Language Detection Using Copy Model

Organization:	Universidade de Aveiro
Course:	TAI - Teoria Algorítmica da Informação
Date:	Aveiro, Portugal, 29/05/2023
Students:	Afonso Campos, 100055 Daniela Dias, 98039 Hugo Gonçalves, 98497

Project abstract:	Language Detection with resort to copy models.

Index

Introduction

Text Similarity

Text similarity is a key concept in natural language processing (NLP), as it enables comparison and analysis of various texts based on their content, structure, or meaning. It refers to evaluating how similar or related two or more texts are to one another. Text similarity is commonly used in various NLP applications and has several use cases, including:

- Information Retrieval: Text similarity is crucial for search engines to provide accurate and relevant results. By comparing the query text with a large corpus of documents, search engines can retrieve documents most similar to the query.

- Plagiarism Detection: Text similarity identifies instances of plagiarism, where one document is copied from another without proper attribution. By comparing documents or passages, similarity algorithms can highlight potential cases of plagiarism.

- **Language Detection**: Text similarity helps identify the languages in documents and other forms of text (including fragments of text). By comparing the provided text with documents of different languages, we can detect the presence of one or more languages with some degree of confidence.

- Text Summarization: Text similarity helps to summarize large amounts of text by identifying redundant or similar information. Texts with high similarity scores can be merged or filtered to generate concise and informative summaries.

- Clustering and Categorization: Text similarity enables the grouping and categorization of documents based on their content. Similarity measures help in identifying related documents, which can be useful in organizing and structuring large document collections.

- Recommendation Systems: Text similarity is used to recommend similar items or content to users based on their preferences or behavior. By comparing the text of user profiles or items, recommendation algorithms can suggest relevant and similar items of interest.

- Question Answering Systems: Text similarity helps in finding the most relevant answer to a given query. Question-answering systems can rank and retrieve the most appropriate answer by measuring the similarity between the question and the available answers or a knowledge base.

  In this essay, we apply text similarity for **Language Detection**.

Language Detection

Here, we consider the problem of determining the "similarity" between a target text, *t,* and some reference texts, r_i. For instance, each r_i could be a text sample from a specific language, and *t* could be a text whose language needs to be identified.

The traditional approach to solving this classification problem starts with extracting and selecting features. These features are then fed into a function that maps the feature space to the set of classes and performs the classification. However, this approach can be challenging since selecting the smallest set of features with enough distinguishing power to tackle the problem isn't easy and intuitive.

Our approach avoids this challenge by adopting an information-theoretic approach to the classification problem that bypasses the feature extraction and selection stages. Representing the original data using a small set of features can be viewed as a specific type of lossy data compression. We take advantage of compression algorithms to measure the similarity between files.

For each class, represented by the reference text r_i, we create a model that serves as a good description of r_i – a model that requires fewer bits to describe r_i compared to other models. In essence, it acts as an efficient compression model for the members of the class r_i. Consequently, we assign *t* to the class that corresponds to the model requiring fewer bits to describe it, that is, to compress *t*.

The chosen compression model is the **Copy Model** (studied in the previous Lab Work). A copy model is a method for data compression that views data sources as being produced of previously seen, replicating parts and that each new symbol will have already occurred in the past.

Copy Model

The first stage of our approach resulted in the development of a program, named ***lang***, responsible for applying a Copy Model to a target text *t* and reporting the estimated amount of information (i.e. number of bits) required to compress *t* according to a reference r_i.

Obtain Copy Model Parameters

Before computing our Copy Model, we must first obtain its parameters. Similarly to the previous Work Lab, these include the alpha, the window size, and the threshold.

In this Work Lab, however, we have explored the possibility of using finite-context models to provide the probabilities of the non-hit symbols. If *-nFC* (non Finite Context) is omitted, the Copy Model runs with Finite Context. Otherwise, it runs with the previous approach.

Name	Description	Flag	Options
Alpha	Constant used to smooth Hit Probability	-a	]0, 5]
Window Size	Size of the window (k characters)	-k	Integer (positive number)
Threshold	Limit to stop Copy Model	-t	]0, 0.5]
Target File Path	Path to the file containing the target text *t* that will be analyzed	-i	String
Reference File Path	Path to the file containing the reference text r_i	-r	String
Output File Path	Path to the output file (*lang* results)	-o	String
Finite Context	Flag to use Finite Context	-nFC	---

Table 1: Input arguments for Copy model

For this assignment, we have also simplified the Copy Model and removed other features that are considered unnecessary. The last changes included removing the following parameters:

Name	Description	Flag	Options
Serializer Type	Type of Model to use for the Generator	-s	0 - Positional 1 - Probabilistic
No Information	Flag to only run the model for the Generator	-nI	---
Logging Level	Logging Level (Info by default)	-l	1 - Info 2 - Debug
No Serialization	Flag to only run the Copy Model	-nS	---

Table 2: Removed Input arguments for Copy model

The Copy Model parameters are provided to the program through the command line, parsed, and validated accordingly. In these steps, we make use of a class called *LangInputArguments*, which serves several purposes:

- Parsing Command-Line Arguments: The *parseArguments* function takes the *argc* and *argv* parameters and iterates over the arguments, assigning their values to the appropriate member variables of the class based on the provided flags.

- Validating Arguments: The *checkArguments* function validates the parsed arguments to ensure that all the required arguments are provided and their values are within the specified ranges. If any argument is missing or invalid, an error message is printed, and the function returns false. Otherwise, it returns true to indicate that all arguments are valid.

- Providing Access to Argument Values: The class provides getter methods (*getReferenceFilePath*, *getTargetFilePath*, etc.) to retrieve the values of the parsed arguments. These methods allow other parts of the program to access the argument values in a controlled manner.

- Printing Usage Instructions: The *printUsage* function displays the usage instructions and options for the program. It provides a helpful message to guide users on how to provide the required command-line arguments and their corresponding flags.

By encapsulating the argument parsing, validation, and access logic within the *LangInputArguments* class, the code promotes modularity and code reusability. Other parts of the program can simply create an instance of this class, call the necessary functions to parse/validate the arguments and retrieve the values as needed.

Finite-Context

One commonly used approach for representing data dependencies involves the application of Markov models. In our context with lossless data compression, we can use a specific type known as a discrete-time Markov chain or **Finite-Context model.**

A Finite-Context model allows us to gather statistical information of high order from an information source. It assigns probability estimates to symbols in an alphabet based on a context derived from a fixed and finite number of previous outcomes. The primary objective is to predict the next outcome of the source by inferring a model from the observed past sequence of outcomes.

The probability of an event, *e*, is estimated based only on the relative frequencies of previously occurred events:

$$P(e|c) \approx N(e|c) \ / \ \Sigma \, N(s|c)$$

However, this approach suffers from the problem of assigning zero probability to events that were not seen during the construction of the model. To overcome this, we use a "smoothing" parameter to estimate the probabilities:

$$P(e|c) \approx (N(e|c) \ + \ \alpha \,) \ / \ (\Sigma \, N(s|c) \ + \ \alpha|\Sigma|)$$

Obtain Model

To run the Copy Model with the target text *t*, we must first generate a model based on the reference text r_i.

We first create a ReferenceReader responsible for reading sequences from a reference file (taking into account the given window size, *k*). This reader provides access to the read sequences and other related information, abstracting away the details of file I/O and sequence handling, making it easier to work with sequences in the context of model generation.

With the ReferenceReader ready, we can start the ModelGenerator. It's worth noting that, due to the improvements described in the previous sections, our generator creates the following two distinct models with the provided reference:

- Positional Model
- Finite-Context Model

While the Positional Model was imported from the previous assignment - a model that stores the positions of the various sequences of size *k* in the file -, the second model was added to use Finite-Context when calculating the probabilities of the non-hit symbols.

Here, the ModelGenerator reads sequences from the ReferenceReader, until there are no more sequences left and keeps track of information useful for each of the mentioned models.

For each sequence, it adds the current position of the sequence to a map called *sequencePositions*, which takes sequences of characters with size *k* and a vector of the positions where the sequence occurs - **relevant to the Positional Model**.

At the same time, for each sequence, it extracts the last character, along with the sequence without the last character, and it counts the occurrences of each finite context with a map called *finiteContextCounts* - **relevant to the Finite-Context Model**.

Finally, we can obtain the Positional Model (i.e. we obtain the map of sequences strings to vectors of integers, representing the positions of each sequence) and the Finite-Context Model (i.e. we obtain the map of last characters to maps of sequence prefixes and their corresponding counts).

Obtain File Info

Before compressing the target text, *t*, with the generated Copy Model, we must first read the target text in order to extract useful metrics, such as:

- alphabet: The set of unique characters present in the file.
- size: The total number of characters in the file.

In this first pass, the following characters aren't included in the alphabet:

- Tab Character
- New Line Character
- Carriage Return Character
- Comma
- Semicolon

These metrics are needed in order to run the generated Copy Model. After all, to calculate the estimated amount of information required to compress the target text *t* (according to a reference r_i) with the positional model, the Copy Model calculates the probability of correct predictions and the complementary probabilities of the remaining **alphabet** characters.

Run Copy Model

In this step, we run the generated Copy Model using the provided input arguments, the file information obtained in the first pass, and the generated model. For this, we make use of readers with different purposes and different files as input: CopyModelReader (**target file**) and SequentialReader (**reference file**).

The Copy model executor is run with the specified alpha, threshold, and finite context settings. After execution, the files are closed, and a message is printed indicating the completion of the Copy Model execution. Finally, we can store the output generated by the Copy Model executor and later use it in *findlang* and *locatelang*.

Copy Model Reader

The CopyModelReader is responsible for reading and managing the window of characters read from the target used in the Copy Model. In other words, it is responsible for starting/resetting the current window (function *readWindow*) and expanding the current window (function *expand*).

When the Copy Model is initialized, the current window starts empty. Hence, the reader starts by reading the first window of characters from the target file and expands it until the window has the desired size. Expanding - i.e. reading the next character and adding it to the end of the current window - is only possible if the file is open and if we haven't reached the EOF.

During the execution of the Copy Model, the window will be expanded until it meets a certain threshold (we'll go more in detail about the Copy Model in the following sections).

If the window has expanded beyond the specified window size and the Copy Model has reached the threshold, the reader resets the window by shifting the window to a certain offset and expands it again from that point on. The offset is calculated considering how many characters we have expanded with the following expression:

- *this->currentWindow.size() - this->windowSize + 1*.

For example, in the word "cortina", "corti" is the current window (*currentWindowSize* 5) we have set the *windowSize* equal to 3. We need to shift the window 3 times (5-3+1) to have "ti" as the current window. After, we will expand the window again and have "tin" as the current window.

If the window hasn't expanded beyond the specified window size and the Copy Model has reached the threshold, the reader simply shifts it once and expands it to obtain a window with the desired size. Finally, while the Copy Model doesn't reach the given threshold, the reader keeps expanding the window.

Sequential Reader

The SequentialReader is responsible for reading the text from the reference file into memory. This reader was adapted from the RandomAccessReader developed in the previous assignment. While it keeps the same interface (functions *readFile*, *getCharacterAt*), it now handles characters with a variable number of bytes (ranging from one to four bytes).

This adaptation was motivated by variable-sized UTF-8 characters in Greek, Romanian, and Slovak languages. With the RandomAccessReader, we could randomly access the contents of a file directly without loading it to memory. For this, we would obtain a position, multiply by the size of a single *char* (1 byte) and obtain the corresponding character. However, this solution doesn't consider characters with more bytes as we don't know the size of the previous characters.

This problem also motivated us to consider each character as a *string* instead of a *char*, allowing characters composed of more than one byte to be read and interpreted correctly independently of their size.

Copy Model Logic

We kept the same Copy Model logic from the previous assignment and adapted it to include Finite-Context for calculating the probabilities of the non-hit symbols. Suppose the user has set the Copy Model with the useFiniteContext flag. In that case, the probability of non-hit symbols isn't calculated by distributing the complementary probability of the probability of a correct prediction by the remaining characters in the alphabet. Instead, it is calculated based only on the relative frequencies of previously occurred events, in other words, by dividing the number of times the current character occurs in the given context by the total number of times each character in the alphabet occurs in that same context.

The purpose of the smoothing parameter is to prevent zero probabilities and handle cases where certain characters have not been observed within the context.

- probabilityOfFail = (countOfCharWithContext + alpha) / (countOfAlphabetWithContext + alpha * (int)fileInfoReader->getAlphabet().size())

Generate Output

The Copy Model generates metrics that capture the amount of information, information per symbol, and information per iteration generated during the execution - relevant information for further analysis or processing. In summary, the generated output includes

Reference Path:
The path for the reference text, r_i, used in the previously generated model.

Target Path:
The path for the target file used to run the Copy Model.

Information Amount:
The total amount of information, i.e., the number of bits required to compress the target text *t*.

Information Per Symbol:
The amount of information per symbol (from the alphabet) calculated in bits.

Information Per Iteration:
The amount of information in each iteration during the execution of the Copy Model calculated in bits.

All these metrics are written to a separate file to be accessible by the *findlang* and *locatelang* programs.

Language Recognition System

Based on the program *lang*, we have built a language recognition system, ***findlang***, that, from a set of examples from several languages (reference texts, r_i), provides a guess for the language in which a text *t* was written.

Obtain *Findlang* Parameters

The first step of the ***findlang*** script is to parse any command line arguments to execute the language identification according to user specifications. The possible command line options are as follows:

Name	Description	Flag	Options
Executable Path	Executable of the C++ program *lang* which implements the Copy Model logic	-e	String
Reference Files Directory (required)	Path to the directory containing the language references	-r	String
Interactive Mode	Flag to run the language recognition in interactive mode	-it	---
Target File (required if not in Interactive Mode)	Path to the target file *t* we wish to classify	-t	String
Output File Path	Output path for the *lang* program	-o	String

Table 3: Input arguments for the findlang script

This is done with the built-in Python library *sys* in conjunction with a custom python class **InputArguments**. Argument validation is also carried out with the method check_arguments in which required arguments are checked. A default value is assigned to the **Output File Path** in case the user doesn't specify the parameter.

Interactive Mode

Should the user have selected to run the program in interactive mode, a temporary file, "tempTarget.txt" is written. This file's contents are given by the user through terminal input and the path to this new file is then used in place of the **Target File** input argument for the rest of the script.

Testing the Various Languages

With all the input arguments now processed, we are ready to test our target text against the various language references. We iterate over the files in the **Reference Files Directory** (which must be properly labeled with the language name as the file name) and, with each possible language reference, we estimate the total amount of information resulting from the Copy Model compression.

This is done through the *execute_lang* function that uses the *subprocess* Python module to run the ***lang*** program (detailed above) through its executable path (stored in the **InputArguments** instance). The executable is then run with four command line arguments:

- *Reference Path* (-r): the path to the reference currently being tested as a result of iterating over the Reference Files Directory;
- *Target File Path* (-i): the path to the target file (the input argument defined in the *findlang* script);
- *Output File Path* (-o): the equally named input argument in the *findlang* script.

After the execution of the *lang* program, the program's output stored in **Output File Path**, is read to fetch the total amount of information of the target content after compression. These results are then progressively stored in the *information_per_file* dictionary with the language name as the key and the total amount of information after compression as the value.

Determining the Language

With all the language references tested and the respective results stored, we can classify the contents of the target file concerning its language. Theoretically, the compression will be greater when using a reference of the same language as the target text. It is then safe to assume that the language of the reference file that resulted in the highest compression (and, therefore, the lowest amount of total information) will be the language of the target *t*.

With this in mind, we sort the *information_per_file dictionary* by value and fetch the first key-value pair, corresponding to the reference and language that resulted in better compression.

Confidence

Due to the existence of very similar languages (for example, Portuguese and Spanish), it is possible for the *findlang* script to sometimes miss-classify a certain target. With this in mind, we found it relevant to define a metric that represents our confidence in the classification result based on how close the compression results were to other references.

This metric, **confidence**, is calculated as a percentage. A confidence of 100% signals absolute confidence in the classification result, while a confidence of 0% will mean the contrary.

To calculate this metric, we calculate the difference between the total amount of information of the worst compression and the best (*CR*, compression range) and then look at the difference between the total amount of information between the two best compression references (*CU*, compression uncertainty). Confidence is then computed using the following formula:

$$Confidence = \frac{CU}{CR} * 100$$

Finally, all results are printed to the terminal, including the final language classification, the confidence of such classification and 3 runner up languages sorted by their confidence.

Language Location System

Based on the program *lang*, we have built a language location system, ***locatelang***, that can process text *t* containing segments written in different languages and obtain the character position in which the segment starts (as well as the language in which the segment is written).

Input Parameters

The first step of the ***locatelang*** script is to parse any command line arguments to execute the language localization according to user specifications. The possible command line options are as follows:

Name	Description	Flag	Options
Executable Path (required)	Executable of the C++ program *lang* which implements the Copy Model logic	-e	String
Reference Files (required)	Path to the directory with the reference files	-r	String
Interactive Mode	Flag that allows to input text in the terminal and use it as target text *t*	-it	---
Target File (required if not in Interactive Mode)	Path to the file with the target text *t*	-t	String
Window Size	The size of the window for the Low Pass Filter	-m	int
Optimization Level	Level of optimization to be used	-O	int (1 to 3)
Window Overlap	Window overlap - in percentage	-o	float (0 to 1)

Table 4: Input arguments for the locatelang script

This is done with the built-in Python library *sys* in conjunction with a custom python class InputArguments. Argument validation is also carried out with the instance method *check_arguments,* in which required arguments are checked, and a default value is assigned to the **arguments**, in case the user doesn't specify the parameter.

Generating Targets

To create multilingual text files, we developed a Python script ***mix_text*** which uses single-language reference files to output a multilingual file and its respective language bounds within the file contents. This script defines and runs a function *mix_languages* that takes 3 arguments:

- *lang_files*: A dictionary with a string representing the language of a single-language file as key and the file's path as the respective value;
- *n_languages*: Integers specifying the number of languages to be included in the final output;
- *size*: Integer specifying the size, in characters, of the output.

This function samples *n* languages from the available options in *lang_files* and constructs an output file with random segments of text selected from the single-language files. The resulting file is built with small increments, with each increment randomly selecting one of the initially chosen languages, then extracting a random segment with 3 to 15 total words from the language's file. Finally, the segments are added to the output file's contents. The increments are done until the desired number of characters is reached.

A second file is created to keep track of the span of the various languages within the output file. Each line of this file is of the format "`{language} -> ({starting_index}: {ending_index})`", with *starting_index* signaling the start of a language's span within the file's contents (the index of the character where the segment begins) and *ending_index* signaling the end of the span.

We can then use both the output file and the results file to test our ***locatelang*** script, with the results file giving us an easy way to obtain the *locatelang* accuracy.

Identifying Language Segments

As discussed, *locatelang* aims to identify different segments of languages inside a target file as well as the start and end position of each segment. To achieve this, we need to go through the entire target file and check, at each point, which reference is better compressing the target sequence. The reference (i.e., language) that is compressing better should be the one that is indeed present in the target.

As mentioned in a previous section (and observable in Appendix I), our *lang* program generates, as output, the amount of information in each position of the target sequence.

Character-By-Character Comparison

Our first approach was to check, at each position (p_i), which reference had the smallest amount of information and consider that language as the correct one for character *i.* We quickly realized that we could not achieve great results with this simple approach as it doesn't consider any context (words, for example), making the program tremendously unstable in detecting the correct language.

Our approach to mitigate this problem was to implement a system of counters. Considering that the language doesn't change from character to character (we may have several words of the same language before changing to a different language), we created a set of counters, one for each language. Each one of the counters was increased every time that language was detected. We also used a variable to keep track of the "current language" (i.e., the language with the higher number of hits).

We would change this "current language" to a different one when the value in the counter of a different language had a higher value than the counter of the current language. Once this happened, we would reset all the counters and start the process again.

Although this process fixed the instability in language detection, it brought a different problem - it is biased towards the language that occurs more often in the target file. Therefore, if we had *Portuguese* in the target file for 1000 consecutive characters, we would need approximately another 1000 consecutive characters of a different language to change the "current language". We have implemented some patches to this mechanism, such as decreasing the counter of the current language every time a different language is detected. However, this wasn't enough to ensure good results for any kind of target with any sort of language distribution.

Low Pass Filter with Average

Looking at the results obtained from the first approach, we learned that comparing the languages, character by character, would not be the best approach to recognize the language in text segments. This is mainly because natural languages are composed of words, and therefore, there is no need to evaluate the language of every single character without context.

Our second approach uses a sliding window (of configurable size) that slides through the target text and calculates the average information of the characters inside the window for each one of the reference languages. The reference language that has the lowest average should be the correct one. Using this method, we can evaluate an entire word or sentence (depending on the size of the window), which will reduce the instability of the detection (i.e., we will stop detecting many different languages in a few sequential characters) and will also take into consideration some context but will not be biased towards any language like happened on our first approach.

It is worth noting that, to reduce the instability, we only change to a different language if the average of the current window for a certain reference language is below 2 - i.e., if at least some characters were recognized as making part of the reference.

Every time we slide the window, the new window can overlap the old window by an X amount of characters or not. The number of characters to be overlapped can be configured using an input argument, which sets the number of overlapped characters as X% of the window size. By default, the overlapping is disabled. Please check more details in the section "Input Parameters".

Obtaining Accuracy

To evaluate and improve our language detection system, it is important to measure the **accuracy** of the recognition. Once we generate our own targets by mixing up several languages (more details in the “Generating Targets” section), we know exactly the language of each character in the target. To obtain the accuracy, we simply compare the real language of each character in the target to the one indicated by the *locatelang* program. We count a hit if the languages are the same and a miss otherwise. Therefore, the formula below gives the accuracy (in percentage).

$$Accuracy = \frac{Hits}{Hits + Misses} * 100$$

Optimizing Results

By analyzing the obtained results for different targets, we realized that our program often identified languages that were not present in the target. Most of those languages were identified in a residual number (once or twice), however, they were reducing the accuracy of the detection. In contrast, we have discovered that the languages that were indeed present in the target file were almost always the ones that were detected most times in the target.

```
{'spanish.txt': 36, 'greek.txt': 31, 'czech.txt': 11, 'italian.txt': 5, 'portuguese.txt': 2,
          'french.txt': 2, 'romanian.txt': 2, 'polish.txt': 1, 'english.txt': 1}
```

Snippet 1: Languages detected for target with Greek, Spanish, and Czech and the number of times they were detected.

As we can see in the Snippet above, for a target with 3 languages - Greek, Spanish, and Czech - many other languages were found, such as English, Polish, and Romanian. However, most of these languages were detected only a few times, while the languages that were indeed present in the target were the most detected ones.

This motivated us to create an optimization mechanism that would remove the languages with fewer hits and keep the ones with higher hits. By removing languages with fewer hits, we are slowly approximating the languages present in the target and removing the ones identified incorrectly.

We do this with an iterative process. At each new iteration, a threshold is calculated (based on the optimization level), and all the languages below that threshold are eliminated. Then, we repeat the process by re-calculating the identified languages and their counts (i.e., rerunning the *locatelang*) and calculating a new threshold to remove the residual languages. The threshold at each iteration is given by:

$$Threshold = \lceil \frac{mean(languages)}{len(languages)} * optimization\ level \rceil$$

Evidently, this mechanism is not perfect and can accidentally remove languages that were indeed present in the target depending on the number of iterations used (the higher the number of iterations, the higher the probability of removing “correct” languages). If this happens, we will see a huge decrease in the final accuracy of the detection instead of improving it. To mitigate this problem, we have configured three different levels of optimization that can be adjusted through the *-O* input argument:

- **Level 1** (-O 1): Provides the less aggressive optimization. It can remove residual languages, i.e., ones identified once or twice in the target.

- **Level 2** (-O 2): This is more aggressive than level 1 and can improve the accuracy values even more. It has a higher risk of removing a language that is indeed present in the target file.

- **Level 3** (-O 3): The most aggressive optimization. It has a somewhat high probability of removing a language present in the target but can obtain great results in some situations.

Each optimization level adds one iteration to the process of removing languages and increases the threshold used to remove the residual languages.

Results

Taking into account the recommended number of different languages for our problem (at least 20), we have collected reference texts in the following languages: Lithuanian, Polish, Italian, Hungarian, Swedish, Slovenian, Greek, English, Romanian, Danish, Slovak, Finish, Spanish, Estonian, Portuguese, Bulgarian, Czech, Dutch, German, French. The decision to choose each language was motivated by the following reasons:

- The presence of linguistic relationships between the considered language and other included languages.
- The presence of characters that are unique to the considered language.

In the following sections, we study the results obtained from the execution of both *findlang* and *locatelang* and we achieve multiple conclusions, ranging from the quality of our implementations to the linguistic relationships between the chosen languages.

FindLang Results

Lithuanian

Time	Window Size	Detected Language	Confidence	Similar Languages		
12.236	3	lithuanian	67.65	'FINISH'	'ROMANIAN'	'HUNGARIAN"
13.396	5	lithuanian	79.48	'PORTUGUESE'	'ESTONIAN'	'HUNGARIAN"
15.673	10	lithuanian	100	'POLISH'	'ITALIAN'	'HUNGARIAN"
14.446	12	lithuanian	100	'POLISH'	'ITALIAN'	'HUNGARIAN"

Polish

Time	Window Size	Detected Language	Confidence	Similar Languages		
12.684	3	polish	36.9	'SLOVAK'	'CZECH'	'DUTCH"
14.033	5	polish	69.57	'SLOVAK'	'CZECH'	'SLOVENIAN"
16.411	10	polish	100	'LITHUANIAN'	'ITALIAN'	'HUNGARIAN"
14.922	12	polish	100	'LITHUANIAN'	'ITALIAN'	'HUNGARIAN"

Italian

Time	Window Size	Detected Language	Confidence	Similar Languages		
12.316	3	italian	33.1	'ROMANIAN'	'PORTUGUESE'	'SPANISH"
13.276	5	italian	40.38	'PORTUGUESE'	'ROMANIAN'	'SPANISH"
15.42	10	italian	80.06	'PORTUGUESE'	'SPANISH'	'ROMANIAN"
14.392	12	italian	81.6	'SPANISH'	'PORTUGUESE'	'ROMANIAN"

Hungarian

Time	Window Size	Detected Language	Confidence	Similar Languages

12.404	3	hungarian	48.08	'SLOVAK'	'ENGLISH'	'GERMAN"
13.225	5	hungarian	79.59	'ENGLISH'	'SWEDISH'	'CZECH"
15.396	10	hungarian	97.95	'SLOVENIAN'	'SLOVAK'	'SPANISH"
14.402	12	hungarian	100	'LITHUANIAN'	'POLISH'	'ITALIAN"

Swedish

Time	Window Size	Detected Language	Confidence	Similar Languages		
12.345	3	swedish	32.83	'DANISH'	'DUTCH'	'GERMAN"
13.284	5	swedish	45.93	'DANISH'	'GERMAN'	'DUTCH"
15.375	10	swedish	80.7	'DANISH'	'LITHUANIAN'	'POLISH"
14.415	12	swedish	100	'LITHUANIAN'	'POLISH'	'ITALIAN"

Slovenian

Time	Window Size	Detected Language	Confidence	Similar Languages		
12.451	3	slovenian	5.05	'CZECH'	'SLOVAK'	'DANISH"
13.734	5	slovenian	46.68	'SLOVAK'	'CZECH'	'POLISH"
16.629	10	slovenian	97.19	'CZECH'	'LITHUANIAN'	'POLISH"
14.936	12	slovenian	100	'LITHUANIAN'	'POLISH'	'ITALIAN"

Greek

Time	Window Size	Detected Language	Confidence	Similar Languages		
12.374	3	greek	97.69	'DUTCH'	'LITHUANIAN'	'POLISH"
13.818	5	greek	100	'LITHUANIAN'	'POLISH'	'ITALIAN"
16.349	10	greek	100	'LITHUANIAN'	'POLISH'	'ITALIAN"
15.331	12	greek	100	'LITHUANIAN'	'POLISH'	'ITALIAN"

English

Time	Window Size	Detected Language	Confidence	Similar Languages		
12.446	3	english	47.9	'DUTCH'	'FRENCH'	'SPANISH"
13.499	5	english	52.94	'DUTCH'	'DANISH'	'FRENCH"
16.192	10	english	97.81	'ROMANIAN'	'LITHUANIAN'	'POLISH"
15.339	12	english	100	'LITHUANIAN'	'POLISH'	'ITALIAN"

Romanian

Time	Window Size	Detected Language	Confidence	Similar Languages		
12.381	3	romanian	54.98	'PORTUGUESE'	'ENGLISH'	'SLOVAK"
13.714	5	romanian	54.03	'ITALIAN'	'PORTUGUESE'	'SPANISH"
16.28	10	romanian	100	'LITHUANIAN'	'POLISH'	'ITALIAN"
14.447	12	romanian	100	'LITHUANIAN'	'POLISH'	'ITALIAN"

Danish

Time	Window Size	Detected Language	Confidence	Similar Languages		
12.567	3	danish	23.78	'SWEDISH'	'DUTCH'	'ENGLISH"
13.762	5	danish	31.57	'SWEDISH'	'DUTCH'	'ENGLISH"
16.361	10	danish	100	'LITHUANIAN'	'POLISH'	'ITALIAN"
14.82	12	danish	100	'LITHUANIAN'	'POLISH'	'ITALIAN"

Slovak

Time	Window Size	Detected Language	Confidence	Similar Languages		
12.482	3	slovak	16.59	'CZECH'	'SLOVENIAN'	'POLISH"

14.075	5	slovak	27.75	'CZECH'	'SLOVENIAN'	'POLISH"
16.537	10	slovak	77.36	'CZECH'	'SLOVENIAN'	'LITHUANIAN"
14.805	12	slovak	100	'LITHUANIAN'	'POLISH'	'ITALIAN"

Finish

Time	Window Size	Detected Language	Confidence	Similar Languages		
12.445	3	finish	59.61	'ESTONIAN'	'SWEDISH'	'DUTCH"
13.251	5	finish	59.85	'ESTONIAN'	'SWEDISH'	'DANISH"
15.266	10	finish	100	'LITHUANIAN'	'POLISH'	'ITALIAN"
14.367	12	finish	100	'LITHUANIAN'	'POLISH'	'ITALIAN"

Spanish

Time	Window Size	Detected Language	Confidence	Similar Languages		
12.37	3	spanish	9.19	'PORTUGUESE'	'ITALIAN'	'FRENCH"
13.49	5	spanish	22.34	'PORTUGUESE'	'ITALIAN'	'FRENCH"
16.032	10	spanish	61.35	'PORTUGUESE'	'ITALIAN'	'FRENCH"
15.976	12	spanish	80.69	'PORTUGUESE'	'ITALIAN'	'LITHUANIAN"

Estonian

Time	Window Size	Detected Language	Confidence	Similar Languages		
12.326	3	estonian	45.85	'DUTCH'	'FINISH'	'SWEDISH"
13.574	5	estonian	55.3	'FINISH'	'SWEDISH'	'ENGLISH"
16.845	10	estonian	97.33	'FINISH'	'LITHUANIAN'	'POLISH"
14.318	12	estonian	100	'LITHUANIAN'	'POLISH'	'ITALIAN"

Portuguese

Time	Window Size	Detected Language	Confidence	Similar Languages		
12.428	3	portuguese	3.63	'SPANISH'	'ITALIAN'	'ROMANIAN"
14.528	5	portuguese	24.43	'SPANISH'	'ITALIAN'	'ROMANIAN"
16.516	10	portuguese	81.45	'SPANISH'	'FRENCH'	'LITHUANIAN"
15.68	12	portuguese	84.51	'SPANISH'	'LITHUANIAN'	'POLISH"

Bulgarian

Time	Window Size	Detected Language	Confidence	Similar Languages		
12.343	3	bulgarian	99.6	'POLISH'	'ITALIAN'	'ENGLISH"
13.207	5	bulgarian	100	'LITHUANIAN'	'POLISH'	'ITALIAN"
15.625	10	bulgarian	100	'LITHUANIAN'	'POLISH'	'ITALIAN"
15.764	12	bulgarian	100	'LITHUANIAN'	'POLISH'	'ITALIAN"

Czech

Time	Window Size	Detected Language	Confidence	Similar Languages		
12.443	3	czech	13.27	'SLOVAK'	'SLOVENIAN'	'POLISH"
13.727	5	czech	28.87	'SLOVAK'	'SLOVENIAN'	'POLISH"
16.01	10	czech	72.74	'SLOVAK'	'LITHUANIAN'	'POLISH"
15.285	12	czech	100	'LITHUANIAN'	'POLISH'	'ITALIAN"

German

Time	Window Size	Detected Language	Confidence	Similar Languages		
12.38	3	german	41.61	'DUTCH'	'DANISH'	'FRENCH"
13.55	5	german	55.89	'DUTCH'	'SWEDISH'	'DANISH"
17.286	10	german	98.81	'DUTCH'	'LITHUANIAN'	'POLISH"

15.471	12	german	100	'LITHUANIAN'	'POLISH'	'ITALIAN"

Dutch

Time	Window Size	Detected Language	Confidence	Similar Languages		
12.382	3	dutch	54.64	'SWEDISH'	'DANISH'	'ENGLISH"
14.052	5	dutch	47.5	'DANISH'	'SWEDISH'	'GERMAN"
16.091	10	dutch	98.58	'GERMAN'	'FINISH'	'LITHUANIAN"
15.719	12	dutch	100	'LITHUANIAN'	'POLISH'	'ITALIAN"

French

Time	Window Size	Detected Language	Confidence	Similar Languages		
12.352	3	french	49.41	'SPANISH'	'ENGLISH'	'ROMANIAN"
13.963	5	french	57.86	'ENGLISH'	'SPANISH'	'DUTCH"
15.196	10	french	94.71	'ENGLISH'	'SPANISH'	'ITALIAN"
14.796	12	french	92.05	'ENGLISH'	'ITALIAN'	'LITHUANIAN"

Accuracy

Firstly we need to study the performance of the implemented program, *findlang*. This means studying how accurate it was at identifying the language in the target text *t*.

By observing the following figure (and the previous tables with all the unprocessed results), we can conclude that the accuracy was 100% for all languages, independently of the selected window size.

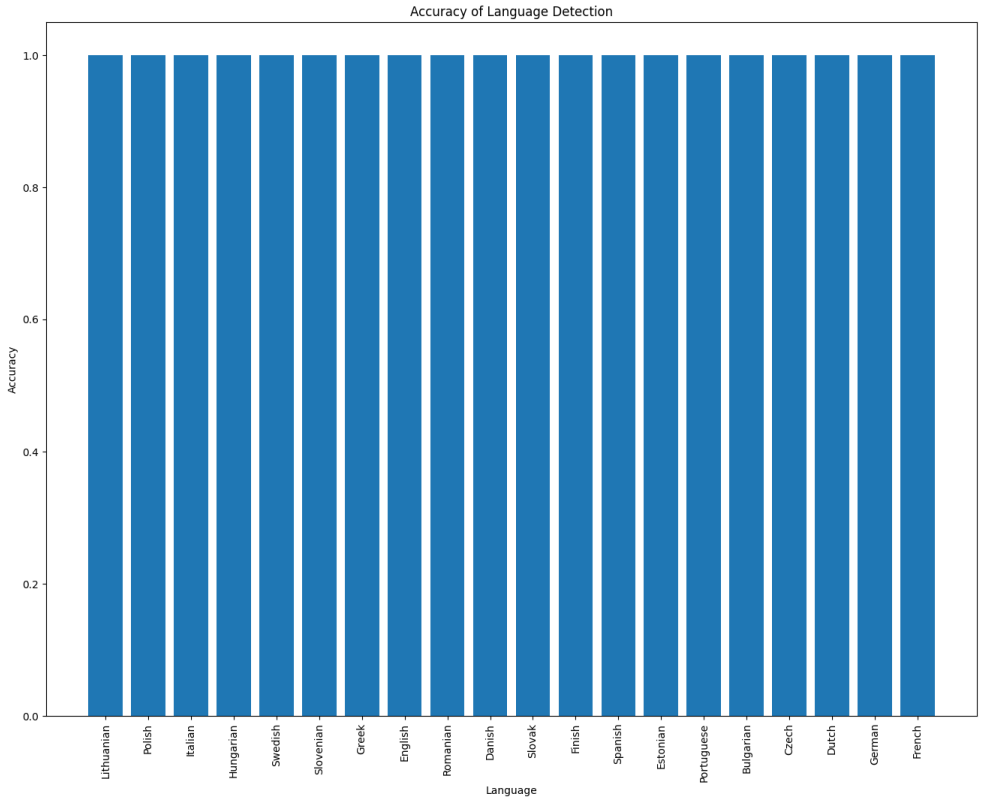

Fig 1: Accuracy of Language Detection

Window Size

For this assignment, we varied the value of the window size to study the impact it had on the accuracy and confidence of *findlang* (in detecting the language of the target text). In the previous section, we have already concluded that the window size had no impact on the accuracy of the results.

Hypothesis: The greater the window size, the greater the confidence that the detected language is indeed the language of the target text.

This hypothesis takes into account that, when we increase the window size, the Copy Model has a higher probability of identifying complete words and it is capable of better identifying the context of the different segments of text.

Fig. 1 helps us visualize that, indeed, the confidence increases as the window size increases for all possible languages.

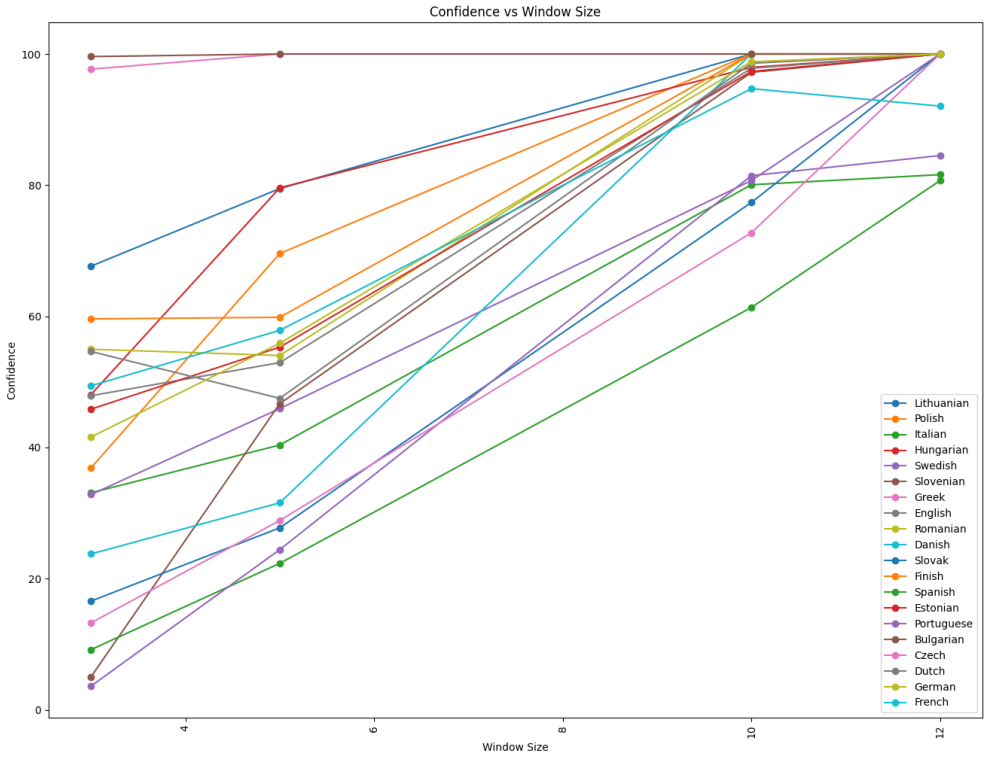

Fig 2: Confidence vs Window Size for All Languages

Language Relationships

Throughout history, languages have evolved and developed, giving rise to diverse linguistic systems. Language relationships encompass the connections between different languages, revealing shared origins, influences, and interactions.

One fundamental concept in language relationships is language families. Languages can be classified into different language families based on their genetic relationship, indicating their shared origins from a common ancestor or their descent from one another.

Hypothesis: Languages from the same language families (based on their genetic relationship) will be detected as similar languages more often.

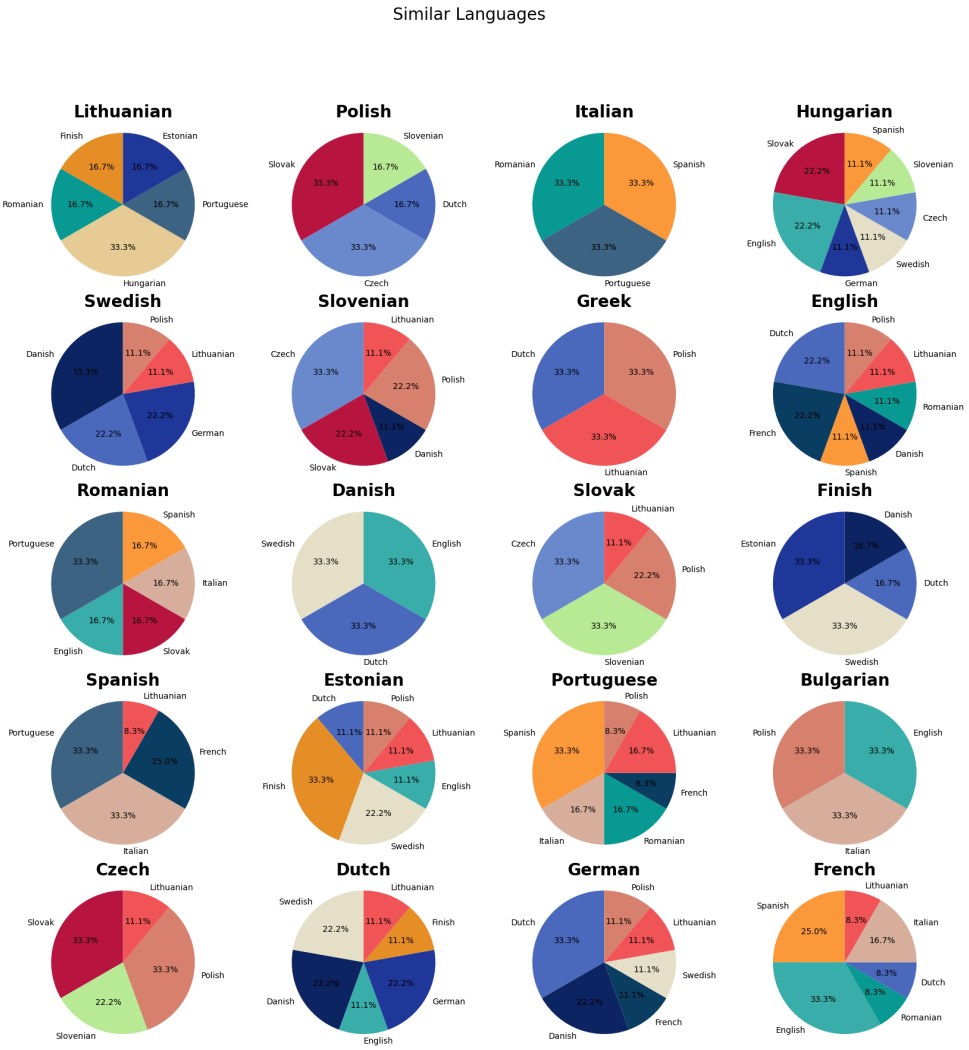

Fig 3: Similarities detected between languages

For instance, the Romance languages like Portuguese, Spanish, Italian, French, and Romanian are considered to have a linguistic genetic relationship as they all evolved from the spoken Latin of ancient Rome.

This relationship can easily be confirmed if we consider the similarity percentages (obtained from the detected similar languages) between the mentioned Romance languages. From Fig. 3, we can obtain the following similarity percentages:

Portuguese
33.3% Spanish, 16.7% Italian, 16.7% Romanian, 8.3% French

Spanish
33.3% Portuguese, 33.3% Italian, 25.0% French

Italian
33.3% Portuguese, 33.3% Romanian, 33.3% Spanish

French
25.0% Spanish, 16.7% Italian, 8.3% Romanian

Romanian
16.7% Portuguese, 25.0% Italian, 8.3% Spanish

Additionally, these similarities can be used to perform hierarchical clustering and group the languages in terms of similarity (i.e. hierarchical clustering creates groups so that objects within a group are similar to each other and different from objects in other groups).

The resulting clusters can then be visually represented in a hierarchical tree called a dendrogram, a diagram that shows the hierarchical relationship between elements as long as we can measure their similarity to each other. This type of visualization is useful for detecting language families.

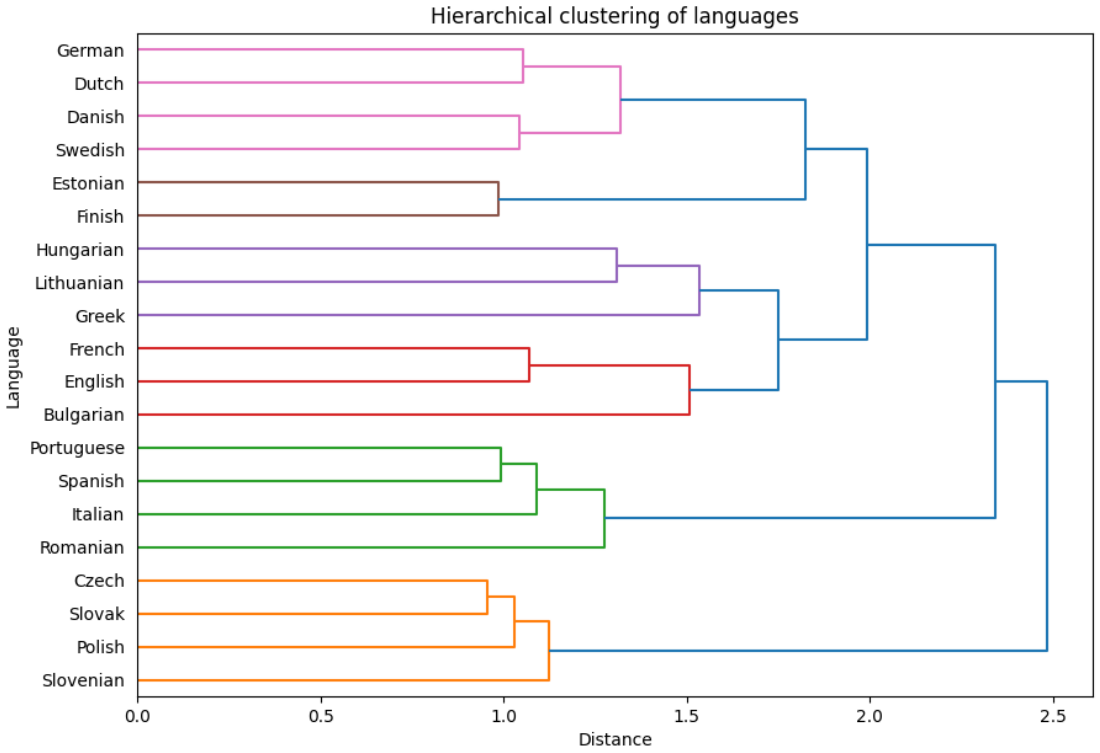

Fig 4: Hierarchical clustering of languages

By analyzing the figure Fig 4, we can detect the following language families:

- **Finnic Languages**: Estonian, Finish
- **Germanic Languages**: German, Dutch, Danish, Swedish
  - **North Germanic Languages**: Danish, Swedish
  - **West Germanic Languages**: German, Dutch
- **West Slavic Languages**: Czech, Slovak, Polish, Slovenian
- **Romance Languages**: Portuguese, Spanish, Italian, Romanian

While language relationships provide insights into possible linguistic relationships, there can be exceptions. For example, adopting and intermixing specific languages can complicate the correlation between language and genetic relationships.

This explains the similarity seen between the French and English languages. French is a Romance language descended from Latin with German and English influences. At the same time, English is a Germanic language with Latin and French influences.

LocateLang Results

General Results

To illustrate our results, we created a visual way, using colors in the terminal to show the languages and where they were located. We can check which language was identified and where using the colors and the legend.

Fig. 5: Locate Lang results for the english_polish_czech target. Default parameters. No Optimization.

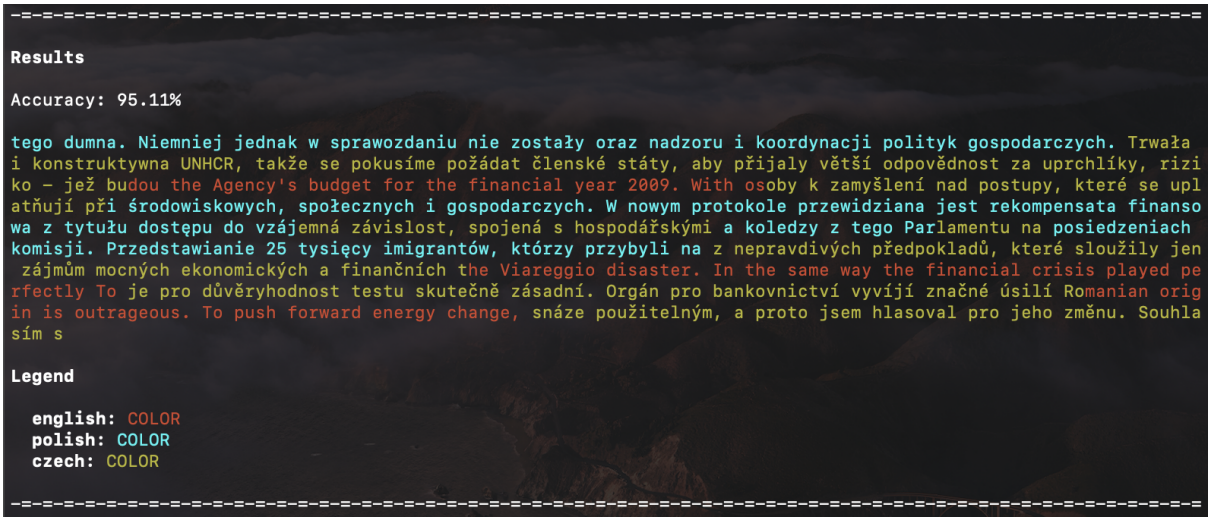

Fig. 6: Locate Lang results for the english_polish_czech target. Default parameters. Optimization Level 3.

Fig. 7: Locate Lang results for the french_romanian_greek target. Default parameters. No Optimization.

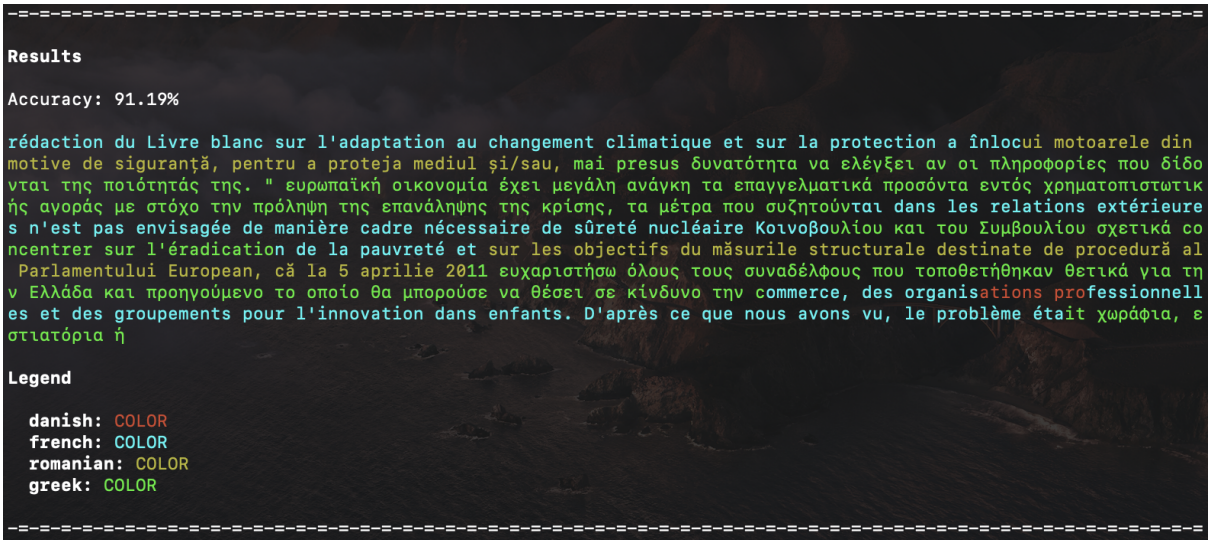

Fig. 8: Locate Lang results for the french_romanian_greek target. Default parameters. Optimization Level 3.

To understand the limits and power of our Language-Location system, we have measured the accuracy obtained by the program by varying the input parameters for the different targets. The best accuracy results for the different targets and the input parameters used to obtain them are listed in the table below.

	English Polish Czech	French Italian Hungarian	French Romanian Greek	German Estonian Finish	Greek Spanish Czech	Swedish German Slovak
Accuracy	**95.41 %**	**95.83%**	**95.25%**	**97.63%**	**99.26%**	**94.67%**
K	10	12	10	5	5	12
Window Size	3	3	5	15	15	3
Window Overlap	0	0	0.4	0.4	0.2	0.2
Optimization	1	0	0	0	2	1
Finite-Context	True	True	False	True	True	False

Table 5: Best accuracy results for the different tested targets.

As we can observe in Table 5, our *locatelang* implementation obtained an accuracy of 99.26% for the target with Greek, Spanish, and Czech. Once these results were obtained for targets with 1000 characters, our locatelang could identify the language in 992 characters, only failing in 8 characters!

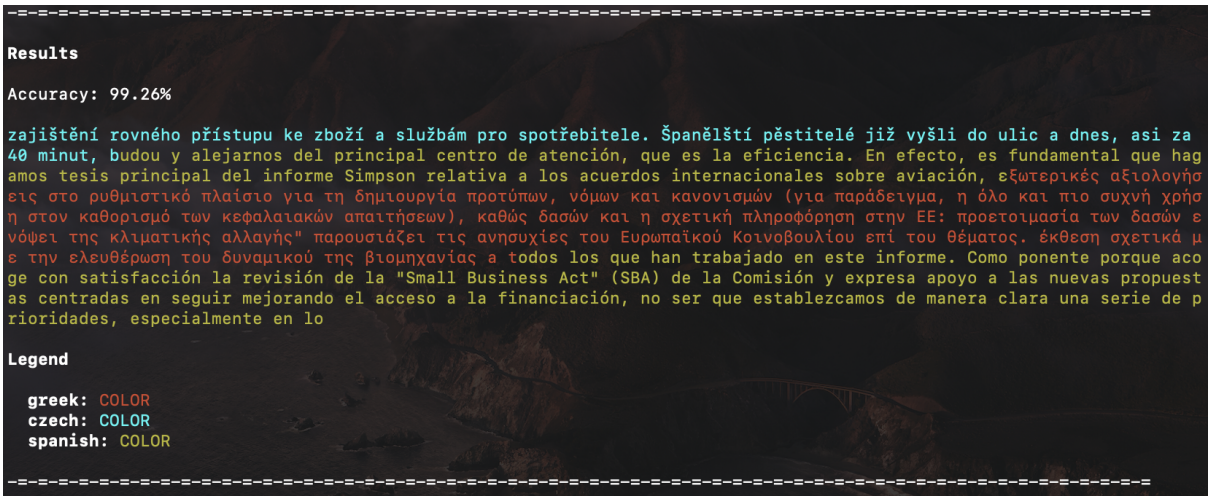

Fig 9: Output provided by the *locatelang* program for the target with the best accuracy.

Optimization Effectiveness

As discussed above, we have implemented a mechanism to optimize the results for the LocateLang program. These optimizations are performed independently of the Copy Model (Lang) results.

Hypothesis: Higher optimization levels should provide higher values of Accuracy.

Fig. 10: Accuracy for different targets with different Optimization Levels

Analyzing the figure above, we can see that the optimization mechanism works for the considered parameters - K=3, Window Size = 10, no window overlapping, and Finite-Context enabled in the copy model. For almost every target, the optimization turned on outperforms the accuracy obtained when the optimization is turned off. Besides that, we can understand that each optimization level can improve the accuracy even more than the preceding one. It is worth noting that the last level of optimization (level 3) is the one that provides higher accuracy for every single target, as expected.

However, as discussed in the “Optimizing Results” section, higher optimization levels may sometimes not reflect higher accuracy values. Higher optimization levels are more aggressive and can remove correctly identified languages, reducing accuracy. This situation can be verified in the figure below, where we changed the K size from 3 to 5. This change reduced the number of false positives (languages identified as being in the target that were not there). It impacted the optimization process, leading it to remove correctly identified languages.

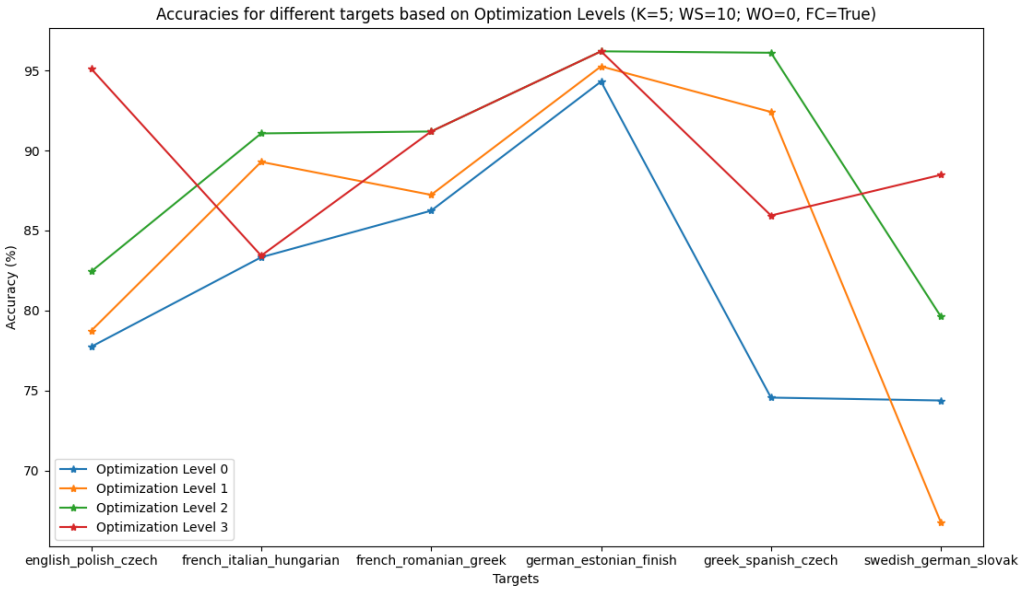

Fig. 11: Accuracy for different targets with different Optimization Levels - Optimization Level 3 can be too aggressive.

As we can see, for most of the targets, Optimization Level 3 was not capable of improving the results, quite the contrary, it worsened the accuracy results for most of the targets.

Although Optimization is a great resource to increase accuracy, it should only be used when there are many false positives. The different Optimization levels are also very important and allow the user to control the mechanism aggressiveness and combine higher values of K with the Optimization to obtain better accuracy.

Accuracy (in the function of K)

It is important to understand how the different values of K (window size for the Copy Model) can influence the accuracy of the Locate Lang program. Before looking at the results, we have established the following Hypothesis.

Hypothesis: Low values of K should result in low accuracy. High values of K should result in low accuracy. Median values (not too low or high) should result in better accuracy.

With lower K values, we may often identify the languages incorrectly. If we have K=2, we will find sequences of 2 characters in the Copy Model that match the reference model. This means that in a Portuguese word, we may find sequences of two characters that appear in other languages too. This will lead us to incorrectly identify the Portuguese word as a different language, increasing the number of false positives and reducing the accuracy. With high values of K, K=10, for example, the process will be reverted, i.e., we may fail to identify a word in Portuguese because the word size is smaller than K. Therefore, we may identify the word as a different language and not Portuguese. Consequently, values of K that are neither too high nor too low may find the best accuracy.

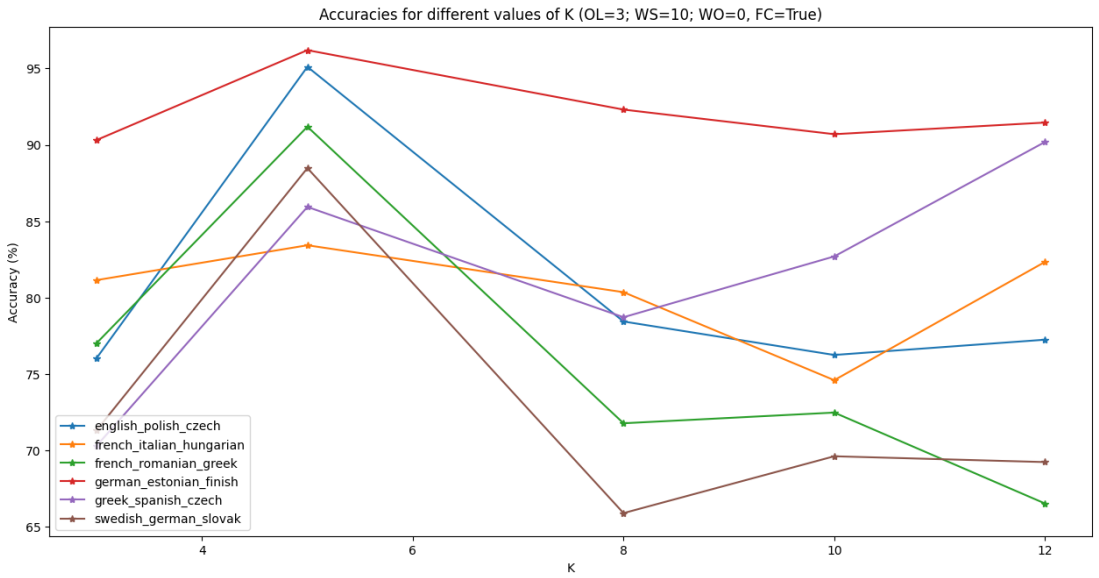

Fig. 12: Accuracy values in function of the CPM K values.

As shown in the figure above, for K=5, we have the best accuracy values for almost all targets. As the value of K increases, the accuracy decreases with occasional increases that may be explained by the different averages of characters p/ word for the different languages. Therefore, these results corroborate our initial hypothesis.

Finite-Context Model Impact on Accuracy

As detailed in the above sections, we have implemented a Finite-Context model in our *Lang* implementation to calculate the probabilities of the non-hit characters. The idea was to check whether this adaptation would improve our Locate Lang system's accuracy.

Hypothesis: Using the Finite-Context model should increase the accuracy as we use context from the reference to calculate non-hit probabilities.

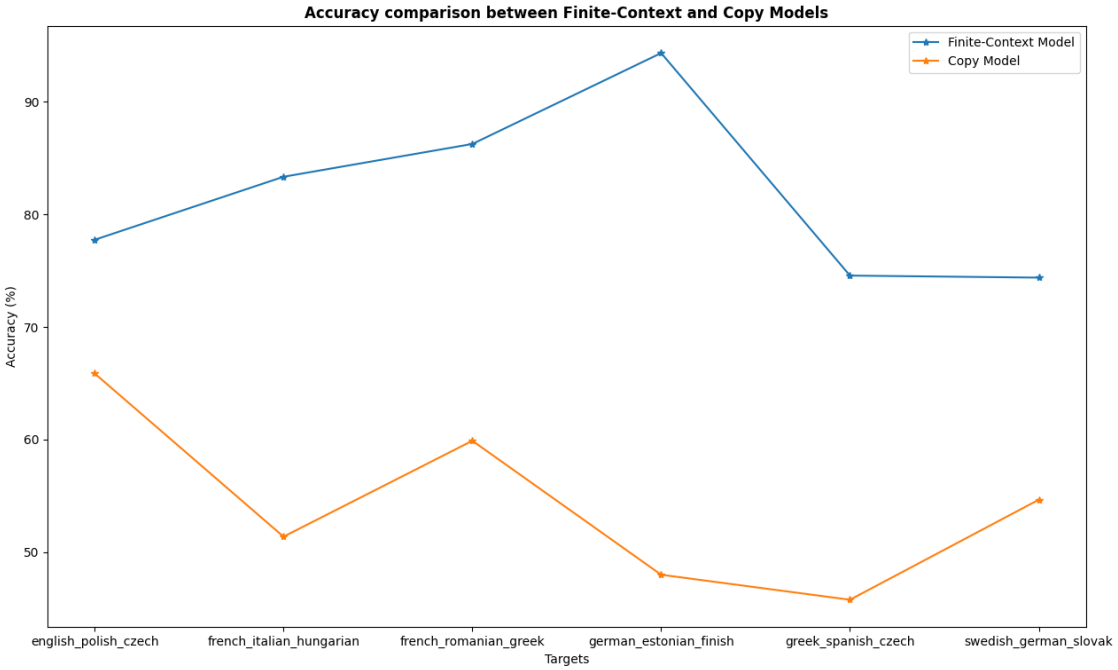

Fig. 13: Accuracy obtained for different targets using the Finite-Context and original Copy Model.

As shown in the figure above, our results corroborate our initial Hypothesis - using the Finite-Context model indeed improves the accuracy of the language location system - we get higher results of accuracy for each target when using the Finite-Context model compared to when we use the Copy model.

Impact of Target Size in Accuracy

All the results presented above were obtained for targets with 1000 characters. It is important to check however if the size of the targets has any influence on the language detection system, i.e. if they influence the accuracy of the system.

From a theoretical point of view, the size of the targets should not influence the accuracy of the system once we are only adding more words to be identified. It is worth noting that when we talk about the size of the target, we are only discussing the size of the target (amount of characters) and not the relative size of the different pieces of languages (consecutive characters of a given language).

Hypothesis: The accuracy should not be influenced by the target's size.

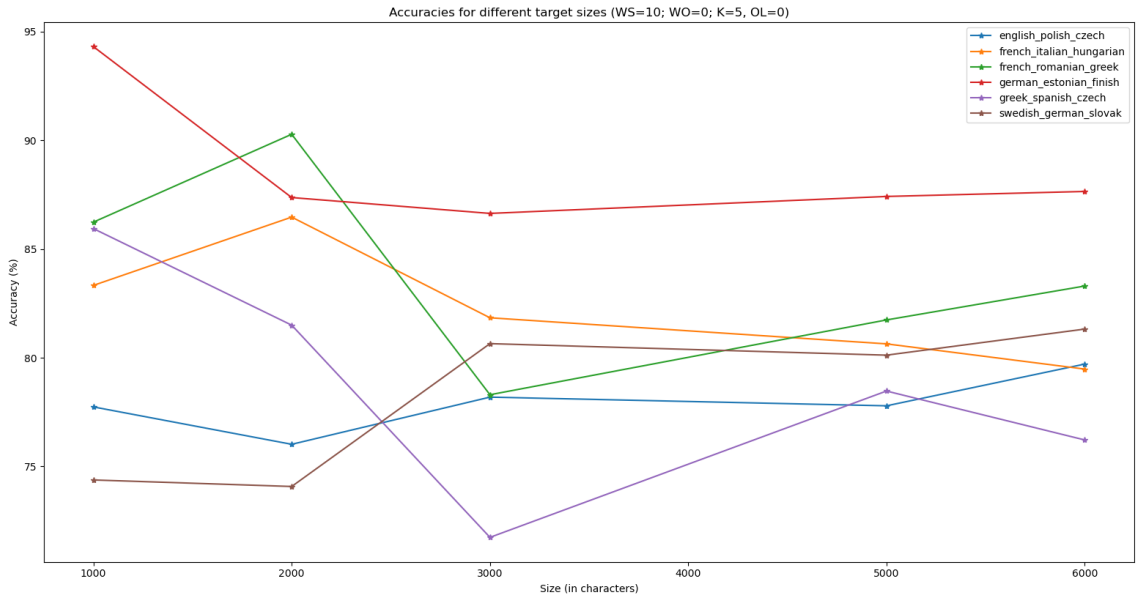

Fig. 14: Accuracy obtained for different targets in function of the target size.

As we can observe in Fig.14, the accuracy of the *locatelang* oscillates for smaller targets (i.e. targets with fewer characters). However, as we increase the target size, we see a reduction in this oscillation. By looking at the results, we are compelled to affirm that if we continued to increase the target size, we would slowly approximate the accuracy of the different targets towards a common point of around 80-90% of accuracy.

Therefore, although we were expecting our accuracy to not be influenced by the size of the target, in practice, we experienced some oscillations in the accuracy for different target sizes. We can conclude that the size of the target can influence the accuracy of smaller targets; we believe that for bigger targets, this effect must be unperceived.

Impact of References Size in Accuracy

As explained in the above sections, the *Lang* program was trained on references from 20 different languages. Each one of these references had an approximate size of 4 MB. It is important to study whether the size of the references can be a determining factor in the accuracy of the *locatelang* program.

From a theoretical standpoint, the size of the references should affect the accuracy of the model. Smaller references contain less information to train the model and can impact directly its efficiency as some words for a language may not be recognized because they were not present in the references, for example.

Hypothesis: The size of the references should impact the accuracy of the *locatelang* program.

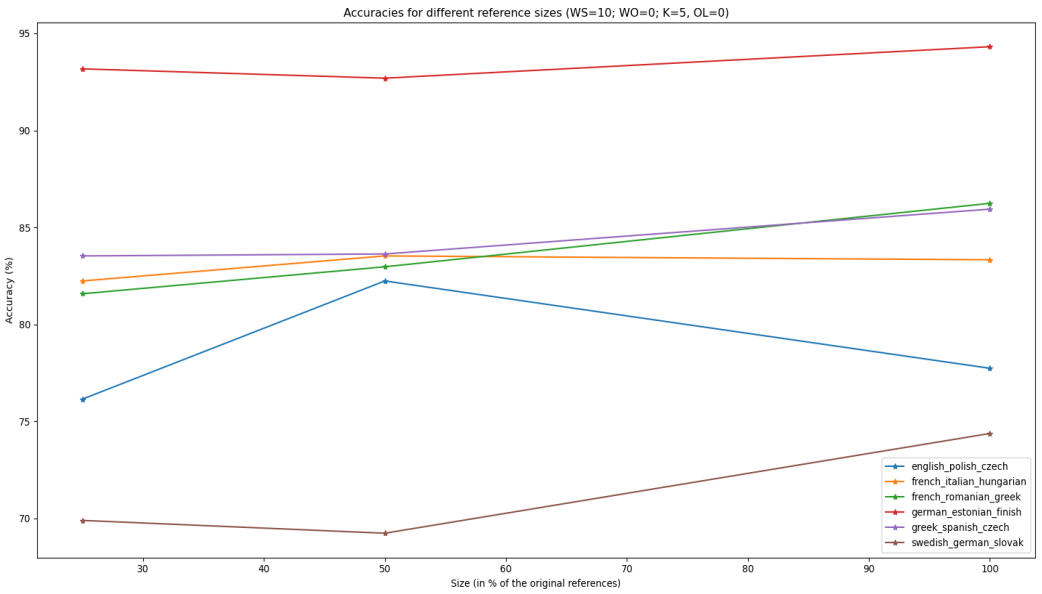

Fig. 15: Accuracy obtained in function of the reference sizes.

As shown in Fig. 15, as we decrease the reference size, the accuracy tends to be slightly reduced. If we continued to reduce the size of the references, the accuracy would suffer a higher reduction up to the point it would be near 0% (when the references were empty).

Therefore, we can confirm that, as expected, reducing the size of the references will reduce the accuracy of the model. However, the accuracy reduction is not as drastic as we initially anticipated. This emphasizes the importance of the quality of data vs the amount of data to train this kind of model, i.e., it is more important to have data with higher quality (high diversity of words and sentences) than a higher amount of data with low quality.

References

Text Similarity Applications
https://spotintelligence.com/2022/12/19/text-similarity-python/

Genetic Relationship (linguistics)
https://en.wikipedia.org/wiki/Genetic_relationship_(linguistics)

Romance Languages
https://en.wikipedia.org/wiki/Romance_languages

Germanic Languages
https://en.wikipedia.org/wiki/Germanic_languages

Finnic Languages
https://en.wikipedia.org/wiki/Finnic_languages

Slavic Languages
https://en.wikipedia.org/wiki/Slavic_languages

Appendix

Program *lang*

```
Reference: ../references/europarl20/polish.txt
Target: ../references/single_language/italian_target.txt
Total Information: 1854.71
Information p/ Symbol: 3.02563
Information p/ Iteration:
3:5.12928
4:5.12928
5:0.5
6:8.11172
7:8.11172
8:0.5
9:2.29863
10:2.29863
…
620:3.95138
621:3.95138
622:3.95138
623:0.5
624:3.12725
625:3.12725
626:5.12928
```

Appendix.1: Output of program *lang*