UNIVERSITÀ DELLA CALABRIA

DIPARTIMENTO DI MATEMATICA E INFORMATICA

Master's Degree in *AI&CS*

Deep Learning - Project report

LLM - Detect AI Generated Text

Group members

Alessia Donata Camarda (242466)

Giuseppina Pia Varano (247753)

Cristian Porco (252049)

ACCADEMIC YEAR 2023/2024

1 Introduction

In recent years, **large language models (LLMs)** have become increasingly sophisticated and *capable of producing texts that are difficult to distinguish from those written by humans*. Ethical and professional concerns also arise regarding the responsible use of LLMs, both in terms of the spread of misleading information and impact on student's skill development.

These concerns led to the establishment of a Kaggle competition [8] that challenges participants to develop a machine learning model which can accurately detect whether an essay was written by a student or an LLM.

2 Detect AI Generated Text

2.1 Dataset Description

The competition provided a starting dataset called `llm-detect-ai-generated-text`. It comprises essays written by students or generated by a variety of LLMs. All of them were written in response to one of seven essay prompts.

However, two main datasets were used to train our models:

- `daigt-v3-train-dataset` [9]: it was created by merging the starting dataset and some samples generated giving in input to different LLMs the prompts available from the competition dataset. Below, we report the LLMs used and the number of samples generated by each LLM (view *Table 1*). Data cleaning has already been partially applied by the creator of the dataset (they removed null and duplicated values).

- `extracted-features-dataset`: it was generated on the fly with the aim of training an SVM model that recognizes features in texts and classifies the input of the previous dataset based on these features. For each text of the daigt-v3-train-dataset, some functions are applied on it with the aim of extracting relevant features from it. We will explain this choice and this process better in the following sections.

LLM	Number of samples generated
persuade_corpus	25996
chat_gpt_moth	2421
llama2_chat	2421
mistral7binstruct_v2	2421
mistral7binstruct_v1	2421
original_moth	2421
train_essays	1378
llama_70b_v1	1172
falcon_180b_v1	1055
darragh_claude_v7	1000
darragh_claude_v6	1000
radek_500	500
NousResearch/Llama-2-7b-chat-hf	400
mistralai/Mistral-7B-Instruct-v0.1	400
cohere-command	350
palm-text-bison1	349
radekgpt4	200

Table 1: `daigt-v3-train-dataset`: number of samples generated using the i-th LLM.

2.2 Methods

2.2.1 Preprocessing

Since our architecture uses different models within it, the preprocessing on the daigt-v3-train dataset is the same up to a certain point, then it changes depending on the type of model that will use the text of the dataset as input.

1. First of all, the data is undersampled with the aim of making the dataset balanced. Undersampling is carried out taking care to not eliminate from the dataset texts produced by LLMs whose contribution is lower than the others (i.e. the number of samples generated by the specific machine is lower than the number of the other machines); this is done to prevent the creation of bias within the dataset, otherwise the sentences would have been generated only by machines whose patterns could be easily learned given the large number of samples within the dataset. To achieve this, the `daigt-v3-train-dataset` was divided by prompts, and then the number of samples was balanced between those generated by artificial intelligence and those by humans, maintaining as threshold the minimum of the two.

2. Then, the texts in the dataset are modified in order to remove emojis or characters identifiable as such from within them. In particular, we are afraid that, if too much human-correlated, those chars can be associated by the network directly to a label rather than to the another one.

Preprocessing for features extraction

Using the `spaCy` library [1], we conduct our features extraction. Using an English language model that

contains pre-trained linguistic information, including part-of-speech recognition, syntactic parsing, entity recognition, and more, we analyze the texts of the daigt-v3-train-dataset and we obtain useful linguistic information. In particular, we are interested in: length of every text, average lengths of sentences and paragraphs, use of punctuation and stop words. Using the `scikit-learn` library [4], we include in our preprocessing the *Term Frequency-Inverse Document Frequency* (TF-IDF) analysis:

- Term Frequency (TF) measures the frequency of a word in a specific document.
- Inverse Document Frequency (IDF) measures the importance of a word across the entire corpus.

TF-IDF is the product of TF and IDF and is designed to give more weight to words that are important in a specific document but not common across the corpus. Using the TfidfVectorizer, we create a count vector for each document, where each element of the vector represents the frequency of a specific word in the text. Since the count vector is created starting from the data present in our training set, during the prediction and evaluation phase, it is necessary to pass as input to the TfidfVectorizer the vocabulary created by it after being trained on the training set.

Preprocessing for using Bert and GPT pretrained Models

Transfer learning is a machine learning technique where a model trained on one task is adapted for a second related one. Instead of training a model from scratch for a specific task, someone can start with a pre-trained model on a different but related task and fine-tune it for the one at hand. This is particularly useful when you have a limited amount of labeled data for the target task. Pre-trained models have already learned useful features and patterns from the data they were trained on: in natural language processing, for example, models like BERT or GPT have been pre-trained on vast amounts of text data. When applying these pre-trained models to a new task, one can either use them as feature extractors (extracting useful features from the input data) or fine-tune them on a smaller dataset related to the specific task. The idea is that the knowledge gained from learning one task can be transferred and leveraged for another task, potentially leading to faster convergence and improved performance, especially when the amount of data is low, and this is the reason why in our setup, we deemed it necessary to leverage transfer learning with two models.
Before using the pre-trained models of Bert and GPT, it is necessary to divide the texts into tokens. To do this, we use the corresponding tokenizers of Bert and GPT. They return similar output, as well as their behavior is similar, but what they use and how they generate tokens is slightly different.

- Bert Tokenizer: it does not simply divide the text into tokens, it usually uses a subword tokenization process which is useful for a correct tokenization. The output of the tokenizer is mainly composed of the following elements:
  - input_ids: a sequence of unique identifiers representing tokens in the text. Each substring is mapped to a unique identifier within the BERT vocabulary (the tokenizer uses a specific vocabulary to map each word or substring of text to a unique identifier).
  - attention_mask: a binary mask indicating which positions are filled with valid tokens and which are padding positions. This mask allows the model to ignore padding tokens during training.
- GPT Tokenizer: it breaks text into tokens using a "whole-word-based" tokenization approach, meaning that the tokens correspond to the actual words in the text. Furthermore, GPT can

dynamically deal with new tokens or words that are not present in the vocabulary during training, thanks to its text generation capability. Its output is similar to the Bert Tokenizer one but the use of the attention_mask is different: the attention mask, in fact, is not needed; GPT uses an "autoregressive" attention mechanism in which each position in the sequence has access to all previous positions, but not subsequent ones.

The idea of using the same dataset on two different pre-trained models implies that the same information gets truncated based on the model used. More specifically, BERT takes a maximum of 512 tokens as input, while GPT-2 takes 1024 tokens. After the development of BERT and attempts at fine-tuning, numerous methods have been proposed in recent times to avoid losing information from long sentences in the process from tokenization to model embedding. We mention below some proposals that have been considered during the development of the project:

- chunking or segmentation;
- sliding window approach: the window is let slide over the long paragraph, in order to create overlapping segments of the text. This way, information from adjacent segments is shared, and the model can capture context across boundaries;
- consider using models like Transformer-XL or Longformer [2];
- use of hierarchical or multi-level models where a high-level model processes larger chunks and a low-level model refines the representation.

Despite each of these models appearing to be a good solution for handling long sentences, computational complexities are non-negligible factors when working with concatenated pre-trained models. While the complexity of BERT scales as $o(n^2)$, *Pappagari et al.* [11] propose two models beyond BERT, RoBERT and ToBERT, which scale as $o(n \cdot k)$ and $o(n^2/k^2)$, respectively. These complexities translate into timeframes that are incompatible with the runtimes imposed by platforms with accelerators: herefore, both the inputs to GPT and BERT models have been truncated to 512 tokens for consistency across inputs.

2.2.2 Architectures

We have employed an ensemble architecture in our project: we combine the results of different prediction models to improve the overall performance and generalization of the system. In particular, we use a stacking strategy: multiple models are trained independently, and a final model (usually called meta-model) is then trained to make predictions based on the outputs of the base models. Our architecture can therefore be summarized as in the following image:

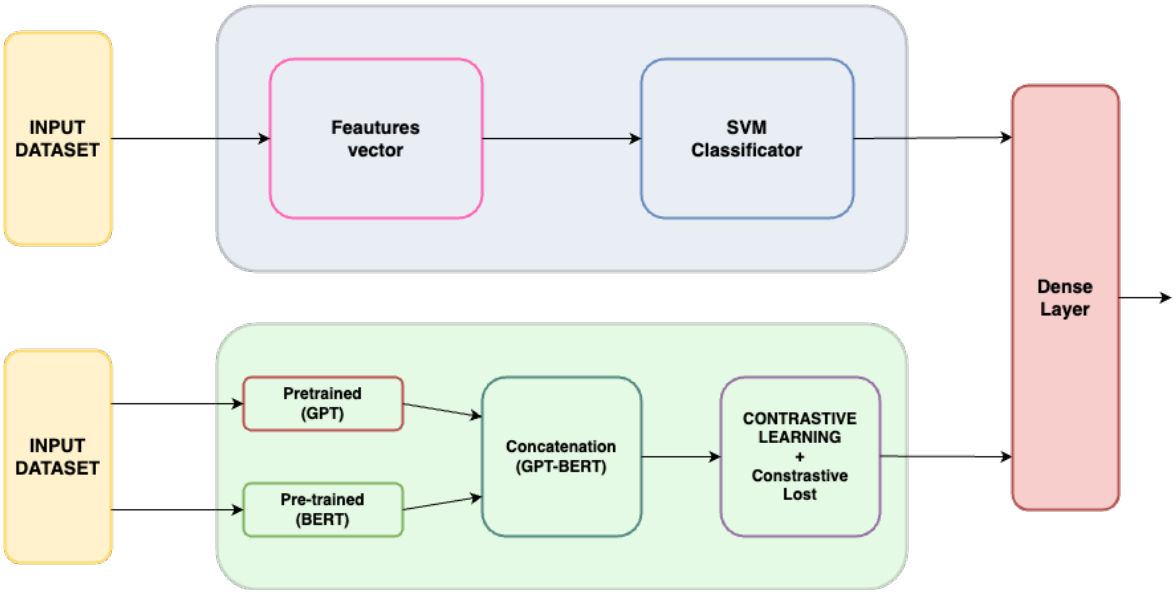

Figure 1: Project architecture

We have decided to use the following basic models for these reasons: several papers [5] highlight the good generalization capacity of machine learning models in the task we are dealing with. In particular, the paper cited demonstrates the good performance of SVMs: indeed, the main goal of SVMs in classification problems is to try to identify the hyperplane (a kind of plane in space) that is equidistant from the closest instances of each class, called support vectors. Moreover, works like Zellers' [13] discuss the ability of LLMs to recognize texts generated by them more easily than texts not generated by them. The need to use pre-trained models led us to create a hybrid model that could encompass at least one of the sentence-generating models from the input dataset and a second one representing the state-of-the-art in large language models. For the reasons mentioned above, the choice fell on GPT-2, which constitutes our dataset of 2421 samples, and on BERT. In attempting fine-tuning on it, BERT has proven to be a valid starting point in the study of large language models over the past years [10, 12, 11].

By combining the capabilities of BERT and GPT into a single overall representation, we tried to let the model benefit from the information captured by both pre-trained models. What we have in output from the model that uses both Bert and GPT is, therefore, the concatenation of the representations of the input sentences obtained from both models. Given such representations, then, one of the best techniques we can use to work on this latent space is contrastive learning. Its main goal is to learn data representations in a space where similar examples are positioned close to each other, while different examples are separated by a significant distance, which is exactly what we really need to be able to correctly classify our texts. In the contrastive learning framework, samples are typically augmented to create positive pairs and negative pairs: positive pairs consist of similar samples, while negative pairs consist of dissimilar samples. The model learns to map positive pairs close together and negative pairs far apart in the learned representation space [3]. In our case, we used the function provided by the TensorFlow framework for Supervised Contrastive Learning [7], which aims to minimize the following loss function:

$$\mathcal{L}^{self} = \sum_{i \in I} \mathcal{L}_i^{self} = -\sum_{i \in I} log \frac{exp(z_i \cdot z_{j(i)}/\tau)}{\sum_{a \in A(i)} exp(z_i \cdot z_a/\tau)} \tag{1}$$

2.2.3 Training and Experiments

Accelerators

The TPU (Tensor Processing Unit) is a type of specialized processor designed to perform specific operations used in neural networks and deep learning and to work efficiently with frameworks like TensorFlow. The use of the TPU during the training phase has proven to be necessary for performance reasons: GPU and CPU are not powerful enough to train large neural networks and the time required to train them without using the TPU can exceed hours. Therefore, using TPU allowed us to reduce training time and improve energy efficiency. To use this accelerator, however, we needed to modify the project notebook so that its use was enabled, the fitting operations used consistent batches, and the model definition was performed using the accelerator. However, it must be specified that the TPU was used only in certain phases, in particular, during the tokenization and training of the encoder which uses these tokenizations as input, during the training of the classifier and of the final model, and to obtain predictions from these models . Our SVM model, however, unfortunately could not be trained using the TPU as the necessary libraries are configured to use only the CPU. The scapy library can be configured to use the GPU but, since it is only possible to use one accelerator at a time, we decided to use the TPU directly so as to make at least the operations discussed above faster.

The training phase: a deep insight into the structure

Below we illustrate the results obtained during the training phase of the different branches of the model. In particular, we report the predictions produced by each model, the area under the curve, the loss and accuracy trend of the trained neural networks.

Starting from the top branch of the architecture, features were extracted from the dataset upon which the SVM model was trained. Knowing that, the results on the test, validation, and training sets are listed below.

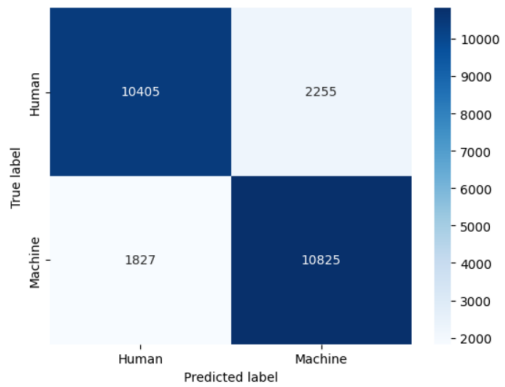

Figure 2: Confusion matrix for the training set generated using the SVM model.

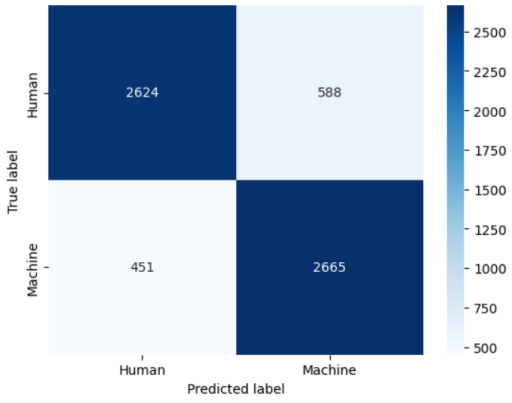

Figure 3: Confusion matrix for the validation set generated using the SVM model.

With a machine learning model, as mentioned in the paragraph above, we already achieve good results on the classification task: the SVM model is thus able to distinguish between sentences generated by the LLM and human ones. Anyways, the problem lies in the fact that a classification based on text-extracted features heavily relies on the text itself: it's possible that depending on the source

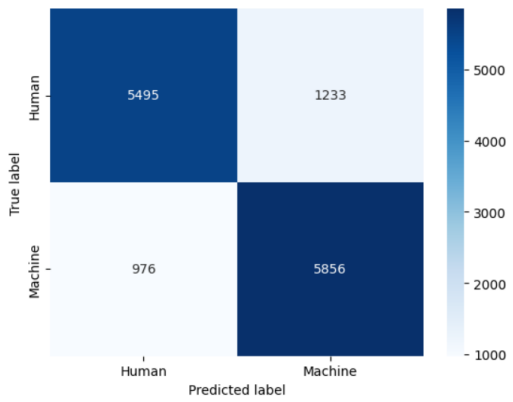

Figure 4: Confusion matrix for the test set generated using the SVM model.

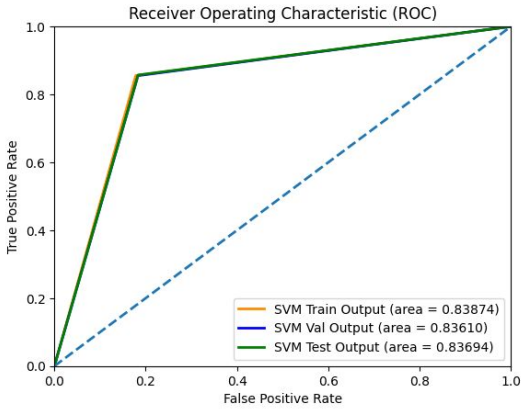

Figure 5: ROC curve calculated from the training, validation, and test sets of the SVM model.

type or the research context, the SVM may still not be able to complete the task. It is indeed known that many AI detection networks need to be trained on different types of sources, they can be papers from a certain field, poems, newspaper articles, etc.

Moving now to the lower branch of the architecture, we notice how the dataset was entirely given as input to a hybrid model containing BERT and GPT, both pretrained. From this, the model's representations were obtained, concatenated, and fed into a dense layer. The loss function used in this case is the one mentioned in 1. Below, we report the representations in the latent space after minimizing the contrastive loss and the trend of the loss function during the training phase. The internal structure of the model is reported in the Appendix.

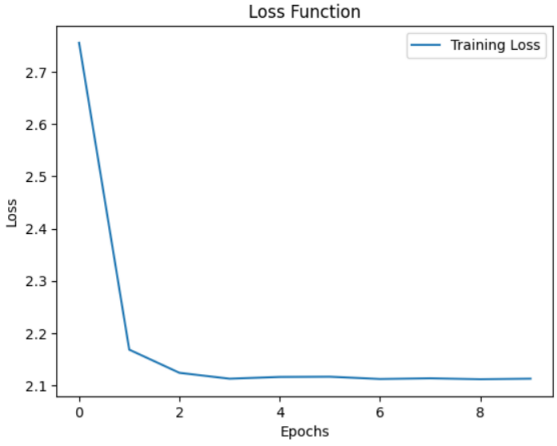

Figure 6: Plot of the loss function obtained by training the encoder on the training set.

Once the "contrasted" representations were obtained, it was possible to build a classifier that could determine which sentences were synthetic and which were human. Therefore, below, we report the metrics and the structure of the classifier, which includes, in addition to the encoder mentioned above, also a dense layer and a dropout layer.

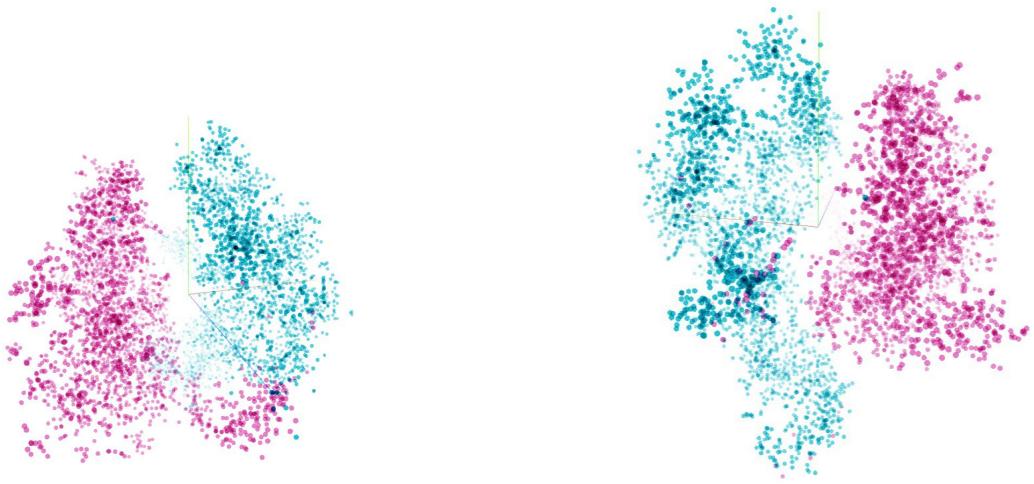

Figure 7: Representation of embeddings in the latent space from different perspectives.

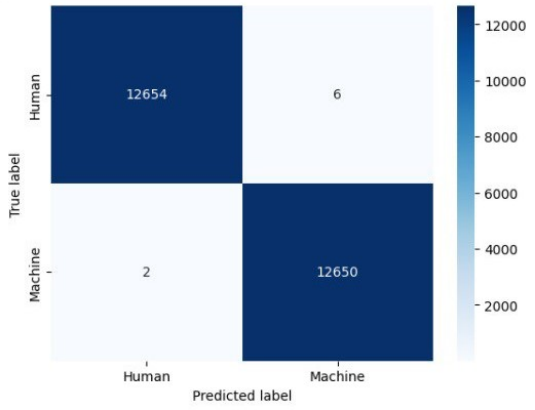

Figure 8: Confusion matrix for the training set generated using the classifier obtained by the pretrained models.

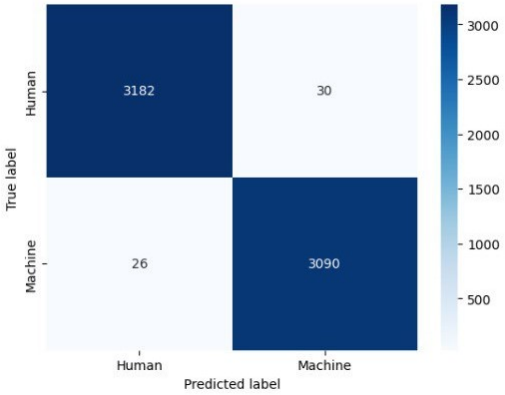

Figure 9: Confusion matrix for the validation set generated using the classifier obtained by the pretrained models.

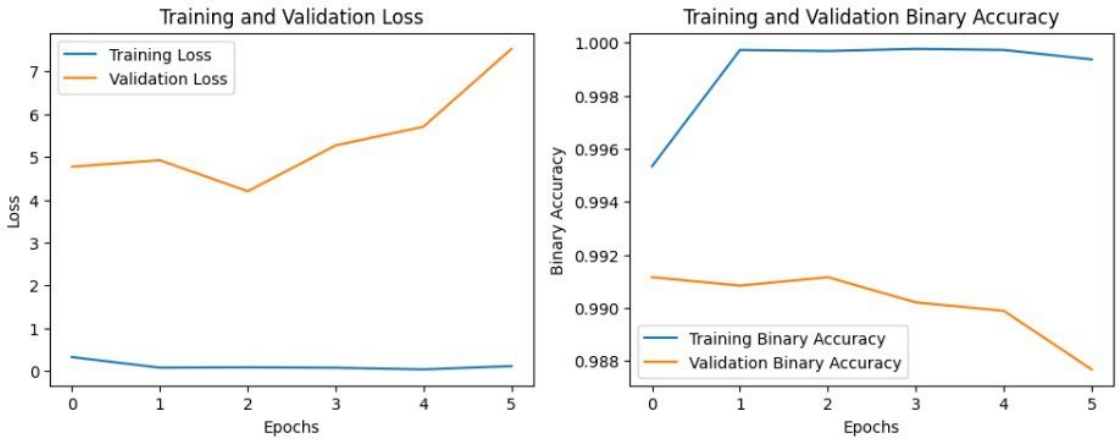

Figure 12: Plot of the loss functions obtained by training the encoder on the training set.

We can notice how the classifier derived from contrastive learning can lead to excellent classification: the ROC curves highlight how close it is to the ideal model, with an AUC of almost 0.99. However,

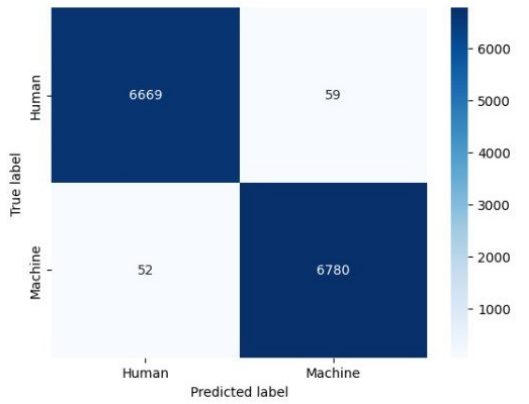

Figure 10: Confusion matrix for the test set generated using the classifier obtained by the pretrained models.

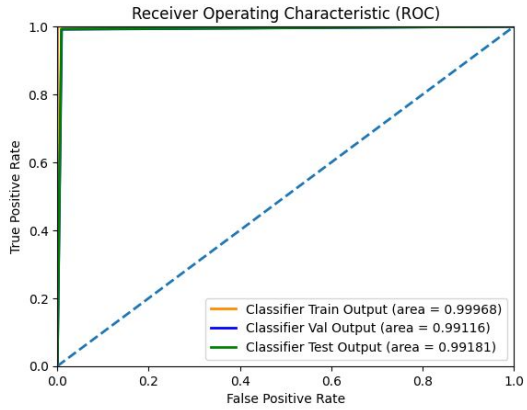

Figure 11: ROC curve calculated from the training, validation, and test sets of the classifier obtained by the pretrained models.

it's evident that such a high-performing model must hide an intrinsic overfitting, as evidenced by the trend, even within just a few epochs, of the validation loss compared to the training loss.

Finally, we analyze the metrics obtained from training the final model. As expected, its performance is well beyond expectations. However, as we will highlight in the next section, these already suggest the presence of overfitting.

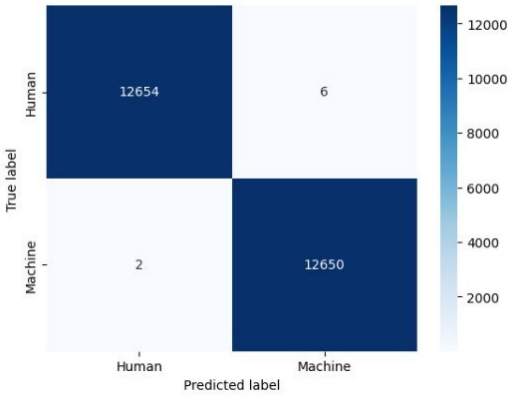

Figure 13: Confusion matrix for the training set generated using the final model.

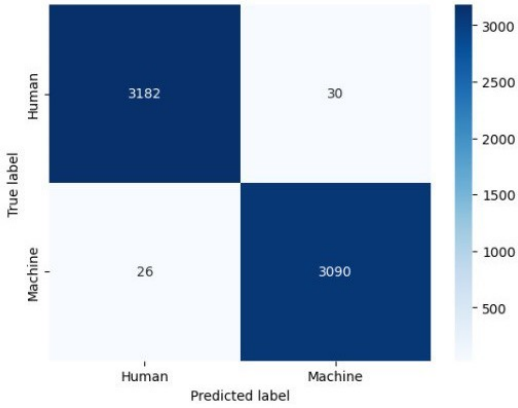

Figure 14: Confusion matrix for the validation set generated using the final model.

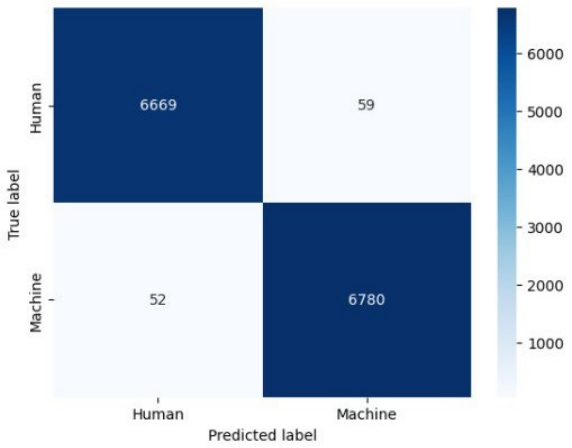

Figure 15: Confusion matrix for the test set generated using the final model.

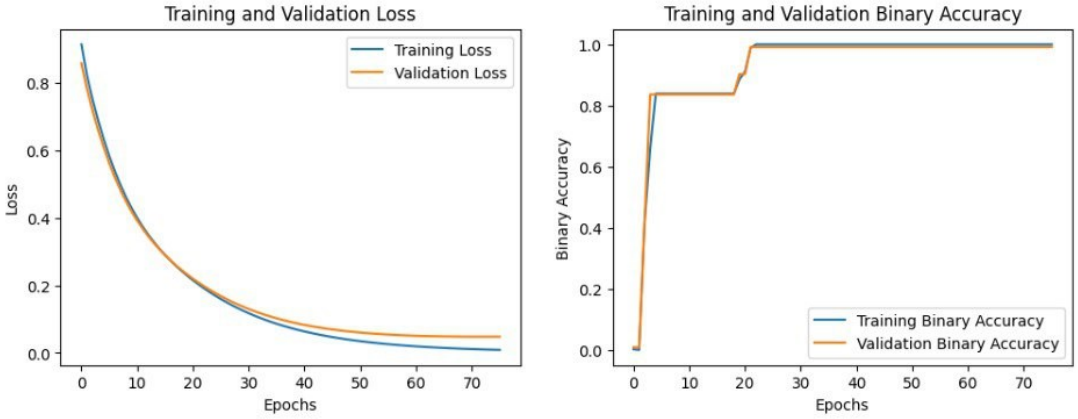

Figure 16: Plot of the loss functions obtained by training the final model on the training set.

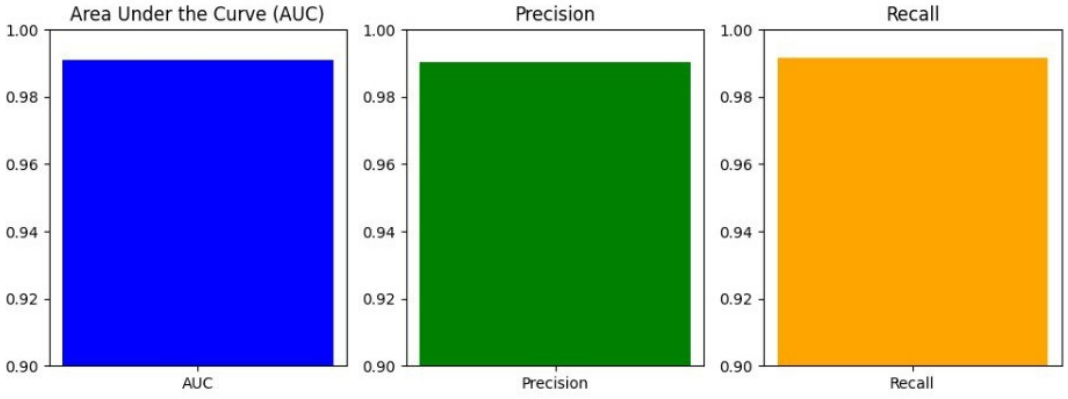

Figure 17: AUC, precision and recall metrics for the final model.

3 Results

3.1 Final Results

After submitting the code to the competition and obtaining the scores with which the performance of our model is evaluated by Kaggle, we can discuss the obtained results.

Although not completely evident from the obtained metrics, we can conclude that our model overfits. This can already be understood from the behavior of the classifier built using the GPT and Bert pre-trained models: the validation loss increased rather than decreased while the training loss decreased and stabilized, in the same way the accuracy of the validation set decreased while the training set one increased until it stabilized. One of the reasons why the model might initially appear to be not overfitting is the strong similarity of the texts within the dataset used. Since the data is very similar, the model is able to well classify the validation set and the test set built from the starting dataset, while, actually, with different data the performance could degrade. The impression is that the model is behaving well but its generalization capability can be improved.

The use of an ensemble in which such powerful methodologies and techniques are used, such as the pre-trained Bert and GPT models, contrastive learning and support vector machine, cannot fail to generate overfitting. This is predictable also given the small amount of data available and, in our case, the small amount of computational power.

3.2 Proof of Challenge Participation

Figure 18: Proof of participation in the competition and scores obtained.

4 Conclusions

In this section we specify some of the actions that could be carried out to improve the model and make it more performing.

4.1 Future developments

4.1.1 The sliding window

The use of sliding windows with pre-trained models like BERT may be necessary in certain contexts to handle text sequences longer than what the model can process in a single pass. Windows of appropriate sizes can be created and slid through the text to encompass the entire sequence. However, it is crucial to handle overlaps between the windows to ensure a continuous comprehension of the context. It

should be noted that the sliding window approach may pose challenges, such as the duplication of information in the overlaps between windows.

4.1.2 Replacing pretrained models

Alternatively to the sliding window, as mentioned above, it is possible to replace the simple BERT model with one of the Longformer models already available in online libraries (see HuggingFace [6]), always considering that a better implementation of the attention system almost always coincides with a greater computational requirement.

4.1.3 Reduce overfit

To reduce the overfit introduced into the final model, we could follow two main strategies:

- Fine-tune Bert and GPT pretrained models and some parameters used during model training and performance evaluation (for example, the choice of the probability threshold beyond which a prediction is considered to belong to class 1 or 0).
- Introduce regularization layers within the final model or increase the probability of drop out present in the classifier built with Bert and GPT.

References

[1] Explosion AI. *spaCy: Industrial-strength Natural Language Processing in Python*. URL: https://spacy.io/.

[2] Iz Beltagy, Matthew E. Peters, and Arman Cohan. *Longformer: The Long-Document Transformer*. 2020. arXiv: 2004.05150 [cs.CL].

[3] Ting Chen et al. *A Simple Framework for Contrastive Learning of Visual Representations*. 2020. arXiv: 2002.05709 [cs.LG].

[4] scikit-learn contributors. *scikit-learn: Machine Learning in Python*. URL: https://scikit-learn.org/stable/.

[5] Kadhim Hayawi, Sakib Shahriar, and Sujith Samuel Mathew. *The Imitation Game: Detecting Human and AI-Generated Texts in the Era of ChatGPT and BARD*. 2023. arXiv: 2307.12166 [cs.CL].

[6] *Hugging Face*. URL: https://huggingface.co/.

[7] Prannay Khosla et al. *Supervised Contrastive Learning*. 2021. arXiv: 2004.11362 [cs.LG].

[8] Jules King et al. *LLM - Detect AI Generated Text*. 2023. URL: https://kaggle.com/competitions/llm-detect-ai-generated-text.

[9] Darek Kłeczek. *daigt-v3-train-dataset*. 2023. URL: https://www.kaggle.com/datasets/thedrcat/daigt-v3-train-dataset/.

[10] Yinhan Liu et al. *RoBERTa: A Robustly Optimized BERT Pretraining Approach*. 2019. arXiv: 1907.11692 [cs.CL].

[11] Raghavendra Pappagari et al. *Hierarchical Transformers for Long Document Classification*. 2019. arXiv: 1910.10781 [cs.CL].

[12] Victor Sanh et al. *DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter*. 2020. arXiv: 1910.01108 [cs.CL].

[13] Rowan Zellers et al. *Defending Against Neural Fake News*. 2020. arXiv: 1905.12616 [cs.CL].

5 Appendix

5.1 Support Vector Machines

Support Vector Machines (SVMs) are a type of machine learning algorithm used for classification and regression problems. The main goal of SVMs in classification problems is to *try to identify the hyperplane that is equidistant from the closest instances of each class, called support vectors.*

The ability of SVMs to handle even non-linearly separable data is improved through the "kernel trick". This involves applying a nonlinear transformation to the data, projecting it into a higher-dimensional space where linear separation is possible.

SVMs can be extended to handle multi-class classification problems using strategies such as *"One-vs-All"* or *"One-vs-One"*. Furthermore, SVMs are equipped with a **regularization parameter** (C) that controls the trade-off between obtaining a larger margin and accurately classifying training instances. A higher value of C can lead to a narrower margin but higher accuracy on training. SVMs can be sensitive to outliers, as these instances can greatly influence the position of the hyperplane. Using appropriate regularization can help mitigate this impact.

In conclusion, SVMs are a powerful tool for tackling classification and regression problems, known for their effectiveness in large spaces and the ability to handle even complex data through kernel tricks.

5.2 Internal structure of encoder and classifier

```
Model: "model"
__________________________________________________________________________________________________
 Layer (type)                Output Shape              Param #   Connected to
==================================================================================================
 bert_input_word_ids (Input  [(None, 512)]             0         []
 Layer)

 gpt_input_word_ids (InputL  [(None, 512)]             0         []
 ayer)

 bert_patterns_recognizer (  TFBaseModelOutputWithPooli  108310272  ['bert_input_word_ids[0][0]']
 TFBertModel)                ngAndCrossAttentions(last_
                             hidden_state=(None, 512, 7
                             68),
                              pooler_output=(None, 768)
                             , past_key_values=None, hi
                             dden_states=None, attentio
                             ns=None, cross_attentions=
                             None)

 gpt_patterns_recognizer (T  TFBaseModelOutputWithPastA  124439808  ['gpt_input_word_ids[0][0]']
 FGPT2Model)                 ndCrossAttentions(last_hid
                             den_state=(None, 512, 768)
                             , past_key_values=((2, Non
                             e, 12, 512, 64),
                              (2, None, 12, 512, 64),
                              (2, None, 12, 512, 64),
                              (2, None, 12, 512, 64),
                              (2, None, 12, 512, 64),
                              (2, None, 12, 512, 64),
                              (2, None, 12, 512, 64),
                              (2, None, 12, 512, 64),
                              (2, None, 12, 512, 64),
                              (2, None, 12, 512, 64),
                              (2, None, 12, 512, 64),
                              (2, None, 12, 512, 64)),
                              hidden_states=None, atten
                             tions=None, cross_attentio
                             ns=None)

 concatenate (Concatenate)   (None, 512, 1536)         0         ['bert_patterns_recognizer[0][
                                                                 0]',
                                                                  'gpt_patterns_recognizer[0][0
                                                                 ]']

==================================================================================================
Total params: 232750080 (887.87 MB)
Trainable params: 232750080 (887.87 MB)
Non-trainable params: 0 (0.00 Byte)
__________________________________________________________________________________________________
Model: "encoder-with-projection"
__________________________________________________________________________________________________
 Layer (type)                Output Shape              Param #   Connected to
==================================================================================================
 bert_input_word_ids (Input  [(None, 512)]             0         []
 Layer)

 gpt_input_word_ids (InputL  [(None, 512)]             0         []
 ayer)

 model (Functional)          (None, 512, 1536)         232750080 ['bert_input_word_ids[0][0]',
                                                                  'gpt_input_word_ids[0][0]']

 flatten (Flatten)           (None, 786432)            0         ['model[0][0]']

 dense (Dense)               (None, 64)                50331712  ['flatten[0][0]']

==================================================================================================
Total params: 283081792 (1.05 GB)
Trainable params: 283081792 (1.05 GB)
Non-trainable params: 0 (0.00 Byte)
```

Figure 19: Internal structure and amount of parameters of the encoder model.

```
Model: "classifier-gpt-bert-representations"
_________________________________________________________________________________________________
 Layer (type)                Output Shape              Param #   Connected to
=================================================================================================
 bert_input_word_ids (Input  [(None, 512)]             0         []
 Layer)

 gpt_input_word_ids (InputL  [(None, 512)]             0         []
 ayer)

 model (Functional)          (None, 512, 1536)         2327500   ['bert_input_word_ids[0][0]',
                                                       80         'gpt_input_word_ids[0][0]']

 flatten_1 (Flatten)         (None, 786432)            0         ['model[1][0]']

 dropout_74 (Dropout)        (None, 786432)            0         ['flatten_1[0][0]']

 dense_1 (Dense)             (None, 1)                 786433    ['dropout_74[0][0]']

=================================================================================================
Total params: 233536513 (890.87 MB)
Trainable params: 786433 (3.00 MB)
Non-trainable params: 232750080 (887.87 MB)
```

Figure 20: Internal structure and amount of parameters of the classifier model.