Identifying Synthetic Text using a Finite Context Model

Bruna Simões *brunams21@ua.pt*
Daniel Ferreira *danielmartinsferreira@ua.pt* Tiago Carvalho *tiagogcarvalho@ua.pt*

***Abstract*—The problem of determining if ChatGPT rewrote a text consists of deciding whether, for a given text, that text was written by ChatGPT or not. Most detectors use feature extraction and selection operations to perform the classification and operate heavily to determine the smallest set of features necessary to solve the problem correctly. Since the representation of the original data by a small set of features, used to train the models, can be seen as a form of *lossy data compression*, it is proven that data compression can be used to address classification problems, which removes the need to use a separate feature extraction and selection stages. The following report implements a solution based on this idea and infers the accuracy, precision and recall of a model based on the information-theoretic approach.**

***Index Terms*—Lossy Data Compression, Large Language Models, Finite-Context Models.**

I. Introduction

ADVANCED Language Models are computational models notable for their ability to achieve general-purpose language generation and other natural language processing tasks such as classification, that is, the ability to determine which set of categories an observation belongs to. However, with the increasing ability and increasingly large datasets used to train these models, comes the problem of identifying if a certain text corresponds to a human-generated or artificially generated one.

Current methodologies implement strategies to extract features and select the smallest set that retains enough discriminant power to tackle the problem. However, since the representation of the original data used to train these large language models can be seen as a form of *lossy data compression*, compression algorithms can measure the similarity between files, solving the classification problem and bypassing those stages.

Therefore, by creating two different models that have a certain reference text, that is, the text to be analyzed, one being a ChatGPT model, and the other, being a human model, the model which requires fewer bits to describe the text to be analyzed consists on the model which is more similar to the reference text, and therefore, can identify the model which the reference text was most likely generated by.

II. Method

The developed methodology consists of a C++ program that analyzes three texts, two to populate the ChatGPT and Human models and another to analyze and infer its classification based on the computed results. The following section provides information about how these models are created, how data is populated and how the classification is made on the text to be analyzed.

A. Finite-Context Models

In order to represent data dependencies and therefore identify patterns, the Markov Models were used. In lossless data compression, the discrete-time Markov chain - also called finite-context model - is used as a specific type.

This model can collect data from a certain information source. A probability estimate to the symbols of the alphabet is assigned according to the fixed context computed over a finite number of past occurrences, that is, based on a certain fixed context, there is a certain amount of times that the symbol to be compressed has appeared with the same context, and therefore, that compression will depend on the number of past outcomes. If a context has appeared with relatively higher frequency in a certain model, then the compression of the symbols that consist of that context will be low.

Therefore, the two models, which are based on the past of the sequence of outcomes, are assigned significantly different probability estimates, since the contexts of the file to be analyzed will need less compression for the model it is more similar to.

Considering that $x_1^n = x_1x_2...x_n, x_i \in \Sigma$ corresponds to the sequence of outputs (symbols from the source alphabet Σ), then the *k*-order Markov model verifies that

$$P(x_n|x_{n-1}...x_{n-k}) = P(x_n|x_{n-1}...x_{n-k}...)$$

where the context $c = x_{n-1}...x_{n-k}$ corresponds to the described sub-sequence.

Since a letter in a word is generally heavily influenced by the preceding letters, that is, the context, Markov models become more useful than other compression models, since they are associated with the context preceding a certain symbol of the alphabet.

B. Count Collection

Each model is represented by a reference text r_i, in which the program performs counts on the apparition of each symbol within a certain context for that reference text. For example, considering a certain context c, it is counted the number of apparitions of each symbol of the alphabet following that context. If a symbol is not present following a certain context, then the count number for that context is zero. However, a symbol not following after a certain context does not imply that the same symbol never appears in the reference text.

Therefore, for the two classes, two different reference texts are provided as inputs: one is a "good description" of a ChatGPT and another of a human/non-AI model.

As mentioned above, a model which consists of a "good description" of a reference text consists of a model that requires few bits to compress a certain text, given the fact that text is similar and has the same patterns as the model it is described by.

In order to have reliable models which are representative of their classes, it is provided as input a series of texts rewritten by ChatGPT, and given to the corresponding ChatGPT model and a series of texts written originally by humans to the corresponding human model. For each model, appearances of the alphabet symbols are counted for each found context. This can be expressed as a table of contexts and their respective counts, where the lines correspond to all combinations of contexts with a fixed size k and columns correspond to all the alphabet symbols. For example, for an alphabet $\Sigma = \{0, 1\}$ and $k = 2$, all possible context combinations C would be given by $C = \{00, 01, 10, 11\}$. However, for a text $r_i = 0011$, the context $c = 10$ is never present, and therefore it is inferred that all counts for that context are zero.

The program stores the count information as an *hashmap* where the keys correspond to a k-sized string and the values correspond to another *hashmap*. For this second *hashmap*, the keys correspond to the alphabet letters, that is, a character, and the values correspond to the number of times that character has appeared in the context defined by the *string*, that is, its count.

The steps to populate each of the model tables are as follows:

- **Alphabet processing:** The program iterates through each character in the inputted texts and stores it in an *array* of characters. As mentioned above, the program also accepts filters as input, and therefore, for each symbol, if a symbol is a character that corresponds to one of the filtering categories, or if the symbol is neither an alphanumeric, punctuation or a space symbol, then it is ignored, and the program does not add it to the alphabet list of symbols.
- **Symbol processing:** Similarly in alphabet processing, the text is filtered in such a way that certain symbols are ignored when they meet the same criteria described above. If the symbol is not ignored and passes through the filtering process, it is then verified if the last registered k symbols, which correspond to the current context, have the same size as the inputted k value. This verification is made to ensure that the first k symbols do not appear in the table, since the context size is insufficient to be inserted in the table. For the remaining characters in the text that already have a k-sized context, if the context is found in the table, then the counter for the current symbol given that context is incremented by one.
  Additionally, a parameter *sizeLimit* is taken into consideration due to the program's natural restrictions in the number of bits an integer can store. If the counter has surpassed that parameter, then all the counters of the table are divided by half. If the context is not found, however, then it is initialized in the table, where all the symbols in the alphabet are initialized with a value zero for that context, and the char that corresponds to the symbol being analyzed is initialized to one, symbolizing the first found character for the given context.
- **Updating context:** After updating the table, the context needs to be updated in order to accommodate the next symbol to be processed. Therefore, if the context has size k, the least recent symbol will be removed from the string, and the current analyzed symbol will be added.

C. Compression Methodology

After proceeding with counting all the instances for each found context in the provided texts, the program analyzes the text provided for classification character by character. For each character found, the context window is updated, so as to store the last k symbols before the character being encoding.

For each character in the text to be analyzed, the program will calculate the cost of encoding that character considering each of the models. Since each model has different texts as a reference, their *hashmaps* are naturally also different, and therefore, the cost of encoding a character is different for each model. Therefore, for each character being analyzed, the cost of encoding that char is calculated for each model and added to the total number of bits for each model.

The steps to infer the encoding for each character are as follows:

- **Symbol filtering:** Similarly to the alphabet and symbol processing in the counting phase, the symbols to be encoded are equally filtered, and the symbols that match the filtering criteria are ignored.
- **Symbol processing:** In the symbol processing stage, if the current context doesn't yet have size k, then the number of total symbols is incremented by one and the context is incremented with the symbol being processed, and no compression is made. However, if the context has size k, then the symbol will be further processed. For each model, the symbol is processed by finding the same context in the corresponding *hashmap*, and it case that context is found, then the probability for that symbol with that context will be inferred based on the relative frequency principle:

$$P(e|c) \approx \frac{N(e|c)}{\sum_{s \in \Sigma} N(s|c)}$$

  Where e is the current symbol being analyzed, c is the k-sized context and s corresponds to all the symbols in the alphabet, acquired in the *Count Collection* phase. However, since this probability would be equal to zero in cases where no context is found or even when there are symbols counted for the current symbol and current context, a smoothing parameter was applied to estimate these probabilities, given by

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

where α is the smoothing parameter.
The average number of bits necessary to encode the current symbol is characterized by this probability and is calculated based on it.
The model which has the total least number of bits corresponds to the model that is more similar to the text being analyzed and therefore, the analyzed text is classified based on it.

D. Parameter Tuning

A fundamental aspect of our implementation is the capability to fine-tune parameters to adapt the compression algorithm to various data characteristics and optimize performance and classification reliability. These parameters include:

- **Size limit** (s): The size limit, as mentioned above, determines the maximum allowed number count that a *hashmap* can have. When counting the symbols and incrementing them in the respecting table, when encountering a counter that is the same number as the size limit, then all the elements of that hash table become halved. This parameter is useful in situations where the reference texts are too extensive and surpass the program's native data number storing capacity.
- **Filters** (f): The filters, as also mentioned above, filter both the texts to be analyzed and the reference texts to decrease some computational effort for symbols that are not relevant to provide a good description for a model. These filters include:
  - `n`: Filter all the characters that are numbers.
  - `s`: Filter all the characters that are symbols.
  - `e`: Filter all the characters that are spaces.
  - `c`: All letters are converted to their uppercase, and therefore, all texts become case-insensitive.

  Several filters can be applied at once, for example, indicating the filter as "`ns`" means all numbers and symbols will be filtered, but spaces will not, and the letters maintain their case sensitiveness.
- **Smoothing Parameter** (α): Aids in smoothing the probability distributions, ensuring that symbols not yet observed are still given a chance of occurrence, thus preventing zero-probability issues.
- **Context size** (k): Dictates the size of the sequence of symbols considered as a context, influencing the model's sensitivity to patterns within the data.
- **File/directory to be analyzed location** (t): Location of the file(s) to be analyzed and classified based on the number of bits for each model. Can either be a directory or a file. If it corresponds to a directory, then all the files in that directory will be analyzed.
- **File/directory as a ChatGPT reference location** (g): Location of the file(s) to be used as a reference for the ChatGPT model. Can either be a directory or a file. If it corresponds to a directory, then all the files in that directory will be analyzed.
- **File/directory as a Human reference location** (h): Location of the file(s) to be used as a reference for the human/non-AI model. Can either be a directory or a file. If it corresponds to a directory, then all the files in that directory will be analyzed.
- **True label of the file/directory to be analyzed** (b): Restrictively GPT/Human, referees to the original input source, i.e. to the text to be analyzed.
- **Path to the results file** (o): Specify the name of the file where the results should be displayed. If this parameter is not passed as an argument, the results will be displayed solely on the user's terminal. Each line in the file indicates respectively the k, $alpha$, size limit, filters used, the true label of the file/directory to be analyzed, if the resulting model is more similar to ChatGPT or not, the reason between the number of bits using the ChatGPT model and the number of bits using the human model and the time, in seconds, it took to analyze that file.

III. Results

For the construction of the human and synthetic models, the program was fed, paragraph-by-paragraph, the content of Moby-Dick by Herman Melville[1] and partially of the Complete Works of Shakespeare by William Shakespeare[2] as well as their corresponding rewritten version. The synthetic versions were produced using OpenAI's[3] gpt-4-turbo and gpt-3.5-turbo models.

To validate the proposed solution, a series of tests were conducted with numerous sentences constructed from random quotes provided by the Goodreads API.[4]

Of the program's possible input arguments, each was used as a test variable to isolate its effects on performance metrics and runtime.

IV. Discussion

The setup featuring $k = 5$, $\alpha = 0.1$, sizelimit $= 5000$, and filter $=$ nsc secures the best accuracy and recall rates as seen in I. This suggests that a moderately intricate model with strict filtering criteria is more efficient at distinguishing between types of text. The significant performance of this configuration can be partly attributed to the exponential increase in hashtable entries as k increases and more complex filters are applied. With a larger k, the model incorporates a broader context, resulting in a richer, albeit more complex, hashtable. Different filters, especially those that are less restrictive in character filtration, contribute to this complexity by increasing the number of potential entries as seen in Figures 3 and 4. This is further exacerbated in scenarios where case sensitivity is considered, effectively doubling the possible entries in the hashtable due to distinct upper and lower case representations.

On the other hand, a configuration aimed at swift processing, such as $k = 3$, $\alpha = 0.5$, sizelimit $= 5000$, and filter $=$ nsce, strikes an optimal balance between performance and speed, making it suitable for real-time applications. The model's responsiveness to variations in k and α is pronounced,

[1] https://www.gutenberg.org/ebooks/2701
[2] https://www.gutenberg.org/ebooks/100
[3] https://platform.openai.com/docs/overview
[4] https://www.goodreads.com

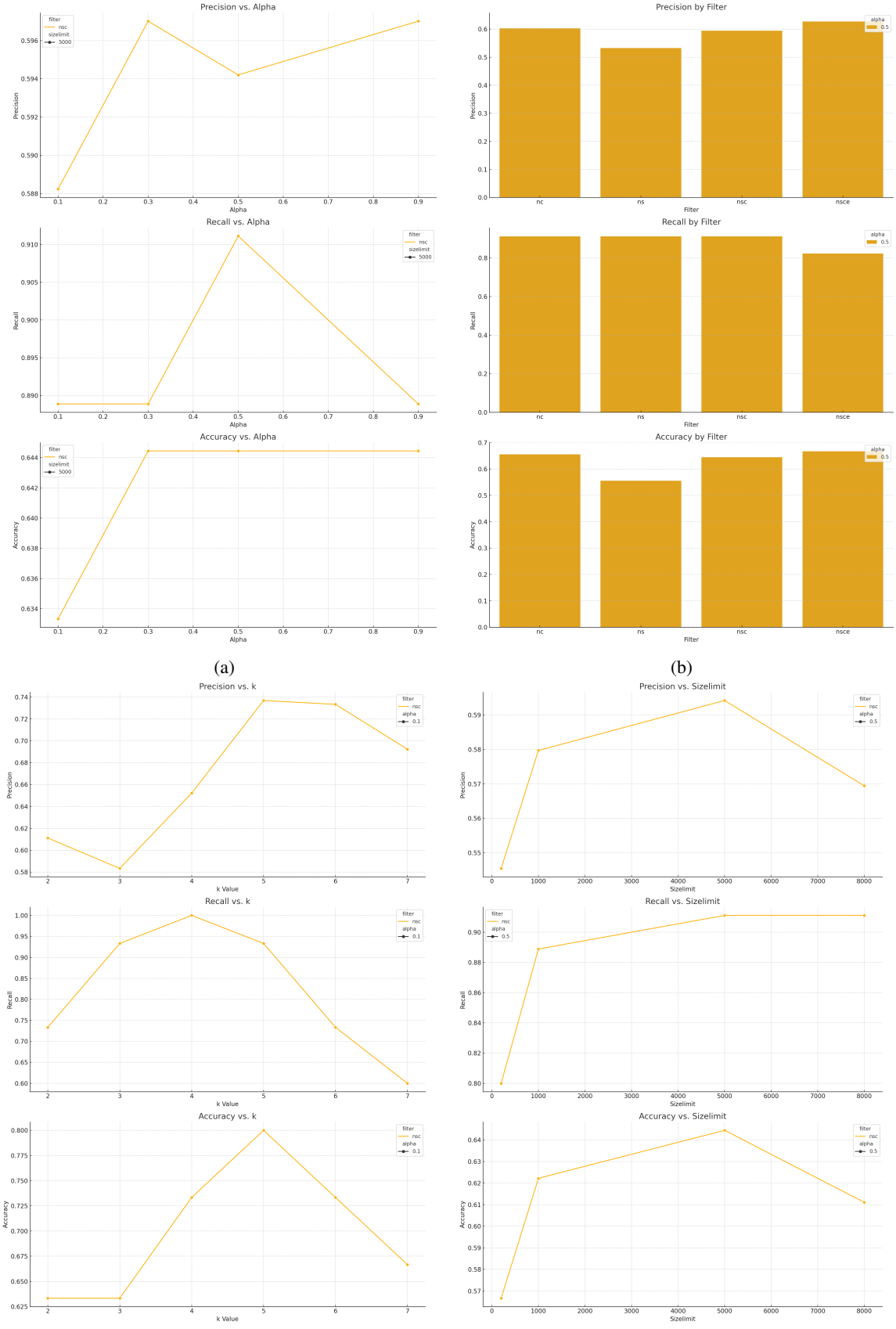

Fig. 1: Collection of precision, recall and accuracy values adjusting to different values of input alpha, filter, k and size limit.

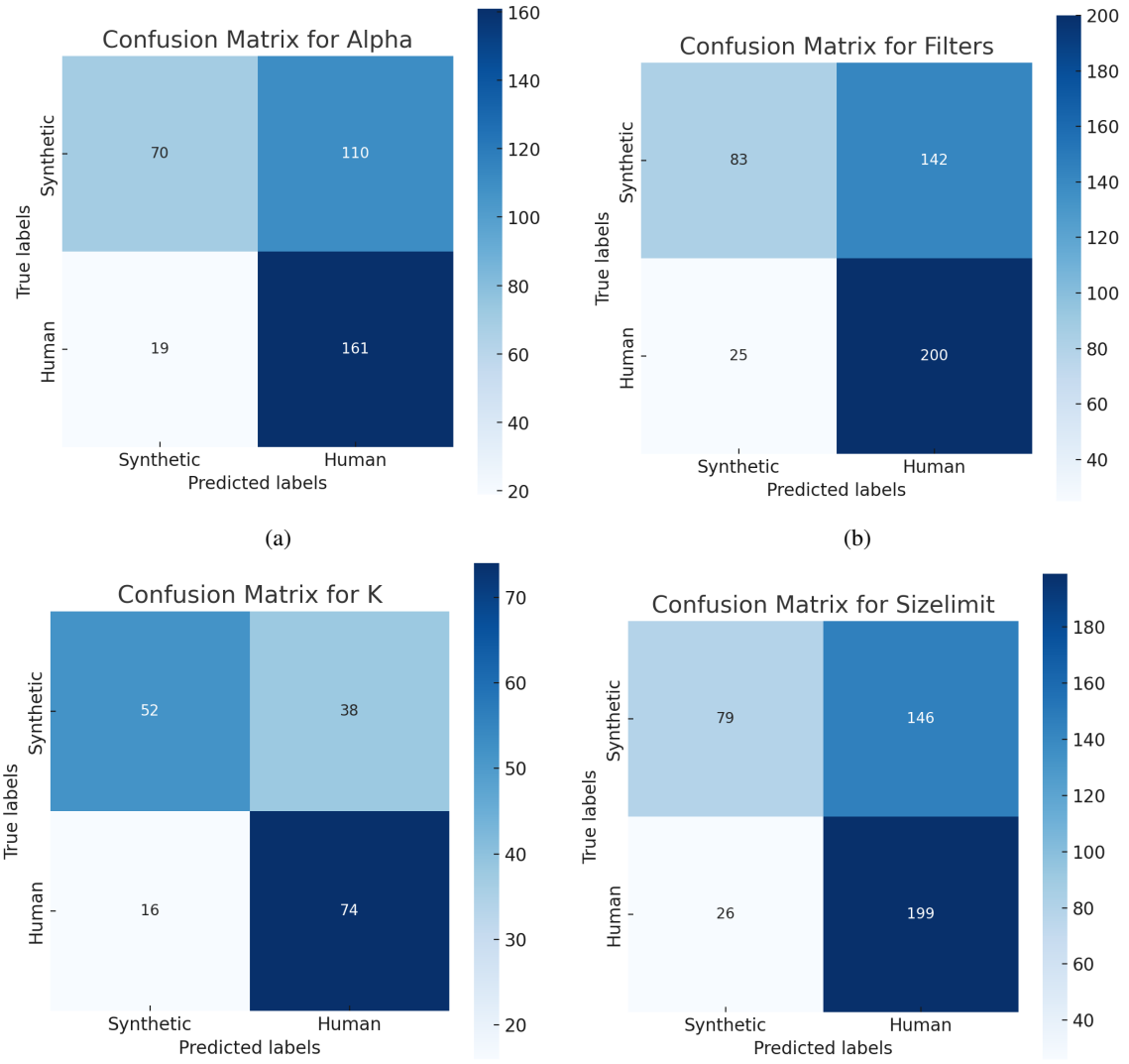

Fig. 2: Confusion matrix for each independent various input arguments.

with higher k values boosting recall but requiring more computational resources due to the increased number of hashtable entries. Generally, there exists a distinct compromise between performance indicators and processing time, implying that lower k values might be preferable in scenarios demanding quick responses, even at the cost of slight reductions in accuracy.

V. Conclusion

We delineate an approach to text classification by harnessing data compression strategies, specifically through the application of finite-context models. Results show satisfactory metrics when time constraints are not an issue. This methodology particularly excels in scenarios where detailed contextual analysis is paramount for distinguishing between human-generated and synthetic texts. The parameter tuning capability provides flexibility, allowing the model to be adapted based on the specific requirements of the task at hand. Future work could explore the integration of additional contextual features or the application of hybrid models that combine the strengths of various data compression techniques to enhance classification accuracy further. The proposed approach not only opens new avenues in text analysis but also contributes to the broader field of data compression by demonstrating its utility in practical applications beyond traditional file compression.

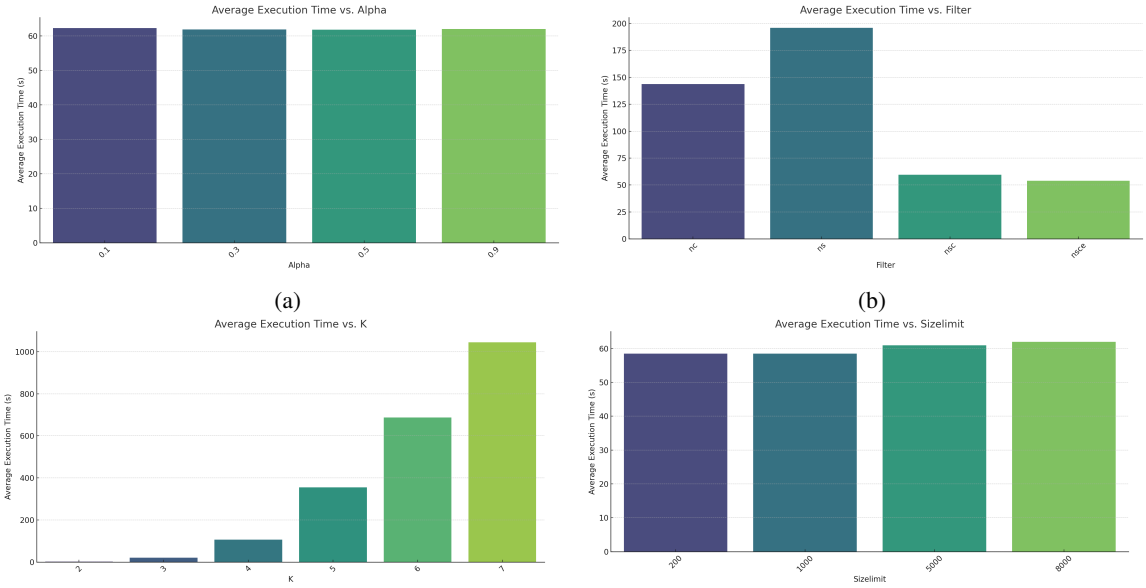

Fig. 3: Average execution in function of multiple input values alpha, filter, k and size limit.

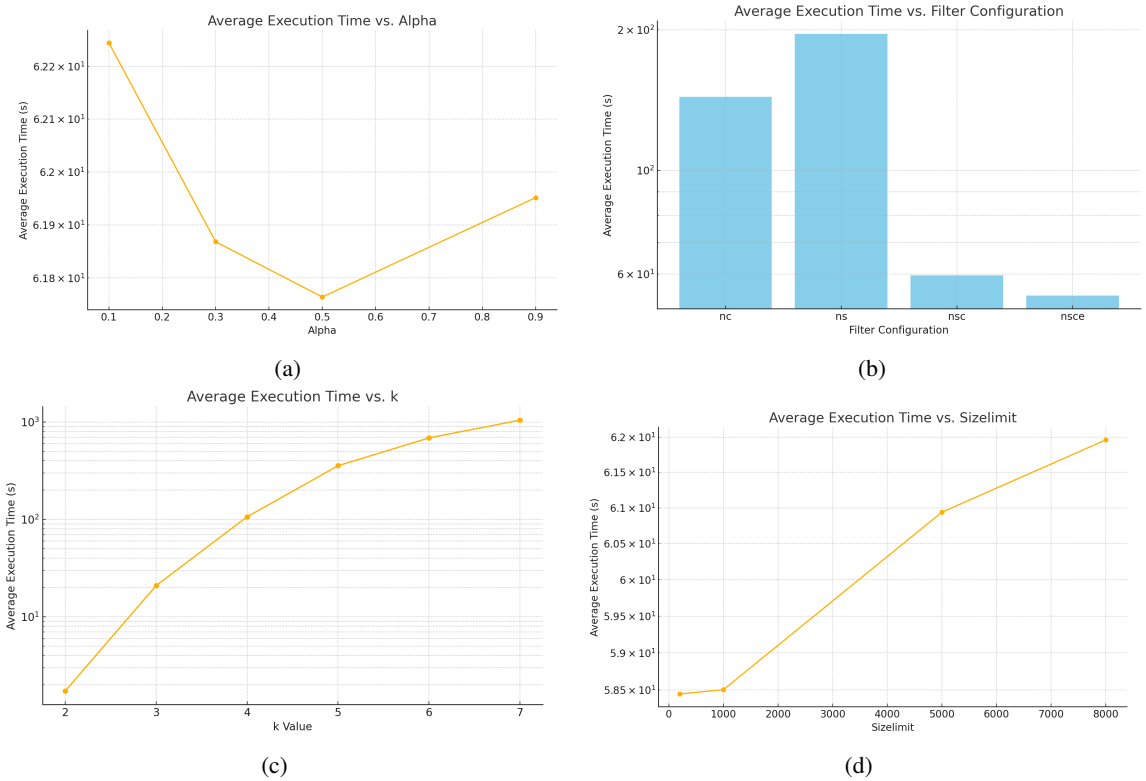

Fig. 4: Average execution (in logarithmic scale) in function of multiple input values alpha, filter, k and size limit.

TABLE I: Optimal Parameter Combinations Based on Top-Performing Metrics

k	alpha	sizelimit	filter	Precision	Recall	Accuracy	Avg Time (s)	Top Metric
5	0.1	5000	nsc	0.737	0.933	0.800	355.66	Accuracy, Recall
4	0.1	5000	nsc	0.652	1.000	0.733	106.31	Recall
3	0.5	5000	nsce	0.627	0.822	0.667	53.94	Avg Time