AI Generated Text Detector

Lab work № 2

Authors

Inês Baptista (98384)

João Correia (104360)

Daniel Ferreira (102885)

2023/2024

Algorithmic Information Theory

Universidade de Aveiro

Aveiro

1 Indice

Contents

2 Introduction

This project aims to develop a program named `was_chatted` that can predict whether a given text file or string was authored by AI or a human. The approach entails creating a model based on natural language processing (NLP) techniques and information-theoretic principles to make these predictions.

AI (`rc`) and humans (`rh`) exhibit distinct writing styles. Humans often produce more diverse and unique texts, sometimes with grammatical errors, while AI demonstrates grammatical precision and tends to generate sentences with consistent structure and vocabulary. The information content, or **entropy**, of these two types of text will differ.

To predict the origin of a target text (`t`), two AI/NLP models are constructed: one for each class (`rh` and `rc`). These models are trained on text samples from both sources, enabling them to learn the distinct writing styles and patterns of each class. To aid in the learning process, the models utilize **finite-context models**, a form of discrete-time **Markov chains**. These models provide estimates of probabilities for symbols within an alphabet, conditioned on a fixed, finite sequence of previous outcomes. This context-sensitive approach helps the models better understand and mimic the language patterns of each class.

This project tests the *information-theoretic* hypothesis that compression algorithms can be employed to measure the similarity between files. Given a target text (`t`), the number of bits required to compress it is estimated, and the text is then classified based on which model compresses it with **fewer bits**.

In this report, we detail the steps of implementing the models, the decisions taken during the work and include relevant results.

3 Theory

3.1 Markov Models

A finite-context model, also known as a discrete-time Markov chain, uses high-order statistical information from an information source to predict the next outcome based on a conditioning context of past outcomes. Let $x_1^n = x_1 x_2 \ldots x_n$ represent the sequence of outputs (symbols) generated by the information source until time n.

In a k-order Markov model, the probability of the next symbol x_n given the previous k symbols is given by:

$$P(x_n \mid x_{n-1} \ldots x_{n-k}) = P(x_n \mid x_{n-1} \ldots x_{n-k}). \tag{1}$$

The goal of the model is to estimate the probability of the next outcome based on the observed past.

3.2 Estimating Probabilities

To estimate the probabilities, we collect counts that represent the number of times each symbol occurs in each context exemplified by table 1. Here, $N(s|c)$ represents how many times symbol s occurred following context c.

x_{n-3}	x_{n-2}	x_{n-1}	$N(0\|c)$	$N(1\|c)$
0	0	0	10	25
0	0	1	4	12
0	1	0	15	2
0	1	1	3	4
1	0	0	34	78
1	0	1	2	15
1	1	0	17	9
1	1	1	2	2

Table 1: Counts of occurrences for each symbol following each context

Given a context c, the probability of symbol s following context c can be estimated as:

$$P(e \mid c) \approx \frac{N(e \mid c)}{\sum_{s \in \Sigma} N(sd \mid c)}, \tag{2}$$

where $N(s \mid c)$ is the number of times symbol s occurred following context c, and Σ is the alphabet. For example, we may estimate the probability that a symbol ‘0‘ follows the sequence ‘100‘ as:

$$P(0|100) = \frac{N(0|100)}{N(0|100) + N(1|100)} = \frac{34}{34 + 78} \approx 0.30 \tag{3}$$

This method of estimating the probability of an event e is based solely on the relative frequencies of previously occurred events and can lead to assigning zero probability to events that have not been seen in the model. To mitigate this issue, a smoothing parameter α is added in probability estimation:

$$P(e \mid c) \approx \frac{N(e \mid c) + \alpha}{\sum_{s \in \Sigma}(N(s \mid c) + \alpha|\Sigma|)}. \tag{4}$$

4 Dataset

In an attempt to find a dataset, we explored platforms like Kaggle to find data on texts authored by both AI and human writers. However, many available datasets were too small for our needs, so we decided to compile our own dataset from various sources. The AI-generated texts were primarily generated using *ChatGPT*, but we also included texts from other large language models such as *GPT4*, *Claude*, *PaLM*, and *Llama-70b*.

The benefits of not using a single-source are **reduced bias** and **improved generalization**, enabling the model to perform better on unseen text. To train and evaluate the model, we require both a training set and a testing set. The training set is used to train the model and the testing set, which is not used during training, provides an unbiased evaluation of the model's performance.

4.1 Dataset Creation

We created our custom dataset by combining three different datasets from Kaggle that contained both target classes. The sources were:

- **Augmented data for LLM - Detect AI Generated Text**: This dataset was obtained from Kaggle and contains a file named `final_train.csv` [3].
- **DAIGT-V4-TRAIN-DATASET**: This dataset was obtained from Kaggle and contains a file named `train_v4_drcat_01.csv` [7].
- **DAIGT V2 Train Dataset**: This dataset was obtained from Kaggle and contains a file named `train_v2_drcat_02.csv` [6].

We published our custom dataset on Kaggle [5]. All the files in the dataset were previously cleaned, for example, duplicate entries were eliminated. The most important files are:

- `final_train.csv`: this file is 935.26 MB and it corresponds to the **training set** that we used.
- `final_train_balanced_char_count.csv`: this file is 693.95 MB and a balanced version of the `final_test.csv`. It corresponds to the **training set** we used.
- `final_test.csv`: this file is 193.72 MB. This the file we use as the **testing set**.

The testing dataset was sourced from **Augmented data for LLM - Detect AI Generated Text** [3] and has the exact structure of the training data.

4.2 Dataset Description

Every file in our dataset (training and testing) contains the same two columns: *Text* and *Label.*

- **Number of columns:** 2
  - *Text*: Contains the text content.
  - *Label*: Contains the label data (1 for AI-generated text and 0 for human-generated text).
- **Number of unique values in balanced training set**: 330,014 text entries

A human-written example from the dataset:

> *Label:* 0
> *Text:* We should keep the Electoral College for a number of reasons. While it is usually thought of as "outdated" or unfair, it actually serves to balance the influence of different regions and populations across the country.

5 Dataset exploration

To be able to find imbalances and patterns in our dataset, we performed Exploratory Data Analysis (EDA) on the unbalanced dataset in the file `final_train.csv`. This analysis was conducted using the script `data_analysis.py`.

Class Distribution We examined the portion of examples for each target class. In Figure 1, we verify that there are more texts with label 0 than 1. This made us realize the dataset required balancing of the data to eliminate model **bias** towards human-written texts (label 0).

Count of labels:

- Label 0: 249,316 instances
- Label 1: 165,007 instances

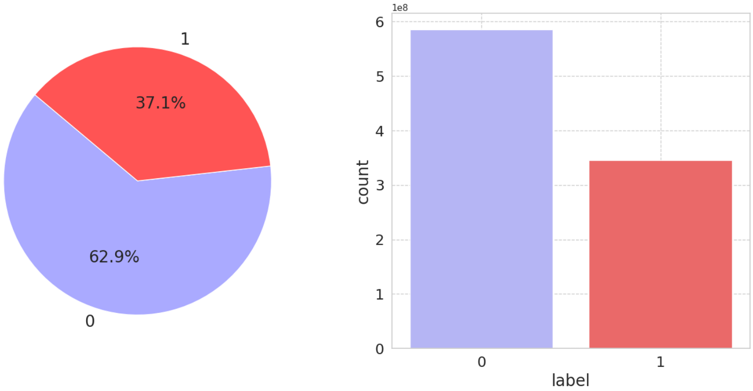
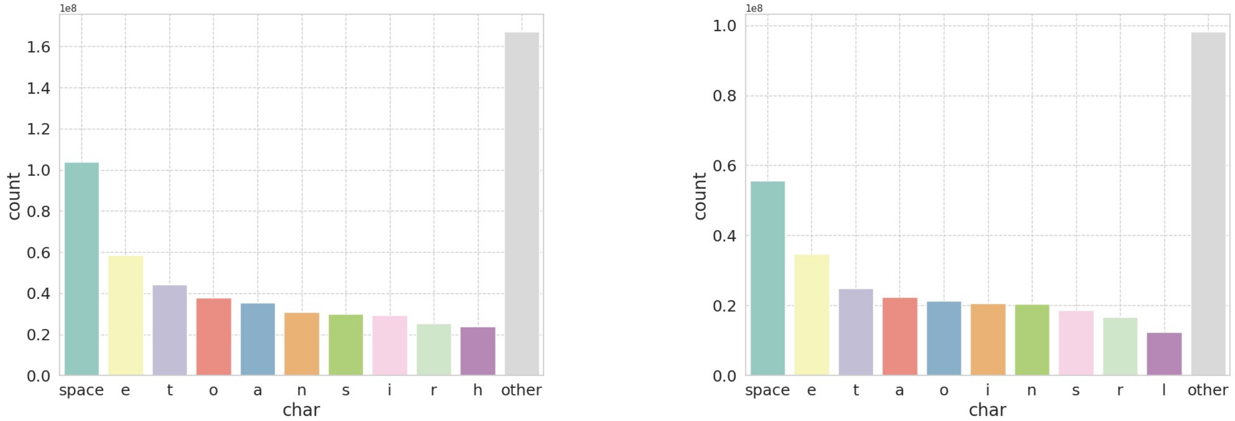


Figure 1: Proportion of each class in the dataset.

Character Count Distribution We then examined the distribution of characters in texts from each class to help us identify patterns.

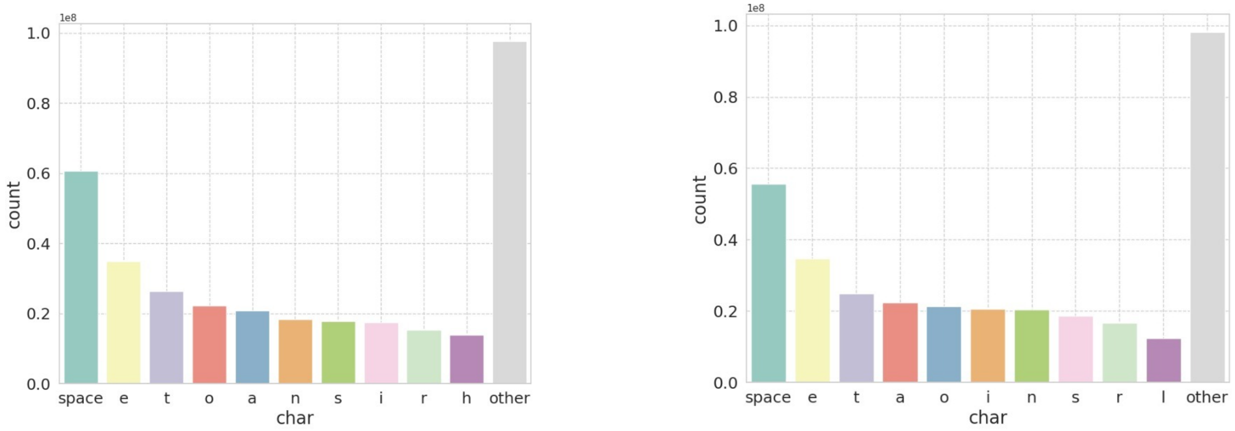


(a) Human texts letter distribution - Label 0

(b) AI texts letter distribution - Label 1

Figure 2: Comparison of the letter distribution in the unbalanced file for both classes - human-written text and AI-generated text

In both cases, the most common characters are ‘e’ and ‘t’ which might be related to the fact that both files are written in English and these letters appear in a lot of common words - typically refered as *stop words* [2] such as the word ‘the’. However, there are divergences, for example, the third most frequent letter is ‘o’ for human texts, and ‘a’ for AI texts which makes us hypothesize that our method might be efficient for discerning between both classes.



(a) Human texts letter distribution - Label 0

(b) AI texts letter distribution - Label 1

Figure 3: Comparison of the letter distribution in the balanced file for both classes - AI-generated text and human-written text

Balancing the datasets can significantly change the distribution of letters, as human-written texts initially made up 62.9% of the data and now are at 50.0%. However this reduction didn’t alter the letter distribution of the texts, except for the fact that there are less ‘spaces’.

6 Dataset Preprocessing

The preprocessing steps were aimed to clean and balance the dataset for optimal model training and evaluation. The preprocessing stage involved the following steps:

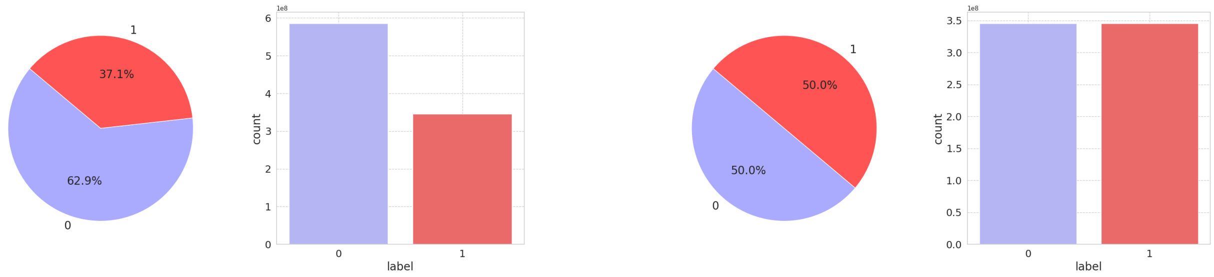
6.1 Cleaning Data

- **Removing deduplication**: We checked the original datasets for duplicate text entries (overlapping texts) and removed them, ensuring the data is unique.
- **Handling Missing Values**: Any missing values in the data were identified and removed.

6.2 Training Dataset Balancing

To minimize **bias** in the model, it is essential to balance the training dataset. Balancing ensures equal representation for both classes. The balancing was achieved by equalizing the character count across both classes and was performed with the script `balance_dataset.py`.

As seen in Figure 7, the dataset now contains a balanced distribution of total characters for both labels. Reducing the **bias** towards a specific class enhances the model **accuracy** and predictive abilities.



(a) Unbalanced Dataset

(b) Balanced Dataset

Figure 4: Comparison of the training dataset before and prior to balancing.

6.3 Testing set processing

The testing dataset doesn't need to be balanced since it will only be used in the **evaluation** stage, a stage where **bias** is not a problem as we are only validating the model and not actually training it.

Table 2: Testing-to-Training Ratio and Characteristics of Training and Testing Datasets

Metric	Training Set	Testing Set
Size (MB)	736.34	193.72
Percentage of Total Size	79.17%	20.83%
Total Unique Values	330,014	86,587
Label 0 Instances	55,845	55,714
Label 1 Instances	30,742	30,163

Testing-to-Training Ratio: The final balanced training dataset has `736.34 MB`, comprising 79.1% of the total data. The testing dataset is smaller with `193.72 MB`, comprising 20.9% of the data (see Table 2).

7 Methodology

Developed in C++, the main program is `was_chatted.cpp`. It estimates the number of bits required to compress texts with different models, including a model trained on AI-generated text and another one trained on human-written texts and assigns the `input_file` to the class that generates less bits.

The program accepts parameters as follows: `input_file+` represents the input file(s) to be tested, while `-m model_file` specifies the file where the trained model is saved. The program is designed to handle an indefinite number of classes through the `-m model_file` parameter, enabling its application to multi-class problems such as language detection, where there are multiple languages (e.g., English, Portuguese, Spanish).

7.1 Model Trainer

The `trainer.cpp` program trains a finite context model using one or more files `training_set+` while allowing customization through parameters such as `-k order` for the model order or window size, `-s smoothing_factor` for the (α) used in Equation 4, `-a alphabet` for the alphabet considered for training and `-r scaling_factor` that will be explained later on. The program also supports an option `-i` to ignore letter case during training.

7.1.1 FiniteContextModel class

The implementation of **FiniteContextModel** utilizes an `std::unordered_map` for efficient data storage, allowing for quick retrieval, updates and insertions. This data structure `context_counts` is essential for calculating the probability of symbols within a given context. It is defined as `unordered_map<string, EventMap>`, where EventMap is a structure that contains:

```
struct EventMap {
    unordered_map<char, uint32_t> events;
    uint32_t total;
};
```

- `events`: This `unordered_map` stores the count of occurrences for each character (`char`) within a specific context.
- `total`: This variable maintains the total count of events that occurred in the context.

This structure allows the model to efficiently compute the probability of a given symbol within a context by eliminating the need to calculate `total` during inference.

circular_buffer

A `circular_buffer` facilitates counting by efficiently retrieving the last k characters from the most recent context read in a file. Its fixed-size promotes efficient memory use by reusing space and discarding older contexts, eliminating the need for dynamic memory allocation. The `circular_buffer` implementation can be found in the file `circular_buffer.hpp` and was adapted from implementation in this article [4] from *EmbeddedArtistry*. The buffer is represented by `circular_buffer<char> buffer(k)`.

update

The `update` method is responsible for updating the model's context counts based on an input text or file. It processes an input source (`ifstream` for file or `string` for text) with a circular buffer of size `k`, updating context counts for each valid character using the `increment` method.

```
void update(ifstream &input);
void update(string &input);
```

increment

The `increment` function scales down the `counts` if they exceed the maximum limit `UINT32_MAX`. When the total event count (`counts.total`) reaches `UINT32_MAX`, the function divides the event counts (`counts.events`) and total count by a `scaling_factor`. This factor defaults to 2 but can be customized by the user - a higher value leads to more imprecision but might be necessary for larger files.

```cpp
virtual void increment(EventMap &counts, const char &event) {
    if (counts.total == UINT32_MAX) {
        cerr << "Warning: Event count has reached maximum size (UINT32_MAX). Scaling down counts." << endl;
        for (auto &pair : counts.events) {
            pair.second /= scaling_factor;
        }
        counts.total /= scaling_factor;
    }
    counts.events[event]++;
    counts.total++;
}
```

This method acts as a **fail safe**. In our case, we found that `uint32_t` was sufficient, as our training dataset is not very large — around ≈ 700 MB — so the fail safe was never triggered. However, this approach can **reduce** model precision, albeit not drastically, since exact counts are not required for our purposes.

estimate_bits

The `estimate_bits` method estimates the bits needed to encode text or files based on context counts. It has two overloads:

- *estimate_bits (context-event):* Calculates bits needed to encode a specific event (character) within a context, using the `probability` method to determine the probability and returning its negative base-2 logarithm.
- *estimate_bits (text):* Calculates total bits needed to encode text/files by using `estimate_bits (context-event)` for each character, with optional context count updates.

probability

The `probability` method calculates the probability of an event (character) in a context by dividing the event count by the total context count with a smoothing factor.

```cpp
float probability(const string &context, const char &event);
```

This method is exclusively used in `estimate_bits (context-event)` for event probability calculations.

7.1.2 FiniteContextModelTrainer class

The **FiniteContextModelTrainer** class also used to facilitate the training of finite context models based on input text data. It systematically trains models by processing text data and initializing all the necessary structures.

Support for different data source types This class also uses the *function overloading* [8] technique for C++ software design pattern in the function `train()` to support training with data of different types, including input files, text strings, and text files.

7.2 Model Saving

It's possible to save the models as a binary file with either default or custom filenames. The method `save()` in the class **FiniteContextModel** serializes model parameters such as `id`, `k`, `smoothing_factor`, `ignore_case`, and `alphabet`. It also serializes the model's data, including context counts and occurrences of events within each context.

7.3 Model Loading

The `load` function in the **FiniteContextModel** class reads a binary file to initialize the model's state. After reading the models parameters, it loads the data for each context into an EventMap `struct`, that keeps track of the occurrence counts of different events (characters) within each context.

7.4 Model Evaluation

The `finite_context_model_evaluator.cpp` file evaluates previously trained models' performance and reliability on the `testing_set`. The model `-m model_file+` and input files `testing_set+` should be provided as arguments. The script calculates the time taken to load the models and evaluates each input file and other metrics.

The class **FiniteContextModelEvaluator** class facilitates the models evaluations. It calculates the evaluation time for each model and multiple metrics such as **accuracy**, **precision**, **recall**, and **F1-score**. It also includes the calculation of the **confusion matrix**, useful to summarize the model's predictions in a matrix format, enabling us to see clearly the correct and wrong predictions.

Support for different data source types This class also uses the *Template Method* software design pattern in the function `evaluate()` to support evaluations using regular text files and strings.

7.5 Optimization Strategies

As a proof of concept, a version with **Morris Counter** was implemented for approximate counting. The class for this version is named **ApproximateFiniteContextModel**.

8 Results and Analysis

In this section of the report, we will detail how the model behaves, both in terms of training and evaluation. In the training stage, we will cover how different parameters affect the accuracy and the training time of the model. We will also verify experimentally if our model was able to predict correctly both the target classes, proving our hypothesis that an *theoretical-information* approach is a good way to distinguish AI and human written texts.

The model was trained using the training dataset `final_train_balanced_char_count.csv` with 693.95 MB. To read the `CSV` files, a `CSV` *Parser* [1] library was used. The model was trained not only with the CSV files but also with regular text files and strings using the same **FiniteContextModelTrainer** class. After training, the models were saved as binary files for later use.

8.1 Impact of context length k on model performance

As the context length (k) increases, the model's complexity also increases, requiring more resources and time to train. This is because higher k values require more calculations and processing, as the model must consider longer sequences of context. This additional complexity can provide more nuanced understanding of the patterns within the text. To test this, we recorded the accuracies while varying k, displayed on Figure 5.

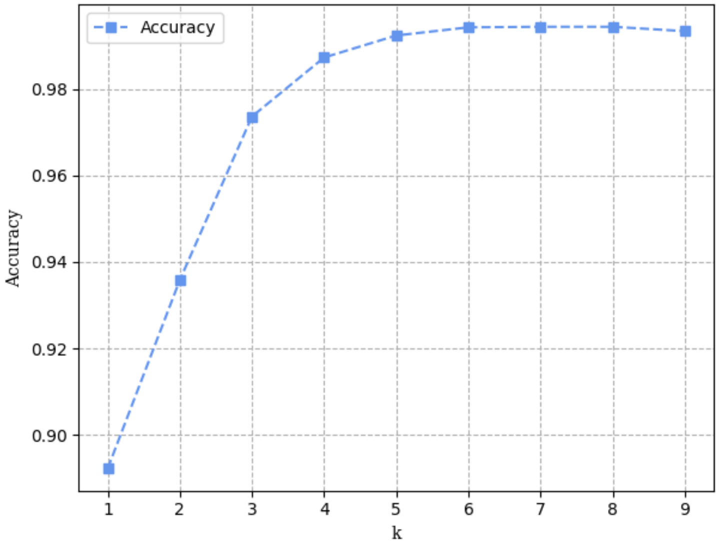
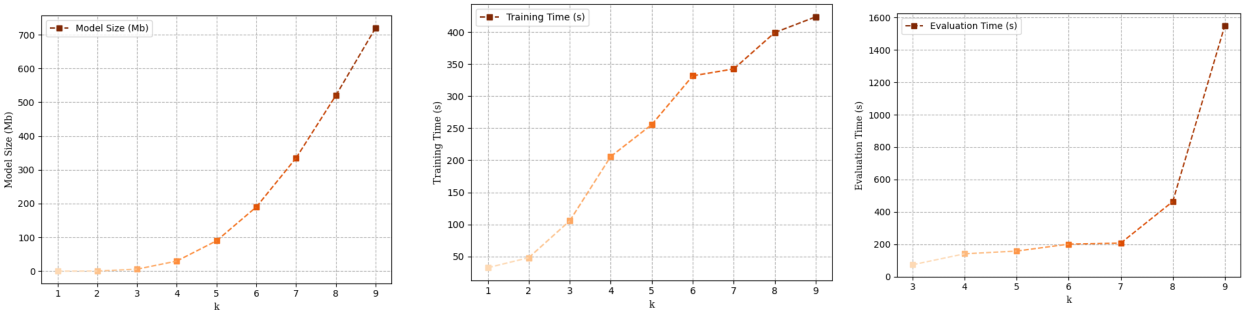


Figure 5: Accuracy as a function of the context length k

As seen in the Figure above, the accuracy increases rapidly with a higher k an it reaches a *plateau* when $k = 6$. This means that higher context lengths don't offer new information to the model. This is about the size of a word in English [9] which could mean that AI and human texts differ mostly in terms of the words that are used. Training times were also recorded for each model with varying context lengths (k).

Table 3: Model Size (MB) and Training Time as a function of the *context length* k.

k	Model Size (MB)	Training Time (s)
1	0.2	33.62
2	0.6	47.69
3	6	105.52
4	30	205.92
5	90	255.67
6	190	331.86
7	335	342.60
8	520	363.68
9	720	423.94



(a) Model Size (MB) as a function of k

(b) Training time (s) as a function of k

(c) Evaluation Time (s) as a function of k

Based on the analysis of the Figure 7 and Table 3, it is evident that the results align with our expectations. As the value of k increases, indicating a larger context window, both the size of the models and the time required for training proportionally increase. This observation is consistent with the theoretical understanding of finite-context models.

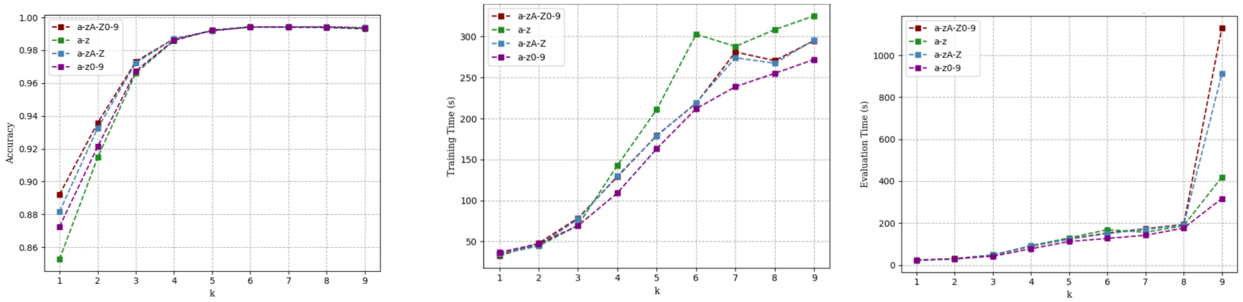
As we will see later, it becomes apparent that the size of the alphabet also influences the training and evaluation times. As the alphabet size grows, reflecting a character set, the time required for both training and evaluation increases.

Overall, these findings reaffirm the necessity of balancing model complexity with computational resources and time constraints, because we will see acuracies very close to each between models that take twice the amount of time to train and 5 times the size.

8.2 Alphabet parameter tuning

The choice of alphabet, the set of characters used in the model, can significantly impact model performance depending on the data set. To explore the influence of different alphabets on our models, we experimented with some predefined sets of characters.

The following graphs illustrate the performance metrics obtained from training models with different alphabet configurations:



(a) k vs Model Size (Binary File)

(b) k vs Training Time

(c) k vs Training Time

Figure 7: Comparison of the different alphabets

Our experiments with different alphabet configurations reveal minimal variation in model performance, indicating that the choice of alphabet has a limited impact on our specific task. However, a consistent trend is observed: as alphabet size increases, both training and evaluation times rise. This relationship between alphabet size and processing time makes logical sense in these types of algorithms.

While models trained with larger alphabets exhibit slightly higher accuracy in our case, the improvement is modest. This suggests that incorporating a broader range of characters may marginally enhance the model's ability to capture nuanced linguistic patterns but only for lower k values.

In summary, while alphabet choice may influence performance to some extent, its impact is minimal in our context. The primary determinant of model behavior is the alphabet size, with larger alphabets requiring more time for training and evaluation. Any gains in accuracy from larger alphabets must be weighed against the associated computational costs.

8.3 Model predictions

Approximately 50 combinations of the parameter values for context length k and smoothing Factor s were tested to find the one that would yield the best results.

- ⋆ **Context Length (k)**: $k = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$
- ♢ **Smoothing Factors (s)**: $s = [0.5, 0.75, 1, 1.25, 1.5]$

Table 4: Combination that yielded the best accuracy.

Parameter	**Value**
`smoothing_factor`	0.5
k	7
alphabet	`[a-zA-Z0-9]`
Training time (s)	342.5
Evaluation time (s)	206.5
Average evaluation time (s)	206.5
Total Bytes (MB)	35.14
Accuracy (%)	99.43

Our best model has **99.4%** accuracy as seen in table 4. In terms of time efficiency, the model performed the training in a reasonable amount of time.

The confusion matrix for the model is presented in Table 5. It reveals that the model achieves a very high **true positive rate** for predicting human-written text and a high **true negative rate** for predicting AI-written text which are the values we are aiming at maximizing in predictive model. The model was wrong in only a few occasions as the values for **false positive rate** (predicted wrongfully that a text was written by a human) and **false negative rate** (predicted wrongfully that a text was written by an AI) are quite low.

The evaluation metrics for the model are summarized in Table 6. The model also shows *recall* of 99.51%, meaning it correctly identifies almost all AI-written texts. Similarly, the model's *precision* of 99.63% indicates that most texts it predicts as AI-written are indeed AI-written. The *F1-score* of 99.57% combines these two metrics and suggests a highly accurate model. The formulas used to calculate these metrics can be found in the appendix.

Table 5: Confusion Matrix for the best model.

		Predicted	
		Human	AI
Actual	Human	55,560	285
	AI	205	30,537

Table 6: Model Evaluation Metrics for the best model.

Accuracy	**Recall**	**F1-Score**	**Precision**
99.43	99.51	99.57	99.63

9 Conclusion

The model demonstrates exceptional performance in distinguishing between human-written and AI-written texts. With an accuracy of **99.43%**, the model consistently and reliably classifies both classes. The high recall of **99.51%** indicates that the model correctly identifies almost all AI-written texts, while the precision of **99.63%** indicates that most texts predicted as AI-written are indeed from AI sources. The F1-score of **99.57%** underscores the balance between precision and recall.

The results confirm that the **information content** of AI written text is very different from the **information content** human-written text. They also suggest that a *theoretical information* approach using tools like compression is a good approach to building a model to classify text on wether it was written by AI or not.

Reference List

[1] *CSV-Parser*. Accessed 6 May 2024. May 2024. URL: https://github.com/vincentlaucsb/csv-parser.

[2] Kavita Ganesan. "What Are Stop Words? — Opinosis Analytics". In: *Opinosis Analytics* (Mar. 2023). URL: https://www.opinosis-analytics.com/knowledge-base/stop-words-explained.

[3] J. D. Herrera. *Augmented Data for LLM - Detect AI-Generated Text*. Kaggle. Available at: https://www.kaggle.com/datasets/jdragonxherrera/augmented-data-for-llm-detect-ai-generated-text. 2023.

[4] Phillip Johnston. "Creating a Circular Buffer in C and C++". In: *Embedded Artistry* (Dec. 2022). URL: https://embeddedartistry.com/blog/2017/05/17/creating-a-circular-buffer-in-c-and-c.

[5] *TAI-DAIGT*. Accessed 6 May 2024. May 2024. URL: https://www.kaggle.com/datasets/danielnivalis/tai-daigt.

[6] TheDrCat. *DAIGT V2 Train Dataset*. Kaggle. Available at: https://www.kaggle.com/datasets/thedrcat/daigt-v2-train-dataset. 2023.

[7] TheDrCat. *DAIGT-V4 Train Dataset*. Kaggle. Available at: https://www.kaggle.com/datasets/thedrcat/daigt-v4-train-dataset. 2023.

[8] Contributors to Wikimedia projects. "Function Overloading". In: *Wikimedia Foundation, Inc.* (Apr. 2024). URL: https://en.wikipedia.org/wiki/Function_overloading.

[9] Ann Wylie. "What's the Best Length of a Word Online?" In: *Wylie Communications, Inc.* (Jan. 2024). URL: https://www.wyliecomm.com/2021/11/whats-the-best-length-of-a-word-online.

Appendix

Metrics Formulas

Recall (Sensitivity or True Positive Rate):

$$\text{Recall} = \frac{\text{True Positives (TP)}}{\text{True Positives (TP)} + \text{False Negatives (FN)}}$$

Precision:

$$\text{Precision} = \frac{\text{True Positives (TP)}}{\text{True Positives (TP)} + \text{False Positives (FP)}}$$

F1-score:

$$\text{F1-score} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$