DETI - 2023/2024

Markov Models for Text Classification

TAI - Assignment 2

Diogo Paiva	103183
João Fonseca	103154
Rafael Gonçalves	102534

Algorithmic Theory of Information

Professor: Dr Armando J. Pinho

Professor: Diogo Pratas

May 9, 2024

Contents

Chapter 1

Introduction

1.1 Purpose of the report

The purpose of the report is to document the work developed by our group, for the second assignment of the Algorithmic Theory of Information course (2023/24), where we were tasked with the exploration of Markov models for text classification.

1.2 Repository on GitHub

The source code of the program *was_chatted* and the scripts used for the analysis and validation of the results can be found at our GitHub repository [1].

1.3 Text classification and compression algorithms

Text classification typically involves feature extraction and selection operations. The resulting features are then fed into a function that maps the feature space onto the set of classes and performs the classification. However, representing the original data by a small set of features can be seen as a special form of lossy data compression, since it's difficult to choose the smallest set of features that retains enough discriminant power to solve the problem.

To overcome this problem, we use compression algorithms that measure the similarity between files, bypassing the feature extraction and selection stages. The idea is as follows: for each class (there are two in this case), represented by a reference text r_i, we create a model that requires fewer bits to describe r_i. Then we assign to a text t the class corresponding to the model that needs fewer bits to describe it.

[1] https://github.com/joaompfonseca/tai-assignment2-group8

1.4 Markov models in text classification

Let us denote by $x_1^n = x_1x_2 \ . \ . \ .x_n$, $x_i \in \Sigma$, the sequence of outputs (symbols from the source alphabet Σ) that the information source has generated until instant n. A k-order Markov model verifies the relation

$$P(x_n|x_{n-1} \dots x_{n-k}) = P(x_n|x_{n-1} \dots x_{n-k} \dots),$$

where the sub-sequence c = $x_{n-1} \ . \ . \ . \ x_{n-k}$ is called the state or context of the process.

Markov models are particularly useful in text compression, because the next letter in a word is generally heavily influenced by the preceding letters. We can use the frequency of the pairs of letters to estimate the probability that a letter follows any other letter.

To estimate the probabilities the idea is to collect counts that represent the number of times that each symbol occurs in each context and represent in a table. Then to calculate the probability of an event *e*, based only on the relative frequencies of previously occurred events is

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

where α is the smoothing parameter and $N(s|c)$ indicates how many times symbol *s* occurred following context *c*.

Chapter 2

The *Was Chatted* program

2.1 Description

The `was_chatted` program is a command-line tool whose purpose is to accurately determine whether a text was rewritten or not by ChatGPT. It considers two reference texts: one not rewritten by ChatGPT, and another rewritten by ChatGPT. It uses Markov models to generate tables, one for each reference text. Using each table, one is able to estimate the number of bits per symbol a text under analysis would require to be compressed, for the corresponding reference text. If one considers that a lower number of bits per symbol represents a better encoding, therefore indicating a higher similarity with the reference text, by comparing the two values, it is possible to determine the likely origin of the text under analysis.

2.2 Relevant Features

Some relevant features present in this program include:

- **Customised alphabet should be provided** to allow for a better control of the considered symbols during the program execution.
- **Caching into a binary file** of the Markov model tables, based on the Markov model order and a hash value of the considered alphabet, for a given reference text file.
- **Usage of unsigned integers of 32 bits** to store the counts, allowing up to 4,294,967,295 different values.
- **Prevention of overflows in the counts** during table creation and probability calculation. A reduce factor parameter is used to divide all the counts when an overflow is bound to happen.
- **Smoothing factor in the calculation of probabilities** to account for unseen symbols of the alphabet during the construction of the table.

- **Support for logging of results** to a specific file allows us to test and analyse the program outputs for multiple parameter configurations.

2.3 Compile

It is optional since you can use the provided executable inside the `bin` directory.

- Run `cd bin` in root.
- Run `cmake .. && make` to compile the program.

2.4 Execute

- Run `cd bin` in root to change to the executable's directory.
- Run `./was_chatted REQUIRED OPTIONAL` to execute hte program.

2.4.1 Required arguments

- `-n rh_file_path`: path to the file containing the text not rewritten by ChatGPT (string).
- `-r ch_file_path`: path to the file containing the text rewritten by ChatGPT (string).
- `-t t_file_path`: path to the file containing the text under analysis (string).
- `-a alphabet_file_path`: path to the file containing the considered alphabet (string).
- `-k markov_model_order`: order of the Markov model (int).
- `-s smoothing_factor`: parameter to smooth the first probability estimation (double).

2.4.2 Optional arguments

- `-h`: shows how to use the program.
- `-d reduce_factor`: factor to reduce the counts of the Markov model to prevent overflow (int, default is 2).
- `-l log_file_path`: path to the file where the log will be written (string, default is empty).

2.4.3 Example

```
./was_chatted\
-n ../example/no.txt\
-r ../example/yes.txt\
-t ../example/test.txt\
-a ../example/alphabet.txt\
-k 6\
-s 0.1
```

2.5 Flow of execution

1. Parse program arguments and print program configuration.
2. Load models of text not rewritten by ChatGPT (*rhModel*) and of text rewritten by ChatGPT (*rcModel*), with a Markov model order k and an alphabet Σ (repeat for each).
   (a) Read the file containing the alphabet and determine its hash value h.
   (b) Using the file name, k and h, construct the cache file path.
   (c) If model for k is cached at the determined location
      i. Load the table from the cache file and return.
   (d) Otherwise, read the contents from the file.
      i. Read the file containing the text and only consider characters contained in the alphabet Σ.
      ii. Generate the table.
         A. Consider a context c with size k.
         B. Consider the next symbol in the text s.
         C. Increment the count on the table of s occurring given c.
         D. In case of a count overflow, divide the counts on the table by a reduce factor.
      iii. Save the table to the cache file.
3. Load analyser for the texts under analysis.
   (a) Read the file containing the texts under analysis.
4. Until there are no lines of text to analyse.
   (a) Estimate the number of bits per symbol of the text under analysis t on both models, *rhEstimatedBps* for *rhModel* and *rcEstimatedBps* for *rcModel* (repeat for each).
      i. Consider the text t.
      ii. Consider a summation S.
         A. Consider a context c with a size k.
         B. Consider the next symbol in the text s.
         C. Consider $N(s|c)$ as being the counts of s given c, $\sum\limits_{a\in\Sigma} N(a|c)$ as being the counts of s given c, a smoothing factor α and the size of the alphabet $|\Sigma|$.
         D. Obtain $P(s|c) \approx \frac{N(s|c)+\alpha}{\sum\limits_{a\in\Sigma} N(a|c)+\alpha|\Sigma|}$ as being the probability of s occurring given c.
         E. Add $-log_2(P(s|c))$ to S.
      iii. Return $S/|l|$.
   (b) If $rhEstimatedBps < rcEstimatedBps$, then the text t is said to have not been rewritten by ChatGPT (*class 0*), otherwise rewritten by ChatGPT (*class 1*).
   (c) Optional: If a file path for the logger is specified, the *rhEstimatedBps*, the *rcEstimatedBps* and the *class* of the text t are written into it.
5. Exit the program.

Chapter 3

Validation

Probabilistic models usually rely on assumptions or make associations that may not reflect the nature of the data being modelled. Validation helps identify systematic errors in the output and also reduce bias towards particular outcomes. In this sense, we conducted several experiments to assess the performance of the `was_chatted` program in different datasets, using common classification metrics (accuracy, precision, recall, F1-score). Furthermore, we tested different values for the program's hypeparameters to understand their impact on the results.

3.1 Metrics

Since we are dealing with a binary classification problem, we gathered the most common metrics to evaluate the performance of a binary classifier [1], from which we selected the following: accuracy, precision, recall and F1-score.

Accuracy is defined as the ratio of the number of correct predictions to the total number of predictions (Eq. 3.1), while precision is the ratio of the number of true positive predictions to the total number of positive predictions (Eq. 3.2) and recall is the ratio of the number of true positive predictions to the total number of actual positive instances (Eq. 3.3). Finally, the F1-score is the harmonic mean of precision and recall (Eq. 3.4).

$$\text{Accuracy} = \frac{TP+TN}{TP+TN+FP+FN} \tag{3.1}$$

$$\text{Precision} = \frac{TP}{TP+FP} \tag{3.2}$$

$$\text{Recall} = \frac{TP}{TP+FN} \tag{3.3}$$

$$\text{F1-score} = 2 \times \frac{Precision \times Recall}{Precision + Recall} \tag{3.4}$$

where TP is the number of true positive predictions, TN is the number of true negative predictions, FP is the number of false positive predictions and FN is the number of false negative predictions.

From the equations, we see that accuracy includes the true negatives, in contrast to the other metrics. Typically, in anomaly detection problems, where the number of negative (normal) instances is often much larger than the number of positive (anormal) instances, we are more interested in detecting the positive instances, so precision, recall and f1-score are more relevant. In a real application of our particular problem, if we consider AI-generated text as an anomaly, these metrics are preferred over accuracy.

3.2 Datasets

Harnessing 2 public datasets (DAIGT train dataset [1] and aadityaubhat/GPT-wiki-intro [2], we were able to collect a variety of texts that were either AI-generated or not.

Table 3.1: Datasets used for model assessment

Dataset	**Human Texts**	**AI Texts**	**Total size**
1. DAIGT-V4-train-dataset	27370	46203	73573
2. aadityaubhat/GPT-wiki-intro	150000	150000	300000

Conducting an Exploratory Data Analysis (EDA) on the datasets, we noticed a class imbalance in the first dataset, with one class having a much larger number of instances than the other. This skewed distribution of classes can lead to biased models, as they would predict the majority class more often. To mitigate this issue, we downsampled the majority class to have the same number of instances as the minority class.

As we needed samples for building the Markov models and others for analysis, we split the datasets into training and testing sets: the first dataset is divided with a 80-20 ratio and the second dataset can be downloaded with approximately 70-30 ratio.

In order to make the results more reliable, we used a k-fold Cross-Validation (CV) strategy, also referred as v-fold CV in the literature [2]. In a nutshell, the dataset is divided into k folds, where k-1 folds are used for training and the remaining fold is used for validation. This process is repeated k times, with each fold being used once as a testing set. The final performance is the average of the k iterations (Fig. 3.1). As default, we use 10 folds.

[1] https://www.kaggle.com/datasets/thedrcat/daigt-v4-train-dataset/data?select=train_v4_drcat_01.csv
[2] https://huggingface.co/datasets/aadityaubhat/GPT-wiki-intro

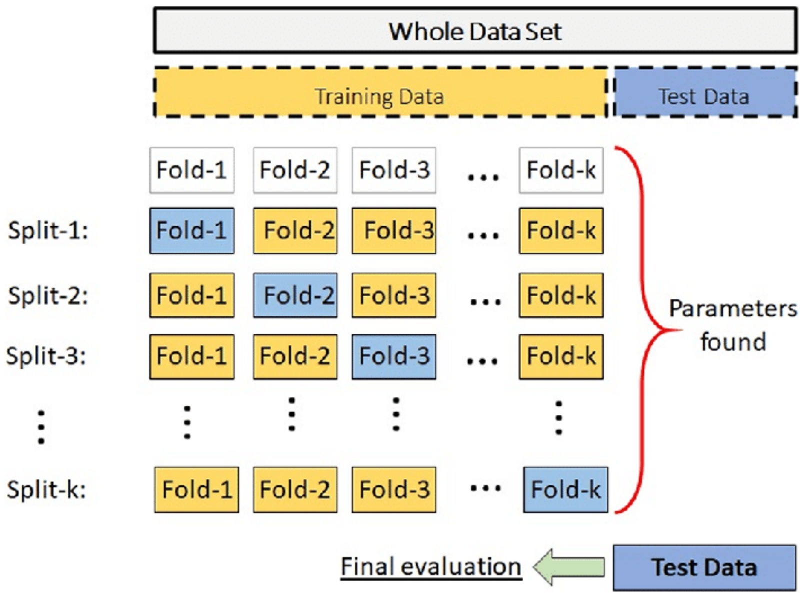

Figure 3.1: K-fold Cross-Validation [3]

3.3 Experiments

In order to find the best configuration hyperparameters for our solution, we carried out a series of experiments. After experimenting with many different alphabets, we concluded that the best for the DAIGT dataset was all the lowercase letters from the English alphabet, the numbers 0-9, and the white space. As for the WIKI dataset, it was similar to the previous one, with the addition of all the corresponding uppercase letters from the English alphabet.

As a first step, we intended to find the best value for the order of the Markov model, k, and for the smoothing factor, s. We accomplished this by running the program on both datasets, keeping all hyperparameters fixed and only varying k and s. As can be seen in Fig. 3.2, by varying k in the DAIGT dataset, we have $k = 8$; on the WIKI dataset, as can be seen in Fig. 3.3, we have $k = 4$, since we couldn't run experiments for a larger value of k on our computers. As for the smoothing factor s, the best value for the DAIGT dataset was $s = 0.1$, as shown in Fig. 3.4, and for the WIKI dataset was also $s = 0.1$, as shown in Fig. 3.3.

After observing the results obtained for each dataset, we can easily conclude that the best dataset for our purpose is DAIGT, as it gives more consistent results. Therefore, the next experiments were carried out using the DAIGT dataset, with $k = 8$ and $s = 0.1$.

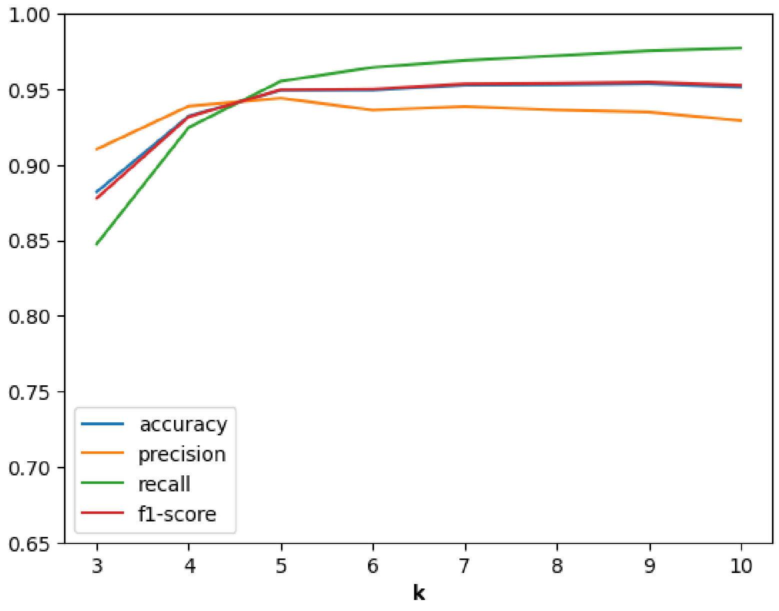

Figure 3.2: Varying Markov model order *k* using the DAIGT dataset (lowercase)

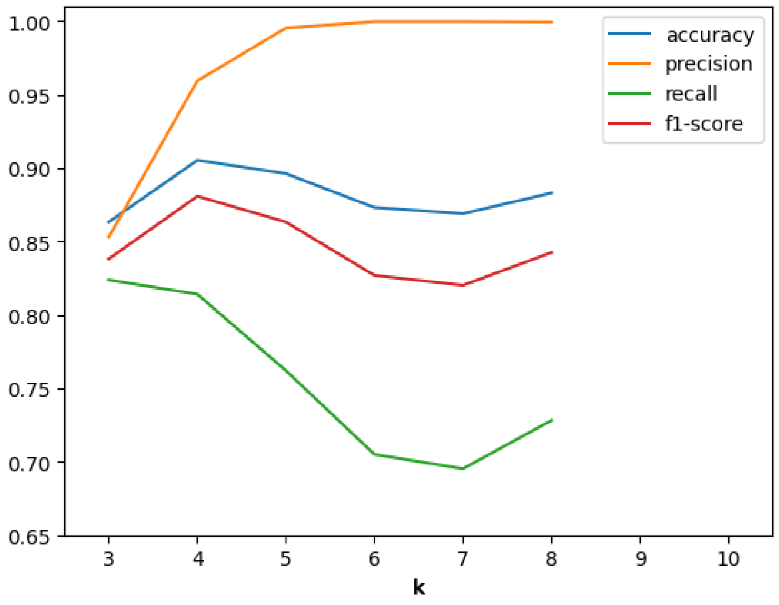

Figure 3.3: Varying Markov model order *k* using the WIKI dataset

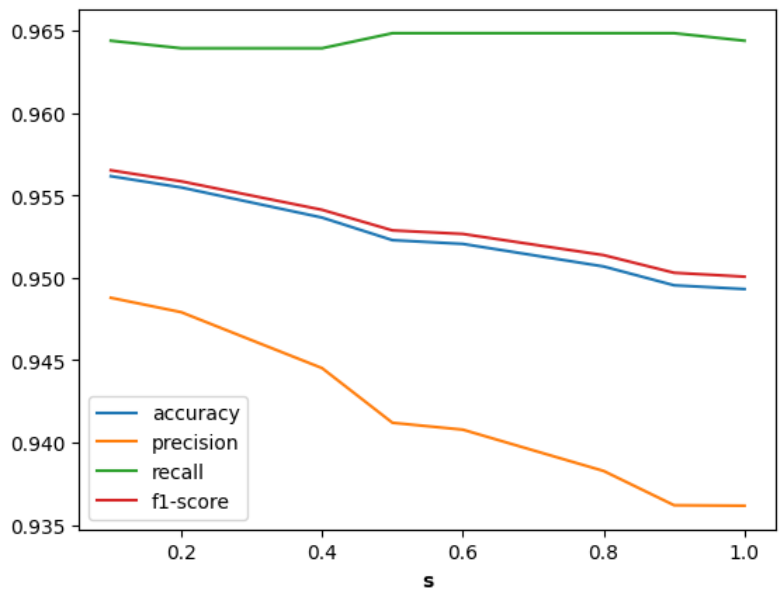

Figure 3.4: Varying smoothing factor s using the DAIGT dataset (lowercase)

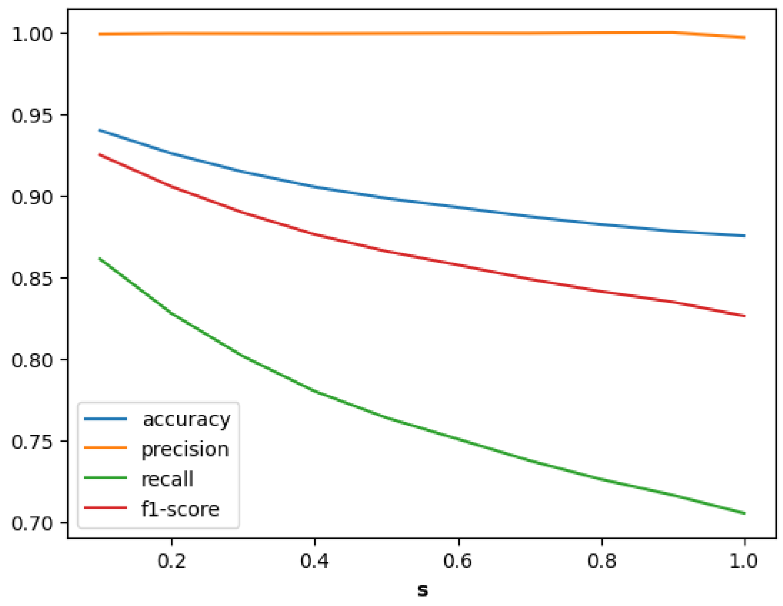

Figure 3.5: Varying smoothing factor s using the WIKI dataset

Considering the DAIGT dataset, and $k = 8$ and $s = 0.1$, we wanted to see if variations in the size of the text under analysis or the size for each entry in the training set would have an impact on the validation metrics.

As seen in Fig. 3.6, the results of the metrics improve as the size of the text under analysis increases. This can be attributed to the fact that a larger amount of information available allows for a more precise classification of the text at hand. When it comes to the size of the entries for generating the tables, there is a sudden improvement when it goes from 3×10^7 to 3.5×10^7

characters, as seen in Fig. 3.7. The reason for this unexpected behaviour is uncertain, but it makes sense in a way, since more training data (more information) provides a better representation of the class. We also observe that recall is very high, while precision is a bit lower, from which we can infer that the model is very good at detecting AI-generated text, but it also classifies some human-generated text as AI-generated.

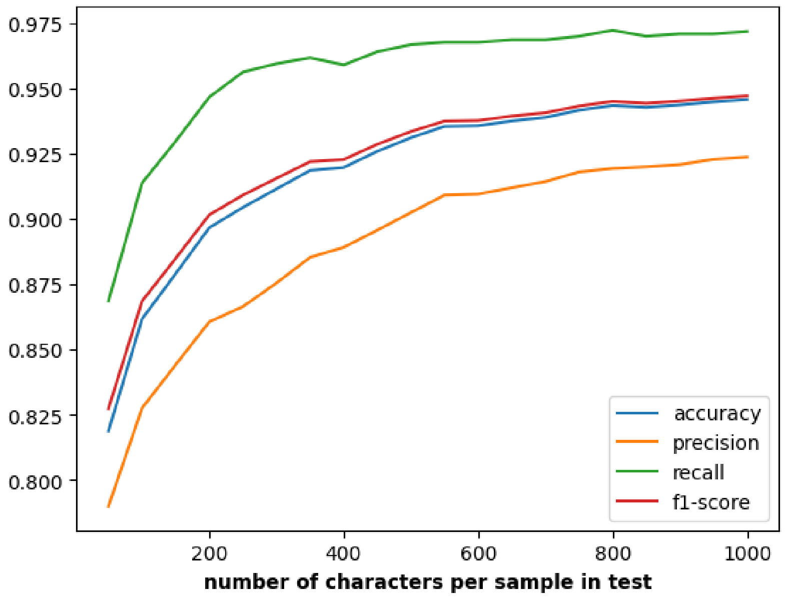

Figure 3.6: Varying size of the text under analysis using the DAIGT dataset (lowercase)

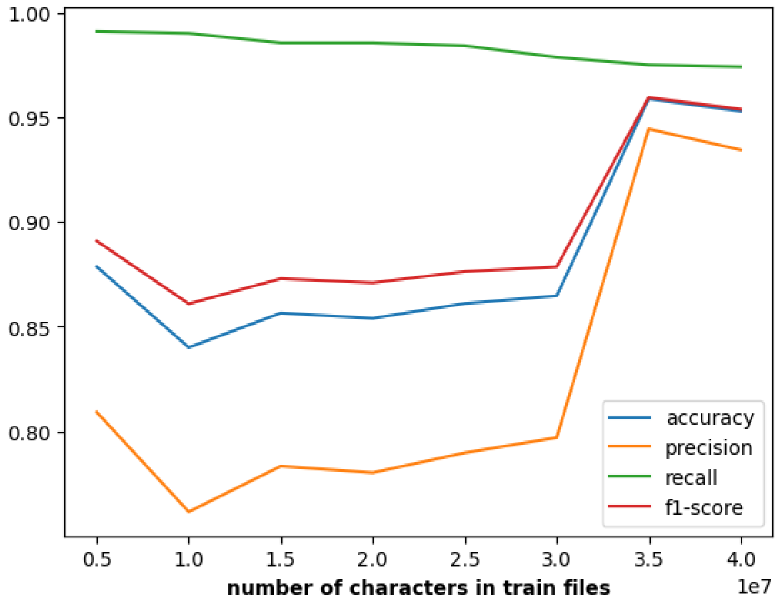

Figure 3.7: Varying size of the training text using the DAIGT dataset (lowercase)

3.4 Best results

3.4.1 Hyperparameters

- **Dataset:** DAIGT dataset.
- **Alphabet:** a-z, 0-9 and white space.
- **Markov model order:** $k = 8$.
- **Smoothing factor:** $s = 0.1$.
- **Reduce factor:** 1.

3.5 Metrics

- **Accuracy:** 96.53%
- **F1-Score:** 96.57%
- **Precision:** 95.53%
- **Recall:** 97.63%

Chapter 4

Final considerations

The present assignment aimed to develop a command-line tool to determine whether a text was generated by ChatGPT or not. The program was validated using two publicly available datasets, and the results were promising, proving that there is a correlation between the true label of a text and the number of bits per symbol produced with the Markov models. Regarding hyperparameter tuning, our evaluation relied on common classification metrics, such as accuracy, precision, recall and F1-score. We tested different configuration grids for the Markov model order and the smoothing factor, which affect the model performance. The best combination for DAIGT-V4-train-dataset was a Markov model order of 8 and a smoothing factor of 0.1, using an alphabet with lowercase letters, numbers and space. It is also important to study the impact of the references length or the target length on the model performance, so we varied then and conclude that they contribute to a performance improvement. Nevertheless, the program is overall very sensitive to the hypeparameters and dataset.

Finally, of all the implemented features, we would like to emphasise the user-defined alphabet, the caching mechanism, the prevention of overflows through a reduction factor, the results logging and the cross-validation strategy. These features were essential for the program's performance and reliability.

References

[1] Sebastian Raschka. An overview of general performance metrics of binary classifier systems. *CoRR*, abs/1410.5330, 2014.

[2] Sylvain Arlot and Alain Celisse. A survey of cross-validation procedures for model selection. *Statistics Surveys*, 4, January 2010.

[3] Ender Sevinç. An empowered adaboost algorithm implementation: A covid-19 dataset study. *Computers Industrial Engineering*, 165:107912, 01 2022.