

# Finite Context Models for the Detection of Human and ChatGPT written text

Raquel Paradinha 102491      Paulo Pinto 103234      Miguel Matos 103341  
Filipe Antão 103470

**Teoria Algoritmica da Informação**  
**Departamento de Electrónica, Telecomunicações e Informática**  
**Universidade de Aveiro**

May 8, 2024

## Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Finite Context Models . . . . .	3
<b>2</b>	<b>Dataset</b>	<b>4</b>
<b>3</b>	<b>Implementation</b>	<b>4</b>
3.1	<i>main</i> . . . . .	4
3.2	<i>FileReader</i> . . . . .	4
3.2.1	<i>readTextFromFile</i> . . . . .	4
3.2.2	<i>readTextsFromDirectory</i> . . . . .	5
3.3	<i>MarkovModel</i> . . . . .	5
3.3.1	Constructor . . . . .	5
3.3.2	<i>trainModel</i> . . . . .	5
3.3.3	<i>compressFileSize</i> . . . . .	5
3.3.4	<i>calculateCompressionStats</i> . . . . .	6
3.3.5	<i>classify</i> . . . . .	6
3.3.6	Accessors . . . . .	6
3.4	<i>gui</i> . . . . .	6
<b>4</b>	<b>Results</b>	<b>8</b>
<b>5</b>	<b>Discussion</b>	<b>9</b>

## **Abstract**

In this report, we explain our methods for implementing a finite context model and using it to model two different datasets, consisting of text not written by ChatGPT and text written by ChatGPT, to create a program able to distinguish one from the other. The final solution is tested, and results are presented and discussed.

# 1 Introduction

In a time when artificial intelligence has occupied almost every aspect of human life, its ability to generate text practically indistinct from the ones authored by humans brings significant benefits and more prominent concerns. With the advance of large language models, like ChatGPT, the difference between human and machine-written content tends to be thinner, bringing ethical and sometimes legal issues and an easier spread of misinformation.

This way, it becomes crucial to have mechanisms that help identify the difference between the artificial and human texts.

In this work, we introduce an implementation of a system that aims to minimize this preoccupation. Using the principle that the similarity between files can be identified through compression algorithms, we developed a program that allows the user to test a file of its choice, with the support of a graphical user interface, to make the experience more user-friendly [3]. Program filters can also be personalized to fulfill each user's preferences better.

## 1.1 Finite Context Models

Finite Context Models use a finite sequence of previous data points to predict future events, effectively capturing the dependencies within a limited context. This method is an extension of the principles underlying Markov chains — models where the probability of transitioning to a future state depends solely on the current state and not the sequence that preceded it. These models are adept at handling sequential data, making them particularly useful for text analysis.

This predictive capability can be adapted to classify data by learning the patterns or sequences typical to specific datasets. This adaptability extends their utility beyond text analysis to applications like genomic sequence classification or speech recognition. By analyzing sequences and determining the likelihood of occurrence of various events, these models help make informed decisions based on the statistical significance of the observed patterns.

For this assignment, we will consider two types of data - text *not* written by ChatGPT and text written by ChatGPT. In essence, I will use one of these models over both datasets to infer two probability tables and then use them to check which table the given text fits better.

## 2 Dataset

The dataset the model uses to construct its table of probabilities is of utmost importance to the classification task since data that does not accurately represent the different classes will influence the model in the wrong direction. Thus, the provided data must contain relevant and diverse text within classes. Too little data and low diversity will make the program unable to generalize well across different text types.

To collect our data, we tried the following approaches:

1. Manually prompted ChatGPT to rewrite text excerpts gathered by us
2. Made a script that repeatedly sent requests to OpenAI's API to convert non-get text to text rewritten by ChatGPT
3. Searched for publicly available datasets

The first two approaches were initially pursued since we wanted to build our tailored dataset for the problem. However, manually prompting ChatGPT's user interface proved too slow and tedious to gather meaningfully sized data, so we resorted to creating a simple Python script to convert text files by repeatedly asking the API to rewrite the text chunk by chunk. However, OpenAI's API is not freely accessible, so building a large dataset using this method would have its cost, which disincentives its use.

Thus, we searched for publicly available datasets containing the data we wanted, precisely two sequences corresponding to the original non-gpt text and the rewritten version. We found two suitable sources of text and compiled entries for both to construct a dataset we deemed adequate. We did not consider the entirety of the datasets since they were too large and would lead to exaggerated compute times. All text was written in English; the first [1] dataset contained the first paragraph of Wikipedia entries and their respective rewriting using ChatGPT, and the second [2] consisted of a massive compilation of Human vs AI written text.

## 3 Implementation

### 3.1 *main*

This file contains the main function of our Markov model program. It receives five arguments:

- *path\_to\_not\_rewritten\_texts\_folder*: Path to the folder containing non-rewritten texts
- *path\_to\_rewritten\_texts\_folder*: Path to the folder containing rewritten texts
- *path\_to\_target\_text\_file*: Path to the target text file that needs to be classified
- *alpha*: Smoothing parameter for the Markov model
- *context\_size*: Context size (order) of the Markov model.

The primary objective of this file is to classify a target text file using the Markov model trained using two sets of reference texts: non-rewritten (original) texts and rewritten texts. Based on the results obtained from the Markov model compression, the classification model aims to determine whether the target text is closer to the original or rewritten texts.

### 3.2 *FileReader*

The file *FileReader* defines and implements two functions for reading text from files. These functions are encapsulated within a *FileReader* class, which provides methods to read text data from a single or multiple files within a directory. Additionally, a utility function, *toLowerCase*, is used for standardizing text by converting it to lowercase.

#### 3.2.1 *readTextFromFile*

This method accepts a file path as input, reads the file's contents in binary mode into a string stream, and converts the resulting string to lowercase using the *toLowerCase* function. The method returns the converted string. If the file cannot be opened, it logs an error message and returns an empty string.

### 3.2.2 *readTextsFromDirectory*

This method accepts a directory path as input and iterates through all regular files in the directory using `std::filesystem::directory_iterator`. For each file, it reads the file's content into a string stream and converts it to lowercase using the `toLowerCase` function. The converted text data from all files is stored in a vector, which is then returned.

## 3.3 *MarkovModel*

The *MarkovModel* class is utilized as a statistical tool for predicting the next character in a sequence based on a given context of previously observed characters. It serves a dual purpose in text compression analysis and classifying texts as human-written or generated by AI, such as ChatGPT.

The *MarkovModel* class is composed of two member variables:

- **ALPHABET:** Lists all unique characters encountered during the model's training. This list is crucial for defining the possible outputs the model can predict.
- **ALPHABET\_SIZE:** Represents the number of unique characters in the *ALPHABET* and is used to manage the complexity and calculations within the model.

### 3.3.1 *Constructor*

Initializes an instance of the *MarkovModel* class with specified parameters where *contextSize* defines the historical context used for predictions and *alpha* is a smoothing parameter that aids in managing rare or unseen contexts in training data.

### 3.3.2 *trainModel*

The *trainModel* function is integral to constructing a Markov model tailored to the patterns within a corpus of text data. Its primary purpose is to learn the statistical relationships between sequences of characters, considering a fixed-size context window. This process begins by traversing through the provided texts, extracting sequential segments of characters to build a representation of the language's structure.

During this traversal, the function tallies the occurrences of characters following specific contexts, effectively constructing a mapping of context to character probabilities. A smoothing method ensures robustness and accounts for unseen combinations, adding a small constant *alpha* to each count. This step prevents zero probabilities and maintains uncertainty for unseen contexts. Additionally, it constructs the alphabet of characters encountered, facilitating subsequent text generation or prediction tasks.

Following the data collection phase, the function refines the learned probabilities, normalizing them to ensure they adhere to a valid probability distribution. Moreover, it handles situations where specific contexts are absent by initializing their probabilities uniformly across the alphabet. This meticulous learning process encapsulates the nuances of the language within the Markov model, empowering it to generate coherent text or make informed predictions based on the observed character sequences.

### 3.3.3 *compressFileSize*

The *compressFileSize* function estimates the potential reduction in file size achievable through compression using the Markov model's learned probabilities. It traverses through the provided text, segmenting it into context-sized chunks. Each chunk represents a context window, capturing the sequence of characters the model analyzes to predict the subsequent character.

During this process, the function examines whether the Markov model contains information about the current context and the following character. If such information is available, it retrieves the associated probability. This probability is critical in determining the number of bits required to represent the character in a compressed file. The function utilizes the  $-\log_2$  function to convert the probability into its binary representation, contributing to the overall estimation of compressed file size.

In cases where the model lacks information about the current context or the subsequent character, the function resorts to assuming an equal distribution of probabilities across the alphabet. This assumption ensures a fair estimation of the compressed file size, even in scenarios where specific

context-character pairs are unseen during training. By summing the computed bits across all context windows in the text, the function provides an insightful estimate of the potential compression achieved by leveraging the Markov model's learned probabilities.

#### 3.3.4 *calculateCompressionStats*

The *calculateCompressionStats* method tries to access the compression efficiency achieved by the Markov model when applied to a given text. It encapsulates this evaluation by computing two essential metrics: the compression ratio and the percentage.

To initiate this assessment, the method first computes the text's original size in bits, considering each character occupies 8 bits. This initial size is the baseline for comparison against the compressed size, which is determined by invoking the *compressFileSize* method on the provided text.

Upon obtaining the compressed size, the method calculates the compression ratio, which is defined as the ratio of the original size to the compressed size. This metric offers insight into how many times smaller the compressed file is compared to the original text and indicates the degree of compression achieved.

Additionally, the method evaluates the compression percentage, representing the proportion of space saved through compression relative to the original size. It quantifies the compression efficiency regarding a percentage reduction in file size, providing a more intuitive understanding of the compression outcome.

By returning a pair containing both the compression ratio and the compression percentage, the method furnishes a comprehensive summary of the Markov model's compression performance on the given text, aiding in assessing its effectiveness in reducing file size.

#### 3.3.5 *classify*

This method is crucial in discerning whether a given text is more characteristic of human-generated or ChatGPT-generated content. It accomplishes this task by leveraging Markov models trained on two distinct datasets: human-generated texts (*rhTexts*) and ChatGPT-generated texts (*rcTexts*). Only this function is called on the main file; it calls the others and returns the result, which is AI- or human-generated.

At the onset, the method initializes two separate Markov models, one for each dataset, with identical context sizes and smoothing parameters. It proceeds to train these models on their respective datasets, capturing the statistical patterns and dependencies present in the texts.

Following model training, the method evaluates the compression statistics of the provided text using both Markov models. By invoking the *calculateCompressionStats* method for each model, the compression ratio and percentage are obtained, which serve as indicators of the compression efficiency achieved when encoding the text with each model.

Subsequently, the method compares the compression ratios obtained from the two models. Suppose the compression ratio derived from the model trained on human texts (*rhModel*) surpasses that from the model trained on ChatGPT texts (*rcModel*); it concludes that the provided text is more likely to be of human origin. Otherwise, it attributes the text to ChatGPT-generated content.

By encapsulating this classification logic, the classification method provides a streamlined approach to differentiating between human-generated and ChatGPT-generated texts based on their compression characteristics, offering valuable insights into the nature of the provided text.

#### 3.3.6 Accessors

The *getAlphabet* method returns the alphabet learned from the training data. This alphabet comprises all unique characters encountered during the training process. By returning this alphabet as a string, the method provides insight into the diversity of characters present in the text corpus.

Similarly, the *getAlphabetSize* method retrieves the size of the alphabet learned by the Markov model. This size represents the number of distinct characters in the training data. By returning this value, the method offers a quantitative measure of the richness of character diversity within the text corpus.

### 3.4 *gui*

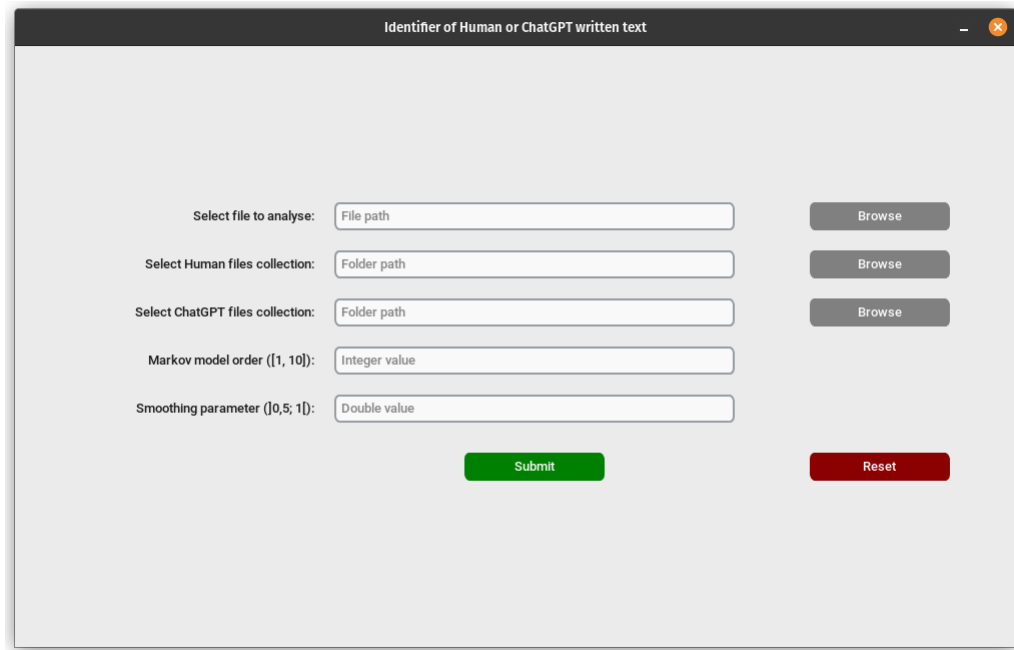
To make our program more appealing to users, we developed a graphical user interface (GUI) using Python's tkinter and customtkinter libraries.

With this feature, the user can easily feed the main program with his preferred values, having at his availability the following input fields:

- **File to Analyze:** File path of the text to analyze;
- **Human Files Collection:** Directory path containing a collection of human-written text files for training the classifier;
- **ChatGPT Files Collection:** directory path containing a collection of ChatGPT-generated text files for training the classifier;
- **Markov Model Order:** An entry field to specify the order of the Markov model. Valid values are in the range  $[1, 10]$ ;
- **Smoothing Parameter:** An entry field to specify the smoothing parameter used in the model, where valid values lie in the range  $]0.5; 1[$ .

Besides this, the three first fields have a *Browse* button that allows users to select files or directories via a graphical file dialog. In addition, the GUI has *Submit* and *Reset* buttons, as seen in Figure 1, to pass the input data to the main program and to clean all the input data, respectively.

When the user parameters are submitted, the classifier program is executed, and the result is displayed at the top of the GUI.



The image shows a graphical user interface titled "Identifier of Human or ChatGPT written text". It features five input fields on the left, each with a corresponding "Browse" button on the right. The input fields are labeled: "Select file to analyse:" (File path), "Select Human files collection:" (Folder path), "Select ChatGPT files collection:" (Folder path), "Markov model order ([1, 10]):" (Integer value), and "Smoothing parameter (]0.5, 1[):" (Double value). At the bottom center is a green "Submit" button, and at the bottom right is a red "Reset" button.

Figure 1: Graphical User Interface.

## 4 Results

Tests were conducted to ensure accuracy and robustness in both our finite context models. A test dataset was added to our project, including texts written by humans and from ChatGPT. A bash script named *tests.sh* was created to conduct the tests. This script iterates through the files within our test dataset and runs the program to check the text in the file against our models and conclude whether ChatGPT wrote that text.

The figures below show the accuracy of the outcome solution against the gathered test dataset. Overall, the model performed much better at distinguishing human text than GPT written text, reliably doing so over the tested context sizes. However, for GPT written text, context lengths of around 5 and 6 characters start to hinder the model's performance too much.

On the other hand, varying the alpha value didn't seem to affect the model's performance too much, but a high alpha value did show an averse effect on classification of GPT written text at moderate context sizes.

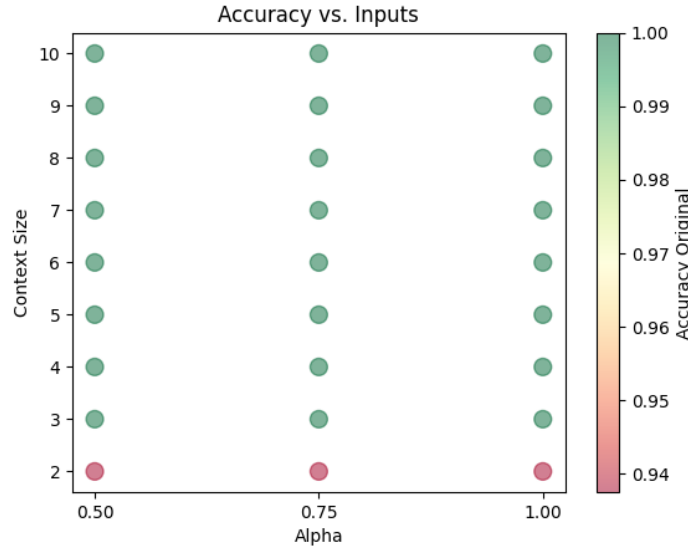


Figure 2: Accuracy against non-GPT written dataset texts

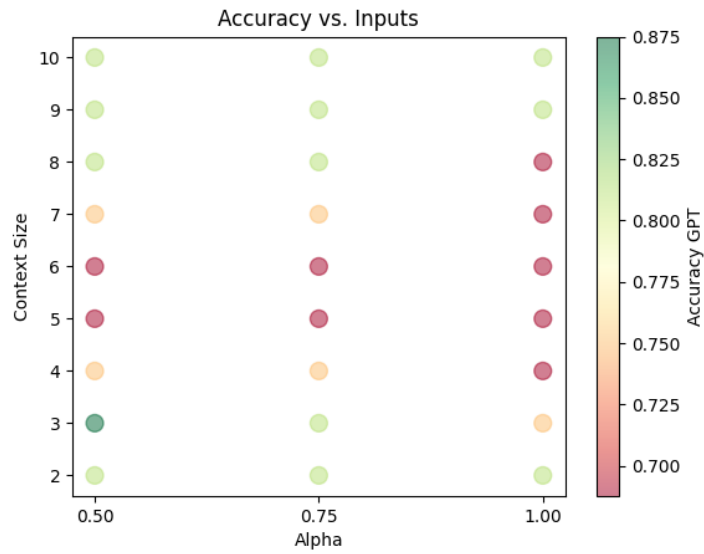


Figure 3: Accuracy against GPT written dataset texts



## 5 Discussion

Regarding the results obtained for the developed program, we can affirm that our implementation meets the expectations, reaching excellent results, especially in identifying text written by humans.

Furthermore, the existence of a GUI that allows the user to interact with our system quickly adds value and genuine utility to the whole process.

Nonetheless, during the development process, we had some ups and downs, mainly with constructing the dataset and trying different approaches to get a more representative dataset for the problem since poor base data would greatly hinder the task at hand.

## References

- [1] aadityaubhat. *GPT-wiki-intro*. URL: <https://huggingface.co/datasets/aadityaubhat/GPT-wiki-intro> (visited on 03/21/2024).
- [2] Shayan Gerami. *AI Vs Human Text*. URL: <https://www.kaggle.com/datasets/shanegerami/ai-vs-human-text/data> (visited on 03/22/2024).
- [3] Armando J. Pinho. *Some notes for the course ALGORITHMIC INFORMATION THEORY*. Accessed: 2024-03-16. 2023.