Assignment 2 - Language detection using Copy Models for compression

Algorithmic Information Theory (2022/23)
Universidade de Aveiro

Martinho Tavares, 98262, martinho.tavares@ua.pt
Nuno Cunha, 98124, nunocunha@ua.pt
Pedro Lima, 97860, p.lima@ua.pt

May 29, 2023

1 Introduction

The goal of this assignment is to implement a language detection system using the previously developed copy model from the last assignment. The system is capable of detecting the language of a given text, if it is one of the languages used to train the model. To implement this, a new version of the copy model was developed, called lang, which received a reference text to train the model and a target text to compress by copying from the reference. The main idea is that if we use a reference text of a given language then we should get a better compression if the target text matches the provided language, since the same language patterns are to be expected, and so better copies can be made. Then, we can identify the language of a given target text in terms of the language of the provided reference. If using multiple references, we can comparatively identify a language, by checking which of them provided a better compression of the target text.

One of the other new features implemented in the lang model is the ability to use a finite-context model for the probabilities of non-hit symbols, instead of using the uniform and frequency probability distributions.

With the new model implemented, two other scripts were developed to test the performance of the model and obtain the language prediction in an entire text file or in specific sections of a text. The first script is called `find_lang`, which receives a target text and uses all the existing references to predict the language of the target text, returning the language with the highest probability of being the target's language. This is done by executing the lang model for each reference text and comparing the total information obtained for each language, returning the reference language with the lowest information.

The second script is called `locate_lang`, which receives a target text and, like the `find_lang` script, executes the lang model for each reference text, but

instead of returning the language with the lowest information, it returns the language with the lowest information for each section of the target text. This is done by saving which language has the lowest information for each step/character of the target text, and then checking the intervals where the language is the same, returning the language with the lowest information for each interval.

In this report we will present the methodology used to implement the lang model, the `find_lang` and `locate_lang` scripts, in more detail in section 2, and the results obtained by testing the scripts with different target texts in section 3. And finally, in section 4, we will present the conclusions of this assignment.

2 Methodology

For the development of the different components, we utilized different approaches. The lang model was written in C++, extending and improving the copy model program developed in the previous assignment.

However, the `find_lang` and `locate_lang` scripts were developed in Python. These scripts essentially perform many independent runs of the lang model using different references, and parse the output as they desire. Since practically all of the heavy processing is performed by the lang model, we can afford the loss of performance that running a script in Python entails in turn for the much increased development simplicity. In fact, in practice we noticed that most of the execution time spent on these scripts is in the lang model, as the result processing in Python is relatively simple. For both scripts, parameters of the lang model can be specified.

Even then, the `find_lang` and `locate_lang` take too much time waiting for the results of the lang model. Therefore, to ease this large bottleneck, we added support for multiprocessing to both of these scripts. Since different lang models with different references can be run completely independently, without any need of interprocess communication, we can easily launch multiple processes. The ideal option would be to integrate multithreading directly into the lang model program itself, in C++, which allows greater memory efficiency as the target text doesn't have to be stored in memory multiple times, one for each process. However, we decided to simply opt for multiprocessing, since multithreading would heavily complicate the source code of the lang model, and also because multiprocessing provided a performance boost good enough for our purposes. This allowed us to dedicate more time in other areas of the assignment, such as improving the lang model and the scripts.

If, while performing multiprocessing, any of the processes fails for some reason, then the script by default exits outputting the obtained error from the failed process.

In each of the following sections, we provide further development details on each of the components.

2.1 Lang model

The developed lang model is heavily based on the copy model built for the first assignment. The major change was adapting the existing program to train a copy model on a given reference text and only perform predictions on a given target text, instead of performing both training and predicting on a single message. This allows estimation of the number of bits required to compress the target text using copies from the reference text.

For the lang program, the following changes were done from the first assignment's copy model:

- UTF-8 support: We are using as references multiple kinds of languages. As such, the ASCII encoding used in the first assignment's copy model is not enough to correctly represent the symbols of every language, since not all alphabets fit in that encoding. The code worked perfectly for UTF-8 text, as it just effectively handled bytes. However, it could cutoff symbols, and it was harder to debug and visually analyze the behaviour of the copying. Therefore, we changed the code to support UTF-8 text, handling now symbols as wide chars instead of chars and strings as wstrings.

- Repeat-like wrapping for the past initialization: Since we want to start training/copying since the very first symbol in the reference/target message, we need to consider some kind of context when at that symbol, i.e. we need to somehow train and test the model using the first k symbols, even if we require k symbols of previous context for each of them, which doesn't naturally exist. For the first assignment, the strategy that we followed to solve this problem was to initiale this k-sized past with the most frequent symbol in the entire message. This was an easy solution to implement, but it's much less appropriate when applied for the specific use case of this assignment. Since we are dealing with natural language text, it doesn't make sense to incorporate k repeated instances of a single letter, as that will definitely not occur for a reasonable value of k. For the first assignment we mainly dealt with a chromossome sequence, but in this case this issue becomes more obvious.

  Therefore, for the lang program we decided to initialize the k-sized past using the last k symbols of the message. That is, we perform a repeat-like wrapping on the text. This way, we ensure the usage of a context in natural language for the first k symbols. The difference between both approaches is illustrated in figure 1

  We have to handle the initialization of the past carefully for both the reference and target texts (note that they are in different character arrays in memory). When training on the reference text, we initialize its past with its last k symbols. For the target text, we require some context for the first k symbols before we start predicting. Therefore, we initialize in the same way, using the last k symbols of the reference text. As such, it will be as if the target text was concatenated at the end of the reference text.

- Usage of `string_view`: In the previous assignment, we used strings to represent the k-sized context/window as we advanced on the text. For the copy model itself, we had an hash table whose keys were these k-sized windows, i.e. k-sized strings. This had the disadvantage that, effectively, the entire content of the message being trained on would be reflected in the keys of this hash table. Since the message itself is already being stored in memory, the storage of keys in the hash table more than duplicates the amount of memory that the message's content ends up occupying. Additionally, advancing through the text involved altering the current k-sized window by appending the new symbol at the end and removing the oldest symbol at the beginning. This operation is not particularly efficient, especially when the string's content already exists and, in practice, all that is being done is the advancing of a pointer.

  Therefore, we decided to switch these strings to string views. String views just require a pointer to the beginning of a string and the string size. This allows the lang program to be more memory efficient by simply pointing to the message content that is already in memory, from which copies are made. Additionally, this enforces a fixed size for the strings, which for our case makes sense as the size is always a fixed value k.

- Usage of `unordered_map`: Our previous copy model made heavy use of the `map` data structure. However, this structure enforces order, which we don't take advantage of, and is not the approximation to an hash table that we desired, contrary to `unordered_map`. As such, in the lang program we changed all uses of the `map` structure to the `unordered_map`, with noticeable improvement in performance.

- Removed ReadingStrategy: In the first assignment, we initially planned to evaluate different approaches to reading the message from disk: either entirely into disk, or in binary directly from disk in case we could not load the entire message into memory. We ended up not exploring different approaches other than storing all the content into memory, and so for the lang model we scrapped the abstraction of a reading strategy, incorporating the functionality that was in the respective classes directly into the lang copy model.

Other than these changes, various bugfixes and slight optimizations were done on the previous copy model. Due to the new approach of training on independent reference and target texts, the code had to be restructured and more responsibilities separated to ensure only training occurs on the reference and prediction is reserved only for the target.

For instance, after training on the reference we have an hash table of the counts of each symbol in the text. These counts are required by the probability distributions, more specifically the `FrequencyDistribution`. However, the probability distributions have to take into account the *target's* alphabet, not the reference's. Therefore, after performing the initial pass over the target text to determine its alphabet, these counts are updated to remove any symbols that

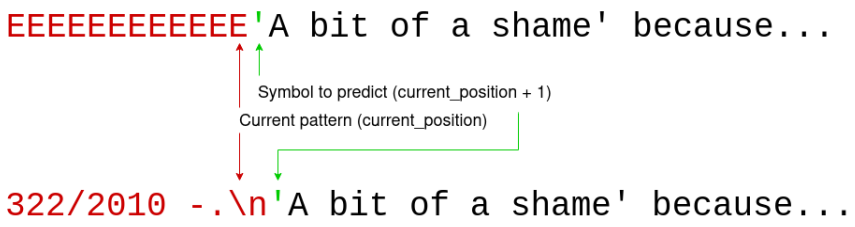


Figure 1: Illustration of the new past initialization, with the difference between the previous version (above) and the new version (below). The english reference text is used as an example.

did not appear in the target text and include those that appeared in the target text but not in the reference. These new symbols, that didn't previously have counts, will have their counts initialized with 1 to avoid reporting an infinite amount of information for them. Do note that, after passing over the target text, the counts of the symbols both in the reference text and target text are not updated, i.e. only counts actually done on the reference text are used. This is done because it doesn't make much sense to include the symbol counts on the target text when the objective of the program is to compress solely based on a model trained on a reference text. Additionally, we mainly want to use this program to compare compression of one same target text using many different references, and so it's not relevant to include the target symbol counts since they would be the same between all runs.

Another point worth making is that most copy pointer managers, which are the copy model components responsible for determining which copy pointer to use at each context, do not perform differently every time the same context is seen in the target text. This is because the array of copy pointers to choose from does not change while traversing the target text anymore, only during training on the reference. As such, `NextOldestCopyPointerManager` is the only manager whose behaviour changes for a given context while predicting on the target.

2.1.1 Verbose modes

The lang program was required to output the total number of bits estimated to be necessary to compress the target text based on the reference text. To simplify output analysis by the `find_lang` script, we added the minimal verbose mode specified with the parameter `-v o`, which simply prints to standard output nothing more than the total number of bits required for compression.

Additionally, for the `locate_lang` script, we needed to know what was the reported information content in bits for each symbol in the message, to correctly identify in which language was each text segment. Therefore, we modified the previous "machine" verbose mode, specified with the parameter `-v m`, which used to print the whole probability distribution in CSV format to standard

output. This output was previously saved to disk using shell redirection, and since the CSV format was very wasteful in terms of space a lot of disk memory was necessary for these files.

Since now we are only concerned with the probability of the actual symbol in the message, we only need one of those values of the probability distribution at every step. As such, we opted to instead of printing the floating-point probability values to standard output, to write them directly in sequence to a file in binary format. This allows for a densely-packed representation of the probability values over the message since no newlines or separators are used and the values are written in their full binary content instead of a character string representation in digits each occupying a byte, saving tremendously in space when compared with the previous assignment's approach. We can then load these values in a highly efficient manner from the `locate_lang` script using the NumPy library. Do note that these these floating-point values, declared as `double`, are assumed to occupy 64 bits.

Since the new "machine" verbose mode outputs the probabilities to a binary file, we added the possibility to declare to which file they are written, by specifying it as an "option parameter": `-v m:X`, where `X` is the path of the file. If only `-m` is passed, then the output is written to "lang.bin" by default. If the specified file already exists, then a prompt asking for ovewrite permission is presented.

2.1.2 Finite-context model

We explored the possibility of using a finite-context model, or discrete time Markov chain, as the default distribution to report for the symbols that we did not predict at each step. In this model, we try to estimate the probability that a symbol occurs after a given k_f-sized context by counting occurrences of the symbol after that context in the past. We allowed this model to have an arbitrary order, i.e. the size k_f of the context used. We can specify a $k_f \leq k$, where k is the size of the context in the copy model itself.

Since we already had a family of classes representing the reported probability distributions, as subclasses of the `BaseDistribution` class, then we created the finite-context model as a subclass as well, named `FiniteContextDistribution`. Implementation involved keeping track of counts of symbols after k_f-sized subcontexts we saw while training on the reference text. For that, we used an hash table with the context as the keys and the symbol counts as the values. The symbol counts themselves were also implemented using hash tables, with the symbols as the keys and the counts as the values. These tables are updated at every training step on the reference.

After training, we initialize the internal distribution variable, which contains the probability distribution that is reported at each step, with zeros for each symbol of the target alphabet. Then, for every hash table of counts in the hash table of subcontexts, we remove all the counts of the symbols that are not in the alphabet, since they won't be used.

Then, at every single prediction step, we update the internal distribution variable to store the probabilities of every symbol based on the obtained symbol

counts for the current k_f-sized subcontext. The probabilities are obtained using the following formula:

$$P(e|c) = \frac{N(e|c) + \alpha}{\sum_{s \in \Sigma} N(s|c) + \alpha|\Sigma|},$$

where e is the symbol of which we want to calculate the probability of occurring, c is the subcontext used, Σ is the target alphabet, $N(x|y)$ is the count of symbol x after sub y and α is a smoothing parameter to avoid probabilities of 0.

In case no counts were obtained for the current k_f-sized subcontext, then all counts are treated as 0 and the reported distribution is a uniform distribution.

This implementation requires recalculating, at each prediction step, the probabilities of all symbols at each subcontext found in the target text. If we found the same subcontext multiple times, then the distribution would be recalculated at each one of them. It can be argued that this is wasteful, and that the distribution can be cached somehow. In fact, we attempted to calculate all probability distributions for each subcontext in the main hash table beforehand, instead of keeping track of the counts. However, this is extremely expensive in terms of memory usage, since normally we only keep counts of a few symbols for each subcontext, but with this approach we need to store counts for all symbols of the target alphabet. Even after trying to use arrays instead of hash tables to store the probability values, the memory usage was too high (upwards of gigabytes). Surprisingly, as well, the performance didn't even increase. Since the target text was usually smaller than the reference text, most of the work was being performed in the preemptive calculation of the probability distributions, most of which were never going to be queried for. As such, we opted for the lazy calculation of the probability distributions, since in practice the performance footprint was not very noticeable and the memory usage was much less.

2.2 Find lang

The find lang script was required to, given a target text and a bank of references, determine in which language it was written. This is done by running the lang program for each combination of the target text and a reference text from the bank, getting the reported total information content in bits using the minimal verbose mode `-v o`, and reporting the reference text whose reported value was the lowest.

The script doesn't take into account its overall performance to, for example, indicate that it doesn't know which language the target text is in, i.e. it always says that the target text is one of the supplied references' languages.

We also added a metric to the script's output, called *confidence*. Confidence dictates how sure is the script that the reported language is actually the target text's language. This metric only takes into account the relative performance of the lang program with the different references, and not the performance of the

lang program overall. Confidence simply measures how confused the script is in the reported language. We essentially compare the lowest reported information value with the second lowest information value. If these values are very different between each other, then we can be more sure that the language that reported the minimum is actually the target text's language (within the known references at least). However, if they are very similar, then there is confusion between at least two languages, and we can't be very sure that the reported minimum is certainly the target language (this can happen, for instance, for very similar languages).

The confidence value C is calculated using the following formula:

$$C = \sqrt[3]{1 - \frac{m_2}{m_1}^3},$$

where m_k is the k-th minimum information value and $0 \leq C < 1$. This formula, for instance, reports a confidence of $\approx 95.65\%$ if the second minimum is twice as large as the first minimum, $\approx 88.95\%$ if the second minimum is 50% larger than the first minimum, but only $\approx 62.89\%$ if the second minimum is 10% larger than the absolute minimum. If the reported information is the same between both minimums, then confidence is 0%. We opted for this formula since it has very accelerated growth in the beginning (m_2 slightly larger than m_1), allowing us to assert that as long as a small enough distance between both minimums is ensured, then we can be confident about the reported language.

2.3 Locate lang

The `locate_lang` script is comparatively more complex than `find_lang`. Instead of identifying the language of the target overall, it attempts to identify which segments of text are written in one of the languages from the bank of references. Additionally, `locate_lang` can also identify segments of text on which it can't assign a language from the references, and as such takes into account the absolute compression performance of the lang program.

To do this, analysis is done on the reported information content at each position in the target text. We require this data for each reference. This is obtained by running the lang program for each reference using the machine verbose mode `-v m`, which outputs all this information into a binary file in disk. After all these results are saved, they are read from disk and analysis is performed by the script.

Contrary to the `find_lang` script, the `locate_lang` script caches the results of the lang model executions. This was done to speed up analysis since the processing of `locate_lang` itself contains many different parameters, and so we can avoid recalculating all necessary information in every run of the script if we only want to adjust the result processing and not the lang model parameters. This contrasts with the `find_lang` script, whose processing doesn't have any parameters, requiring only parameters to the lang model. The cached results keep track of which target file, which references and which parameters were passed to the lang program.

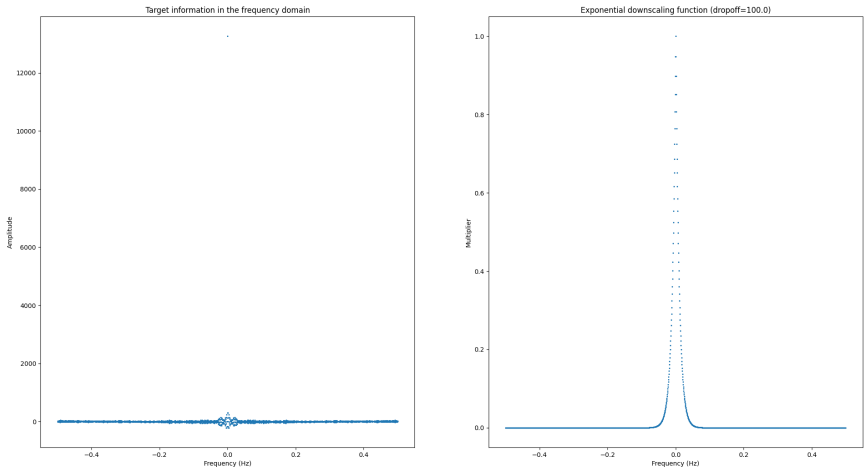


Figure 2: Example of the frequencies composing the compressed information of target *all_languages_random.txt* trained on the greek reference, with default lang model parameters and finite-context distribution with $k_f = 6$. The exponential downscaling function applied to the frequencies is also shown on the right plot.

The information at each target step for every reference is saved in a NumPy matrix M, with a row for each language reference and a column for each position in the target text. This matrix is then manipulated to report the most likely row index corresponding to the probable language at each column, forming a row vector. From this vector, we segment the target indices into continuous groups, returning at which target position each group starts and to which language it belongs to.

Since the information at the rows is very erratic and has many spikes due to the behaviour of the copy model, where copies are only done sometimes and not for very long, the data at each row is filtered using a low-pass filter to smoothen it. The low-pass filter was implemented using the Fast Fourier Transform (FFT). The idea was to get the frequencies that compose the data, downscale the higher frequencies, and then rebuild the signal using the Inverse Fast Fourier Transform (IFFT). For downscaling, we used the exponential function $e^{-\beta|x|}$, where x is the frequency and β is the frequency dropoff, which dictates how strongly should the higher frequencies be filtered. This function is centered at $x = 0$, which allows positive and negative frequencies close to 0 (the lowest) to have an higher multiplier, but the farther they get from 0 the lower the multiplier is. The downscaling is therefore implemented by multipliying the exponential downscaling function with the amplitudes of the frequencies of the data. An example of the frequencies composing the target data and the corresponding downscaling function is presented in figure 2.

With the matrix M representing the information at each step for every reference, we determine for each column, i.e. position in the target text, which

row provides the minimum amount of information, that is which reference compressed the target symbol the best. However, we don't consider all kinds of minimum values. Since for `locate_lang` we want to identify sections of text that we deem unknown, i.e. they likely aren't related to any of the references, then we have to consider some kind of thresholds that the minimum values have to surpass to be truly accepted.

For this we considered two thresholds:

- minimum threshold: the absolute minimum value should not be larger than a given fraction of the second minimum. This threshold was implemented to disregard sections where there is confusion between more than two references. This idea is similar to the *confidence* metric calculated for `find_lang` in section 2.2.

- static threshold: the absolute minimum value should not be larger than a given static threshold in bits. With this threshold, we disregard sections of text where the compression was not sufficient enough to consider that a reference truly represented the target language. This threshold is defined in terms of the cardinality of the target alphabet. More precisely, the static threshold S in bits is given by the formula:

$$S = \frac{\log_2 |\Sigma|}{\gamma}, \tag{1}$$

  where Σ is the target alphabet and γ is a real positive value. This way, we define the static threshold as a fraction of a baseline compression, in which the reported probability distribution at each step is a uniform distribution with each symbol having $\log_2 |\Sigma|$ bits of information. As such, if we set for instance $\gamma = 1$, then the threshold would be extremely forgiving, since as long as a minimum amount of compression is performed the references are considered. However, as γ increases, the threshold becomes more demanding, and requires the copy model to perform well enough to fall under the threshold. We set γ to a value that we consider appropriate enough for our model's performance, such that if when faced with an unkown language the model's performance should be closer to a uniform distribution and above this static threshold.

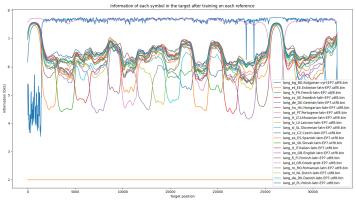
In the end, the following processing arguments are accepted:

- `minimum-threshold`: fraction of the second minimum below which the absolute minimum has to be so that it is considered.

- `static-threshold`: denominator γ that defines the static threshold in terms of the target alphabet's cardinality

- `frequency-filter`: the frequency dropoff of the low pass filter A low-pass filter's frequency dropoff describes the reduction in amplitude or strength of a signal's higher-frequency components as the signal passes through

(a) Example of an output with very low frequency dropoff



(b) Image 2

Figure 3: Example of an output with higher low frequency dropoff

the filter. Low-pass filters attenuate higher frequencies while allowing low-frequency signals to pass. We can see the diference between a low frequency dropoff in figure 3a and a higher frequency dropoff in figure 3.

- `fill-unknown`: make an effort to consider unknown language segments as being of one of the previously identified languages, using forward and backward filling.
- `labeled-output`: print the target text, with segments colored depending on the identified language.
- `plot`: plot the entire matrix M after applying the low pass filter, along with the static threshold.
- `save-result`: save the plot of matrix M to the specified file instead of showing it (only relevant if `plot` was passed).

3 Results

To test the performance of the model and the scripts implemented in this assignment, we used a set of reference texts covering 20 different languages, these being: Bulgarian, Czech, Danish, German, Greek, English, Spanish, Estonian, Finnish, French, Hungarian, Italian, Lithuanian, Latvian, Dutch, Polish, Portuguese, Romanian, Slovak, Slovenian and Swedish. For the target texts, we used a set of texts in the same languages as the reference texts, plus a chinese text, but with different content, which were mainly used to test the `find_lang` script. And finally to test the `locate_lang` script, we combined the previous target files to a single file named *all_languages.txt* and used it as a target text, and also used a small target text called *dog.txt* which also contained some languages mixed in, including chinese, which is not present in our reference files.

3.1 Find lang

To obtain the accuracy of the `find_lang` script, we used the individual language texts and compared the result of the script with the actual language of the text.

Two runs were performed for each language file, one which executed the lang model using its default parameters, such as using a frequency base distribution, and another one which used the finite-context model, instead of the other two options for probability distributions, along with the other default parameters.

For the first run, with default parameters, the script always correctly identified the language of the text, with an accuracy of 1.

$$Accuracy = \frac{correct}{total} = \frac{20}{20} = 1 \tag{2}$$

For the second run, with the finite-context model, the script also correctly identified the language for all the texts used, with an accuracy of 1.

$$Accuracy = \frac{correct}{total} = \frac{20}{20} = 1 \tag{3}$$

To differentiate between the two approaches, we also calculated a confidence value. The way this value is calculated is through the following formula:

$$C = \sqrt[3]{1 - \frac{{m_2}^3}{m_1}},$$

The following tables show the confidence results of the `find_lang` script for the default parameters and the finite-context model, respectively.

By comparing the confidence results of the two approaches, we can see that although both approaches correctly identified the language of the text, the default parameters approach had a much lower confidence value than the finite-context model approach, some of them being even lower than 50%.

The reason for this is that the default parameters approach uses a frequency base distribution, which is not very good at identifying the language of a text, because it does not take into account the order of the characters in the text, which is very important for some languages.

After implementing the confidence value, we also tested the script with a text of a language that was not in the reference files, in this case, chinese. Due to the fact that the script always returns a language, the way to evaluate the accuracy of the script is by checking if the returned language has a very low confidence value.

For the default parameters approach, the script returned the language as hungarian, with a confidence value of 37.7%, which is low, but not low enough considering the other confidence values and the fact that chinese has a completely different alphabet than any of the other languages present in the reference files. In contrast, the finite-context model approach returned the language as swedish, but with a confidence value of 0.0%, which is the lowest possible value, and the correct result for a language that is not in the reference files.

With these results we can conclude that the finite-context model approach is better than the default parameters approach and has a really good accuracy and confidence value.

Language	Confidence
Bulgarian	99.2%
Czech	59.2%
Danish	58.2%
German	60.6%
Greek	98.9%
English	50.5%
Spanish	46.4%
Estonian	53.5%
Finnish	41.2%
French	45.8%
Hungarian	67.1%
Italian	45.1%
Lithuanian	64.7%
Latvian	74.8%
Dutch	35.9%
Polish	72.2%
Portuguese	44.7%
Romanian	74.6%
Slovak	36.1%
Slovenian	30.2%
Swedish	59.8%

Table 1: Confidence results for the `find_lang` script with default parameters

Language	Confidence
Bulgarian	93.1%
Czech	77.3%
Danish	85.5%
German	92.1%
Greek	93.9%
English	92.3%
Spanish	86.4%
Estonian	90.4%
Finnish	89.7%
French	91.7%
Hungarian	92.9%
Italian	90.2%
Lithuanian	90.4%
Latvian	89.8%
Dutch	91.1%
Polish	89.4%
Portuguese	82.7%
Romanian	91.2%
Slovak	77.8%
Slovenian	86.7%
Swedish	89.2%

Table 2: Confidence results for the `find_lang` script with finite-context model

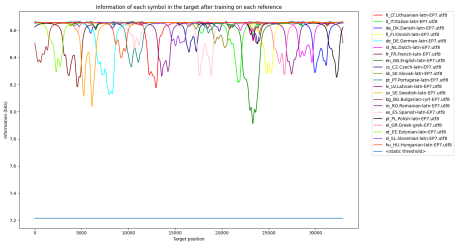


Figure 4: Entropy of each language for *all_languages.txt*,uniform distribution

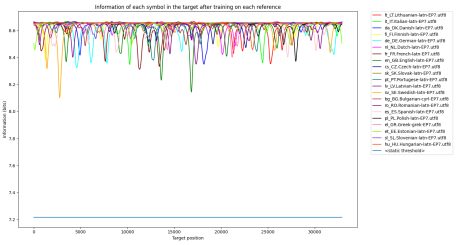


Figure 5: Entropy of each language for *all_languages_random.txt*,uniform distribution

3.2 Locate lang

. . .

For all graphs showing the reported information at each target position, the frequency dropoff used was 1000.

3.2.1 Base probability distribution - Uniform distribution

By using this approach, we can see very clearly the different languages in figure 4 of file *all_languages.txt*, because the entropy of the correct language is much lower than all the others. But by using this approach, we have more difficulty detecting unknown languages (languages that are not in the references). Because even if we define the static threshold at the lowest entropy point of all the correct languages, it will also detect the last chunk of text as French, which is wrong, the last chunk of text in the file is Chinese.

For *all_languages_random.txt*, we can distinguish the languages in the figure 5. We can see the valleys very clearly and infer the language using that information. The same difficulty is seen in this figure: we can't define a static threshold without affecting some of the real languages. (We can't draw a horizontal line that is above all the reference languages and under the Chinese one.)

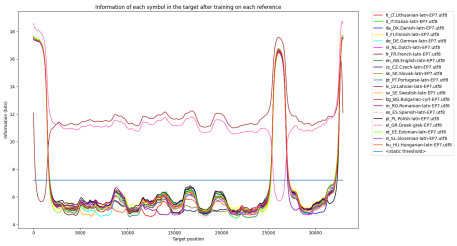


Figure 6: Entropy of each language for *all_languages.txt*,frequency distribution

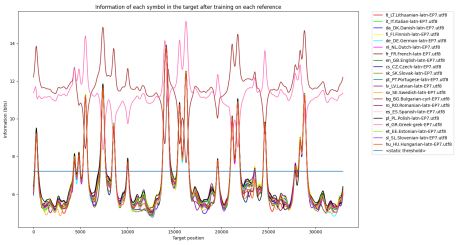


Figure 7: Entropy of each language for *all_languages_random.txt*,frequency distribution

3.2.2 Base probability distribution - Frequency distribution

Using this approach, we can see the different languages in figure 6 of file *all_languages.txt*, although it produces a very different plot compared to the uniform distribution approach.

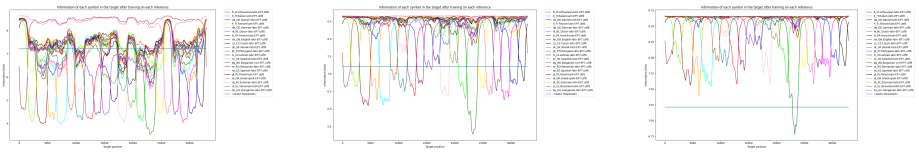
Its still possible to distinguish the languages in figure 7 of file *all_languages_random.txt*, but it is harder to do so, and another difference is between bulgarian and greek languages to the other languages. These have a much higher entropy than the others, due to the fact that these languages use a completely different alphabet than the others.

A downside of this approach is that the static threshold line becomes higher, which means most of the languages will pass the threshold, even in their peak entropy values, at least in the case of the *all_languages.txt* file. For the *all_languages_random.txt* file, the peaks are higher due to each line in the file being a random language.

Still, even with this downside, the script still manages to correctly identify the language of the sections, albeit with less accuracy in the entire interval on the random file, due to the constant change of languages.

3.2.3 Base probability distribution - Finite-context model

To test this approach, various sub parameters were tested, such as the size of the context and the smoothing parameter used to calculate the probability distribution.



(a) $k = 3$ (b) $k = 6$ (c) $k = 9$

Figure 8: Entropy of each language for *all_languages.txt*,finite-context model by changing k

For the context size, k was tested with 3, 6 and 9, and for the smoothing parameter, α, the values tested were 1, 10 and 100.

The results for both files are shown in figures 8 and 9.

By verifiying the results for the context size in figure 8, we can see that the finite-context model produces a much better result than the other approaches, because it is able to clearly distinguish the correct language (the minimum value), with the second best candidate. When increasing k, this difference becomes even more clear, as seen in figures 8a and 8b, although it increases the general entropy value per language, which makes the threshold line to low to detect the correct language in each section.

This problem can also be seen in the random file, in figure 9, where the threshold line is too low to detect the correct language in each section when the context size becomes 6 or 9. Similarly, it is also possible to verify, although not as clearly due to the random nature of the file, that the correct language becomes more clear when increasing the context size.

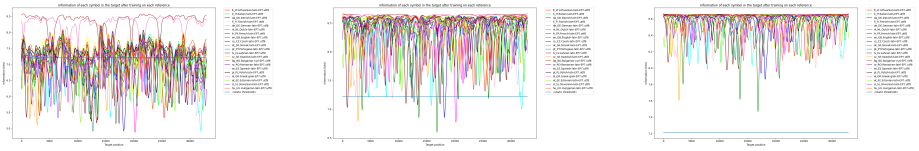
This is due to the threshold not being tailor made for each file, but instead being a static value, which is not ideal, although it is clearly visible what the correct language is in each section due to the difference in entropy values between the correct language and the second best candidate.

Now for the smoothing parameter, α, we can see that the results are very similar for all the values tested, as seen in figures 8a, 10b, with the biggest difference being more noticiable in the random file, in figure 11c, where the static threshold line is too low to detect the correct language. This is due to the fact that increasing the smoothing parameter increases the entropy value of each language, which makes the threshold line too low to detect the correct language in each section.

Through these results, the best value for α was 10 and for the context size was 3, because these helped to distinguish the correct language from the second best candidate, while not increasing the entropy value too much to make the threshold line too low.

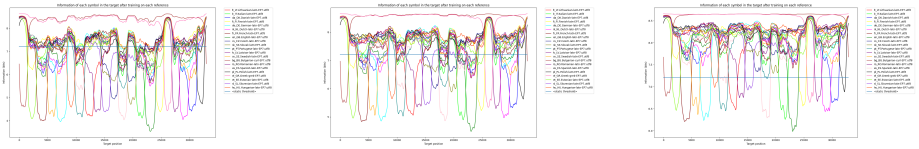
3.2.4 Copy pointer repositioning - Oldest pointer first

To see the effect of the pointer repositioning, we tested the different approaches when using a frequency distribution, which was the default parameter when



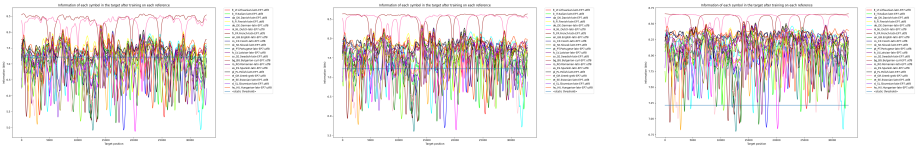
(a) $k = 3$ (b) $k = 6$ (c) $k = 9$

Figure 9: Entropy of each language for *all_languages_random.txt*,finite-context model by changing k



(a) $\alpha = 1$ (b) $\alpha = 10$ (c) $\alpha = 100$

Figure 10: Entropy of each language for *all_languages.txt*,finite-context model by changing α



(a) $\alpha = 1$ (b) $\alpha = 10$ (c) $\alpha = 100$

Figure 11: Entropy of each language for *all_languages_random.txt*,finite-context model by changing α

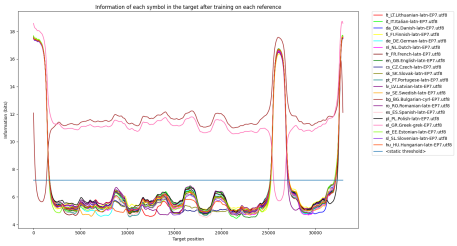


Figure 12: Entropy of each language for *all_languages.txt*,oldest pointer first

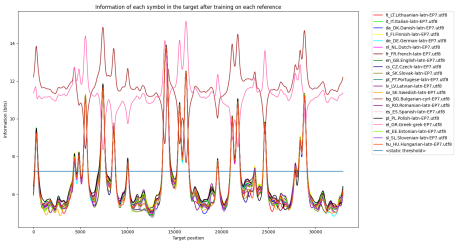


Figure 13: Entropy of each language for *all_languages_random.txt*,oldest pointer first

testing. So these results will be mostly similar we the results shown for the frequency distribution section.

In this approach we can still distinct the different languages, but not as clearly as some of the previous approaches. In the figure 12 we can see that the entropy of the languages are mostly the same, excluding Greek and Bulgarian. One of the positives of this approach is that we can identify much more easily an unknown language (languages that are not in the references) than in some of the previous approaches For the *all_languages_random.txt* file we can see in the figure 12, that the same behavior is seen, but now with slightly more differences between languages.

In contrast to the previous assignment, the lang model is fully trained before compressing the target file. This means that the pointer chosen for each pattern will always be the same. This means that the pointer repositioning will not have much effect on the entropy of the target file, since the pointer will always be the same.

3.2.5 Copy pointer repositioning - Newest pointer first

Like in the older pointer approach, we can still distinct the different languages, using the newest pointer. In both the 14 and 15, we can see that the behavior is extremely similar to the older pointer approach, because the pointer chosen for each pattern will always be the same, just like the older pointer approach.

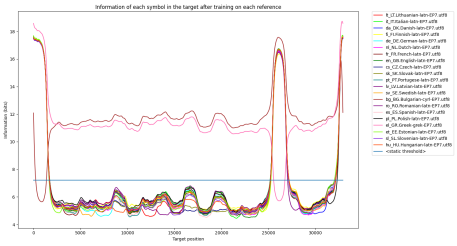


Figure 14: Entropy of each language for *all_languages.txt*,newer pointer first

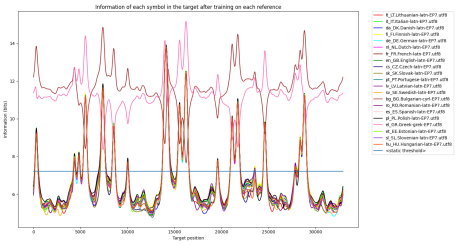


Figure 15: Entropy of each language for *all_languages_random.txt*,newer pointer first

3.2.6 Copy pointer repositioning - Most common prediction

Just like the previous approaches, with this strategy, the pointer chosen for each pattern will always be the same. But in this case, the pointer chosen will be the one with the most common prediction.

Also, like the previous approaches, we can still distinct the different languages, though the results are still identical, as we can see in the figures 16 and 17.

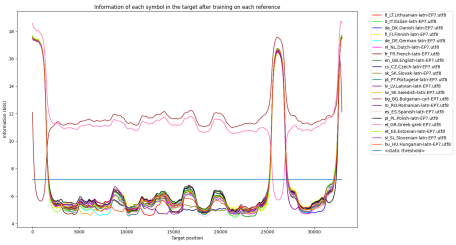


Figure 16: Entropy of each language for *all_languages.txt*,Most common prediction

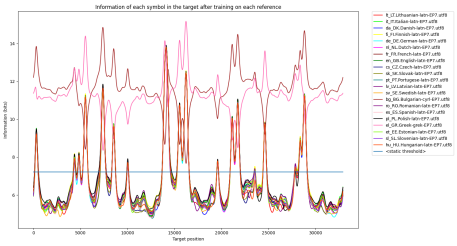


Figure 17: Entropy of each language for *all_languages_random.txt*,Most common prediction

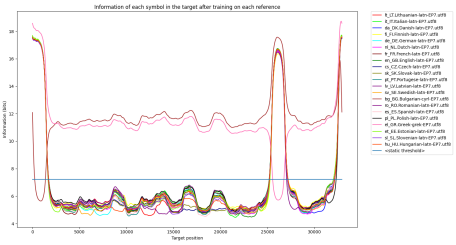


Figure 18: Entropy of each language for *all_languages.txt*,Most common prediction bounded

3.2.7 Copy pointer repositioning - Most common prediction bounded

This version of the most common prediction strategy, is the same as the previous one, but with a bound on the number of pointers that can be chosen. For this test we used the best value for the size of the bound that we verified in the last assignment, which was 100.

Similarly to the previous repositioning strategies, because the model is trained before the compression, the results are identical to the previous approaches. These results can be seen in the figures 18 and 19.

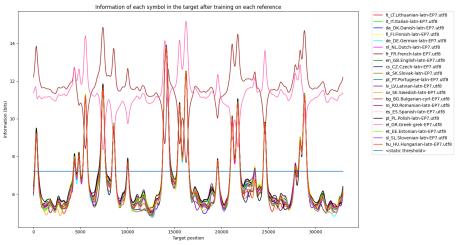


Figure 19: Entropy of each language for *all_languages_random.txt*,Most common prediction bounded

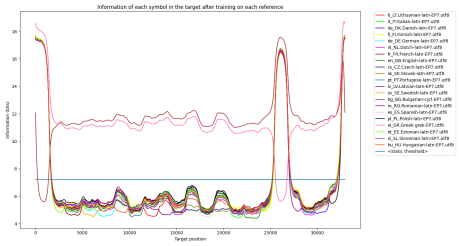


Figure 20: Entropy of each language for *all_languages.txt*,Static threshold

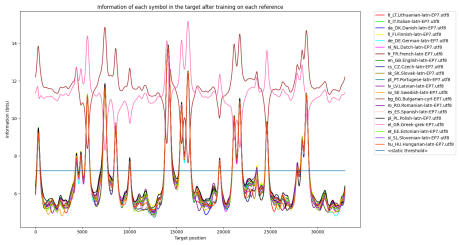


Figure 21: Entropy of each language for *all_languages_random.txt*,Static threshold

3.2.8 Copy pointer threshold - Static threshold

Finnaly, we also tested the different threshold strategies, starting with the static threshold.

In this strategy, the threshold is set to a fixed value, which in this case is 0.5 because it was the one which produced the lowest entropy values in the previous assignment.

In the figures 20 and 21, we can see that the results are identical to the previous approaches.

3.2.9 Copy pointer threshold - Successive fails threshold

In this strategy, the threshold is set to a value that is calculated based on the number of successive fails that the model has. In this case, we used the value that we used in the previous assignment that produced better results, which was 9.

Interestingly, in the figures 22 and 23, we can see that the results, although similar to the use of the static threshold, the entropy values are slightly higher when observing the minimum entropy values, corresponding to the predicted language for the section.

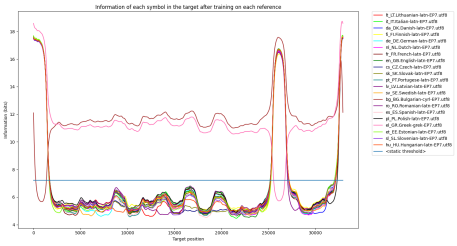


Figure 22: Entropy of each language for *all_languages.txt*,Successive fails threshold

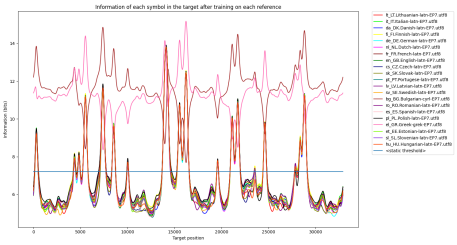


Figure 23: Entropy of each language for *all_languages_random.txt*,Successive fails threshold

3.2.10 Copy pointer threshold - Rate of change threshold

Finnaly, we tested the rate of change threshold strategy with the value 0.01, which is the one that produced the best results in the previous assignment.

By observing the figures 24 and 25, we can see that the difference between the entropy values of the minimun entropy and second minimun entropy is a bit higher than the previous threshold strategies, which means that the model is more confident in the prediction of the language of the section.

This makes this strategy the best one when choosing the threshold to switch pointers.

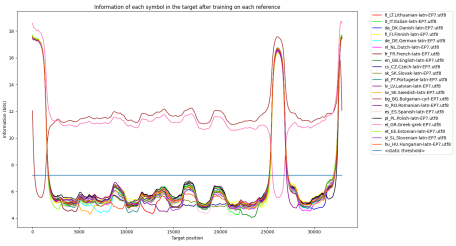


Figure 24: Entropy of each language for *all_languages.txt*,Rate of change threshold

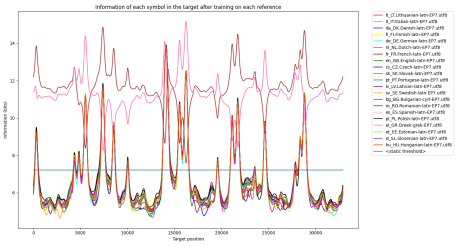


Figure 25: Entropy of each language for *all_languages_random.txt*,Rate of change threshold

3.2.11 Processing parameters - Frequency dropoff

In terms of the `locate_lang` processing parameters, figures 26 and 27 present the effect of the frequency dropoff, i.e. the strength of the low-pass filter, for the *all_languages.txt* and *dog.txt* target files. As can be seen from the plots, there is an ideal range of values for the frequency dropoff, below which the accuracy is very low and above which the accuracy doesn't increase. In this case, we can notice that a frequency dropoff value of around 50 already provides great classification accuracy.

Note that if the frequency dropoff is 0, then it is equivalent to having no low-pass filter applied. Therefore, we can see the heavy influence of the low-pass filter, as applying just some filtering brings the accuracy from 40% without any filter to above 90%. As explained in section 2.3, the copy model's behaviour is very erratic, with spurious and short copies throughout the target text. As such, it's understandable that a low-pass filter heavily improves the classification accuracy, since these spikes of information are attenuated and so we can better study trends over larger segments of the target text instead of symbol-by-symbol.

3.2.12 Processing parameters - Minimum threshold

The minimum threshold dictates how similar is the minimum reference allowed to be to the second minimum reference. We tested values below or equal to 1.0, with 1.0 meaning that the minimum reference can report exactly the same information as the second minimum at a given step. Technically, if the minimum threshold is 1.0, then it has no effect. Figures 28 and 29 present the effect of this parameter on the classification accuracy.

From the plots, we can notice that minimum threshold values below 1.0 decrease the classification accuracy. In the *dog.txt* target the curve is more irregular, but still decreasing. Although the threshold was conceived to reduce ambiguity in the target language when many were likely candidates, it ended up heavily reducing classification performance overall. The reason that likely prevented the program from classifying correctly with this threshold is the transition between languages in the target text. In these transitional sections, the program labels the symbols as unknown since there is confusion, whereas

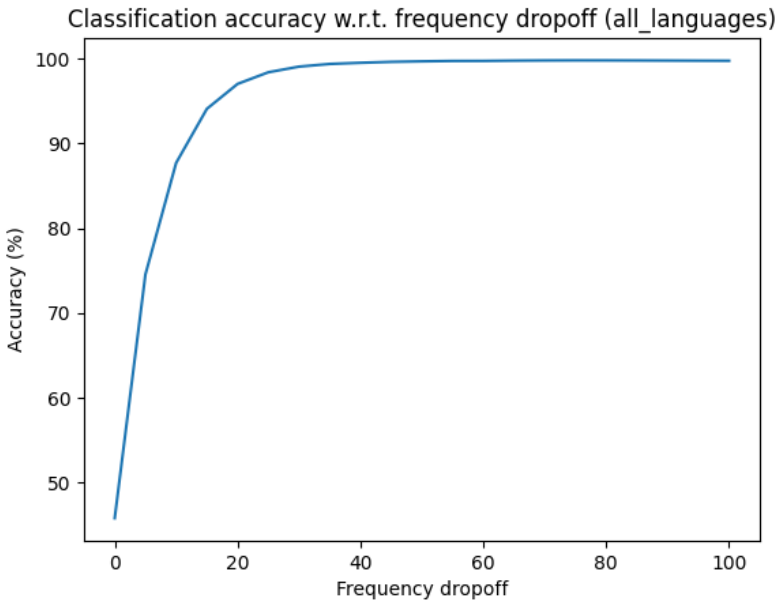


Figure 26: Accuracy with respect to `locate_lang`'s frequency dropoff value, on *all_languages.txt*.

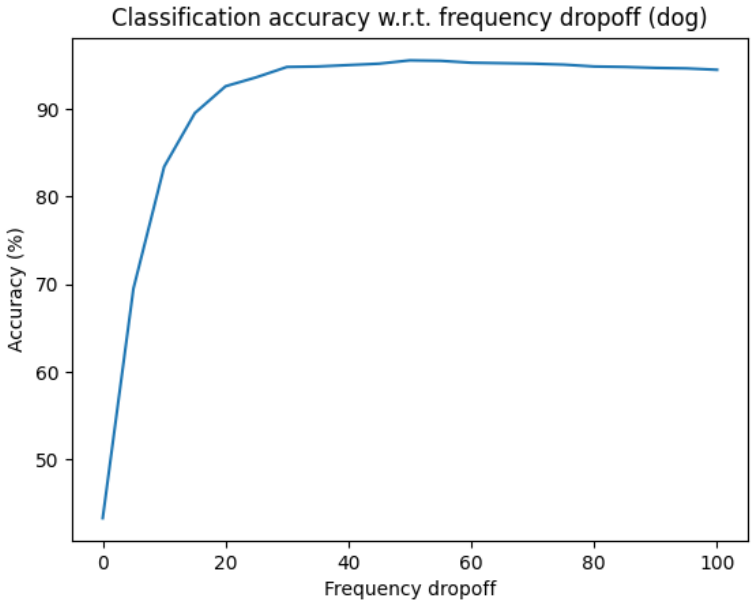


Figure 27: Accuracy with respect to `locate_lang`'s frequency dropoff value, on *dog.txt*.

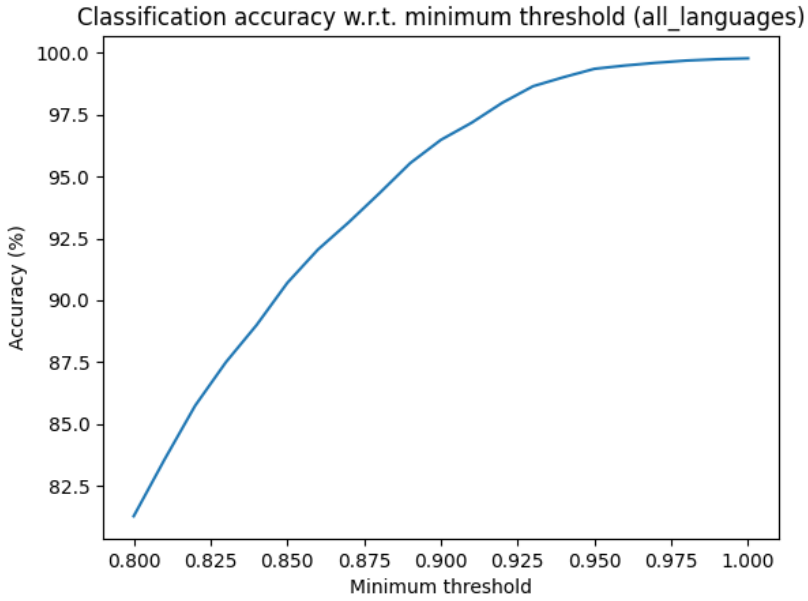


Figure 28: Accuracy with respect to `locate_lang`'s minimum threshold, on *all_languages.txt*.

without any threshold the program tries to make a best effort to classify the perceived language, without reporting any unknowns.

3.2.13 Processing parameters - Static threshold

The `locate_lang`'s static threshold defines a maximum value below which a reference's reported information should be to be considered. More specifically, the parameter value passed to `locate_lang` defines the denominator γ in equation 1, described in section 2.3. We tested different values of γ above 1.0. As previously explained, $\gamma = 1.0$ defines a static threshold that demands a compression performance at least better than reporting a uniform distribution at each step. In practice, if we consider for instance the uniform distribution and the probability distribution based on a finite-context model, then $\gamma = 1.0$ is as if no threshold was applied at all. Figures 30 and 31 present the classification accuracy with respect to the denominator γ of the static threshold.

We can notice that there is a local maximum for both targets. Around $\gamma = 1.2$, we can see that the classification accuracy is the highest, and the larger the value the lower it gets. Curiously, for the *dog.txt* target the classification accuracy is particularly worse for $\gamma = 1.0$ than it is for $\gamma = 1.2$. This makes sense when we consider the purpose of the static threshold. Since the threshold was conceived so that we can identify languages that we may not have trained

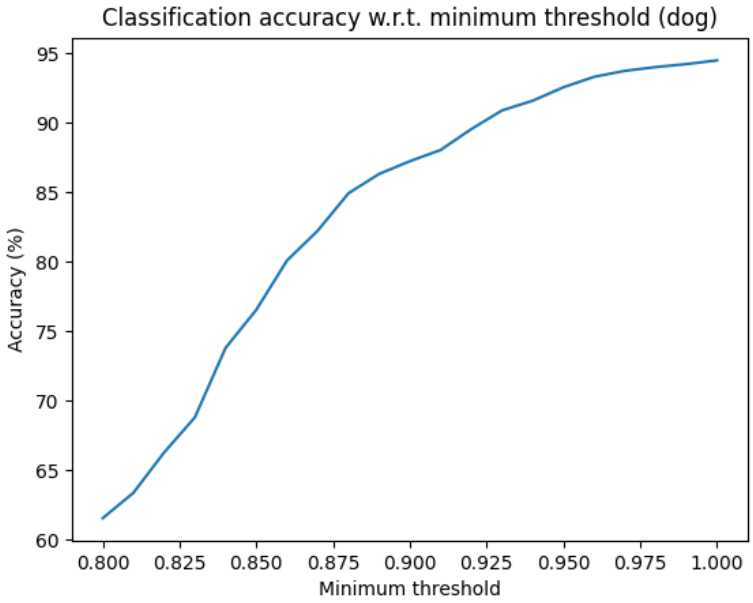


Figure 29: Accuracy with respect to `locate_lang`'s minimum threshold, on *dog.txt*.

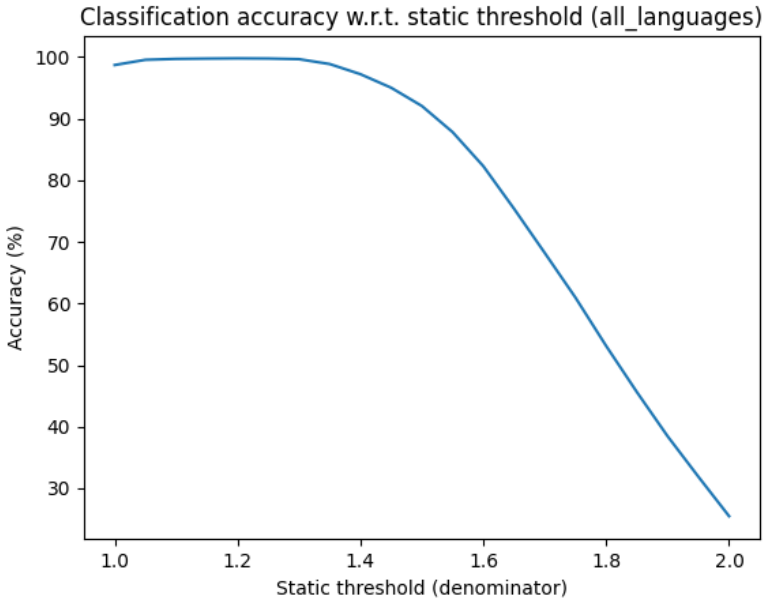


Figure 30: Accuracy with respect to `locate_lang`'s static threshold, on *all_languages.txt*.

on, i.e. are not in the reference bank, then if we set $\gamma = 1.0$ there will be effectively no threshold. As such, we can't identify unknown languages at all. Since both targets have a section in Chinese, which does not have a respective reference, then this section will be guaranteedly misclassified for both targets if $\gamma = 1.0$. The effect is more pronounced in *dog.txt* since the portion that Chinese occupies in the file is larger than in *all_languages.txt*.

Additionally, if γ increases too much, then we are demanding a better performance from the copy model. At this point, it's better to tune this value such that it properly represents the compression performance of the copy model, which we found to be around 1.2.

3.2.14 Variable reference size

We also analyzed the effect of the reference's size on the language classification. As we can see in tables 3 and 4 the accuracy of the language classification increases with the size of the reference. This was the expected behavior since the model has more information to work with and therefore a more complete and complex model. Although we must admit that we weren't expecting such a high accuracy with a reference of only 0.1% of the original reference files. With these results we can conclude that the model is very robust and can work with very small references.

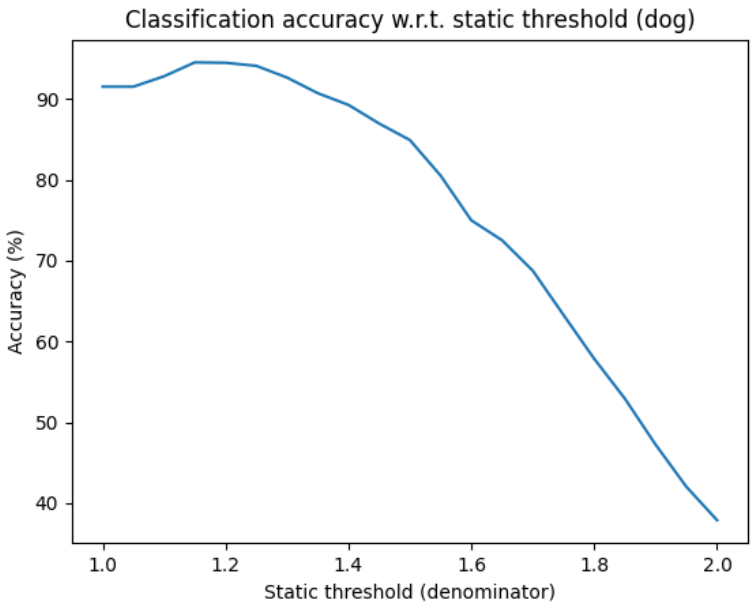


Figure 31: Accuracy with respect to `locate_lang`'s static threshold, on *dog.txt*.

To test this we used the *dog.txt* and *all_languages.txt* files as the target file, the references were made randomly with lines from the original reference files. The size of the reference was varied from 0.1% to 1% with increments of 0.1%, of the original reference files.

We can also conclude that the accuracy of the language classification is affected by the size of the target file, a bigger target file will have a higher accuracy. This happens because the model has more chances to hit the correct language, since the target is bigger.

4 Conclusion

With the development of this assignment, we were able to conclude that our lang model produce good results overall in both scripts developed with some variations dependending on the parameters used, specially when changing to the finite context model approach.

4.1 Find lang

Through the analysis of the results, we can conclude that the best approach to find the language of a text is to use the finite context model. Using the frequency and uniform distribution strategies did also correctly identify the language of

Size of the reference (%)	Accuracy (%)
0.1	87.117552
0.2	92.431562
0.3	93.666130
0.4	93.880837
0.5	93.988191
0.6	93.988191
0.7	94.202899
0.8	94.202899
0.9	94.310252
1.0	94.471283

Table 3: Relation between the size of the reference and the accuracy of the language classification og the *dog.txt* file

Size of the reference (%)	Accuracy (%)
0.1	90.885891
0.2	97.760967
0.3	99.273909
0.4	99.659740
0.5	99.732653
0.6	99.750881
0.7	99.753919
0.8	99.766071
0.9	99.778223
1.0	99.781261

Table 4: Relation between the size of the reference and the accuracy of the language classification og the *dog.txt* file

all target texts texted, although with considerably lower confidence values. In contrast, the finite context model produced much better results with confidence values not lower than 77.3% when testing with the target texts. Another big improvement between the frequency and uniform distribution strategies and the finite context model is the fact that the latter as a much better capacity to confirm that it can't identify a language when is not present in the references, which is a very important feature. For the frequency and uniform distribution strategies, the confidence value for a chinese text was 37.7% for a different language, which is even higher than some correct predictions previously made. In contrast, the finite context model produced a confidence value of 0.0% for the same text, which is the expected result, and guarantees that the model is not making a wrong prediction.

4.2 Locate lang

In the case of the locate lang script, the results were also very positive overall, with the model being able to correctly identify the language of the sections of the target texts, with special emphasis when using the finite context approach, which had a bigger difference in entropy between the correct language and the second most probable language.

By testing the different parameters, we verified that most of the parameters had a very small impact on the results, due to the different nature of the objective between this assignment and the previous one. The only parameter that had a significant impact on the results was the threshold strategy, with the rate of change threshold strategy producing slightly better results than the other two strategies, by creating a slight bigger difference between the entropy values of the correct language and the second most probable language.

5 References