs because small files can have tendencies that will influence a lot the *FCM* due to their occurrences being more probable. However, as the size increases, the general tendencies of the language will appear more and thus have an higher chance of appearing.

Reference Language	Number of Bits
English	3.698
Portuguese	6.912
French	6.104
Spanish	6.494

Table 2: Average number of bits to encode target per Reference Language (Reduced Files)

The size of the target text does not matter as much, the program can predict smaller texts as long as it has good references, but if the text is too small, it can lead to very similar results for different languages. Table 3 has an example of a prediction of a target file in English with just one line, when using the full reference files.

Reference Language	Number of Bits
English	2.434
Portuguese	6.066
French	5.502
Spanish	6.097

Table 3: Average number of bits to encode target per Reference Language (Reduced Target)

The results are not the same but they are somewhat similar to those in the first experiment. Thus if the user does not provide a very small file, the model should still be able to predict the language. Important to note as well that the content matters a lot specially when providing a smaller target file, because if it has many words that are common in multiple languages the results can not be trusted.

3.2. FindLang

FindLang predicts the language of a target text by comparing it with every reference text in the dataset and choosing the one that resulted in the lowest number of bits to encode the target.

In order to test how good the program is, there were tested multiple files with different languages and checked if the program could rightly guess the correct one. There are 20 languages in the datasets and every language represented in the training dataset is also represented in the test dataset.

The program manage to correctly predict **every** language, **100% accuracy**. Just to show some examples, here are the results for a set of languages of their top 5 best matched language from the dataset.

Reference Language
Dutch
Danish
German
English
Catalan

Table 4: Top 5 predicted languages for Dutch

Reference Language
English
Welsh
Dutch
Scottish
Latin

Table 5: Top 5 predicted languages for English

Reference Language
Danish
Swedish
Dutch
Catalan
Latin

Table 6: Top 5 predicted languages for Danish

3.3. LocateLang

Sometimes there are some files that have more than one language. That's here LocateLang comes in. It identifies the positions where it is likely that a language occurs. Although it does not give an exact answer in some cases, it reduces the drastically the number of languages to search for. The parameters can be adjusted by the user in order to obtain better results depending on the target text.

Tests were performed in order to measure how well the program can distinguish languages. Initially, it was provided a file containing only English text in order to prove the accuracy of the model. Figure 1 shows the positions in the target file where the program detected a given language. Each value of the y axis represents a language in the training set. As can be seen in the image, for a text only containing English, the program only detects English as the language that is present.

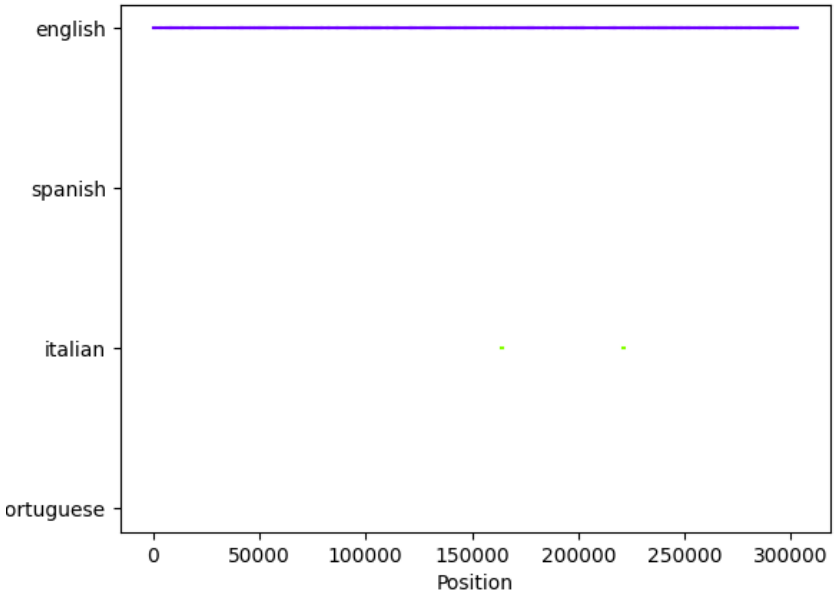

Figure 1: LocateLang - Positions detected for an English text file.

However, the objective of this program is not to predict the language for the whole file, but to predict multiple languages spread across a single text file. The user needs to adjust the parameters to get the best result. A test file was created containing the languages French, Italian, Portuguese and English, in this order. Then the program was executed for this file, using the default parameters. These parameters consist of using a *FCM* with context of 3 and an alpha of 0.001, with a window size of 24. Figure 2 shows the result obtained.

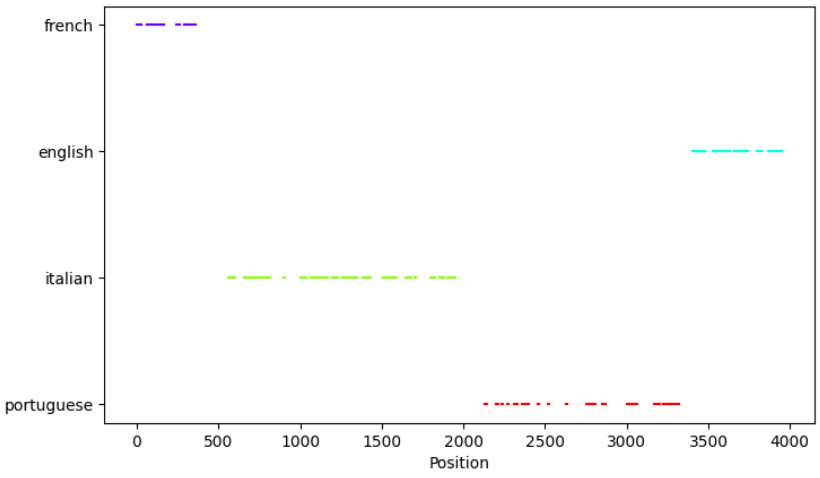

Figure 2: LocateLang - Test file with default configurations.

As can be seen, each line in the y axis represents the labeled language. According to the test file provided, the program does do a good job identifying the languages in the file as well as the correct order at which they appear. There is still some room for improvement.

3.3.1. Sliding Window Size

Some tests can be made adjusting the parameters k, alpha and the window size. Starting with the window size, values were tested to see which size would be the best for this example. Figure 3 shows the program predictions when using a window size of 5.

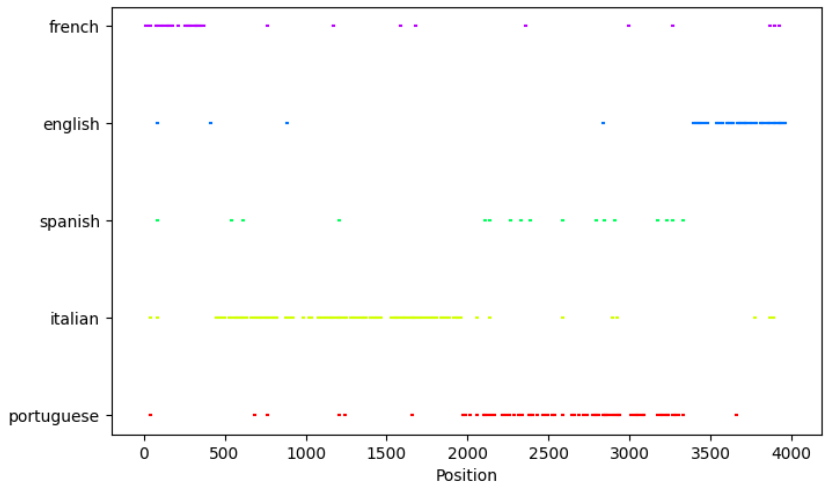

Figure 3: LocateLang - Test file with window size of 5.

When comparing both results it is clear that the first one is better at identifying the language of a test, but with a lower window it might be better to identify individual words of different languages. That is something that will be tested later.

The effect of increasing the window size can be seen in Figure 4.

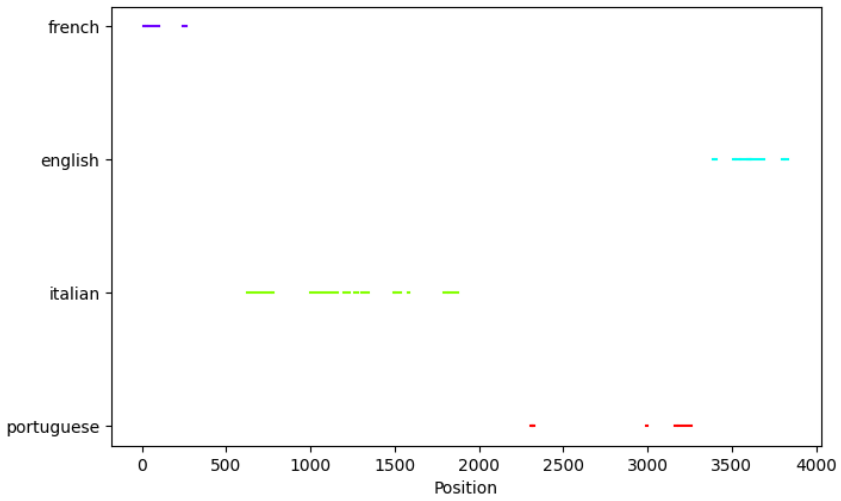

Figure 4: LocateLang - Test file with window size of 100.

For higher values of the window size the program will detect less frequently a language but it will be more sure that in that position there is that specific language. For lower values, it is easier for the program to detect languages and can sometimes detect languages that are not there. This can be interesting because it allows the user to see the relation between the words of 2 languages. For example in Figure 3 it is possible to see that although Spanish is not in the test file, it is mainly detected when Portuguese is detected, which can a sign that languages are related.

3.3.2. Model Parameters

The context of the *FCM* can influence the result depending on the language. Figure 5 shows the results for the default configurations but now with k=6.

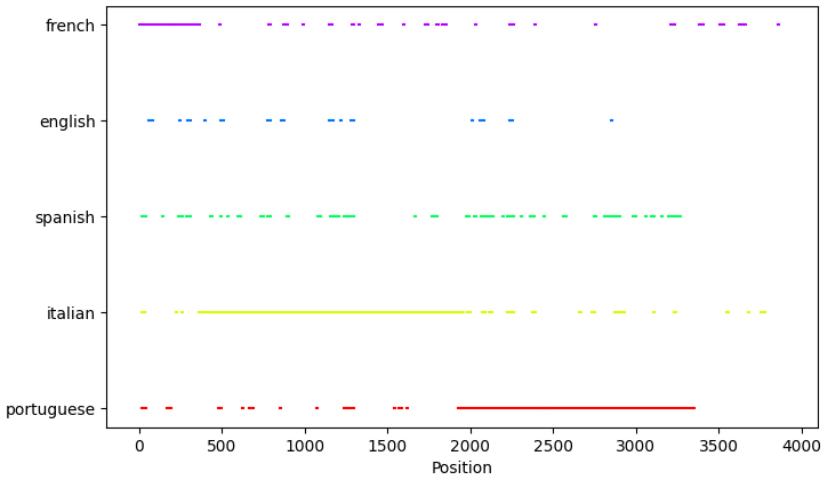

Figure 5: LocateLang - Test file with k=6.

The results are less accurate but one thing to notice is that overall when the correct language is detected it usually stays below the threshold without much oscillations.

The influence of the parameters α is related to the size of the context. It matters more when a language does not have many symbols in the target text. For example, English is harder to detect when the text has many symbols from other languages like Portuguese and French. However, since those two have more symbols in common the *FCM* will not give them uniform probability because they appear in the reference text. The parameter α will affect how the *FCM* deals with the symbols that are not present in the language. If it is high it will give a high probability to symbols that have not appeared. For higher values of k, the probability of a symbol appearing after a context is usually lower so the α will matter more. Thus, it can be harder to detect the language. Figure 6 shows the result for k=5 and α=0.0001 and Figure 7 for k=5 and α=0.0000001.

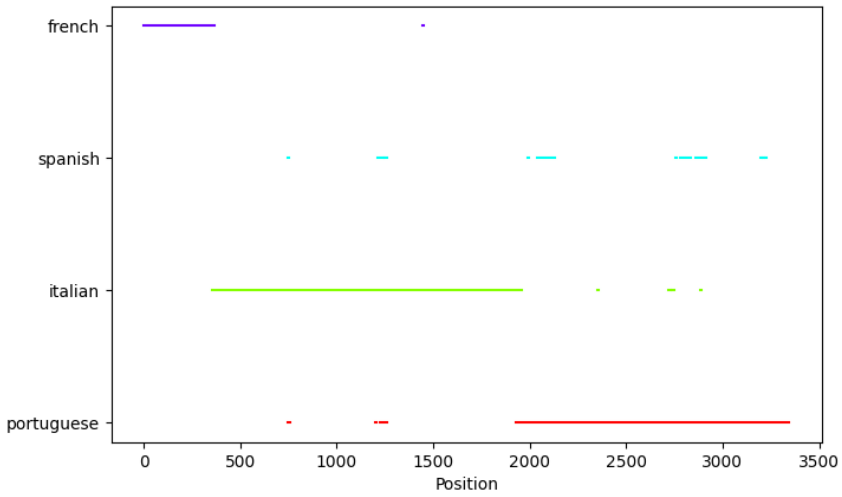

Figure 6: LocateLang - Test file with k=5 and alpha=0.0001.

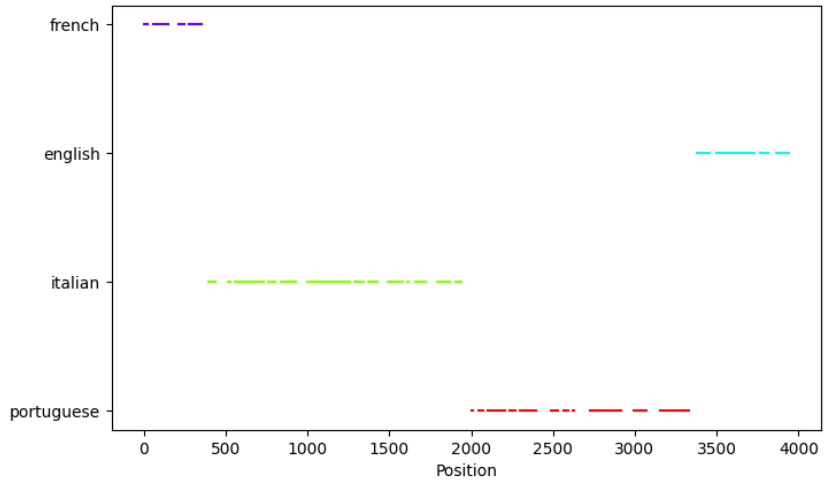

Figure 7: LocateLang - Test file with k=5 and alpha=0.0000001.

3.3.3. Multiple Models and Threshold Strategy

Different languages might have different optimal values for the context size and smoothing parameters. The program allows the user to run multiple values of k and α. The streams of bits needed to encode a symbol are merged into a single one by doing the mean of them taking into consideration their size. Then, the threshold to define if to see if that window is part of a specific language can be given by the user using different options. Max, which takes the average entropy of each of the *FCM*s with the different parameters, and chooses the maximum value of those entropies. Mean, which calculates the mean of the entropies. Entropy, which only takes into consideration the alphabet size and uses the formula $entropy = log_2(alphabet_size)/2$

to define the threshold. Also, the user can choose to have a fixed threshold by providing a numeric value. The choice of the threshold can be important and is specific to the text and languages trying to predict.

Starting with the impact of using different values for k and α, some tests were made in order to obtain better results for the test text. Figure 8 shows the predictions for k=3 and k=4 with a Max threshold.

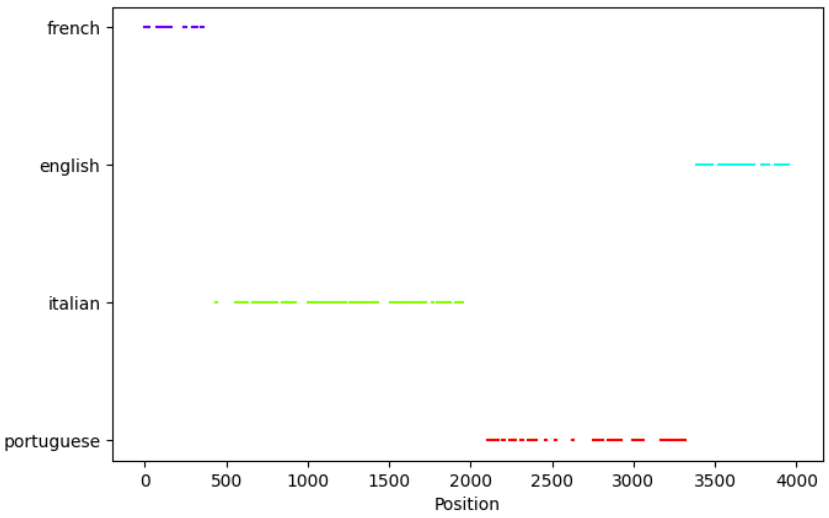

Figure 8: LocateLang - Test file with k=3 and k=4 with Max threshold.

The results are very similar to the initial ones and the reason might because the threshold is the same, since Max chooses the maximum entropy of the *FCM* models as the threshold. Using the Mean threshold strategy the results were similar. Figure 9 is the result of a test using models with k's equal to 3,4 and 5 with a threshold strategy of Entropy.

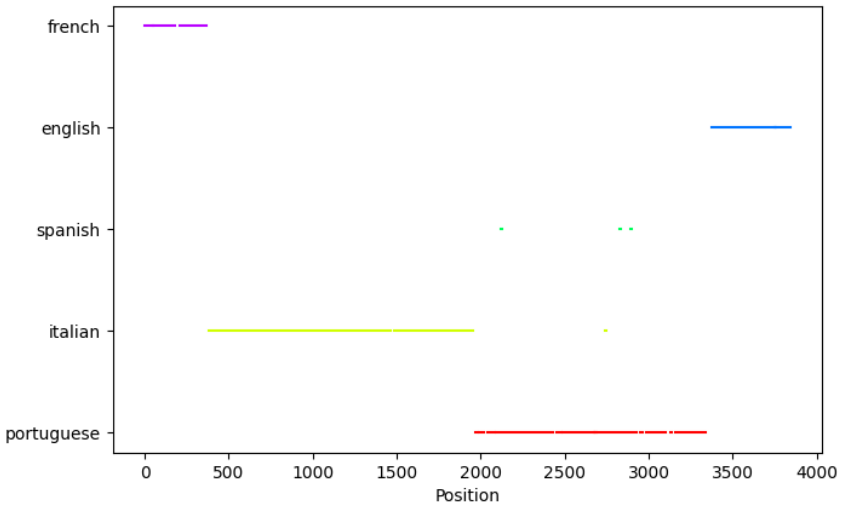

Figure 9: LocateLang - Test file with k=3,4,5 with Entropy strategy threshold.

Although Spanish was detected, the lines are more connected and the results seem to be a bit better overall.

3.3.4. Noise Reduction

The results were very good for now but there are some positions in the text file that are oscillating, entering and leaving the threshold, that can still be improved. For that it is possible to use the noise option, which will allow the user to define a percentage of accepted noise. Whenever a certain window is leaving the threshold, it is added a percentage of the threshold to itself in order to combat noise. Outliers will have less of an importance and the lines in the figures will be more continuous. Figure 10 shows the effect of a noise value of 2 in the model of the Figure 8.

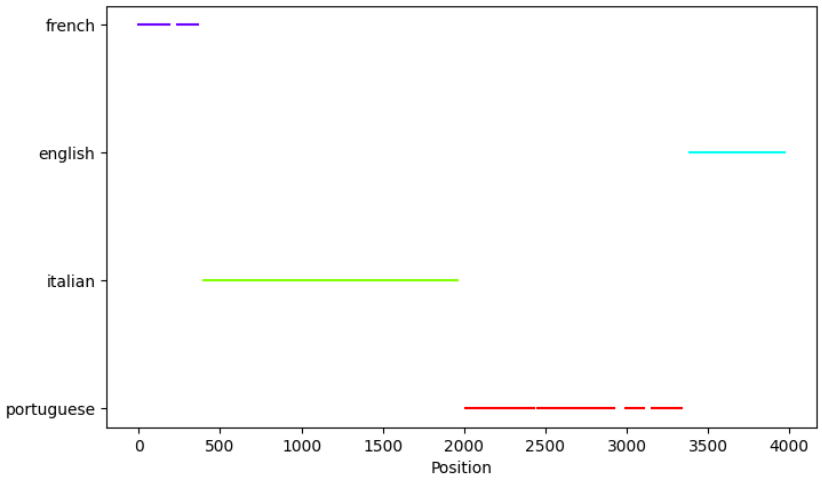

Figure 10: LocateLang - Test file with noise reduction.

Results are more smooth and easier to understand with this value for the noise reduction.

Just to have a different visualization of the data, Figure 11 shows the number of bits for a symbol in a given position for the Portuguese language using those parameters.

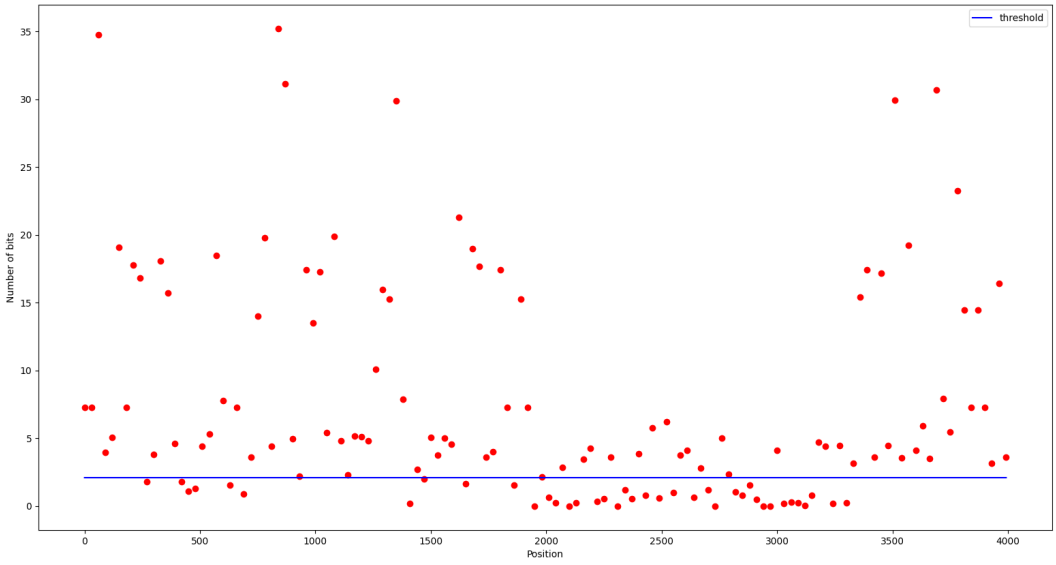

Figure 11: LocateLang - Number of bits for each symbol for Portuguese.

The blue line represents the threshold. When the points are below the threshold, its when it detects the

language, which coincides with the above figure.

3.3.5. Other Results

LocateLang is good when the different languages are separated from one another, but how well does it do when there are foreign words in the middle of another language text. It was created a small test file in Portuguese with some English words in the middle. Figure 12 is the result after adjusting the parameters, specially the window size, which needs to be much smaller.

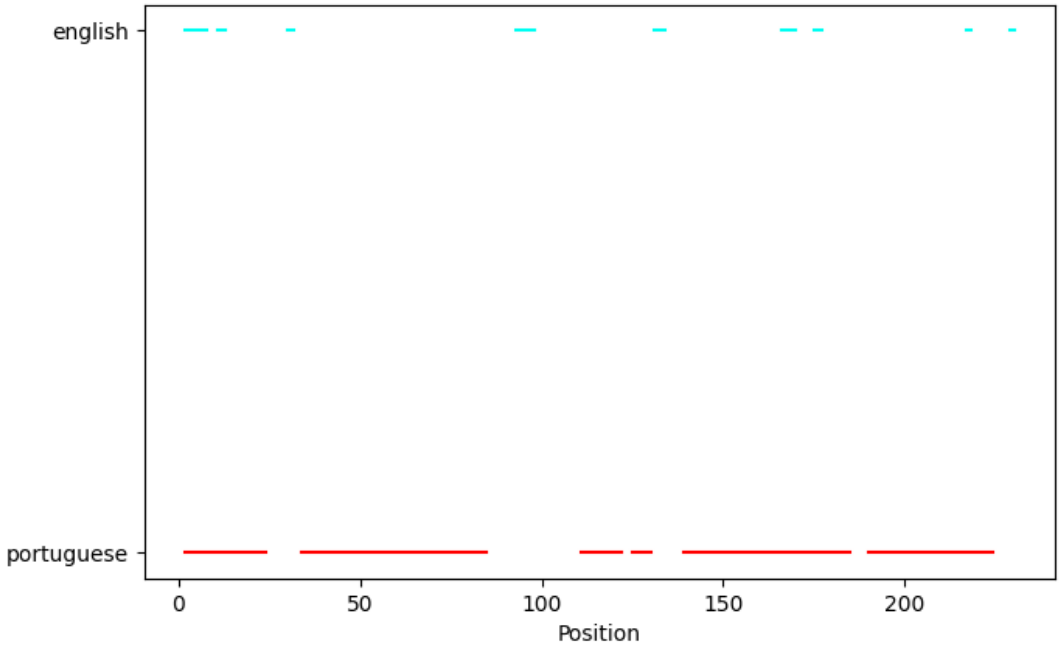

Figure 12: LocateLang - Test with English words in a Portuguese Text.

Words are in positions [26,35], [91,99], [134,139] and [230, 240]. It seems that program could indeed find those words but there are another words that have been marked as English words that are in fact Portuguese, which is understandable, because, although different in many aspects, Portuguese and English share some similarities in many of their words. For such a small example the program did very well.

4. Conclusion

This work allowed us to understand how Finite Context Models can be used to approach the problem of determining the ”similarity” between items. We were able to build a product that successfully implements *FCM* to solve the previously mentioned problem, obtaining reliable results.

After the presentation of the results and analyzing them, we have confidence that our program could be integrated into others, due to its effectiveness. Its limitations can be surpassed with the right configurations from the user. A smart and sharp looking person can automate language identification with the help of this tool. The tool main feature falls in its ability to be configurable and adjustable to every need.