Report of INF582 Data Challenge

Human Written Text vs. AI-Generated Text

Kaggle Team : **CH-EM**
By : Laychiva CHHOUT : laychiva.chhout@polytechnique.edu
Emmanuel Gnabeyeu : emmanuel.gnabeyeu-mbiada@polytechnique.edu

10 March 2023

1 Project Description

The goal of this data challenge was to study and apply machine learning/artificial intelligence techniques to a real-world classification problem and more specifically to predict whether a paragraph is written by humans or generated by an AI.

AI generative text is the process of using artificial intelligence models trained on large amounts of text data to create natural language texts. It has various applications in content creation, summarization, dialogue systems, and natural language understanding. However, the misuse of such technology raises ethical and social concerns. To address this, it is important to develop responsible and trustworthy practices for using and evaluating AI generative text systems. One approach is to use a labeled dataset to extract features that differ between human-written and AI-generated texts and train a classifier to distinguish between them, that is the mission of this challenge for which the evaluation metric is the accuracy, i.e. the fraction of predictions our model got right.

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

where TP = True Positives, TN = True Negatives, FP = False Positives, and FN = False Negatives.

2 Feature Selection/Extraction

2.1 Data cleaning and Text Pre-processing

The performance of text classification models can be greatly enhanced by implementing text preprocessing techniques. Eliminating extraneous or insignificant information from input text can improve the quality of the extracted features and ultimately assist the classification model in more accurately discerning between various categories or classes of text. Despite this, recent advances in NLP have led to the development of pre-trained models that can effectively handle noisy or irrelevant information within the text. In our project, we will conduct text pre-processing and evaluate the models' performance and we will not perform the text pre-processing on the pre-trained models.

2.2 Text processing and feature engineering

To create a representation of human-written and AI-generated texts, we need to extract relevant features that can help distinguish between them. Some features that could potentially differ between human-written and AI-generated texts include :

— Vocabulary richness : The number of unique words used in the text.
— Sentence length : The average length of the sentences in the text
— Punctuation usage : The frequency of different punctuation marks such as periods, commas, and exclamation points can be a useful indicator of the style and tone of the text.
— Word frequency, grammar usage, the style of phrases, syntax, and semantics, etc.

2.3 Feature Extraction

2.3.1 Word Frequency

The frequency of common words such as "the," "and," and "of", and of words in a general text can provide insight into the style and tone of the text. Human-written texts may use more diverse and varied vocabulary compared to AI-generated texts. Word frequency can be extracted using basic NLP techniques such as tokenization and counting, while more advanced techniques such as TF-IDF and LDA(Latent Dirichlet Allocation) can be used to extract more nuanced information about the frequency and distribution of words.

— Counting the frequency of words in the text can be used as a feature to capture the importance of each word in the text and improve the accuracy of the model.

— **TF-IDF** is a composite score representing the power of a given word to uniquely identify the document. It is computed by multiplying Term Frequency(TF) and Inverse Document Frequency(IDF) TF.

2.3.2 Grammar usage and syntax

The syntax of the text refers to the arrangement of words and phrases to create well-formed sentences. Human-written texts may have more complex and varied sentence structures compared to AI-generated texts. The percentage of grammatically correct sentences in the text can be relevant for this task. To extract it as a feature, We can use dependency parsing, Syntactic Parsing, and Part-of-speech (POS).

— **Part-of-speech (POS)** tagging is the process of labeling each word in a sentence with its corresponding part of speech and then indicating the syntactic category of each word in a text(e.g., noun, verb, adjective, etc.). POS tags capture information about the grammatical structure of the text and can help distinguish between human-written and AI-generated texts.

— **Syntactic parsing** is the process of analyzing the grammatical structure of a sentence. This information can be used to capture the relationships between words in the text and improve the accuracy of the model.

2.3.3 Semantics

The meaning of words and phrases can provide insight into the intent and message of a text. Human-written texts may have more nuanced and subtle meanings compared to AI-generated texts. NLP algorithms that can be used to extract semantics include word embeddings, semantic role labeling, named entity recognition(NER), and sequence models such as N-grams(e.g. by varying lengths unigrams, bigrams, trigrams) which are contiguous sequences of N words in a text, and capture information about the syntax and semantics of the text and can be effective in distinguishing between human-written and AI-generated texts. **NER** is the process of identifying and classifying named entities in a sentence (e.g., person, organization, location, etc.). This information can be used to capture the semantic meaning of the text and improve the accuracy of the model.

2.3.4 The style features

The style of a text refers to the way in which it is written, including factors such as tone, register, and use of figurative language. Human-written texts may exhibit more variation in style compared to AI-generated texts. it can be extracted by calculating the sentiment polarity of the text using sentiment analysis which is the process of identifying the sentiment (positive, negative, neutral) of a text. This information can be used to capture the emotional tone of the text.

2.3.5 Pragmatics

The pragmatics of a text refers to its social and cultural context, including factors such as audience, purpose, and genre. Human-written texts may exhibit more awareness and sensitivity to the social and cultural context compared to AI-generated texts. NLP algorithms that can be used to extract

pragmatics include discourse analysis, genre classification, and topic modeling which is the process of identifying the topics present in a text. This information can be used to capture the subject matter of the text.

2.3.6 Word embeddings(Transfer learning)

Word embeddings are dense vector representations of words that capture their meaning and context. We can use pre-trained word embeddings (e.g., GloVe, Word2Vec, FastText) and average or concatenate them to obtain a fixed-length representation of the text. Word embeddings can be effective in capturing the semantics of the text, and the syntactic similarities between words and can help distinguish between human-written and AI-generated texts. These embeddings can be used to replace one-hot encoded vectors in the input layer of a neural network.

2.4 Classification algorithms

Since we are dealing with a binary classification problem (human-written vs. AI-generated), some suitable algorithms are Logistic Regression, Random Forest, Support Vector Machines, Naive Bayes, and Gradient boosting classifiers.

We evaluated the performance of these algorithms using metrics such as accuracy, precision, recall, and F1-score. The training time of these algorithms can vary depending on the size of the dataset and the complexity of the features. SVM and Naive Bayes are usually faster to train than Random Forest.

		accuracy	f1_score	Training Time (s)
Word2Vec	Logistic Regression	0.58250	0.581454	6.829704
	Random Forest	0.55250	0.550251	12.047116
	SVM	0.57875	0.583436	7.797179
	XGBoost	0.53250	0.532500	21.265488
POS	Logistic Regression	0.59500	0.605839	87.242644
	Random Forest	0.68750	0.678663	88.897074
	SVM	0.67750	0.683824	87.737280
	XGBoost	0.67500	0.663212	87.517006
NER	Logistic Regression	0.53500	0.592105	89.571923
	Random Forest	0.59375	0.596273	95.191949
	SVM	0.59250	0.621810	90.510938
	XGBoost	0.55000	0.540816	91.314575

(a)

		accuracy	f1_score	Training Time (s)
Syntax	Logistic Regression	0.59000	0.594059	94.971269
	Random Forest	0.73625	0.737888	96.847464
	SVM	0.70125	0.713772	96.237641
	XGBoost	0.69625	0.697385	95.403530
WordFreq	Logistic Regression	0.59875	0.625438	0.547412
	Random Forest	0.63875	0.659600	6.793451
	SVM	0.58750	0.646681	5.317705
	XGBoost	0.61625	0.621455	5.019554
tfidf_bigram	Logistic Regression	0.59750	0.610169	2.904951
	naive_bayes	0.55500	0.468657	1.647679
	Random Forest	0.57625	0.605355	9.348057
	SVM	0.58750	0.646681	9.488684
	Gradient Boosting	0.59500	0.628440	19.705991

(b)

FIGURE 1 – Summary : Features+Algorithms+Metrics+Time

Comments : The best pipeline in this case in terms of accuracy is syntactic parsing and Random Forest Classifier.

3 Pre-trained Models from Huggingface

Several pre-trained LLM from huggingface [1] have been discovered that are known to perform well in fine-tuned tasks. Those models are BERT, RoBERTa, XLNet, GPT2, etc. However, after our evaluation, we have determined that BERT, RoBERTa, and XLNet demonstrate the highest performance levels. Therefore, we will focus solely on these three pre-trained models in our report.

3.1 Tokenization

Since XLNet, BERT, and RoBERTa share some similarities and can potentially use the same tokenizer, in this challenge we tokenized the input data by using **encode_plus** [2], which is provided by the Hugging Face Transformers library. For each sentence in the dataset, the **encode_plus** function of each tokenizer is called with the following parameters : the sentence to encode, adding special tokens such as [CLS] and [SEP], setting a maximum length of 360, padding or truncating the sentence

1. https ://huggingface.co
2. https ://huggingface.co/docs/transformers/main_classes/tokenizer

to **max_length**, returning attention masks for [PAD] tokens, and returning the encoded tensor in PyTorch format. The encoded tensors are then appended to their respective **input_ids** lists, and the attention masks are appended to their respective **attention_mask** lists. Additionally, the sentence IDs for each sentence in the dataset are collected and stored in the **sentence_ids** list. After all, sentences are encoded, and the **input_ids** and **attention_masks** lists are concatenated into tensors. Finally, the labels and **sentence_ids** are converted into PyTorch tensors.

3.1.1 Architecture of RoBERTa

Since these three pre-trained LLMs models all use the same base architecture (i.e., the transformer architecture), here we will just explain the architecture of RoBERTa which is the most suitable for our problem, (cf. in the next section).

```
==== Embedding Layer ====

roberta.embeddings.word_embeddings.weight              (50265, 768)
roberta.embeddings.position_embeddings.weight            (514, 768)
roberta.embeddings.token_type_embeddings.weight            (1, 768)
roberta.embeddings.LayerNorm.weight                           (768,)
roberta.embeddings.LayerNorm.bias                             (768,)

==== First Transformer ====

roberta.encoder.layer.0.attention.self.query.weight       (768, 768)
roberta.encoder.layer.0.attention.self.query.bias             (768,)
roberta.encoder.layer.0.attention.self.key.weight         (768, 768)
roberta.encoder.layer.0.attention.self.key.bias               (768,)
roberta.encoder.layer.0.attention.self.value.weight       (768, 768)
roberta.encoder.layer.0.attention.self.value.bias             (768,)
roberta.encoder.layer.0.attention.output.dense.weight     (768, 768)
roberta.encoder.layer.0.attention.output.dense.bias           (768,)
roberta.encoder.layer.0.attention.output.LayerNorm.weight       (768,)
roberta.encoder.layer.0.attention.output.LayerNorm.bias       (768,)
roberta.encoder.layer.0.intermediate.dense.weight        (3072, 768)
roberta.encoder.layer.0.intermediate.dense.bias              (3072,)
roberta.encoder.layer.0.output.dense.weight              (768, 3072)
roberta.encoder.layer.0.output.dense.bias                     (768,)
roberta.encoder.layer.0.output.LayerNorm.weight               (768,)
roberta.encoder.layer.0.output.LayerNorm.bias                 (768,)

==== Output Layer ====

classifier.dense.weight                                    (768, 768)
classifier.dense.bias                                         (768,)
classifier.out_proj.weight                                   (2, 768)
classifier.out_proj.bias                                         (2,)
```

FIGURE 2 – Architecture of RoBERTa

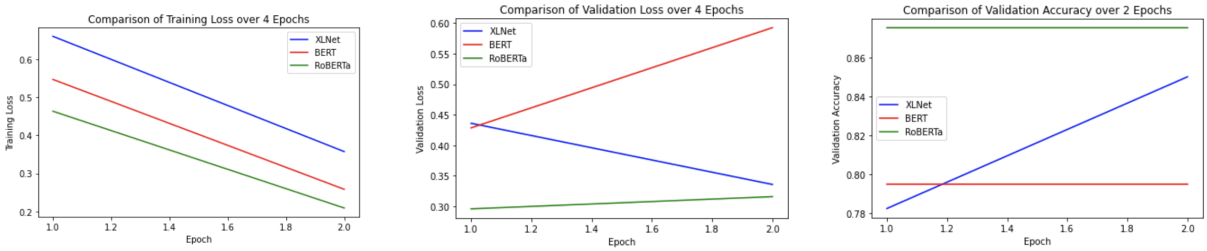
Basically, The RoBERTa model consists of a series of stacked layers, each containing a multi-head self-attention mechanism followed by a feedforward neural network. The architecture is described as below :

- The embedding layer : This layer includes three parts - word embeddings, position embeddings, and token type embeddings - that provide information about the meaning, position, and sentence separation of words in the input sequence.
- Transformer layers : RoBERTa uses a series of stacked transformer layers, each containing a multi-head self-attention mechanism followed by a feedforward neural network. The self-attention mechanism calculates attention scores between different words in the input sequence and uses them to obtain a context vector for each word. The feed-forward neural network then processes the context vectors to produce the output of that layer.
- Output layer : The final transformer layer is followed by an output layer consisting of two linear transformations, which map the output of the last transformer layer to a smaller size and then to the number of classes in the task at hand. The output is passed through a softmax function to obtain class probabilities.

3.2 Parameters used in training and validating

- Data : we used 90% of the training set as training data and 10% of the training set as validation data with **batch_size** = 16
- Optimizer : we used **AdamW** [3] with a learning rate of 5e-5 and an epsilon value of 1e-8.
- epoch : first we use 4 epochs, but after training, we see that after 2 epochs the validation loss increase (as shown in Fig 3(b)) due to the overfitting, so we finally use num_epoch = 2.

3. https ://huggingface.co/docs/transformers/main_classes/optimizer_schedules



(a) Training Loss (b) Validation Loss (c) Validation Accuracy

FIGURE 3 – Model Comparison

3.3 Result

By observing Figures 3(a) and 3(b), we see that the validation loss saturates pretty quickly while the training loss continues to lower. The models are powerful and start to overfit if trained for longer. Overall, we see that RoBERTa performs the best among all since the training loss is much lower than others, the validation loss increases slowly and also it has the best accuracy among all(cf. Figure 3(c)). The following(cf. Table 1) is the result of RoBERTa trained on the environment that we stated in the last section :

epoch	Training Loss	Valid. Loss	Valid. Accur.	Training Time	Validation Time
1	0.46	0.30	0.88	0:00:56	0:00:02
2	0.21	0.32	0.88	0:00:56	0:00:02

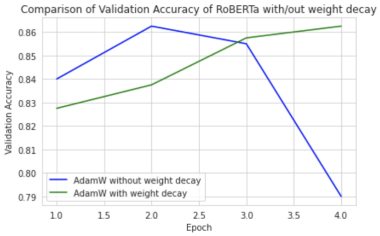
TABLE 1 – RoBERTa results on 2 epochs

3.4 Regularization

To prevent overfitting due to the small size of the training set (4000 tuples), we used weight decay (L2 regularization) which adds a penalty term to the loss function. By using different decay rates for different parameter sets, the model was able to learn better without overfitting.

```
param_optimizer1 = list(bert_model.named_parameters())
no_decay = ['bias', 'gamma', 'beta']
optimizer_grouped_parameters1 = [
    {'params': [p for n, p in param_optimizer1 if not any(nd in n for nd in no_decay)],
     'weight_decay_rate': 0.01},
    {'params': [p for n, p in param_optimizer1 if any(nd in n for nd in no_decay)],
     'weight_decay_rate': 0.0}
]
```

(a) Code snippet



(b) Comparison

FIGURE 4 – Regularization

4 Conclusion

In conclusion, the progress in the NLP field has made pre-trained models more effective for various tasks, including binary text classification with small datasets. However, it's still crucial to have a solid understanding of fundamental techniques like feature selection/extraction, data cleaning, and text pre-processing, since there may be cases where pre-trained LLMs are not applicable. In our opinion, this data challenge has expanded our understanding of NLP and provided insights into the capabilities of current pre-trained models.