Human versus Artificial Intelligence Distinguishment

Group 2

Jeremy Cryer, Jason Raimondi, and Shane Schipper

Shiley-Marcos School of Engineering, University of San Diego

AAI 590 Capstone Project

Professor Anna Marbut

4/15/2024

Introduction

The objective of this project is to determine the origin of text, enabling the capability to distinguish the creation of the content from either human or artificial intelligence (AI). As AI continues to improve rapidly, it is becoming increasingly difficult to differentiate content between being generated by humans or AI. This is a positive feature for many AI applications; however, this also introduces risk for others. Consider scenarios such as fake text or images displayed in news feeds or AI-generated essays submitted by students for coursework. This can result in the spread of misinformation, issues with academic integrity, as well as hurting academic performance.

Essentially anyone with an internet-connected device (e.g., computer, mobile device) can be the end user of the team's AI application, providing this capability to predict content origin. In the couple examples discussed above, anyone could use this application to check text for their origin if they came across something in their newsfeed that they are questioning. In the education example, teachers can make use of this application to check submitted essays and other papers to help uphold academic integrity. Academic integrity has become a large concern due to the capability of AI that even universities, such as the University of Kentucky, have sent out notices to their students to help control this. As stated from a post from the school, it states, "We are committed to upholding academic integrity and ensuring fairness. As part of that process, we want to highlight resources available to support proctoring and plagiarism detection. Earlier this month, Turnitin – a vendor we utilize – began providing an artificial intelligence (AI) detector, integrated with Canvas, which offers an estimate of how many sentences in a written submission may have been generated by artificial intelligence. According to the software developers, educators can use this estimate, as they do with the plagiarism detector output, to determine if further review, inquiry or discussion with the student is needed" (Stokes, 2023). This post came from the university's administration alerting everyone that AI use is a concern for academic integrity, and they are using AI content detectors, such as ones being sought out in this project, to help address this problem.

This project uses data from multiple datasets, all openly available on Kaggle[1]. First, there is an AI versus human text dataset, which contains nearly 500,000 AI and human-generated essays, gathered from multiple sources which is from Gerami, S. (2024). For additional data, this project also uses a dataset for AI-generated versus student-generated text. This is a smaller dataset, with 1,103 samples provided by Dongre, P. (2023). However, it provides some additional samples with added variety. This dataset is also more informal, so it is possible that grammar could prove to be an important factor. These datasets are anticipated to be sufficient for development of a minimum viable solution to the problem.

The primary goal of this project is performing binary classification and providing the result (i.e., generated by human or AI) to the AI application users. To make the application results more appealing and trustworthy, this team also plans to provide the probability (i.e., confidence) of its predictions. As a secondary project goal, the team plans to develop an interactive web application, providing users with the ability to enter or paste in copied text for analysis and obtain prediction results on-demand.

Dataset Summary

For this section, the data selection and preprocessing performed will be discussed. This section will go over the data selection, details, preprocessing, and common relationships noticed as the Exploratory Data Analysis (EDA) and feature engineering processes are performed. Data is the cornerstone of any predictive model, and, if not properly selected and understood, the

[1] https://www.Kaggle.com

model's performance is likely to be inadequate. The quality of the data used to train and test on directly affects the model's performance. Even the best-performing algorithms can prove to be useless when paired with poor data quality.

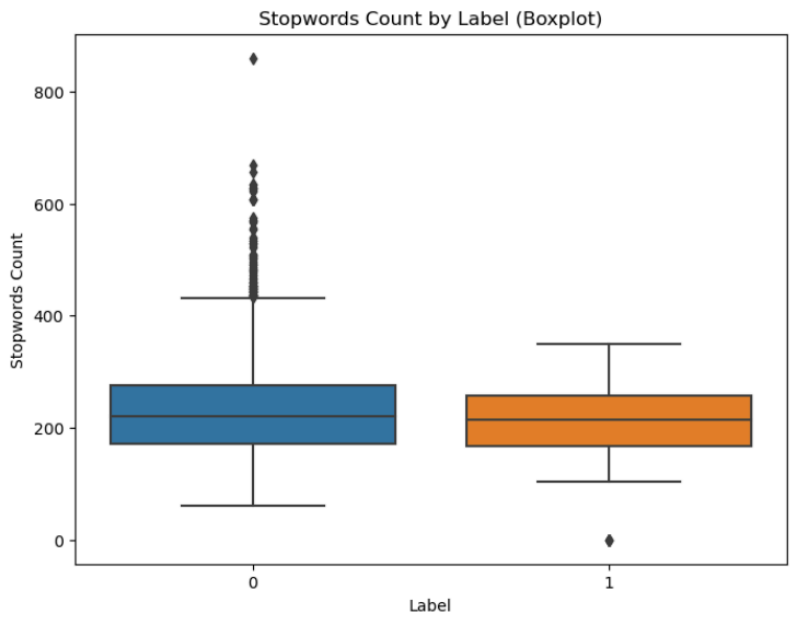
Dataset Selection

The main dataset selected was a very simple yet vast data set. It was found on Kaggle, provided by Gerami, and contains data for text created by both AI and human sources. The dataset contains nearly 500,000 samples of text labeled as either AI or human-generated. No other features came with the raw dataset. A secondary dataset was also used for small batch testing, as it consisted of less formal text and was at more of a high school student writing level. This dataset has the same features as the main dataset, but it is much smaller, with only slightly over 1,000 samples. This dataset was also found on Kaggle, provided by Dongre. This dataset is to represent an edge case, where the grammar from a high school level text is drastically different from expected AI-generated text and will help the team's understanding if the model is performing well on "easy" predictions. This dataset is only being used as an additional gauge of initial performance but will not be used for evaluation of the team's developed models.

EDA, Preprocessing, and Feature Engineering

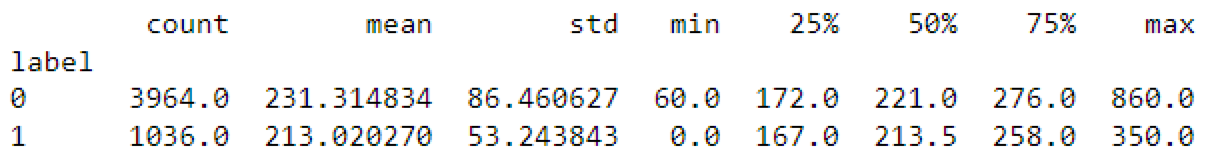
When investigating the main dataset, it was determined that data cleaning and preprocessing needs were minimal. From a data incompletion standpoint, only one data point was missing, so the sample was simply discarded from the dataset. Tokenization is a common feature engineering step for any Natural Language Processing (NLP) problem. As stated on a stackademic blog entry, "Tokenization is very important as it is a base step of feature engineering, and it determines how our model will interpret the data. Thus, it is very important for an NLP Engineer to tokenize the data appropriately in order to avoid confusion" (Patel, 2023). Patel (2023) also mentions other commonly-used steps during the tokenization process, such as considerations to dropping all upper case, punctuation, and stop words. Normally, this is a key preprocessing step for any NLP analysis. However, in this article, it states, “Deleting stop words in the pre-processing stage impacted the classification performance negatively since the selection of stop words play a crucial role in differentiating human and AI” (Islam et al, 2023). Originally, it was contemplated to perform these additional steps, as most NLP models benefit from this, but the logic behind this and the additional article’s insights suggested against doing so. In response to this information, the team performed EDA to understand the difference and potential impact of stop words between the two classes. To accomplish this analysis, a new feature was engineered to list the stop word count for each data sample. According to Figure 1 and Table 1, where label 0 is human-generated and 1 is AI-generated, there is a vast difference in stop words between the two text generation sources.

Figure 1



Note: Stop Words Count by Label

Table 1

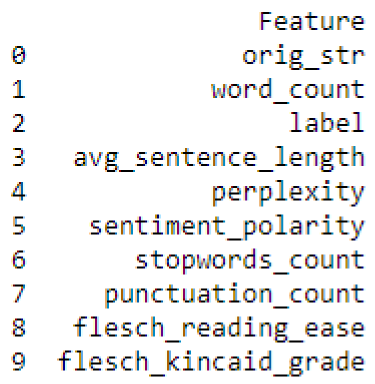


```
         count        mean        std   min    25%    50%    75%    max
label
0       3964.0  231.314834  86.460627  60.0  172.0  221.0  276.0  860.0
1       1036.0  213.020270  53.243843   0.0  167.0  213.5  258.0  350.0
```

Note: Table of Statistics for Stop Words by Label.

This indicates that the extra granularity provided by inclusion of stop words could prove to have predictive power for the team's models. This discovery was one of the strongest relationships identified during EDA that helps support limiting certain preprocessing steps to suit this specific problem. Due to this realization, it was decided to keep stop words during tokenization, such as punctuation and capitalization. Additional features, such as word count, perplexity, sentiment polarity, etc., were also engineered for further consideration, as limited features came with the chosen dataset. A list containing engineered features is displayed in Figure 2.

Figure 2



```
              Feature
0            orig_str
1          word_count
2               label
3  avg_sentence_length
4          perplexity
5   sentiment_polarity
6      stopwords_count
7    punctuation_count
8   flesch_reading_ease
9  flesch_kincaid_grade
```

Note: List Containing Engineered Features

Background Information

For this problem, the end goal is to create an application that can successfully and accurately differentiate between text generated by either human or AI. There have been several models and applications created to this day that can accomplish this with varied approaches.

After researching this problem and existing solutions, successful outcomes were more prominent with traditional machine learning algorithms such as Extreme Gradient Boosting (XGBoost), as well as with deep learning transformer-based model architectures such as DistilBERT. As discussed in this section, the architecture approaches that were attempted to solve this problem will be discussed at a high level.

Traditional Machine Learning Approaches

Traditional machine learning approaches to this problem have largely focused on algorithms such as random forests, Support Vector Machines (SVMs), and XGBoost. Models based on these algorithms are typically trained on datasets composed of meta-text features, as the raw text sequence is incompatible. Generally, with a robust feature set, these various types of models perform well but may be lacking as compared to more complex model architectures, such as transformers.

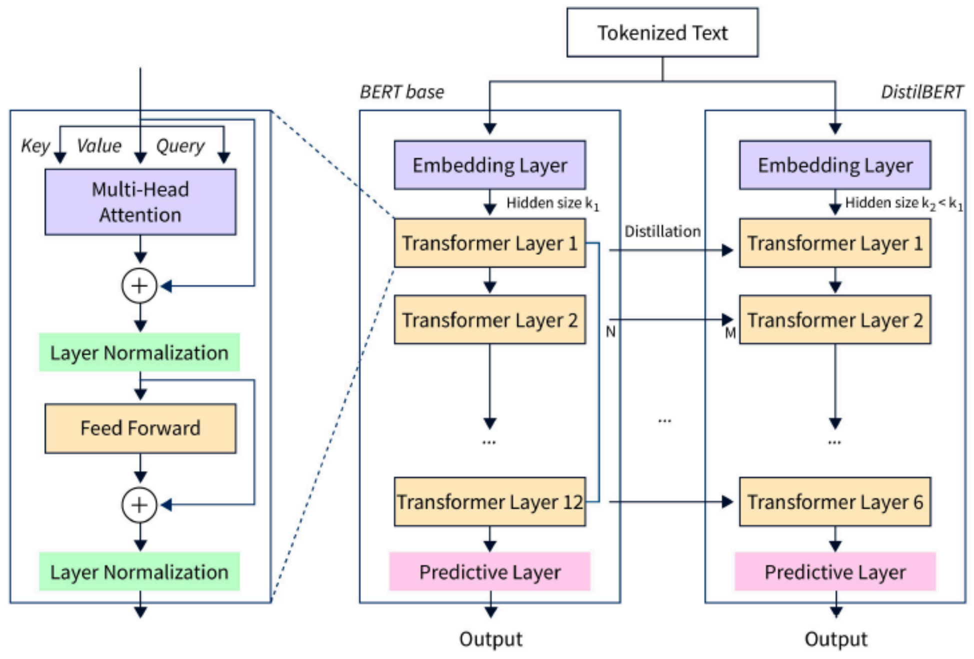
Previous works, such as from Lorenz Mindner (2023), have focused largely on features, such as perplexity, to help perform classification. This feature refers to the predictability of words by a language model based on the previous token. Per Lorenz Mindner (2023), additional features such as word frequency counts, stop word and punctuation counts, and sentiment polarity have also been shown to be effective for this task. A combination of these features should form a robust feature set for baselining traditional machine learning methods for comparison to transformer-based approaches, which will be discussed below.

DistilBERT

DistilBERT is one example of a model architecture that has been proven to work for this application. It was found that Kumar et al. (2024) approached this problem by using a popular LLM, DistilBERT, which is a “distilled”, or, more lightweight version of BERT (Bidirectional

Encoder Representations from Transformers) and trained on the same corpus of text. As the names imply, both variants are transformer-based models containing an encoder but no decoder. The DistilBERT architecture can be seen in Figure 3.

Figure 3



Note: Architecture of DistilBERT (Islam, 2023)

As can be seen when referencing the figure, the architecture takes tokenized text as input and uses an embedding layer, six transformer layers (versus twelve in BERT base), and a predictive layer that generates output. Each transformer layer contains a multi-head attention mechanism, layer normalization, a feed forward neural network, and another layer for normalization.

In this research project, experiments performed by Kumar et al. (2024) concluded that DistilBERT achieved high performance in the task of detecting AI-generated text, around 94 percent overall accuracy. Combined with the efficiency of this distilled version of BERT, it provides a promising solution to this problem.

Experimental Methods

For this problem, the team chose to experiment with three different models, a pre-trained DistilBERT transformer model, a custom PyTorch transformer model, and a model based on traditional machine learning algorithms. Previously mentioned, in the methodology background section, all three models and their conceptual background was discussed. Due to limitations about compute resources available for a large dataset (i.e., nearly 500,000 samples), all three models were presented with a similar challenge. This constraint influenced some of the choices made in how the models were trained, evaluated, and optimized, which is discussed in more detail for each specific model.

Pre-trained DistilBERT

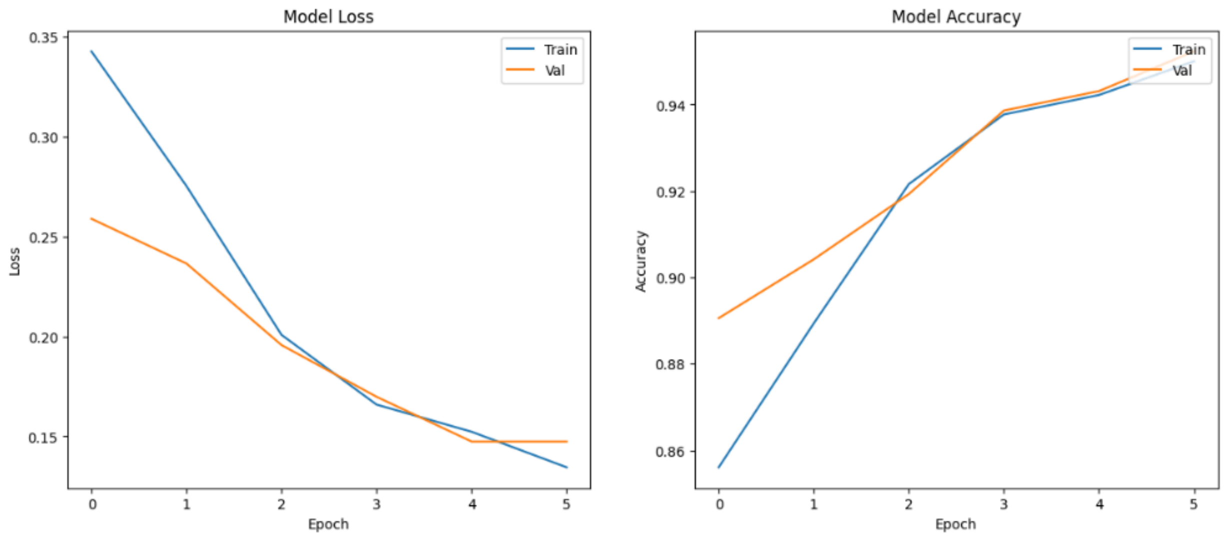
For the pre-trained DistilBERT model, the pre-trained “cased” variant was selected to take into account capitalized words and letters which were deemed likely useful for the prediction being made. This preloaded architecture uses a vocabulary size of 28,996 tokens, 512 maximum position embeddings or sequence length, six transformer layers, 12 attention heads for each attention layer, 3,072 hidden dimensions of the feedforward layer, 20 percent dropout rate for the sequence classifier, and a Gaussian Error Linear Unit (gelu) activation function.

For this and the additional models that will be mentioned, data was split in the same manner. First, either the entire or subset of the dataset is imported. Afterwards, the majority class (i.e., Human - Class 0) is down sampled until it balances the dataset with an equal number of samples as AI - Class 1. The resulting balanced dataset is then split into 90 percent training and 10 percent testing, with an additional split of the training dataset into 80 percent training and 20 percent validation. For each of these splits, stratified sampling and shuffling was used to maintain a class balance and properly mix the samples. Torch datasets were then created while tokenizing and padding the sequences, followed by creation of data loaders to batch the datasets. A batch size of 16 was used, as larger batch sizes (e.g., 32 and 64) were prone to GPU memory issues. A learning rate of 0.001 was configured and Binary Cross Entropy (BCE) loss was used as the loss function during training, as the problem at hand is a binary classification task with only two classes. A training loop with binary accuracy was created, which allowed each training epoch to display loss and accuracy for both training and validation datasets. This same training loop was also used for the custom model.

After experimenting with this pretrained model using different data sizes and carefully reviewing preparation items, the team was not able to obtain good results. Accuracy for both the train and validation datasets was around 50 percent each, which is the same as random chance for these binary predictions. To optimize, the team modified the preloaded model's configuration parameters. First, the vocabulary size was increased from 28,996 to 200,000. The custom transformer's vocabulary was around this size and had good performance, so it was decided to optimize the pretrained model in some aspects to be more like the custom model. It is important to note that vocabulary sizes are not generally this large, however, this custom vocabulary was because the team used a different tokenizer for the custom transformer than the standard tokenizer associated with DistilBERT models. This tokenizer was a function based on words instead of sub-words traditionally configured for a smaller size limit. Next, the following parameters were reduced: hidden dimensions from 3,072 to 1,024, transformer layers from six to two, and sequence classifier dropout rate from 0.2 to 0.05.

Following optimization, the training results were more promising. The training run was configured for 10 epochs but stopped after six due to a timeout on the Google Colab session. However, it was still sufficiently trained at that point, with the train accuracy at 0.9500 and the validation accuracy at 0.9523. The model learning curves can be seen in Figure 4.

Figure 4

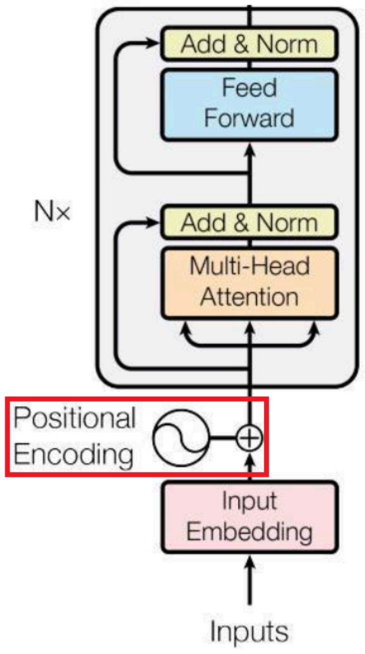


Note: Training - DistilBERT Model Loss and Model Accuracy

Custom Transformer

Due to the research and conclusions with DistilBERT, it was determined to not only experiment using a pre-trained DistilBERT model but to also build a custom transformer model from scratch using the PyTorch framework provided by Paszke et al. (2019). In this case, the framework provides flexibility to create similar layers as the DistilBERT architecture uses. To reduce the complexity level for this model, pre-built transformer layers were used, which already contain the additional components as seen in Figure 3. It was also beneficial to use some modified portions of code from the PyTorch transformer tutorial, provided by "Language modeling" (n.d.), in order to incorporate positional encoding (See Figure 5) which was needed to provide the model with information about the sequences of text.

Figure 5



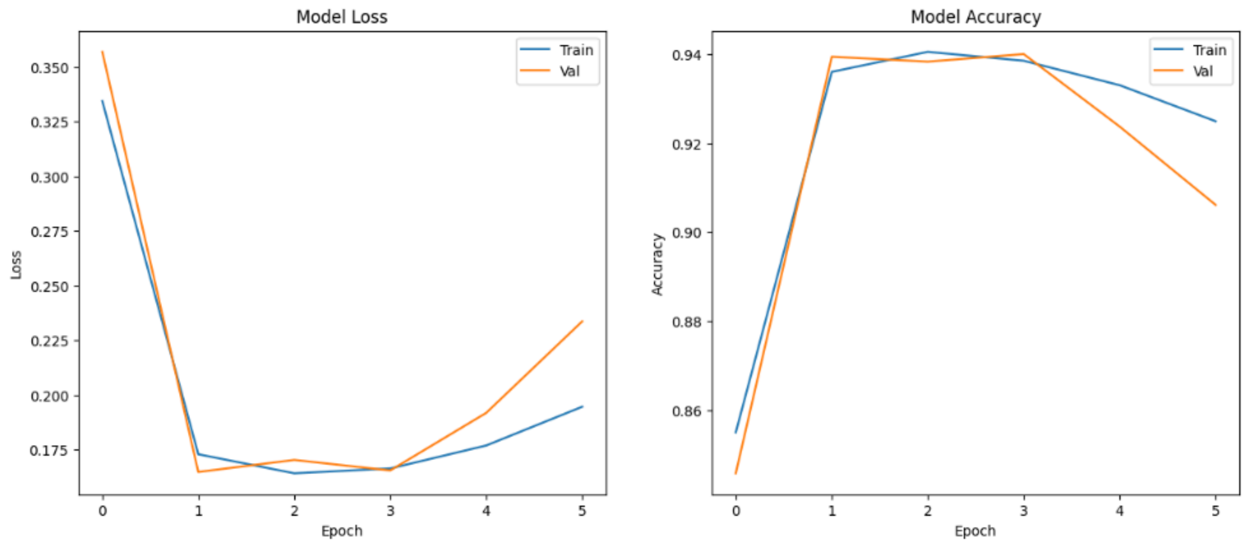
Note: Positional Encoding in Transformer Architecture (“Language modeling,” n.d.)

This architecture enabled more customization and flexibility to the model as compared to the pre-trained models, allowing for more experimentation. With this architecture, vocabulary and trained embeddings were built from scratch using the training dataset.

As partially shown in the Figure 5 diagram, the model architecture contains an embedding layer, positional encoder, two transformer layers, a predictive layer with one output which is then input to a Sigmoid activation function to produce a probability between 0 and 1 for the positive class. As previously mentioned, the vocabulary was trained from scratch and resulted in a total size of 208,251 tokens, which is considerably large as compared to defaults with other models such as DistilBERT. This again was due to the choice of tokenizer function used. In a choice to configure some parameters close to DistilBERT, this model was instantiated also using 12 attention heads and gelu activation. It was decided to use a small number (i.e., 2) of transformer layers since research by Kumar et al. (2024) indicated that a smaller number did not have much effect on performance of a similar task. Due to computational challenges, it was also chosen to use a smaller dimension of 1,024 for the feedforward network than DistilBERT's dimension of 3,072.

In this case, similar data preparation was performed as with DistilBERT, however, this training run did include the entire dataset and used a batch size of 32. Coincidentally, this also ran for six total epochs after stopping due to an unexpected interruption. The train accuracy ended at 0.9249, and the validation accuracy at 0.9061. The model learning curves can be seen in Figure 6.

Figure 6



Note: Training - Custom Transformer Model Loss and Model Accuracy

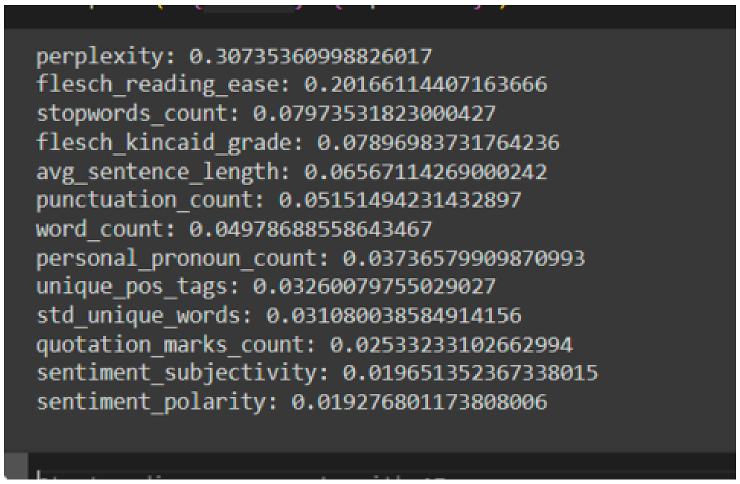
The train accuracy was as high as 0.9385 and the validation accuracy at 0.9400 after the fourth epoch. As can be seen in the previous figure, it appears that the model begins to overfit and lose generalization if trained too long. This insight, along with experimentation, helped us to optimize the previous model. When comparing architectures, the vocabulary size was very different, and the previous model contained a sequence classifier dropout, whereas the custom model did not. Based on performance of the two, the previous was underfitting the data whereas this model began to overfit the data. This is why the increased vocabulary size and reduction in dropout improved the previous model. For future optimization of this custom model, steps would be taken to prevent overfitting after the initial epochs. Incorporating additional dropout and/or regularization techniques may prove to optimize this model further.

Traditional Model

As a form of baselining, the team also trained multiple traditional machine learning models. These models included Random Forest Classifier, Decision Tree Classifier, SVM, and XGBoost. The input data was split into training and testing sets that measured 80 percent and 20 percent of the data, respectively. It was found that the random forest classifier performed the best across all binary classification metrics and achieved roughly 98 percent classification accuracy.

The feature importances of the random forest model were also examined to determine which meta-text features were the most impactful. These can be seen in Figure 7.

Figure 7



```
perplexity: 0.30735360998826017
flesch_reading_ease: 0.20166114407163666
stopwords_count: 0.07973531823000427
flesch_kincaid_grade: 0.07896983731764236
avg_sentence_length: 0.06567114269000242
punctuation_count: 0.05151494231432897
word_count: 0.04978688558643467
personal_pronoun_count: 0.03736579909870993
unique_pos_tags: 0.03260079755029027
std_unique_words: 0.031080038584914156
quotation_marks_count: 0.02533233102662994
sentiment_subjectivity: 0.019651352367338015
sentiment_polarity: 0.019276801173808006
```

Note: Random Forest Model Feature Importance

The three most important features to the classifier were perplexity, flesch reading ease, and stop word count. This indicates that these three features had the most correlation with the division between the two classes. Conversely, sentiment polarity, sentiment subjectivity, and quotation marks count proved to be the three least important features and thus could likely be pruned from the model with minimal accuracy loss.

Results and Conclusion

During this project, the team experimented with three different modeling methods to approach the problem. As discussed throughout this report, the three modeling methods included a pre-trained DistilBERT, a custom transformer model, and traditional machine learning algorithms in an attempt to determine which method would result in the best outcome. Initially, all three approaches had similar results with the custom transformer and traditional algorithms being the only methods capable of being trained on the entirety of training data without computational efficiency becoming an issue. However, after optimization and comparing inference, only one model, the custom transformer, was deemed as the team's best choice for a production environment.

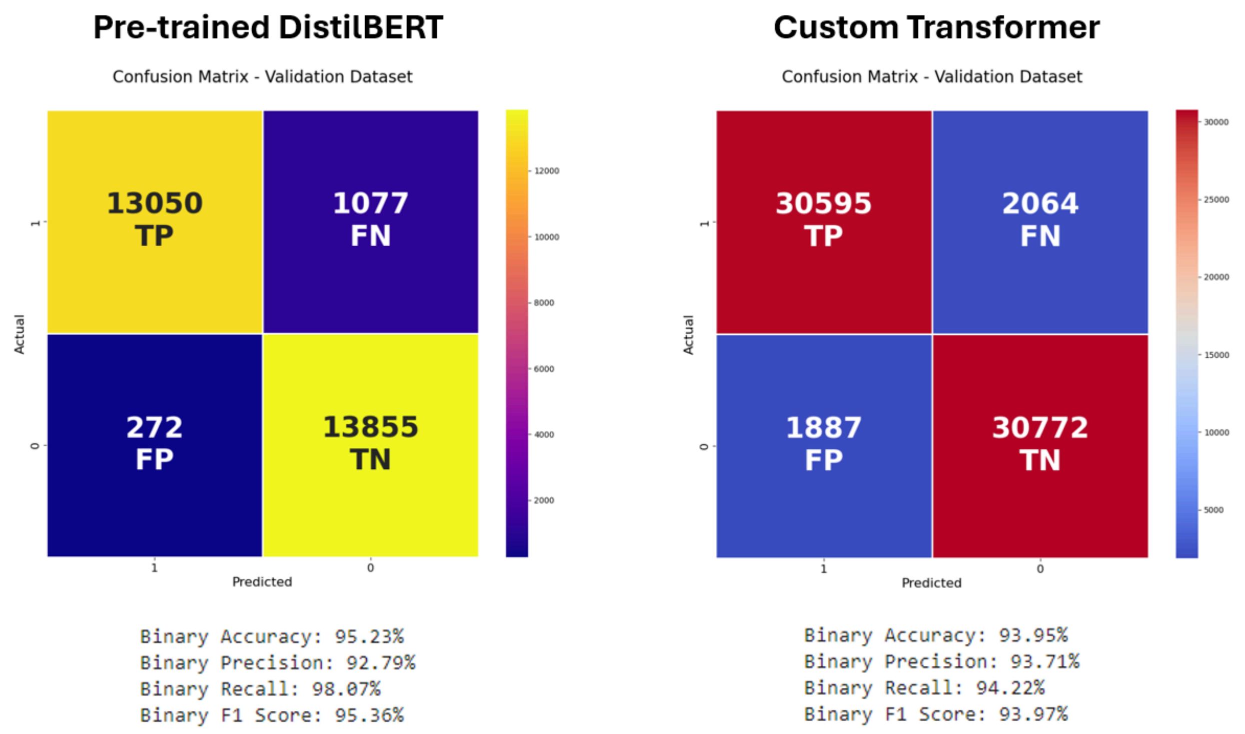
Model Evaluation

To evaluate the team's models, the metrics chosen for comparison include Binary Accuracy, Binary Precision, Binary Recall, and Binary F1 Score. The traditional models displayed good accuracy on the original training and validation sets. However, these models did not generalize well, and, as such, massively over-predicted the positive class on any data not originating from the original sets. The team believes this was caused by large amounts of text length disparities which threw off many of the meta-text metrics being used as features by these models. Thus, it was decided to focus solely on comparing the pre-trained DistilBERT model with the custom transformer model.

Validation Dataset

The evaluation results on the validation dataset for the pre-trained DistilBERT and custom transformer models can be seen in Figure 8.

Figure 8



```
Binary Accuracy: 95.23%
Binary Precision: 92.79%
Binary Recall: 98.07%
Binary F1 Score: 95.36%
```

```
Binary Accuracy: 93.95%
Binary Precision: 93.71%
Binary Recall: 94.22%
Binary F1 Score: 93.97%
```

Note: Results Comparison on Validation Dataset

As mentioned previously, the custom transformer was trained on the entire dataset, and the pre-trained DistilBERT model only on a subset due to resource constraints. When comparing between the two, the DistilBERT model shows better accuracy, recall, and F1 score, whereas the custom transformer has better precision.

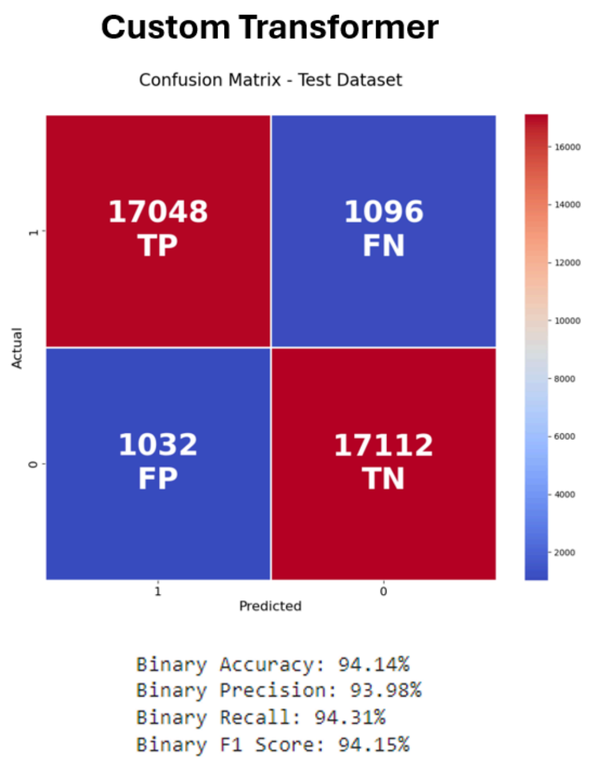
Model Selection

Although the DistilBERT model shows better metrics overall, the team still chose to select the custom transformer as the best model for two reasons. First, its predictions were more consistent. Both the correct and incorrect predictions were very well-balanced for each individual class, whereas the DistilBERT model's false positives were only about 20 percent compared to about 80 percent of false negatives. The second reason is that the DistilBERT model was trained on a much smaller dataset, so the team felt more confident with the custom transformer being trained on the entirety of the dataset for better generalization in production.

Test Dataset

The evaluation results on the test dataset for the custom transformer model can be seen in Figure 9.

Figure 9



```
Binary Accuracy: 94.14%
Binary Precision: 93.98%
Binary Recall: 94.31%
Binary F1 Score: 94.15%
```

Note: Results on Test Dataset

The custom transformer performed similarly although slightly better on the test dataset as to the validation dataset. This is an indication that it is generalizing well to unseen data.

Production Readiness

The team was interested in operationalizing the selected model, so additional efforts were made to do so. As this was not a project requirement, this area will summarize but not go into great detail on all the specifics. In general, the steps taken can be broken down into three different areas: application development, containerization, and application deployment.

Application Development

The team decided to develop a Flask web application with an HTML front-end for users to interact with. Tutorials provided by Ongko (2022) and "Deploy a containerized Flask or FastAPI web app" (2023) assisted, respectively, for building and deploying the application on the Azure cloud platform. Before deployment, the application was first developed in a local environment to provide more flexibility and capability to use existing development environment tools and applications.

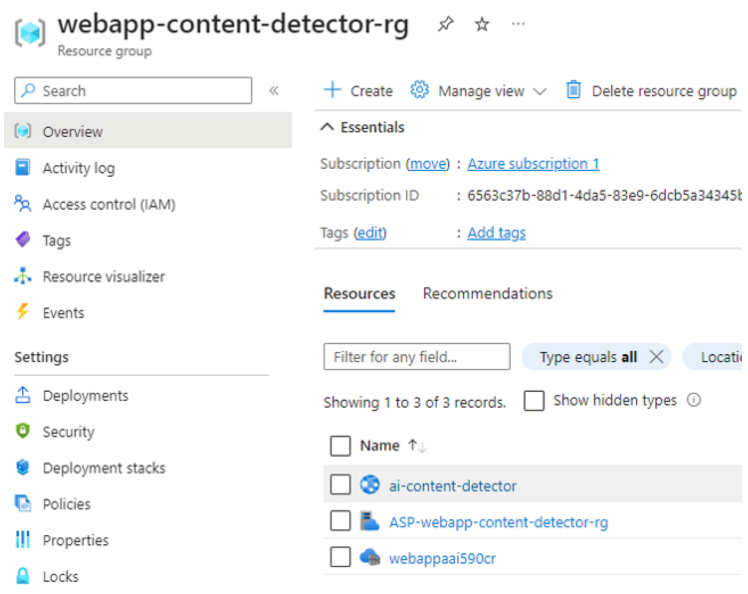
Containerization

To prepare for deployment, the locally developed application was containerized using Docker. This helps to provide a more compatible, portable, and seamless deployment. A Python "slim" base image was used to minimize size, then additional layers were added to support the team's application. Afterwards, a new Docker image was built based on this configuration.

Application Deployment

To deploy on Azure, the general process included creating a resource group where all resources supporting the application were then created within. Specifically, this included an Azure Container Registry, Web App, and App Service Plan. The team's Docker container image was pushed to the Azure Container Registry and then a Web App was created to publish the container. After deployment, the application was confirmed available and serving requests via the public internet. A sample screenshot can be seen in Figure 10 of the Azure web app resources, and another in Figure 11 showing the web app hosted on Azure and accessed via the public internet.

Figure 10



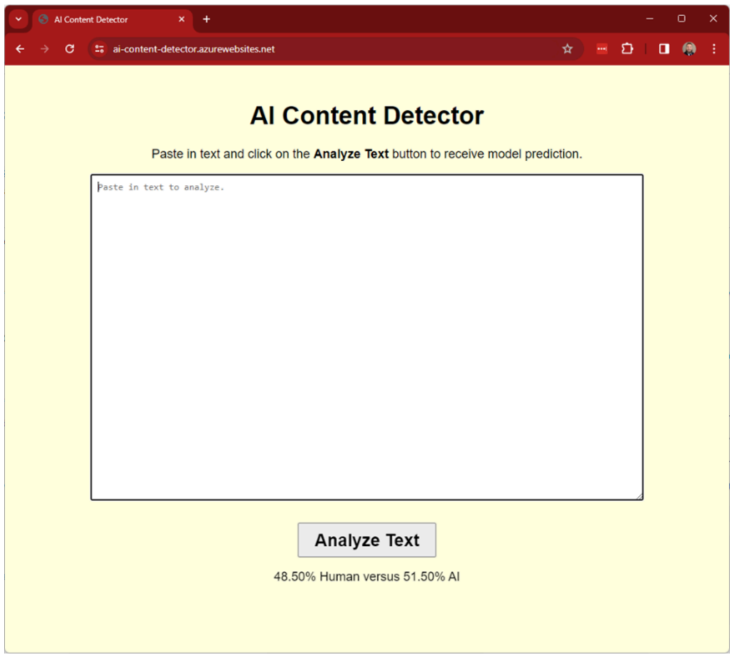
Subscription (move) : Azure subscription 1
Subscription ID : 6563c37b-88d1-4da5-83e9-6dcb5a34345t
Tags (edit) : Add tags

Showing 1 to 3 of 3 records.

Name
ai-content-detector
ASP-webapp-content-detector-rg
webappaai590cr

Note: Azure Web App Resource Group

Figure 11


AI Content Detector
ai-content-detector.azurewebsites.net
AI Content Detector
Paste in text and click on the **Analyze Text** button to receive model prediction.
Paste in text to analyze.
Analyze Text

48.50% Human versus 51.50% AI

Note: Web App Hosted on Azure

Future Improvements

Considering time constraints for this project, the team recognizes that this is still an early iteration and still has room for improvements to be made. If one were to desire a full production release of this solution, some things would have to be reevaluated and improved on the existing system to best prepare for a full release. This primarily revolves around the dataset used to train and test the model.

Based on the results seen, the datasets utilized for all the model training, evaluation, and testing was sufficient. However, for a real-world application, more robust datasets would be desired. The team would want to have a better understanding of the origins of the text content to make sure it is collected from a wide variety of sources to enhance this application and limit the introduction of any possible biases. This is seen to be a concern such that the team decided to create their own custom dataset using an AI interface to create paragraphs of text for inference.

Even though the models performed well using the split datasets that were both trained and tested with, the models did not perform equally as desired with the custom text inputs. However, in terms of this project, the datasets were sufficient in size and quality in order to build and deliver an AI application prototype for an early iteration of this solution.

One aspect that also seemed to overly affect the models was the length of the text corpus. From some preliminary analysis, it was observed that classification accuracy dramatically fell as the length of the text samples decreased. This was particularly evident in the baseline traditional models. Thus, future work should focus on improving the models' robustness to text corpuses with variable length as this would result in a more real-world practical classifier.

Conclusion

Throughout this project, the primary goal was to build the most reliable system possible to allow detection of AI versus human-generated text. The team was challenged with multiple existing methods of obtaining this goal and ultimately pursued three different modeling methods to implement and compare: a pretrained DistilBERT, custom transformer, and traditional machine learning algorithms. In the end, due to some computational restrictions and data limitations, it was found that the custom transformer performed the best for this early iteration and was selected to be implemented in an interactive web application for use. Although the resulting predictions are not as accurate as the team would want for a production scenario, this is a great proof of concept with some beneficial learnings to implement on subsequent iterations centered around the data reliability.

References

[1] *Deploy a containerized Flask or FastAPI web app on Azure App Service*. (2023, December 7). Microsoft Learn. https://learn.microsoft.com/en-us/azure/developer/python/tutorial-containerize-simple-web-app-for-app-service?tabs=web-app-flask

[2] Dongre, P. (2023, December 4). *Detect- ai generated vs student generated text*. Kaggle. https://www.kaggle.com/datasets/prajwaldongre/llm-detect-ai-generated-vs-student-generated-text

[3] Gerami, S. (2024, January 10). *Ai vs human text.* Kaggle. https://www.kaggle.com/datasets/shanegerami/ai-vs-human-text

[4] Islam, N., Sutradhar, D., Noor, H., Raya, J. T., Maisha, M. T., & Farid, D. M. (2023). *Distinguishing Human Generated Text From ChatGPT Generated Text Using Machine Learning.* https://doi.org/10.48550/arXiv.2306.01761

[5] Kumar, P., Ahmed, S., & Sadanandam, M. (2024, February 1). *Distilbert: A novel approach to detect text generated by large language models (LLM)*. Research Square. https://doi.org/10.21203/rs.3.rs-3909387/v1

[6] *Language modeling with nn.transformer and torchtext*. PyTorch. (n.d.). https://pytorch.org/tutorials/beginner/transformer_tutorial.html

[7] Lorenz Mindner, Schlippe, T., & Schaaff, K. (2023). *Classification of Human- and* AI-*Generated Texts: Investigating Features for ChatGPT*. Lecture Notes on Data Engineering and Communications Technologies, 152–170. https://doi.org/10.1007/978-981-99-7947-9_12

[8] Ongko, G. C. (2022, February 3). *Building a machine learning web application using flask*. Medium. https://towardsdatascience.com/building-a-machine-learning-web-application-using-flask-29fa9ea11dac

[9] Patel, P. (2023, September 13). *Unlocking the power of NLP: A deep dive into text preprocessing steps*. Medium. https://blog.stackademic.com/unlocking-the-power-of-nlp-a-deep-dive-into-text-preprocessing-steps-8eb5dfe8b94

[10] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., … Chintala, S. (2019). PyTorch: An Imperative Style, High-Performance Deep Learning Library. *Advances in Neural Information Processing Systems* 32, 8024–8035. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[11] Stokes, S. (2023, April 21). *Upholding academic integrity and ensuring fairness; AI detection*. Office of the Provost. https://provost.uky.edu/news/upholding-academic-integrity-and-ensuring-fairness-ai-detection