

UNIVERSITÀ DELLA CALABRIA

DIPARTIMENTO DI MATEMATICA E INFORMATICA



Deep Learning Project Report

LLM - Detect AI Generated Text

Giovanni Beraldi 242497

Gabriel Tripodi 242784

Table of contents

| | |
|---|-----------|
| 1. Introduction | 3 |
| 2. Detect AI Generated Text | 4 |
| 2.1 Data Description | 4 |
| 2.2 Data Preprocessing | 5 |
| 2.2.1 Text standardization | 6 |
| 2.2.2 Text tokenization | 6 |
| 2.2.3 Vocabulary indexing | 7 |
| 2.2.4 Text encoding | 7 |
| 2.3 Model Architecture | 7 |
| 2.3.1 TF-IDF Classifier | 8 |
| 2.3.2 Attentioned CNN-BiLSTM | 9 |
| 2.3.3 Final Model | 12 |
| 2.4 Training and Experiments | 12 |
| 2.4.1 Data Generator | 13 |
| 2.4.2 Hyperparameters Tuning | 13 |
| 2.4.3 Models Training | 13 |
| 3. Results | 16 |
| 3.1 Final Results | 16 |
| 3.1.1 Attentioned CNN-BiLSTM evaluation | 16 |
| 3.1.2 TF-IDF Classifier evaluation | 17 |
| 3.1.3 Final Model evaluation | 17 |
| 3.2 Prof of Challenge Participation | 18 |
| 4. Conclusions | 18 |

1. Introduction

For this project we participated in a Kaggle competition “*LLM - Detect AI Generated Text*” that challenges participants to develop a deep learning model that can accurately detect whether an essay was written by a student or an LLM. The competition dataset comprises a mix of student-written essays and essays generated by a variety of LLMs. The significance of this task extends beyond the competition, as it addresses the intricate task of authorship identification in written content.

Our primary aim is to craft a robust deep learning model that can navigate the nuances of writing styles, uncovering the subtle distinctions between human and machine-generated essays. This endeavor not only pushes the boundaries of artificial intelligence but also holds practical implications in educational settings, where the ability to differentiate between student and LLM compositions can be of paramount importance.

Throughout this report, we delve into the intricacies of our model development process, exploring the methodologies, challenges encountered, and the ultimate result achieved in our pursuit of building an essay detection system.

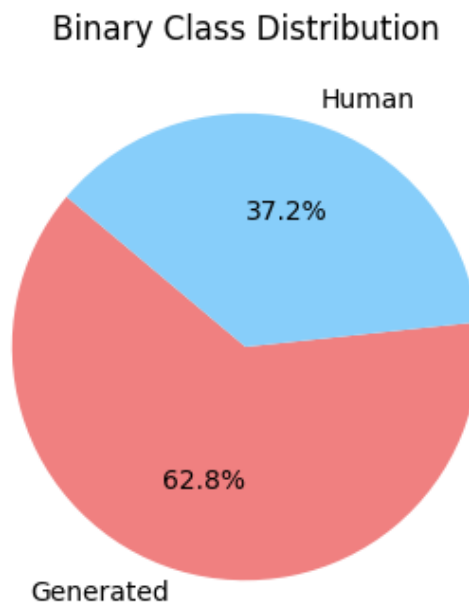
2. Detect AI Generated Text

In this section, we take a closer look at the heart of our project: the dataset and methods we employed to develop our deep learning model.

First, we will analyze the data used to train and evaluate our model. Then we will reveal the magic behind the scenes: how we preprocessed the data, the architecture of our model, and the trials and tribulations of training and testing.

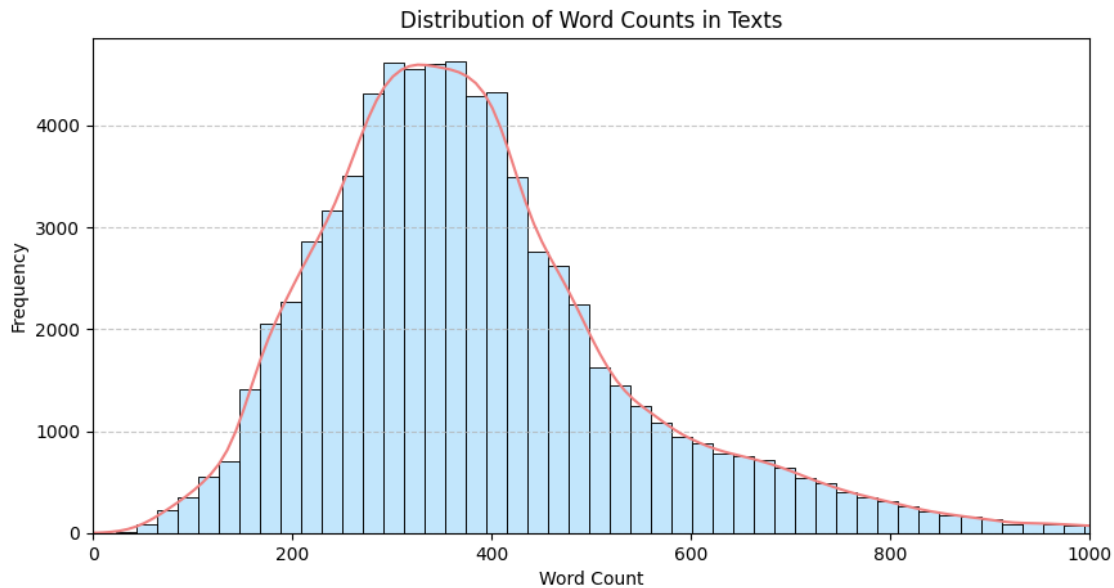
2.1 Data Description

To train and evaluate the model we used the [DAIGT-V4-TRAIN-DATASET \(kaggle.com\)](https://www.kaggle.com/datasets/daigt-v4-train-dataset) dataset. This dataset consists of **73.573 text samples**, written by both humans and LLMs. The distribution of the two classes is as follows:



As we can see from the figure, distribution of the two classes is not perfectly balanced. In fact, the dataset consists of **46.203 LMM generated text** and **27.370 human-written text**. We still decided not to balance the dataset by performing data augmentation (for texts written by humans) or removing samples (for texts generated by LLMs).

To figure out the right length to use during the preprocessing phase, we performed a statistical analysis to understand the distribution of texts according to the word count of the texts itself:



As can be seen from the figure, most texts have an approximate word count between 300 and 400 words.

Although the dataset consisted of multiple columns, we used only the column related to text and class label to train and evaluate our model.

2.2 Data Preprocessing

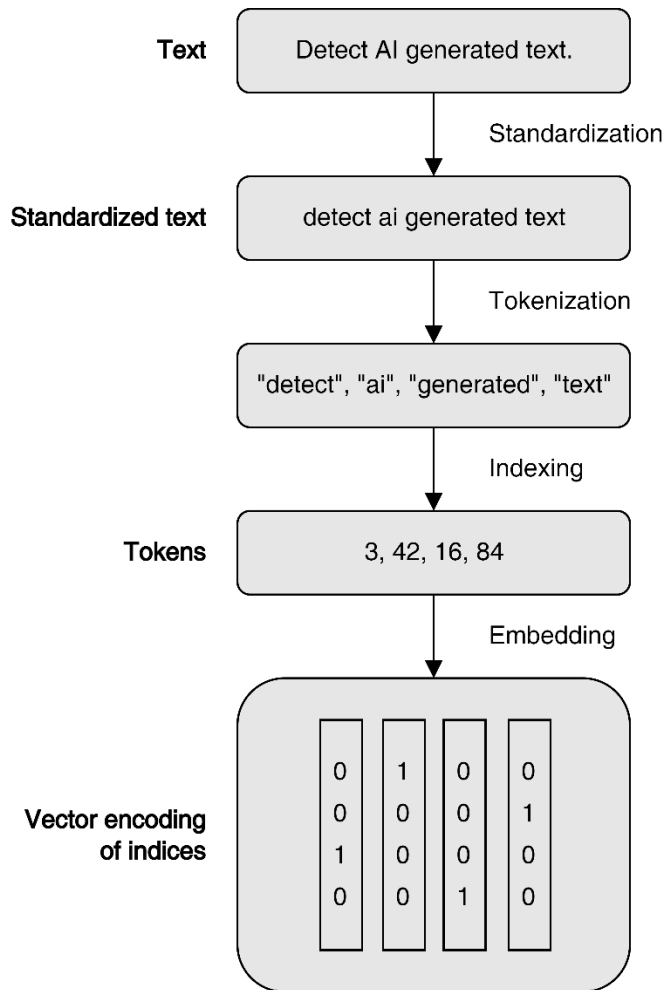
To transform the text into numerical tensors, we used two different approaches to leverage their relative advantages.

The first is the ***bag-of-word approach***, according to which texts are represented as an **unordered set** (“a bag”) **of words** (tokens). However, to recover some local order information we looked at groups of two consecutive token (**bigram**), using the **TF-IDF normalization** for the final encoding.

The second is the ***sequence model approach***: we expose our model to raw word sequences, representing the input text as a vector sequences (2D tensor).

Even though they are different approaches, part of the preprocessing is the same for both:

1. **Text standardization** to make it easier to process
2. **Text splitting** into token representing words or group of word (***tokenization***)
3. **Token conversion into numerical vector**, indexing all tokens present in the dataset



2.2.1 Text standardization

Text standardization is useful to erase encoding differences. We used, both for the bag of words and the sequence approaches, the `lower_and_strip_punctuation` schema, that convert the text to lowercase and remove punctuation characters.

2.2.2 Text tokenization

Once we have standardized the text, we break it up into tokens, using different methods for the bag of words and the sequence approaches:

- For the bag of word approach, we implemented a ***bigram tokenization***, for which tokens are group of 2 consecutive word.
- For the sequence approach, we implemented the ***word-level tokenization***, for which tokens are space-separated (or punctuation separated) substring.

2.2.3 Vocabulary indexing

Once the text is splitted into tokens, we encode each token into a numerical representation, building an index of all terms found in the training data, i.e. the **vocabulary**, and assigning a unique integer to each entry in the vocabulary.

We restrict the vocabulary to only the top **20.000 most common word** found in the training data, ignoring rare terms that have almost no information content.

2.2.4 Text encoding

At this point, we have to encode each text in a numeric form and we have to differentiate between the two approaches. In fact:

- For the bag of word approach, we finally encode the text using the ***TF-IDF encoding***, that for each token take into consideration its frequency in the current text and its frequency in all the text in the training data: terms that appear in almost every text in the dataset are not particularly informative, while term that appears only in a small subset of all text are very distinctive.

The output of the text encoding is thus a vector of 20.000 element, each of which represents the TF-IDF of a token present in the text. The main advantage of this encoding is that we can represent an entire text as a single vector.

- For the sequence approach, we encode the text as an **ordered sequence of word** represented by integer (vocabulary indices). We truncate the text after the first 350 word, so the output of the text encoding is a vector of 350 element, each of which represents the index of a token present in the text. This is a reasonable choice, since the average text length is between 300 and 400 words.

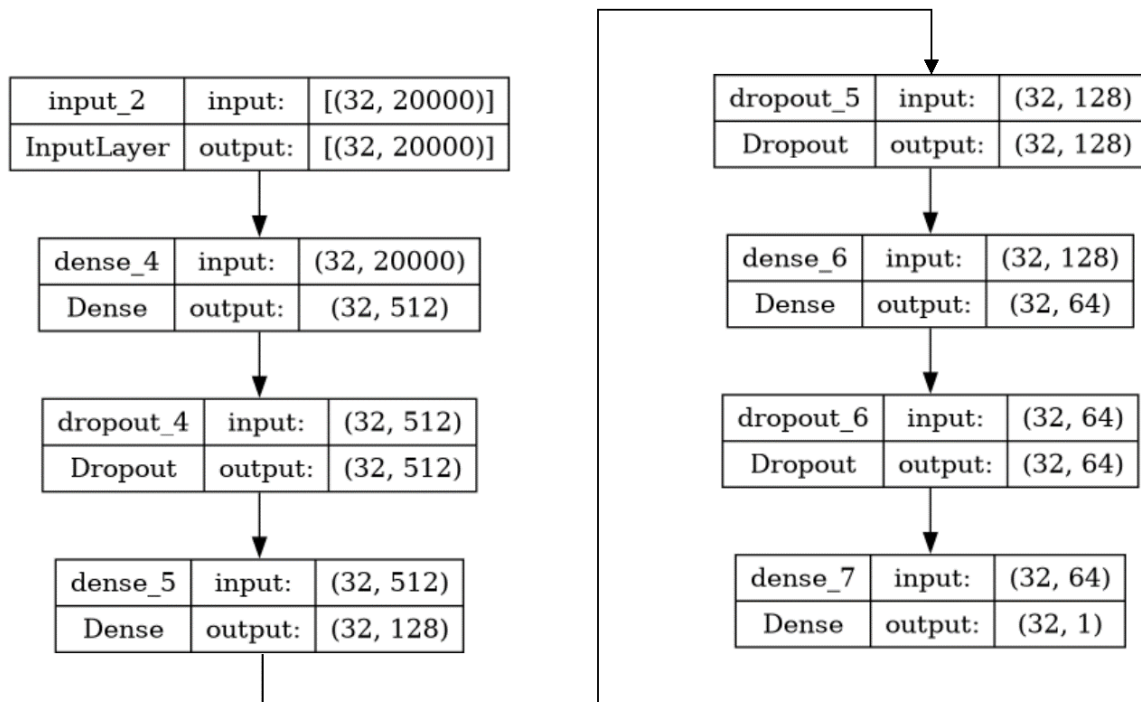
For both TF-IDF and vector of integer text encoding we use the **keras TextVectorization layer**.

2.3 Model Architecture

To take advantage of the two approaches used for text encoding, we decided to opt for the creation of two distinct models, namely a simple **TF-IDF Classifier**, that is a fully-connected feed-forward neural network which receives as input the text encoded with TF-IDF encoding, and an **Attentioned CNN-BiLSTM**, which receives as input the text encoded as a vector of integers. The two models are ultimately merged into a final model, which has been trained to perform weighted averaging of the predictions of the two individual models.

2.3.1 TF-IDF Classifier

This first simple model receives input batches of 20.000-dimensional vector and consists of a list of **keras Dense layers**, separated by **Dropout layers** for better generalization, and return in output the probability that the input essay was huma-written or AI-generated.

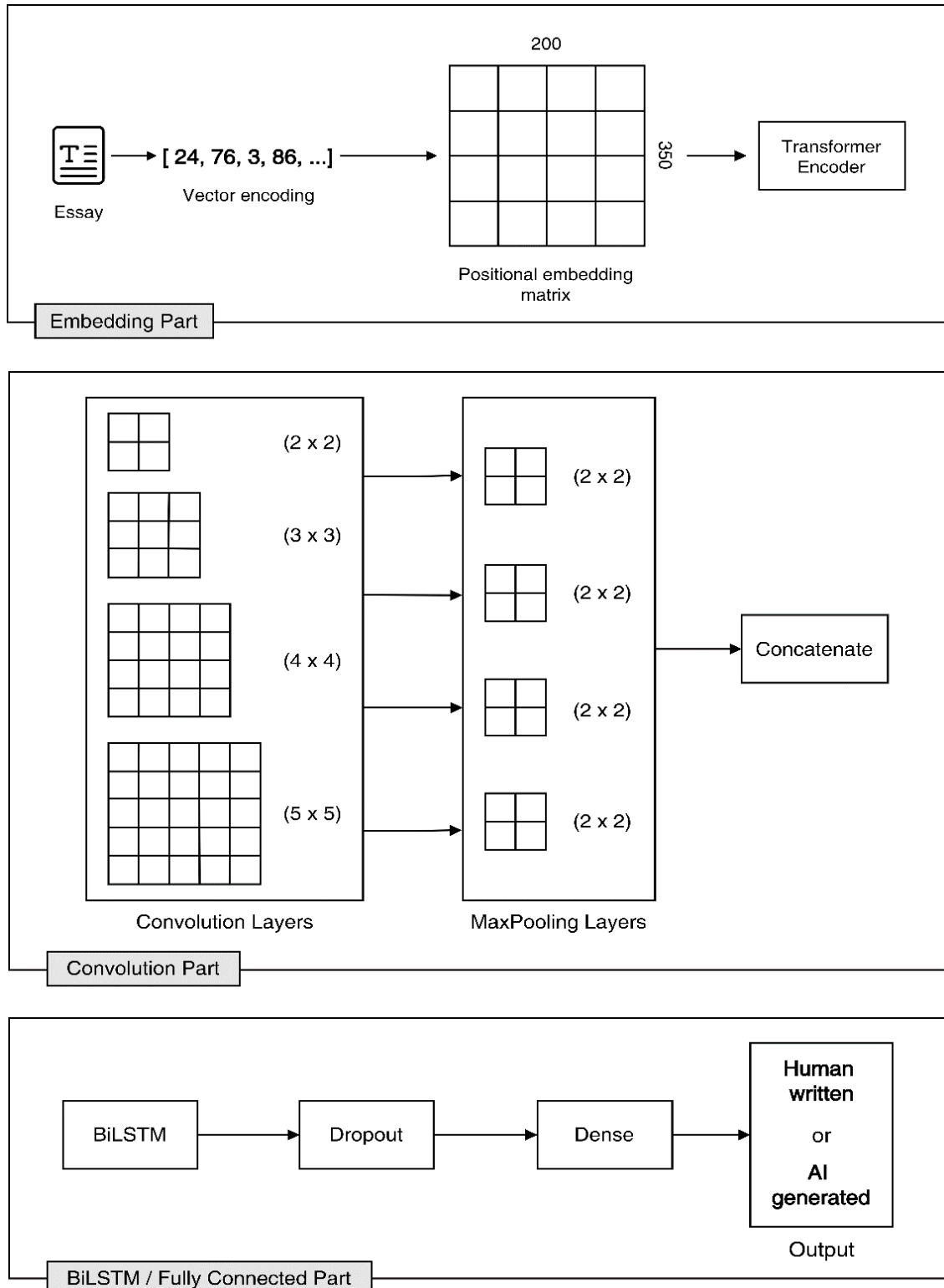


Each Dense layer uses a ***Rectified Linear Unit (ReLU) function***, unlike the last layer which uses the ***Sigmoid activation function*** to return a probability distribution in output.

Although trivial, this model is very useful for **identifying statistical regularities in the input data**, and coupled with the second model has proven its effectiveness by increasing the reliability of the final predictions on the test set.

2.3.2 Attentioned CNN-BiLSTM

This second model, differently from the first one, is a *sequence model*: it was exposed to raw word sequences. The model receives as input a vector of 350 integer indices, representing the tokens (words) present in the essay in the right order, and return in output the probability that the input essay was huma-written or AI-generated.



The Positional Embedding Layer

The first layer of the model is a custom layer that maps each integer in the input (i.e. each word) to a vector, obtaining thus a vector sequences, taking into account also **the order in which the words appear in the text**.

First of all, we compute the word embedding. Instead of learn a new embedding space from the training set, we use the *GloVe pretrained word embeddings* (due to the few number of examples in the training set). To do this, we firstly load the GloVe word embedding to build an *embedding matrix* that subsequently we loaded into the keras *Embedding layer*. This matrix has shape of (max_tokens, embedding_dim), where each entry i contains the embedding-dim-dimensional vector of the word of index i in the vocabulary (built during tokenization). Since we set the number of maximum tokens to 20.000 and used the 200-dimensional pretrained GloVe embeddings, the embedding matrix has shape (20.000, 200). This first embedding layer takes as input a vector of integers of shape (sequence_length) and return a 2D floating-point tensor of shape (sequence_length, embedding_dim), representing the embedding of the text in input. In this case, due to the text encoding, every sequence has length 350, so the output of the layer has shape (350, 200).

At this point, to give the model access to word-order information, **we add the word's position in the sentence to each word embedding**. The final word embedding will have in this way two components: the usual word embedding vector (computed with the previous Embedding layer) and a position vector, which represent the position of the word in the sentence. To do this we encode every integer between $[1, \text{sequence_length}]$, that represent word position in a sentence, to a vector containing values in the range $[-1, 1]$, using the following formula:

$$\begin{aligned} \text{pos}(k, 2 \cdot i) &= \sin \frac{k}{n^{2i/d}} \\ \text{pos}(k, 2 \cdot i + 1) &= \cos \frac{k}{n^{2i/d}} \end{aligned}$$

Computing this for each position k , we obtain a *positional embedding matrix* of dimension (sequence_length, embedding_dim), where each entry i contains the embedding-dim-dimensional vector of the words position i . We subsequently loaded this matrix into another keras *Embedding layer*. This second embedding layer takes as input a vector of integers representing the position of the words of shape (sequence_length) and return a 2D floating-point tensor of shape (sequence_length, embedding_dim), representing the embedding of each position. Also in this case, due to the text encoding, every sequence has length 350, so the output of the layer has shape (350, 200).

The final Positional Embedding layer first calculates the embedding of the sentence using the first Embedding layer and the embedding of the positions relative to the words in the text using the second Embedding layer, and then adds the two embeddings together to obtain the final one.

The Transformer Encoder layer

After the positional embedding of the input essay through the custom Positional Embedding Layer, to modulate the representation of every token in the essay we use a *self-attention mechanism*, that produce *context-aware* token representations, using the representation of related tokens in the sequence. In this way we produce a smart embedding space that provide a different vector representation for a word, depending on the other words surrounding it.

For doing this, we use the *keras Transformer Encoder Layer*, that receives in inputs the positional embedding of the essay and return in output a more useful representation of it: since we are doing text classification, to enrich each token with context from the whole text, we compared the text to itself (the query, key, value inputs to the Multi Head Attention layer are the same), thus focusing on the relationship between pairs of tokens in the sentence.

The Convolution Part

After passing the input through the Positional embedding layer and the Transformer Encoder layer, we obtain a useful representation of the input, that capture semantic information about words and their relationships, which can enhance the overall network's ability to generalize to new, unseen text data.

Convolutional Neural Networks are appropriate for extracting local features from sequence data. In the contest of text classification, these features are represented by n-grams: sentences of n consecutive words, pairs of words, or even single characters. This makes CNNs able to identify important patterns and relationships within short text segments.

For this reason, we use the text embedding as input to a series of 2D Convolution Layers, that can *capture important word sequences or n-grams*, which contribute to the overall meaning of the input text. In particular, this is achieved through the application of convolutional filters of different size across the input text. This filters serve as *n-gram detectors*: each filter searches for a specific class of n-grams and assigns them high scores. Specifically, four filters of different size are applied:

- The first layer applies *bigrams filters* of size 2;
- The second layer applies *trigram filters* of size 3;
- The third layer applies *four-gram filters* of size 4;
- The fourth layer applies *five-gram filters* of size 5.

After each filter, a Max Pooling layer is applied to update and reduce the size of the data. Each convolution layer applies the *Rectified Linear Unit* (ReLU) *function*, introducing non-linearity to the network.

Since CNNs are inherently translation invariant, meaning they can detect patterns regardless of their position in the input, in text classification context this property allows CNNs to identify relevant features regardless of where they appear in the text, improving its ability to better generalize.

The Bi-LSTM / Fully Connected Part

After the Convolution part, the results of all Max Pooling layers are concatenated and reshaped to produce a single fixed size output, used as input to a Bidirectional Long Short-Term Memory network (BiLSTM), which filter the information using its three gates.

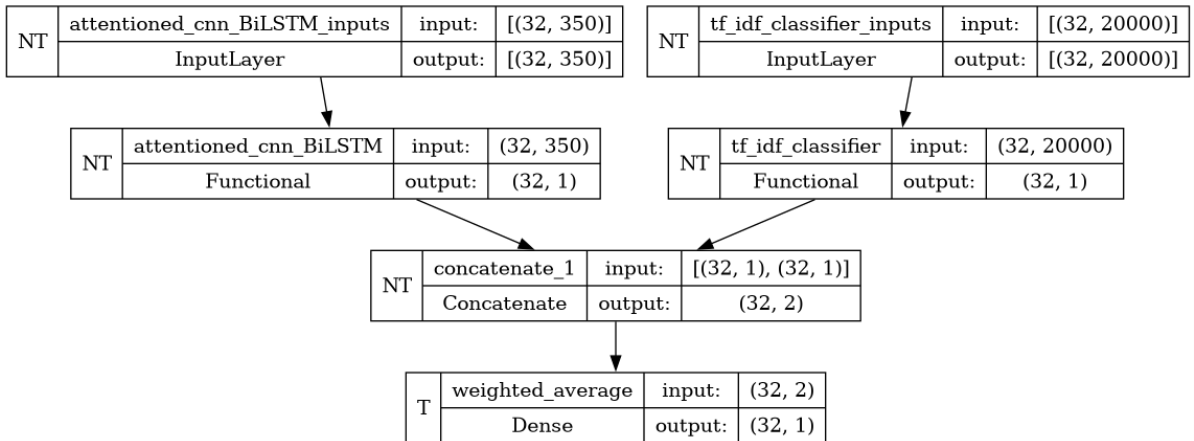
We used the features extracted using CNNs as input to a BiLSTMs because it processes input sequences in both forward and backward directions simultaneously, allows it to grasp how words earlier in the sentence influence the meaning of later ones, capturing long-range contextual dependencies. This ability is crucial for understanding the context and meaning of text.

The output of this step is the input of the last series of fully connected layers. This Dense layers aggregates and integrates the features extracted by the preceding layers of the network. This allows the model to combine information from different parts of the input text and learn higher-level representations that are more discriminative for the classification task.

In particular, the last Dense layer has a single unit and uses the **Sigmoid activation function** to return a probability distribution in output, used to assign classes to essay.

2.3.3 Final Model

The simple TF-IDF Classifier and the Attentioned CNN-BiLSTM models are ultimately merged into a final model. In particular, the probability returned in output by the two models are passed as input to a Dense layer with a single unit, which has been trained to perform weighted averaging of the predictions of the two individual models.



2.4 Training and Experiments

To train and evaluate our models, we firstly split the available data into **training, validation and test set**, using a simple **holdout** method. In particular, we use the 80% of the dataset as training set and the remains 20% as test set: we trained the models on the training set and evaluate it on the test set. The 20% of the training set was finally used to create the validation set.

2.4.1 Data generator

Since for hyperparameter tuning, training, and model evaluation it is necessary to encode the text as a vector of ordered integers (for the Attentioned CNN-BiLSTM) and utilizing TF-IDF encoding (for the TF-IDF Classifier), we have utilized *data generators* that return only the necessary encoded batch of data during the hyperparameter tuning, training, and evaluation phases. This was necessary because the encoding of the entire dataset did not fit completely into the RAM memory of the Kaggle kernel.

2.4.2 Hyperparameters tuning

The reason why we also created a validation set is that we performed *hyperparameters tuning* to choose the right hyperparameters of our models before the training. Since for doing this tuning we use a feedback signal the performance of the models on the validation data, we can't use the test set itself (used for the final evaluation) as validation data: since this is a form of learning, using the test set would cause information leaks (the model start overfitting the test set, making the measure of generalization based on the test set flawed).

To explore the space of the possible decision automatically we used the *KenarTuner* library and in particular the *Hyperband tuner*.

The search space of the hyperparameter tuning for the TD-IDF Classifier model has been determined by the following set of choices:

- Number of units of the first dense layer (value between 128 and 512, step of 128)
- Number of units of the second dense layer (value between 64 and 128, step of 64)
- Number of units of the third dense layer (value between 32 and 64, step of 32)

Instead, for the Attentioned CNN-BiLSTM, the search space has been determined by the following set of choices:

- Number of filters of the four convolution layers (value between 64 and 128, step of 32)
- Dimension of the LSTM layer (value between 128 and 512, step of 128)
- Number of units of the first FFNN dense layer (value between 128 and 256, step of 128)

For both models, the metric that the tuner tried to optimize was the *validation accuracy*, since the goal of the search was to find a model that generalize.

2.4.3 Models Training

Once the hyperparameter search has been completed, we queried the best hyperparameter configurations to the tuner, which we used to create high-performing models that we retrain. For retraining the models we included the validation data, used during the hyperparameter search, as part of the training data (so we used the original training set), thus evaluating performance on the test set after and during the training.

For the training of each model, we used a *batch size of 32 samples* and the following *keras callbacks*:

- **Early stopping:** to interrupt the training when the validation loss is no longer improving for 3 epochs

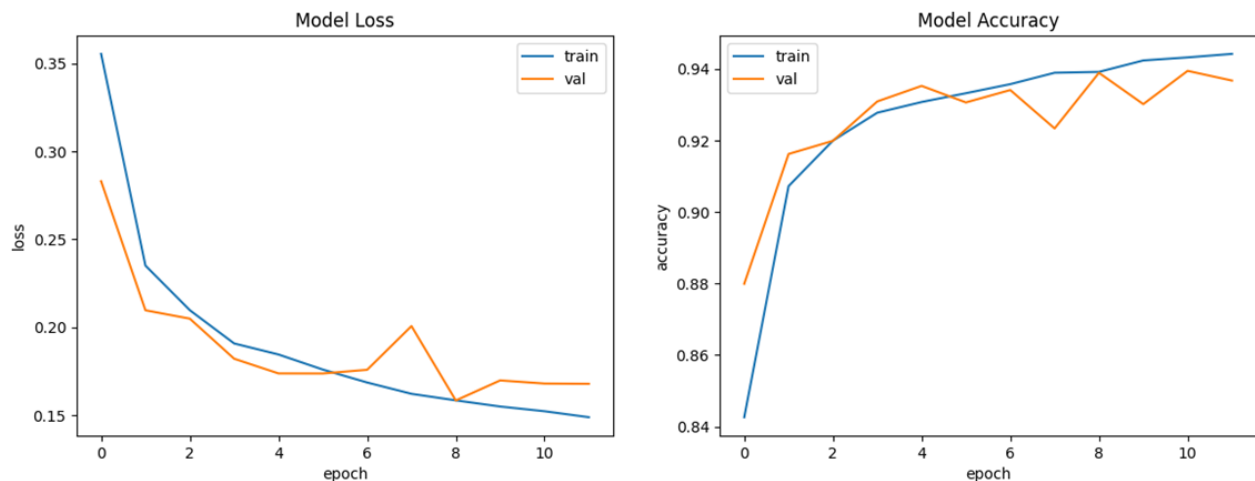
- **Model checkpoint:** to continually save the best model that achieved the best performance at the end of each epoch
- **CSV logger:** to streams epoch results to a CSV file. We after used this CSV file to plot the training metrics

Finally, for all model, we have chosen the following loss function (the quantity that will be minimized during the training) and metric (the measure to monitor during validation):

- **Binary cross-entropy:** loss that computes the cross-entropy loss between true labels and predicted labels.
- **Accuracy:** metric that calculates how often predictions equal labels

Attentioned CNN-BiLSTM training

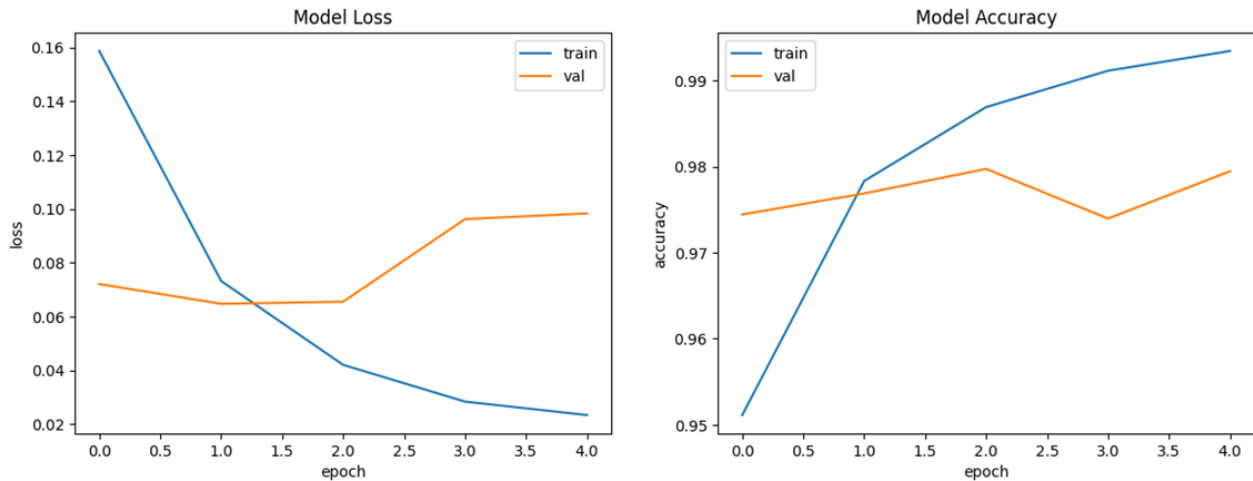
For training the Attentioned CNN-BiLSTM model, we have chosen the adam optimizer and set the maximum number of epochs to 20. The training and validation loss as well as the training and validation accuracy are reported in the following plots:



As can be seen of plots, around epoch number 8 the model starts to *overfit* the training data, so the Early stopping callbacks causes the end of the training.

TD-IDF Classifier training

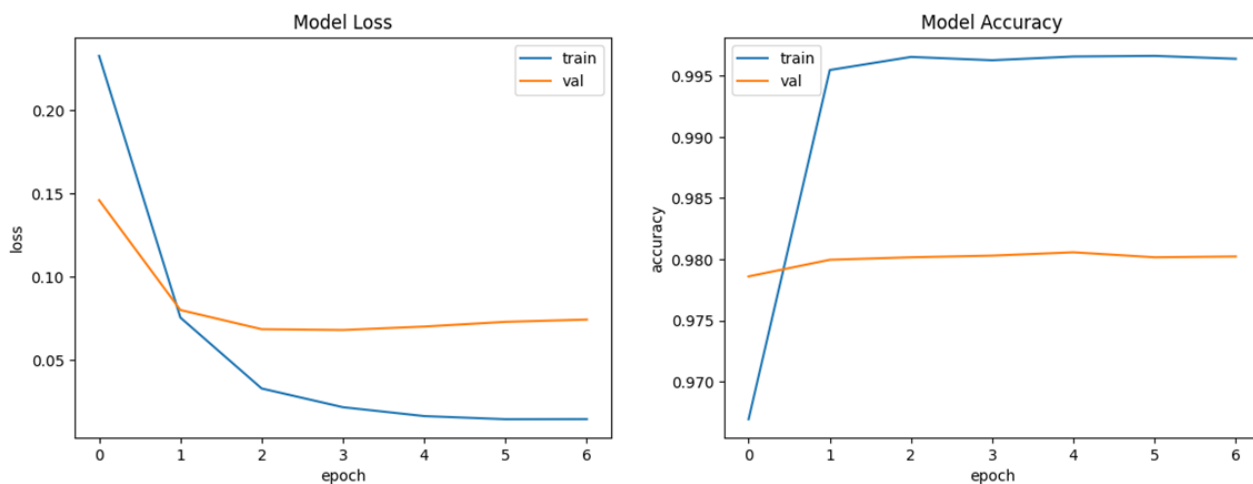
For training the TD-IDF Classifier model, we used the optimizer suggested by the tuner and also in this case we set the maximum number of epochs to 20. The training and validation loss as well as the training and validation accuracy are reported in the following plots:



As can be seen of plots, in this case the **overfitting** starts in the first few epochs, so the Early stopping callbacks causes almost immediately the end of the training. A possible solution might be to change the hyperparameters suggested by the tuner by decreasing the capacity of the model.

Final Model training

Since the Final Model has **only 3 trainable parameters** (the weights and the bias of the one unit of the final Dense layer), the latter was trained for a few epochs only to try to learn how to perform a weighted average of the predictions of the two main models. The training and validation loss as well as the training and validation accuracy are reported in the following plots:



As expected, both loss and accuracy have very good values already at the beginning of the training, since much of the Final Model is made up of already trained models.

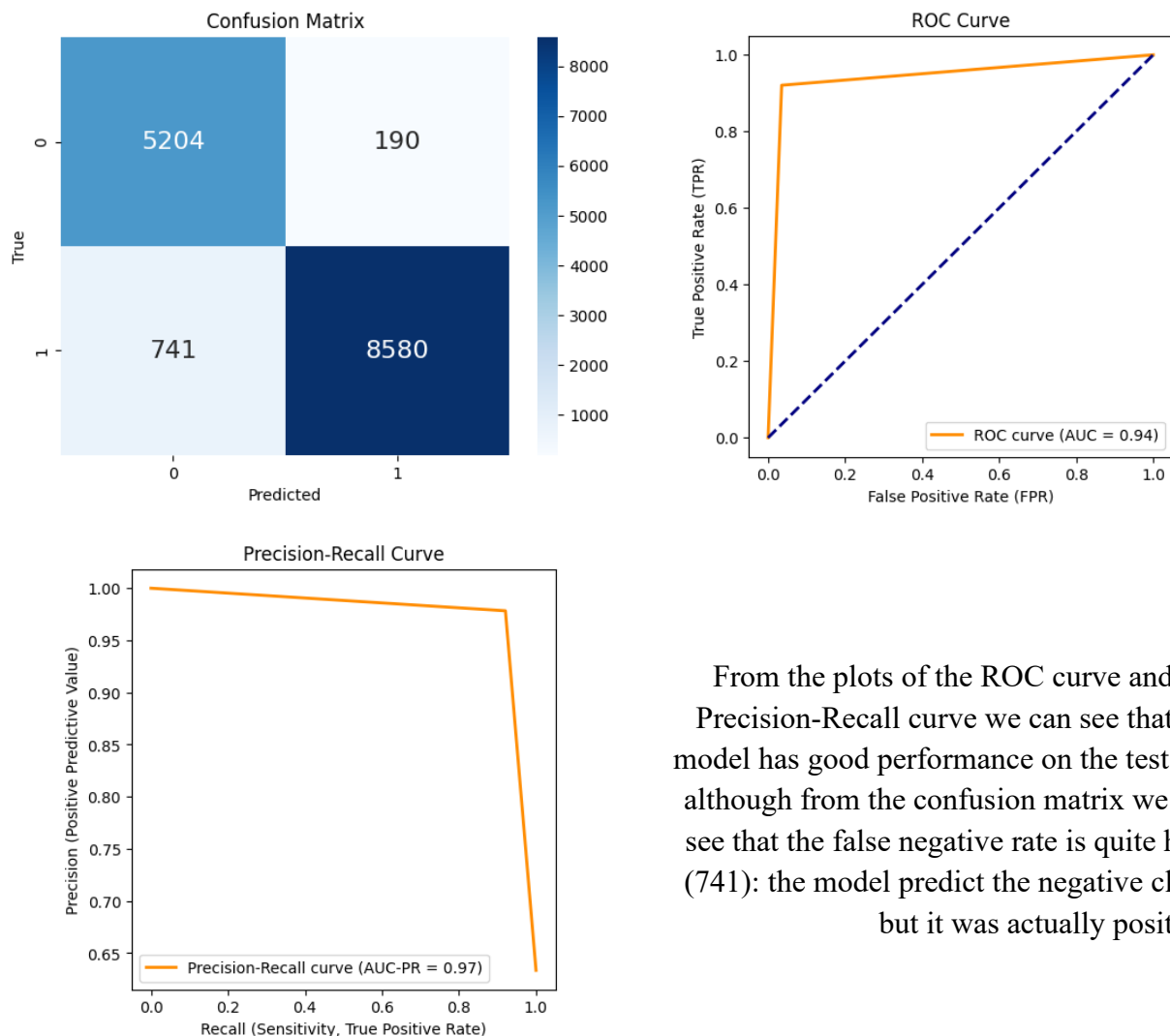
3. Results

In this section, we take a look at the results of our project, considering the performance on our test set and on the competition test set.

3.1 Final Results

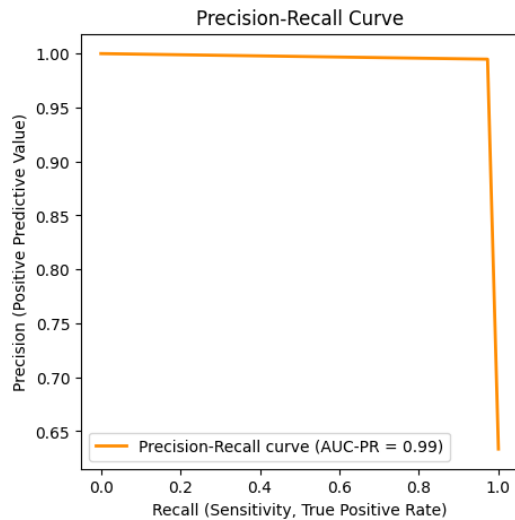
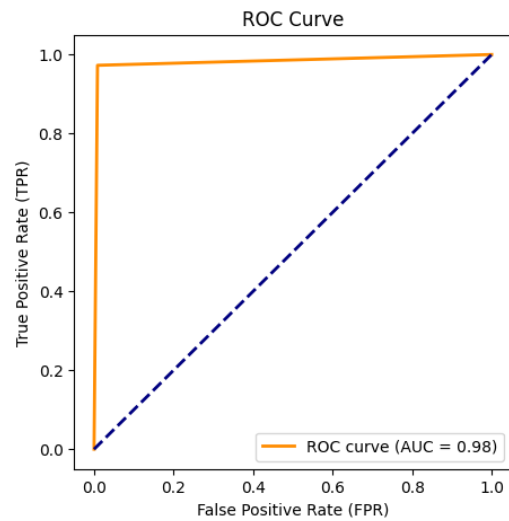
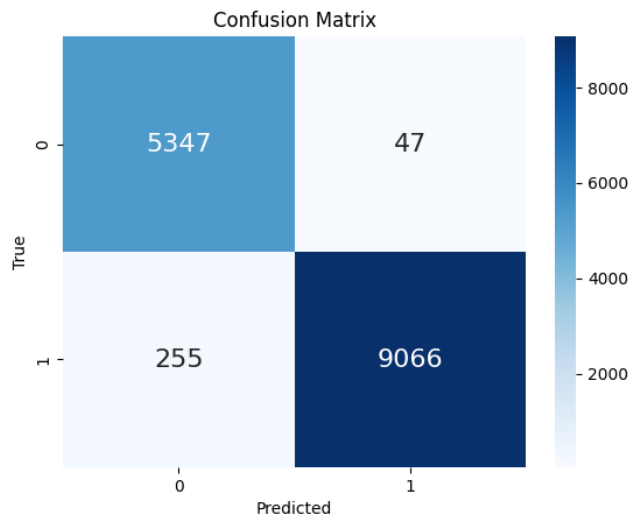
To check if the models have a good performance on data point that the models have never encountered before, we produced some insight for each model, evaluating them on our test set (20% of the original dataset).

3.1.1 Attentioned CNN-BiLSTM evaluation



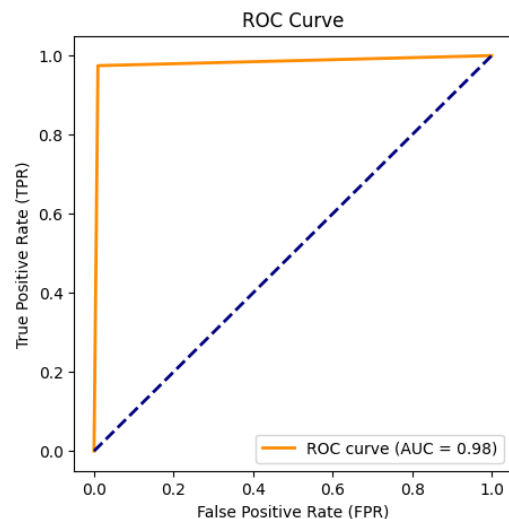
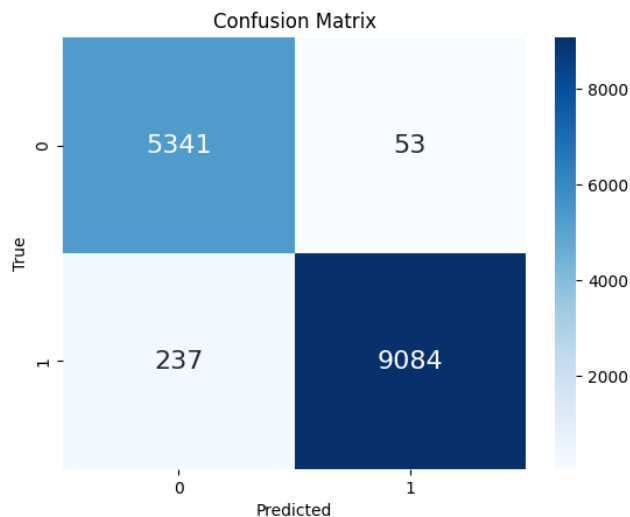
From the plots of the ROC curve and the Precision-Recall curve we can see that the model has good performance on the test set, although from the confusion matrix we can see that the false negative rate is quite high (741): the model predict the negative class, but it was actually positive.

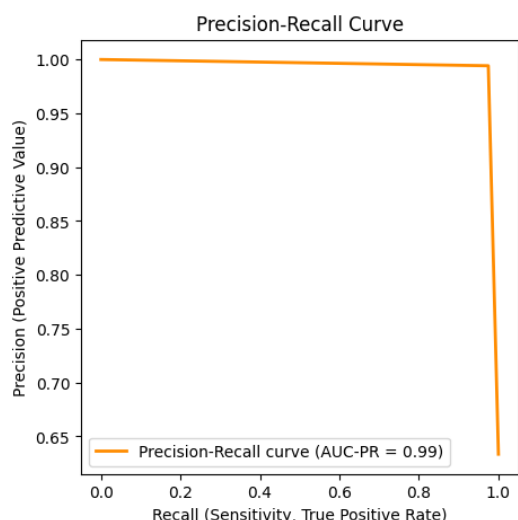
3.1.2 TF-IDF Classifier evaluation



From the plots of the ROC curve and the Precision-Recall curve we can see that the model have better performance w.r.t. the previous one, but also in this case the false negative rate predominates.

3.1.3 Final Model evaluation







From the plots and the confusion matrix we can see that the Final Model is a good ensemble of the two main models and combines the strengths of both models, particularly in how the false positives and false negatives decrease.

3.2 Prof of Challenge Participation

The following are the result on the public and private test set of the Kaggle competition. Although the results are not so great on the private test set, we are satisfied with the result obtained on the public test set, which is close to an accuracy of 0.90.

| Submission and Description | Private Score ⓘ | Public Score ⓘ |
|--|-----------------|-----------------|
|  LLM Detector 4 - Version 4 Succeeded (after deadline) · 5d ago | 0.679424 | 0.872656 |
|  LLM Detector 4 - Version 3 Succeeded (after deadline) · 6d ago | 0.678324 | 0.871327 |

4. Conclusions

In conclusion, this project successfully explored the application of deep learning for binary text classification on the task of distinguish between human-written and AI-generated text.

However, we think that further refinement holds the potential to significantly enhance the model's performance and robustness. Addressing the following limitations could lead to substantial improvements:

- Expanding dataset size and diversity
- Use pre-trained language models
- Explore alternative word embedding techniques
- Employ other regularization techniques
- Experimenting different text pre-processing techniques
- Conduct a more comprehensive hyperparameter tuning