Exploring AI Content Detection with OpenAI API and TensorFlow

Aaron Weiss – MS Applied Statistics

Eastern Michigan University

Dr. Andrew Ross

Faculty Advisor

EMU Department of Mathematics and Statistics

Artificial Intelligence has changed substantially over the past year. The Large Language Model (LLM) has taken over the public consciousness, with public facing chatbot programs finally arriving. It feels like the entire world is racing to figure out what a model like this cannot be used for. One particular use case that has become common, is students using the chatbot function for help on homework and tests at all grade levels, with improved models even being able to pass standardized tests like the Bar exam5 all on their own. As academic institutions come to terms with the new landscape of tools that students can use, some are adapting quickly with methods ranging from allowing the use of chatbots to getting rid of homework entirely. Others are relying on an unproven technology. The technology in question is itself a statistical model, which in theory can detect the difference between content generated by a chatbot, and content created by a human. However, these tools have been shown to work poorly or not all, leaving teachers without reliable methods to know if students are benefitting from ChatGPT or similar services.

Just after setting out to research this topic, OpenAI, the company behind ChatGPT published their own AI detection tool10 which they titled ‘AI Text Classifier’ in a lengthy blog post with a link to then beta version of the tool. Text can be pasted into a box, the classification run, and the model will return a classification, from ‘very unlikely’ to ‘likely’ to be AI-generated. In the blog post detailing the tool, OpenAI states that their tool “…correctly identifies 26% of AI-written text as ‘likely AI-written,’”. So, what does this all mean? Essentially, detecting AI generated content is a task difficult, even the makers of ChatGPT have yet to solve it, and thus it is unlikely that we will solve this task either.

This paper is not an attempt at a solution to AI detection, consistently spotting AI generated text is a complex task that the creators of ChatGPT themselves have yet to complete. This project merely serves as an exploration into whether content that resembles human writing can be generated systematically and different methods of doing so, the characteristics of AI generated content, and creating models to compare current results from other research projects on this topic. After finding an appropriate dataset of articles and blog posts, we used those articles to generate text from ChatGPT, giving the bot the role of students of different grade levels. These texts are then run through different classification models, looking at human against AI writing, but also the different grade levels of AI writing compared against each other. Finally, we will examine the results and discuss possible ways to expand upon this work in the future. Existing tools for AI detection may not be perfect, but perhaps we can contribute something to the conversation.

**LLMs**

What exactly is an LLM? Simply put, it is a statistical model that takes a string of text as input, and outputs text that it predicts is desired based upon the input. The LLM is a tool used in the AI subfield of Natural Language Processing (NLP), which deals with analyzing and predicting text data from all different types of sources. The “Language” part of the LLM is straightforward enough, but the “Large” part is where these models start to shine. GPT 3, which was the most well-known of these models we when started this project, used an unfathomable amount3 of data as its training data. Approximately everything that had been put on the internet up until June of 2021 was downloaded, then arranged in such a way that all that data could be used for model creation, resulting in over 175 billion parameters. We can think of parameters as columns on a spreadsheet, with each row being an observation or training example.

LLMs are not exactly new though, in 1954 the Georgetown-IBM experiment successfully translated 60 Russian sentences into English in the first well-known public demonstration of this type of technology8. Work on this topic stalled for some time, as computing power took time to catch up to the theoretical work of mathematicians, until 2001 when Bengio et al published their paper “A Neural Probabilistic Language Model” 1 which outlined using neural networks to process large amounts of text much faster than had previously been accomplished, but they were still hampered by technological restrictions.

That all changed in 2017 when Google researchers published their paper titled “Attention is All You Need” 2, which introduced the Transformer model. Prior to Google’s paper, neural networks like the recurrent neural network, the convolutional neural network, and the long short-term memory network, had vastly improved over the 1990’s and 2000’s, but they were still expensive to train in terms of computing time and power required. The transformer model was a new type of neural network architecture that allowed for a model to be trained in parallel using many computers which could be equipped with millions of dollars of special processing units designed specifically for AI tasks. For reference, the GPT in ChatGPT stands for “Generative Pretrained Transformer”. Researchers were, practically overnight, now able to train a model with billions of parameters in a fraction of the time it would take to train a “traditional” neural network, and that is where we find ourselves now.

ChatGPT is the fastest growing web app in human history, reaching 100 million users just 2 months after its public launch. Microsoft, their primary investor, had put hundreds of millions into the infrastructure to train the model behind the chatbot, GPT 3, but after the meteoric rise of GPT 3, they raised their investment by ten billion11 in January of 2023. Whether we like it, or are ready for it, this technology will shortly be inescapable. OpenAI has already started licensing their latest model, GPT 4, to companies all over the world who want to integrate AI services into their own products and has begun to offer a subscription service that grants users premium access to the chatbot features. ChatGPT was even used to write some portions of code for this project, in addition to our usage of the API to generate AI written text for our own model building.

In addition to OpenAI’s own classification tool, there are many prominent products that claim to be able to spot AI generated content. ZeroGPT6 and GPTZero7 (we assure you these are two different services) are two of the most notable, and though these tools appear to have improved substantially since the beginning of this project (a recent a viral reddit post showed ZeroGPT predicting that the US Constitution was written by an AI), they remain incomplete and suffer from the same problems as OpenAI’s tool. A single typo anywhere in a text, translating to a foreign language, having ChatGPT write in a foreign language and then using an online translator to go back to English, and even telling ChatGPT to write like a human, can cause the detector to classify the document as likely human written. One possible solution that OpenAI has publicly stated they are working on is a digital watermark9, which forces their model to output text with specific characteristics and patterns that make it easily detectable by tools which are trained to find that watermark.

**Experiments**

The genesis of this project was before many of the AI detection tools had been published, so we had started with the goal of creating a model that could accurately detect the difference between AI and human generated content. Between finding a dataset of 190,000 human written articles15 and blog post from the website Medium.com and beginning to use the OpenAI API14 for their GPT models to generate AI content, we believed we could at least make a competent model. However, shortly after embarking on this journey, OpenAI published their detection tool, and the results were striking. As previously mentioned, their tool correctly classifies AI text 26% of the time, but the success rate for human written text is 91% according to their post about the model. Seeing that AI written text is difficult to classify but human text is relatively easy, and operating under the assumption that any results we could achieve would be below what OpenAI did, we decided to craft some additional experiments.

Seeing posts on social media where AI detection models were fooled when ChatGPT was instructed to “write like human”, this is where we had the idea to generate AI content at different grade levels. We decided to craft a prompt where different grade levels could be easily and systematically inserted, along with other necessary information, and this prompt could be fed to ChatGPT through the python API. These AI written texts could then be used as data for model creation. That said, we still wanted to compare AI vs. human text to see what sort of results we could achieve.

**Data**

In order to generate content though, ChatGPT needs a specific prompt. We had the framework but needed to engineer a specific method. This is where the dataset of Medium.com articles became useful. Through some exploratory analysis we found that there was a large distribution of word counts in the articles. ChatGPT whether through the chatbot interface or through the python API has a limit to how many tokens, or prominent word segments, can be read into the program via a prompt or output in a response, so we had to filter the dataset by word count. Ultimately, we chose to select only articles with between 500 and 1050 words.

Chart, histogram

Description automatically generated

This gave us just over 65,000 articles. The word count was also important as we knew that for creating our model, we wanted to do paired samples. While the article title would be the basis for the content, we decided to give the word count in the prompt, as having articles of similar lengths would eliminate a potential flag. For example, if ChatGPT thinks that 6th graders write shorter texts than college students do, than our model might easily pick up on that, and be able to classify too easily. Additionally, if we wanted to pair our AI samples with our human generated text, word counts should be similar in that regard so that when the human vs. AI model was created, obvious flags like word count would still be accounted for.

Next, we decided to find out the topics of each article. Fortunately, the dataset included tags for each article, as writers on Medium.com use these in order to help people find their articles. We decided to generate content under the role of different age students, so we decided to select topics that were “appropriate” or common for US schools. While this is subjective, the variety of the Medium.com dataset was far too wide for an average student to write about. The top 20 tags shown in the below graph demonstrate this. Sixth graders are not typically writing about data science, the blockchain, or startups.

Chart, bar chart, histogram

Description automatically generated

We ended up choosing six topics: History, Statistics, American History, Geometry, African History, and Literature, and after selecting topics we were left with 1188 articles. From there we randomly selected 50 articles, because we knew we faced limits on total task size within our computational budget. We then constructed prompts based on article title and student grade level, such as:

“Write an approximately 700-word essay like a 10th grade student based on the following title: Capital Cities of Asian Countries”

Then we inserted the API prompt template into a loop and ran the loop for each article and each student level, where we chose 6th grade, 10th grade, and college. The OpenAI API is very straightforward and gave very few problems while working with it. In terms of text generation tasks, it seems to be an invaluable tool, allowing complex systems to be set up quickly and massive amounts of text to be generated with ease. The main section of the code was very short.

message = {"role": "user", "content": f"Write an approximately {word\_count} word essay like a {student} student based on the following title: {title}"}

completion = openai.ChatCompletion.create(

model="gpt-3.5-turbo",

messages=[message],

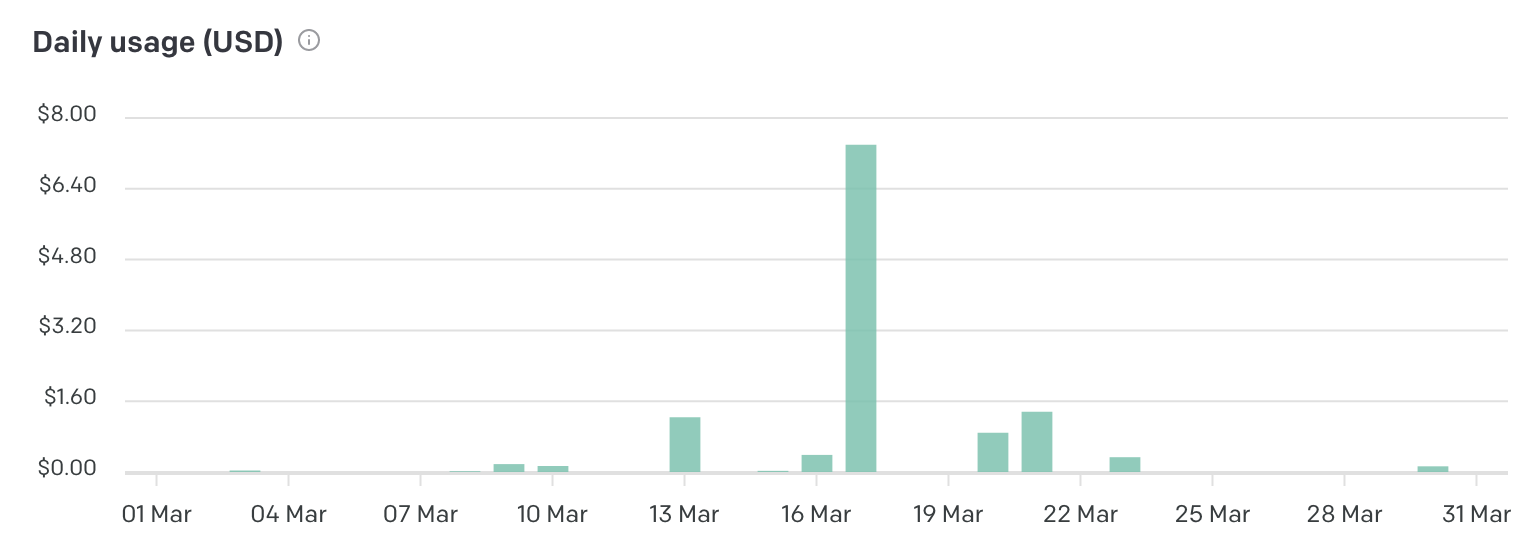
temperature=mytemperature

)

Above is the core of the code for querying the OpenAI model via python. The message variable contained the prompt, structured in the format that GPT needs, containing a role and content. There are other roles available for different purposes, but “user” was appropriate here. The word count, student, and title variables all came from our dataset and was filled in during each iteration of the loop. The model selected for our purposes was the “gpt-3.5-turbo” model. This was the latest model released at the time of this project; GPT 4 was released after we had run the bulk of our experiments.

Temperature is a variable used to determine what the style the model will write with is, lower values are more focused and coherent, where higher value are more diverse and creative but potentially less coherent. After testing different values, we ended up choosing a high value for temperature as this more closely resembled human writing. The 3.5 turbo model was able to produce surprisingly realistic content with ease, based solely an article title, grade level, and word count, with a temperature value of 0.9.

It should be noted that while using the chatbot interface, OpenAI does not charge money for its basic service. The company does offer a paid subscription which prioritizes the user’s activity with the program. However, using the API does cost money. When starting a new account, the company provided an $18 starting grant, but when generating 150 articles between 500-1000 words, we found the cost to be around $0.50. We were able to generate all data for this project within the free trial granted by OpenAI. Model creation is also a process that OpenAI charges for, but again, we were able to upload our custom dataset and train a model within our preview window. Also, a factor in our decision to focus on comparing different grade levels of AI generated text was influenced by not only the cost of generating AI content, but also by the availability of public datasets of AI content. We could not locate a dataset of all AI content, so we chose to refocus the project.



We ended up with two datasets. The first contained only examples of AI written text, with 300 examples in total, and 100 belonging to each grade level. To gather this, we used the Medium article titles to generate 150 examples of AI written text, than did the same process again using the same article titles, yielding 300 examples. For the human written text, we included the 50 articles used to generate the AI text, then randomly selected another 100 articles from the dataset. In total, we have an AI dataset with 300 examples, and a human vs. AI dataset with 450 examples.

Now that we have a dataset, albeit a small one, of AI generated content we need to explore it and see what differences we can find between the grade levels but also against human text. For these purposes we decided to calculate some simple metrics to describe text data, and we chose average sentence length, average grammatical complexity, and sentiment score. For sentence length, we split each text at each period, then count the number of white spaces between each period and average them per text. To calculate grammatical complexity we can use the part-of-speech function from the python library Natural Language Toolkit (NLTK). First we tokenize the data, which is a method of breaking down text sequences into parts and assigning numeric representations to them. When creating models in TensorFlow we also tokenize text data before training our model, as tokenization is a necessary part of almost all NLP tasks. Next we use the part-of-speech function to assign tags to complex words, count the number of complex words, and compute the average per text. Finally, to calculate sentiment we use the TextBlob package, which takes a text and computes the sentiment automatically, then assigns a score between -1.0 and 1.0.

We can visualize the data with some box and whisker charts:

Chart, box and whisker chart

Description automatically generated Chart, box and whisker chart

Description automatically generated

Chart, box and whisker chart

Description automatically generated

From the charts, we can see that there is noticeable difference, sometimes substantial, between our 3 categories of AI generated content. ChatGPT is a remarkable tool, and being able to replicate the more subtle differences of different author types is impressive. A Friedman Chi-Squared test for each metric with n=100 in each group does show that sentence length and complexity have a significant difference, while sentiment does not. The p-values calculated are: 5e-8, 1e-8, and 0.021, respectively. We are using the Friedman test as there is some concern that our samples are not truly independent of each other. While each example of AI content is generated separately from each other, each grade level group has the same prompts used for generation. That is, the 6th, 10th, and college datasets each have an essay about “Capital Cities of Asian Countries” for example.

Chart, box and whisker chart

Description automatically generatedChart, box and whisker chart

Description automatically generated

Chart, box and whisker chart

Description automatically generated

Above are the graphs for human vs all 3 grade levels of AI content in our dataset, where see even more interesting differences. Interestingly, human writing has much more variation in sentence length and grammatical complexity, but less variation in sentiment. Overall, though, we see differences we would expect to see. For human vs. AI, after a Dependent Samples T-Test test, all 3 variables show significant differences with p-values of 0.021, 0.007, and 0.013.

**Modelling**

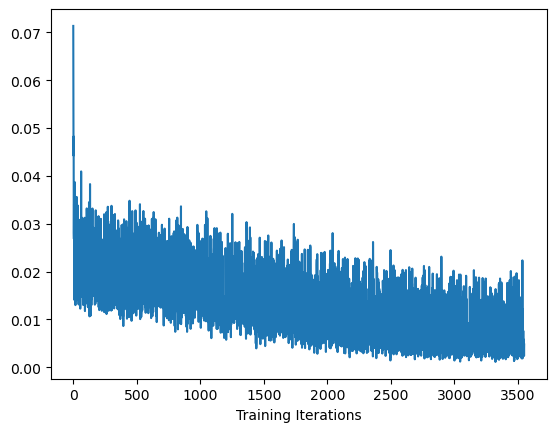
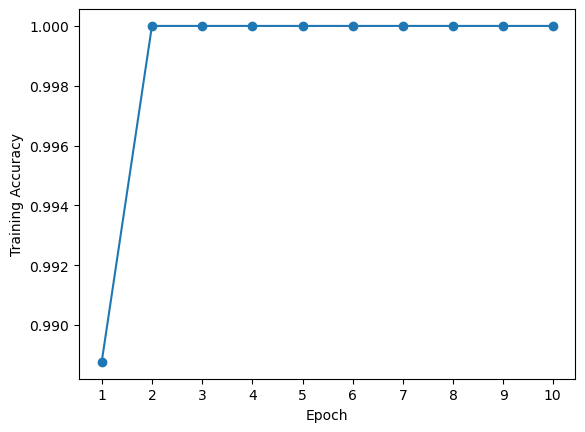
For attempting to predict the category of given text, we have chosen two possible models. The first is Fine-Tuning from OpenAI. Through their API, people can upload their own custom natural language dataset, and using the weights and biases calculated by OpenAI based on the custom dataset, create a classification model. The second model we have chosen is an LSTM, created with the python library Tensorflow. Tensorflow is an open-source python package published by Google that allows for anyone to train neural networks on their own local computers.

The first models that we created were using the OpenAI Fine-Tuning. Their API provides a convenient framework to upload data, do the pre-processing automatically, and queue your model up for training in just a few lines of code that can all be run in the console. In theory, you could even create a dataset in Microsoft excel and run the training entirely from a command prompt without python, but there are ways to execute command prompts in python, making the whole process very painless.

As with the text generation, training models using the OpenAI API is not free. For training models with around 400 examples at about 1000 tokens per example, and 10 iterations through the data, we ended up spending $0.75 per training. Again, this was covered by the introductory grant, but as the amount of data goes up, and number of iterations goes up, so does cost. With that in mind, the cost of building a computer that can compete with the GPT models could cost hundreds of millions of dollars, so for most use cases, the cost is an acceptable trade off.

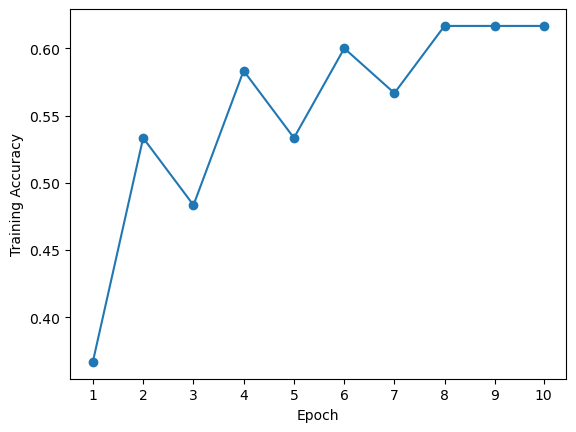
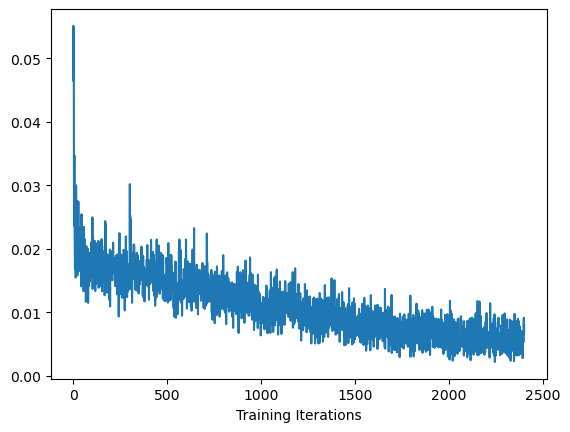
For our first set of models, we chose to go with the Fine-Tuning. This is the latest and greatest tool for NLP classification, and we can upload our own custom dataset to OpenAI’s network and create a custom model using their pre-trained GPT 3. Before we look at the results, we should remember the performance of OpenAI’s classifier tool, which correctly identified AI written text 26% of the time, and human written text 91% of the time, which means their model thought that almost everything it saw was human. This is important for interpreting our results, as we need something to compare to, however we are using a vastly different dataset which is likely much smaller, and much more specific.

Here are graphs of our model’s accuracy and loss function value during the training process, for human written text against all 3 grade levels of AI text. The accuracy data on the graph is collected at the end of each epoch, and the loss data is reported at the end of each training step, and we ran 10 epochs in total. We had trained the model consisting of different grade levels of AI text first and had arrived at 10 epochs as a good number and kept it for this model. Accuracy is on the left, with each dot marker representing an epoch, and loss on the right:



As you can see, we were able to classify with 100% accuracy! Why are we seeing results like this, though, as we know that much larger models are not this accurate? We cannot be entirely certain as Fine-Tuning models do not provide much associated data, but we can speculate. First, it likely has to do with our data, using professional and semi-professional writers to compare to an AI that is playing the role of different age students, one can fathom that the texts would be quite different, which we confirmed earlier in our examination of the data. We cannot pass explanatory variables such as grammatical complexity into the Fine-Tuning model, but Fine-Tuning models could be calculating similar variables and using them in their predictions.

Another factor here is that Fine-Tuning models are pretrained, and we are taking advantage of the fact that GPT 3 has seen almost the entire internet, potentially including our training data. GPT does not behave like a search engine specifically though, it generally can’t recall the specific text of an article from several years ago, so we are not suggesting that the model recognizes the specific articles we are using for training data. We are suggesting that it does have a vague memory of articles like ours and can rely on that to make accurate predictions in our specific case. We would show a confusion matrix of the training data, but it is not interesting, as there are no false positives or false negatives.

We had some idea that creating a model based on a limited dataset might either yield results that were very accurate or very inaccurate, this is why we decided to run another model comparing our 3 different grade levels of AI generated text. Below are graphs describing the training process in terms of accuracy and loss on the training data:  
 

For the 3-class model comparing different grade levels, our results landed at 61% accuracy. Loss was like the human vs AI classification, but our accuracy was much lower. A confusion matrix is quite helpful in visualizing the predictions here. Unlike the graphs above, the confusion matrix comes from the testing data, not the training.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Fine-Tuning** | |  |  |
|  |  | **Predicted** | | |
|  | 6th | 17 | 3 | 1 |
| **Actual** | 10th | 2 | 11 | 4 |
|  | College | 1 | 10 | 8 |
|  |  | 6th | 10th | College |

In the above chart, the darker the square, the higher the frequency of entries in that square. Correct predictions fall along the diagonal from top left to bottom right, so in the top left square, all entries located there were predicted to be 6th grade, and were indeed 6th grade, and the same goes for the middle square and the bottom right regarding 10th grade and college, respectively. So, what does this tell us? We see that 10th grade and college were often mistaken for one another, but 6th grade was very accurate. This does line up with the exploration we did earlier where we saw that for sentence length and grammatical complexity, 10th grade and college were very similar.

Since we do not know much about how exactly OpenAI’s Fine-Tuning models are created, we chose to create some custom models from scratch using the Tensorflow library. These models examine the same class predictions, the three levels of AI text, and human vs AI text, but instead of being only the text they include the metrics that we calculated earlier. One of the main benefits of using Tensorflow is its ability to be customized to any dataset, but there are tradeoffs. Fine-Tuning models can be incredible for NLP classification tasks as GPT may be the most sophisticated language model in existence, but what if we don’t want to perform classification, or we want a more transparent approach to modelling? That’s where libraries to create models on a local machine come into play. If one desires, they could even upload their code to a cloud service that offers access to special computers for training neural networks.

One of the main benefits to running models on your local computer is the ability to modify them through the adjustment of hyperparameters, which are not parameters like the columns of a dataset, but rather parameters that affect the model’s performance. These range from number of layers in a network, the types of layers, the number of neurons per layer, whether to include regularization, what type of regularization to include, to how steep the regularization penalty is. We used the hyperparameter values that performed the best from what we tested, and we ended up training over 20 TensorFlow models on each dataset, as dialing in the hyperparameters for a network is an iterative process. After each model is trained, we slightly change different hyperparameters, run the training again and then record the results. TensorFlow provides a built-in function to record and display results called TensorBoard. This logs data automatically, and results can be viewed in a premade web interface. All of the graphs shown here are the best performing models.

For our purposes, we have a small enough dataset that running it on our local computer is no problem. Getting the model up and running while mixing text and numerical data was tricky, but with some help from the ChatGPT chatbot, we were training models in a short amount of time. Starting with the human vs. AI model, our results are closer to what we expected from the Fine-Tuning.

Chart, line chart

Description automatically generated Chart

Description automatically generated

We do see an odd result here, where our validation accuracy is higher than the testing accuracy, and this is unusual. According to the literature, this type of result can arise from a few different situations. First is not having enough data12, which is a noted problem with this model. Second are techniques used to combat overfitting, such as dropout and regularization. For dropout, at any particular training step, there is a random chance specified by the user that a particular neuron will be omitted from that iteration during the calculation of weights for that layer. This causes training to be less predictable, and thus more dificult. Regularization is a technique that helps models adjust to smaller datasets by adding a penalty term to the regression equation that gets larger as individual weights get larger. This has the effect of rewarding the model for lower weight values, but it also makes training more difficult and validation “easier” by comparison13. We used both of these techniques to enhance our model to the best of our ability.

That said, a model was trained without using droupout, but otherwise using the same hyperparameters as the above model, and our accuracy did go up for both training and testing. Perhaps dropout is not necessary in this case, unless we were to see a greater amount of overfitting.

Chart, line chart, histogram

Description automatically generated Chart

Description automatically generatedß

We see a peak at 79% accuracy around 150 training iterations. In terms of loss, it is dificult to tell from the graph, but the final loss values for training and validation were 0.58 and 0.57 respectively. To give some context to accuracy values, since our the balance of our dataset was one third human written text and two thirds AI written text, that means that if we guess with no information at all, the probability of guessing correct would be two thirds. Knowing that the probability of guessing right is 2/3, we can say:

, with being a scale factor for our accuracy value to get to , and . This is more reflective of what the accuracy would be if our classes were balanced evenly, 59%, and shows that our model did achieve better than guessing.

Looking at the confusion matrix raises more questions than it answers though. Oddly, we see that ai text was rarely predicted to be human writing, almost the exact opposite of the OpenAI classification tool, and human text was often predicted to be AI generated.

Chart, treemap chart

Description automatically generated

While we can’t be entirely certain why this is happening without more research, which is beyond the scope of this project, there are few potential reasons. First, a lack of data is likely a big factor. The Fine-Tuning from OpenAI requires only very small amounts of data because it is a pre-trained model, where Tensorflow (and PyTorch, the other common deep learning python library) can require large amounts of data to create sound models from scratch as those models start with nothing. That said, we wanted to create these models locally as a sort of sanity check, i.e., was what we saw with the Fine-Tuning model what we should expect generally, or was that on outlier in terms of performance? The same philosophy was applied to the 3-class model.

Looking at the grade levels model created in Tensorflow, we see results more in line with what the Fine-Tuning model achieved. Accuracy and loss graphs show very similar results:

Chart, line chart, histogram

Description automatically generatedChart

Description automatically generated

Accuracy was at its peak around 375 epochs at approximately 56% (for context, datasets with 3 equal-size classes have a default accuracy of 33%), so accuracy for validation was substantially higher than guessing even if training was not. Loss values were low and convergent, but at a value of 1.25. We did train a model without dropout for this dataset as well, but results were nearly identical to this one, so we did not include the graphs.

While the results are not great in terms of accuracy, achieving excellent results was not the goal of this project. The confusion matrix for the validation data tells more of the story, though:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **TensorFlow** | |  |  |
|  |  | **Predicted** | | |
|  | 6th | 14 | 3 | 6 |
| **Actual** | 10th | 4 | 6 | 9 |
|  | College | 1 | 2 | 15 |
|  |  | 6th | 10th | College |

As before, correct predictions are along the diagonal, starting in the top left, and we can see that 6th grade and college both were predicted relatively well, but 10th grade writing was often misclassified as the other categories. Overall, with the 3-class model in TensorFlow, it was reassuring to see very similar results to what we saw with the Fine-Tuning model.

As a final exploration, we did take some of the content written by GPT 3 and the original article that we used a prompt and gave it to the chatbot interface and ask it whether it thought it was written by a human or not. Here are the results after asking about the AI written article:

Text

Description automatically generated

And here is the output regarding the original human-written article:

Text

Description automatically generated

**Future Work**

While this work was interesting and we learned quite a bit, there are many ways this work could be improved upon. Here are just a few of the ideas we have had about modifying these experiments:

* Create a variable importance function to run alongside TensorFlow models. Shapley values could be used, or permutation feature importance using the ELI5 library.
* Removing the text from the model and running only the numeric data (average sentence length, etc.). We would not need a neural network for this; we could use a larger variety of models such as logistic regression or gradient boosted machines. Some of these models have built in variable importance functions as well. In addition to modelling only numeric data, we could model only text, like the Fine-Tuning, but with more control via TensorFlow.
* Investigating a method for passing numeric features to the Fine-Tuning method through prompt engineering. If we can pass this data into the model, we should be able to get somewhat better results.
* Experiment with different prompts for data generation. Ours is a straightforward prompt, but in asking GPT 3 to write like a student, and write an essay, we might have been “tipping our pitches”, and this might have allowed unusually high results in the human vs AI experiments.
* Generate a lot more data, in different roles, using different sources as topics. We were limited in the amount of data we could gather and the sources we could gather it from and believe this hampered our ability to craft a good model from TensorFlow.
* Engineer more features of our text data. Burstiness, or increases and decreases of events, is a popular feature that we did not calculate for this experiment. We could also look further into ways to quantify syntax and vocabulary to see if there are additional patterns in AI writing.

It is an exciting time for AI and natural language, as the world begins to see what the LLM is truly capable of. However, we do need to proceed with caution. If ChatGPT can pass the Bar exam and help students plagiarize their homework, it might also be capable of writing malicious social media posts or crafting spam emails. Over the course of this project, we learned quite a bit about using both the chat interface and the API to interact with the GPT model, what it is capable of and what it is not. Though our results were not definitive in detecting AI content, we did learn some of what goes into the task of creating such a model, and some ways a model could be improved.

Bibliography

1. A Neural Probabilistic Language Model – Bengio et al., 2003

<https://jmlr.org/papers/volume3/tmp/bengio03a.pdf>

1. Attention is All You Need – Vaswani et al., 2017.

<https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf>

1. On the Dangers of Stochastic Parrots: Can Language Models Be Too Big? – Bender et al.,

2021

<https://dl.acm.org/doi/pdf/10.1145/3442188.3445922>

1. ChatGPT is Making Universities Rethink Plagiarism – Sofia Barnett, 2023

<https://www.wired.com/story/chatgpt-college-university-plagiarism/>

1. Latest Version of ChatGPT Aces the Bar Exam with Score Nearing 90th Percentile – Debra Weiss, 2023

<https://www.abajournal.com/web/article/latest-version-of-chatgpt-aces-the-bar-exam-with-score-in-90th-percentile>

1. ZeroGPT

<https://www.zerogpt.com/>

1. GPTZero

<https://gptzero.me/>

1. A Brief History of Large Language Models – Bob Weber, 2023

<https://www.linkedin.com/pulse/brief-history-large-language-models-bob>

1. AI Invisible Watermarking: How the ChatGPT Watermark Works – Brittany Garlin, 2023

<https://vistasocial.com/insights/ai-invisible-watermarking-how-the-chatgpt-watermark-works/>

1. New AI Classifier for Indicating AI-Written Text – Kirchner et al, 2023.

<https://openai.com/blog/new-ai-classifier-for-indicating-ai-written-text>

1. Microsoft Invests $10 Billion in ChatGPT Maker OpenAI – Dina Bass, 2023

<https://www.bloomberg.com/news/articles/2023-01-23/microsoft-makes-multibillion-dollar-investment-in-openai#xj4y7vzkg>

1. Improving Your Model’s Accuracy – Apple

<https://developer.apple.com/documentation/createml/improving-your-model-s-accuracy>

1. Image Classification Via Fine-Tuning with EfficientNet – Yixing Fu, 2020

<https://keras.io/examples/vision/image_classification_efficientnet_fine_tuning/>

1. Fine-Tuning – OpenAI, 2022

<https://platform.openai.com/docs/guides/fine-tuning>

1. 190k + Medium Articles – Fabio Chusiano, 2022

<https://www.kaggle.com/datasets/fabiochiusano/medium-articles>

Appendix

All code provided below, and at: https://github.com/aaronomics/grad

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

import datetime

import os

import openai

import ast

import random

# this is a function to find which cols are categorical and which are numerical.

# was using this during eda, but not much anymore.

def find\_cat\_num\_cols(df):

cat\_cols = []

num\_cols = []

for col in df.columns:

if df[col].dtype == object:

cat\_cols.append(col)

else:

num\_cols.append(col)

return cat\_cols, num\_cols

df = pd.read\_csv("medium\_articles.csv")

df['word\_count'] = df['text'].apply(lambda x: len(str(x).split()))

# Set a limit on the maximum word count to be included in the plot

max\_word\_count = 5000

word\_counts = df[df['word\_count'] <= max\_word\_count]['word\_count']

# Plot frequency distribution of word counts

fig, ax = plt.subplots(figsize=(8, 6))

ax.hist(word\_counts, bins=50)

ax.set\_title('Frequency Distribution of Word Counts')

ax.set\_xlabel('Word Count')

ax.set\_ylabel('Frequency')

plt.show()

# Create a new DataFrame with articles between 500 and 1050 words

df\_filtered = df.loc[(df['word\_count'] >= 500) & (df['word\_count'] <= 1050)].copy()

# Find the number of rows in the filtered dataframe

num\_articles = len(df\_filtered)

print(f"There are {num\_articles} articles with a word count between 975 and 1025.")

df\_filtered.reset\_index(drop=True, inplace=True)

df\_filtered.insert(0, 'article\_id', range(1, 1+len(df\_filtered)))

# Set a limit on the maximum word count to be included in the plot

max\_word\_count = 5000

word\_counts = df\_filtered[df\_filtered['word\_count'] <= max\_word\_count]['word\_count']

# Plot frequency distribution of word counts

fig, ax = plt.subplots(figsize=(8, 6))

ax.hist(word\_counts, bins=50)

ax.set\_title('Frequency Distribution of Word Counts of filtered DF')

ax.set\_xlabel('Word Count')

ax.set\_ylabel('Frequency')

plt.show()

# Convert string representation of lists to actual lists

#THIS CAN ONLY BE RUN ONCE AS IT ALTERS DF\_FILTERED

df\_filtered['tags'] = df\_filtered['tags'].apply(lambda x: ast.literal\_eval(x))

# Flatten the nested tag lists

tags = [tag for tags in df\_filtered['tags'] for tag in tags]

unique = list(set(tags))

unique = list(set(unique))

#print(unique)

# Create a pandas series of the tag counts

tag\_counts = pd.Series(tags).value\_counts(normalize=True)

# Get the top 20 tags and their frequency

top\_tags = tag\_counts.head(50)

top\_tags\_freq = (top\_tags \* len(df))

# Filter out tags that occur very infrequently

tag\_counts = tag\_counts[tag\_counts >= 0.001]

# Create the frequency plot

fig, ax = plt.subplots(figsize=(12, 6))

ax.bar(top\_tags.index, top\_tags\_freq, width=0.5)

ax.set\_xticklabels(top\_tags.index, rotation=90, ha='right')

ax.set\_title('Top 20 Tag Frequency')

ax.set\_xlabel('Tags')

ax.set\_ylabel('Frequency')

plt.tight\_layout()

plt.show()

df\_filtered['text'][0]

topic1 = "History"

topic2 = "Statistics"

topic3 = "American History"

topic4 = "Geometry"

topic5 = "African History"

topic6 = "Literature"

topics = [topic1, topic2, topic3, topic4, topic5, topic6]

df\_topics = df\_filtered[df\_filtered["tags"].apply(lambda x: bool(set(x) & set(topics)))]

df\_topics.shape[0] # number of rows

df\_topics

number\_samples = min(100,df\_topics.shape[0])

# first seed used was 131, second to get more training examples is 1313

myseed = 1313 # simpler than 1234 :-)

sample = df\_topics.sample(number\_samples,random\_state = myseed).copy()

sample.head()

# added tags to title list

titles = sample.loc[:, ['title', 'word\_count', 'tags']]

print(titles)

titles

#from openpyxl import Workbook

sample

sample.to\_excel("articles2.xlsx", index=False)

# number\_samples = 10

# sample = df\_topics.sample(number\_samples).copy()

# sample.head()

# titles = sample.loc[:, ['title', 'word\_count']]

# print(titles)

#api\_key = os.getenv("OPENAI\_API\_KEY")

# api\_key = "taken out for privacy" # Ross

print(type(api\_key))

#print(api\_key)

#student = "6th grade"

#student = "10th grade"

student = "college"

student\_list = ["6th grade", "10th grade","college"]

temperature = 0.9

temperature\_list = [0.9] # will generate an essay for each temperature in this list.

# Generate all combinations of student types and temperatures to use,

# basically the Cartesian product (which I sometimes get confused with Kronecker product)

# in R we do this with expand.grid

#[(x, y) for x in range(5) for y in range(5)]

combins = [(s,t) for s in student\_list for t in temperature\_list]

print(combins)

combins\_orig = combins # randomizing the order will change the original list,

# so we're saving a copy here.

### run time test

- running 5 titles, all 3 grade levels, temp = 0.9

- I want to see how long this takes to run, and the cost, then multiply by ten

- run time 12 minutes, so 120 minutes for 50 titles

- cost was $0.02

- hang on to your butts

message = {"role": "user", "content": f"Write an approximately {word\_count} word essay like a {student} student based on the following title: {title}"}

completion = openai.ChatCompletion.create(

model="gpt-3.5-turbo",

messages=[message],

temperature=mytemperature

)

### doing it for real now

- 50 titles, 3 grade levels, temp = 0.9, fingers = crossed

- didn't quite work, API timed out?

- going to try again before I do anything drastic

### current version

This version outputs a single excel file and continuously updates it as new output is generated.

import os.path

# Specify the filename for the Excel file

date\_string = now2.strftime("%Y-%m-%d-%H-%M")

filename = f"responses\_{date\_string}.xlsx"

# Create an empty DataFrame for the responses

responses\_all = pd.DataFrame()

# Check if the Excel file exists

if os.path.isfile(filename):

# If the file exists, load it into a DataFrame

responses\_all = pd.read\_excel(filename)

nrows\_expected = titles.shape[0] \* len(combins)

for index, row in titles.iterrows():

title = row['title']

word\_count = row['word\_count']

tags = str(row['tags'])

combins = combins\_orig.copy()

random.seed(index)

random.shuffle(combins)

# loop through those combinations:

for indx2, row2 in enumerate(combins):

student = row2[0]

mytemperature = row2[1]

message = {"role": "user", "content": f"Write an approximately {word\_count} word essay like a {student} student based on the following title: {title}"}

print(message)

now = datetime.datetime.now()

print("now=", now)

completion = openai.ChatCompletion.create(

model="gpt-3.5-turbo",

messages=[message],

temperature=mytemperature#,

#max\_tokens=500

)

now2 = datetime.datetime.now()

duration = now2 - now

print(f"duration={duration}")

response = completion["choices"][0]["message"]["content"]

responses = pd.DataFrame({

"response": [response],

"title": [title],

"tags": [tags],

"temperature": [temperature],

"input\_word\_count": [word\_count],

"student": [student],

"created": [completion["created"]],

"id": [completion["id"]],

"model": [completion["model"]],

"object": [completion["object"]],

"completion\_tokens": [completion["usage"]["completion\_tokens"]],

"prompt\_tokens": [completion["usage"]["prompt\_tokens"]],

"total\_tokens": [completion["usage"]["total\_tokens"]],

'word\_count': [len(str(response).split())],

'count\_delta': [len(str(response).split()) - word\_count],

"duration": [duration.total\_seconds()],

"start\_time": [now]

})

responses\_all = pd.concat([responses\_all, responses], ignore\_index=True)

i += 1

mean\_dur\_so\_far = np.mean(responses\_all["duration"])

expected\_remaining\_dur = (nrows\_expected-i) \* mean\_dur\_so\_far

print(f"mean\_dur\_so\_far={mean\_dur\_so\_far}, expected\_remaining\_dur={expected\_remaining\_dur}")

# Perhaps write out the Excel file so far.

# We do this outside the student & temperature loop,

# so we only ever write out data on whole groups of student & temperature combinations.

if( index % 1 == 0): #write out the Excel file, as a "checkpoint"

# Write the DataFrame to the Excel file

responses\_all.to\_excel(filename, index=False)

# Get the current date and time

now = datetime.datetime.now()

# Format the date and time as a string

date\_string = now.strftime("%Y-%m-%d-%H-%M")

# Generate the filename with the current date and time

filename = f"responses\_{date\_string}.xlsx"

# Write the DataFrame to an Excel file

responses.to\_excel(filename, index=False)

completion

import time

# Define retry parameters

max\_retries = 5

retry\_delay = 10 # seconds

# Initialize progress tracker

progress = ...

for i in range(...):

retries = 0

while True:

try:

# Make API request

response = openai.api\_function(...)

# Process response

...

# Update progress tracker

progress = ...

if i % save\_interval == 0:

# Save progress periodically

...

# Break out of retry loop if successful

break

except APIConnectionError as e:

# Handle connection error

print(f"API connection error occurred: {e}")

# Check if max retries reached

if retries >= max\_retries:

raise e

# Wait and retry request

print(f"Retrying in {retry\_delay} seconds...")

time.sleep(retry\_delay)

retries += 1

# Save progress after loop finishes

with open('progress.json', 'w') as f:

json.dump(progress, f)

import openai

openai.api\_key = "invalid\_key" # set an invalid API key

try:

response = openai.Completion.create(engine="davinci", prompt="test")

# process the response here

except openai.error.APIConnectionError as e:

handle\_api\_error(e)

import pandas as pd

import datetime

import time

import random

import openai

#from openai.api\_resources import APIConnectionError

#from requests.exceptions import APIConnectionError

#from openai import

#from requests.exceptions import

# Define a function to handle API errors

def handle\_api\_error(e):

print(f"Got API error: {e}")

print("Waiting 60 seconds before trying again...")

time.sleep(60)

# Define a function to handle timeouts

def handle\_timeout(e):

print(f"Got timeout error: {e}")

print("Waiting 60 seconds before trying again...")

time.sleep(60)

def handle\_ratelimit(e):

print(f"Got ratelimit error: {e}")

print("Waiting 60 seconds before trying again...")

time.sleep(60)

def handle\_error(e):

print(f"Got error: {e}")

print("Waiting 60 seconds before trying again...")

time.sleep(60)

# Define a function to save the responses DataFrame to a file

def save\_responses(responses, index):

# Get the current date and time

now = datetime.datetime.now()

# Format the date and time as a string

date\_string = now.strftime("%Y-%m-%d-%H-%M")

# Generate the filename with the current date and time and the current index

filename = f"responses\_{date\_string}\_index{index}.xlsx"

# Write the DataFrame to an Excel file

responses.to\_excel(filename, index=False)

## I need to be able to only run what is necessary, everytime I have to stop it resets the kernel.

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

import datetime

import os

import openai

import ast

import random

import time

def find\_cat\_num\_cols(df):

cat\_cols = []

num\_cols = []

for col in df.columns:

if df[col].dtype == object:

cat\_cols.append(col)

else:

num\_cols.append(col)

return cat\_cols, num\_cols

df = pd.read\_csv("medium\_articles.csv")

df['word\_count'] = df['text'].apply(lambda x: len(str(x).split()))

# Set a limit on the maximum word count to be included in the plot

max\_word\_count = 5000

word\_counts = df[df['word\_count'] <= max\_word\_count]['word\_count']

# Create a new DataFrame with articles between 500 and 1050 words

df\_filtered = df.loc[(df['word\_count'] >= 500) & (df['word\_count'] <= 1050)].copy()

df\_filtered.reset\_index(drop=True, inplace=True)

df\_filtered.insert(0, 'article\_id', range(1, 1+len(df\_filtered)))

# Set a limit on the maximum word count to be included in the plot

max\_word\_count = 5000

word\_counts = df\_filtered[df\_filtered['word\_count'] <= max\_word\_count]['word\_count']

#THIS CAN ONLY BE RUN ONCE AS IT ALTERS DF\_FILTERED

df\_filtered['tags'] = df\_filtered['tags'].apply(lambda x: ast.literal\_eval(x))

# Flatten the nested tag lists

tags = [tag for tags in df\_filtered['tags'] for tag in tags]

unique = list(set(tags))

unique = list(set(unique))

# Create a pandas series of the tag counts

tag\_counts = pd.Series(tags).value\_counts(normalize=True)

# Get the top 20 tags and their frequency

top\_tags = tag\_counts.head(50)

top\_tags\_freq = (top\_tags \* len(df))

# Filter out tags that occur very infrequently

tag\_counts = tag\_counts[tag\_counts >= 0.001]

topic1 = "History"

topic2 = "Statistics"

topic3 = "American History"

topic4 = "Geometry"

topic5 = "African History"

topic6 = "Literature"

topics = [topic1, topic2, topic3, topic4, topic5, topic6]

df\_topics = df\_filtered[df\_filtered["tags"].apply(lambda x: bool(set(x) & set(topics)))]

####

number\_samples = min(50,df\_topics.shape[0])

myseed = 131 # simpler than 1234 :-)

sample = df\_topics.sample(number\_samples,random\_state = myseed).copy()

sample.head()

# added tags to title list

titles = sample.loc[:, ['title', 'word\_count', 'tags']]

#print(titles.head())

student\_list = ["6th grade", "10th grade","college"]

temperature = 0.9

temperature\_list = [0.9] # will generate an essay for each temperature in this list.

#[(x, y) for x in range(5) for y in range(5)]

combins = [(s,t) for s in student\_list for t in temperature\_list]

#print(combins)

combins\_orig = combins

# Define a function to handle API errors

def handle\_api\_error(e):

print(f"Got API error: {e}")

print("Waiting 60 seconds before trying again...")

time.sleep(60)

# Define a function to handle timeouts

def handle\_timeout(e):

print(f"Got timeout error: {e}")

print("Waiting 60 seconds before trying again...")

time.sleep(60)

def handle\_ratelimit(e):

print(f"Got ratelimit error: {e}")

print("Waiting 60 seconds before trying again...")

time.sleep(60)

def handle\_error(e):

print(f"Got error: {e}")

print("Waiting 60 seconds before trying again...")

time.sleep(60)

# Define a function to save the responses DataFrame to a file

def save\_responses(responses, index):

# Get the current date and time

now = datetime.datetime.now()

# Format the date and time as a string

date\_string = now.strftime("%Y-%m-%d-%H-%M")

# Generate the filename with the current date and time and the current index

filename = f"responses\_{date\_string}\_index{index}.xlsx"

# Write the DataFrame to an Excel file

responses.to\_excel(filename, index=False)

print("################################done################################")

start\_row = 108

stop\_row = 10000

#api\_key = os.getenv("OPENAI\_API\_KEY")

#api\_key = # Ross

print(type(api\_key))

#print(api\_key)

# Set up the initial state

responses = pd.DataFrame()

i = 0

nrows\_expected = titles.shape[0] \* len(combins)

# Loop through the titles

for index, row in titles.iterrows():

# if index < start\_row or index >= stop\_row:

# continue

title = row['title']

word\_count = row['word\_count']

tags = str(row['tags'])

combins = combins\_orig.copy()

random.seed(index)

random.shuffle(combins)

# loop through those combinations:

for indx2, row2 in enumerate(combins):

student = row2[0]

mytemperature = row2[1]

message = {"role": "user", "content": f"Write an approximately {word\_count} word essay like a {student} student based on the following title: {title}"}

print(message)

now = datetime.datetime.now()

print("now=", now)

# Define a flag to indicate whether we successfully completed the loop

success = False

# Loop until we successfully complete the api call

while not success:

try:

completion = openai.ChatCompletion.create(

model="gpt-3.5-turbo",

messages=[message],

temperature=mytemperature,

#max\_tokens=500

)

response = completion["choices"][0]["message"]["content"]

# If we successfully completed the api call, set the success flag to True

success = True

except openai.error.APIConnectionError as e:

handle\_api\_error(e)

except TimeoutError as e:

handle\_timeout(e)

except openai.error.RateLimitError as e:

handle\_ratelimit(e)

except:

print("Unknown error")

print("Waiting 60 seconds before trying again...")

time.sleep(60)

now2 = datetime.datetime.now()

duration = now2 - now

print(f"duration={duration}")

response = completion["choices"][0]["message"]["content"]

# Add the response to the responses DataFrame

responses.loc[i, "response"] = response

responses.loc[i, "title"] = title

responses.loc[i, "tags"] = tags

responses.loc[i, "temperature"] = temperature

responses.loc[i, "input\_word\_count"] = word\_count

responses.loc[i, "student"] = student

responses.loc[i,"created"] = completion["created"]

responses.loc[i,"id"] = completion["id"]

responses.loc[i,"model"] = completion["model"]

responses.loc[i,"object"] = completion["object"]

responses.loc[i,"completion\_tokens"] = completion["usage"]["completion\_tokens"]

responses.loc[i,"prompt\_tokens"] = completion["usage"]["prompt\_tokens"]

# added total tokens here

responses.loc[i,"total\_tokens"] = completion["usage"]["total\_tokens"]

responses.loc[i,'word\_count'] = len(str(response).split())

responses.loc[i,'count\_delta'] = responses.loc[i,'word\_count'] - responses.loc[i,'input\_word\_count']

responses.loc[i, "duration"] = duration.total\_seconds()

responses.loc[i, "start\_time"] = now

i += 1

mean\_dur\_so\_far = np.mean(responses["duration"])

expected\_remaining\_dur = (nrows\_expected-i) \* mean\_dur\_so\_far

print(f"mean\_dur\_so\_far={mean\_dur\_so\_far}, expected\_remaining\_dur={expected\_remaining\_dur}")

# Perhaps write out the Excel file so far.

# We do this outside the student & temperature loop,

# so we only ever write out data on whole groups of student & temperature combinations.

if( index % 1 == 0): #write out the Excel file, as a "checkpoint"

# Get the current date and time

# now = datetime.datetime.now()

# Format the date and time as a string

date\_string = now2.strftime("%Y-%m-%d-%H-%M")

# Generate the filename with the current date and time

filename = f"responses\_{date\_string}.xlsx"

# Write the DataFrame to an Excel file

responses.to\_excel(filename, index=False)

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import matplotlib

import seaborn as sns

import nltk

from textblob import TextBlob

df = pd.read\_excel("3class.xlsx")

df2 = pd.read\_excel("human\_ai.xlsx")

def calculate\_metrics(df):

"""

Calculates the average sentence length, average grammatical complexity, and average sentiment

score for each row of text in the input DataFrame.

Parameters:

df (Pandas DataFrame): A DataFrame with a single column of text

Returns:

Pandas DataFrame: A new DataFrame with the columns 'average\_sentence\_length', 'average\_grammatical\_complexity',

and 'average\_sentiment'

"""

# Define a function to calculate the POS tag count for a given sentence

def pos\_tag\_count(sentence):

pos\_tags = nltk.pos\_tag(nltk.word\_tokenize(sentence))

tag\_count = len(pos\_tags)

return tag\_count

# Define a function to calculate the sentiment score for a given sentence

def sentiment\_score(sentence):

blob = TextBlob(sentence)

score = blob.sentiment.polarity

return score

# Tokenize the text into sentences

sentences = df['prompt'].apply(nltk.sent\_tokenize)

# Calculate the average sentence length for each row

df['average\_sentence\_length'] = sentences.apply(lambda x: sum(len(sentence.split()) for sentence in x)/len(x))

# Calculate the average POS tag count for each row

df['average\_grammatical\_complexity'] = sentences.apply(lambda x: sum(pos\_tag\_count(sentence) for sentence in x)/len(x))

# Calculate the average sentiment score for each row

df['average\_sentiment'] = sentences.apply(lambda x: sum(sentiment\_score(sentence) for sentence in x)/len(x))

# Drop the 'sentences' column

#df = df.drop('sentences', axis=1)

return df[['average\_sentence\_length', 'average\_grammatical\_complexity', 'average\_sentiment']]

metrics\_df = calculate\_metrics(df)

df

import matplotlib.pyplot as plt

import pandas as pd

import seaborn as sns

def generate\_boxplots(dataframe, category\_column):

numeric\_cols = dataframe.select\_dtypes(include=['float64', 'int64'])

category\_values = ['6thgrade', '10thgrade', 'college'] # define desired order of labels

sns.set\_style("whitegrid")

for col in numeric\_cols.columns:

fig, ax = plt.subplots(figsize=(5, 3))

title = col.replace("\_", " ").title()

ax.set\_title(title)

boxplot\_data = []

for category\_value in category\_values:

label = category\_value.replace("grade", "") # remove "grade" from label

label = label.capitalize() # capitalize first letter of label

category\_data = dataframe[dataframe[category\_column] == category\_value][col]

boxplot\_data.append(category\_data)

bp = ax.boxplot(boxplot\_data, patch\_artist=True, positions=range(len(category\_values)))

for patch in bp['boxes']:

patch.set\_facecolor('#99ccee')

ax.set\_xticks(range(len(category\_values)))

ax.set\_xticklabels([label for label in category\_values]) # set labels in desired order

plt.show()

generate\_boxplots(df, 'completion')

def generate\_boxplots2(dataframe, category\_column):

numeric\_cols = dataframe.select\_dtypes(include=['float64', 'int64'])

category\_values = ['human', 'ai'] # define desired order of labels

sns.set\_style("whitegrid")

for col in numeric\_cols.columns:

fig, ax = plt.subplots(figsize=(5, 3))

title = col.replace("\_", " ").title()

ax.set\_title(title)

boxplot\_data = []

for category\_value in category\_values:

label = category\_value.replace("grade", "") # remove "grade" from label

label = label.capitalize() # capitalize first letter of label

category\_data = dataframe[dataframe[category\_column] == category\_value][col]

boxplot\_data.append(category\_data)

bp = ax.boxplot(boxplot\_data, patch\_artist=True, positions=range(len(category\_values)))

for patch in bp['boxes']:

patch.set\_facecolor('#99ccee')

ax.set\_xticks(range(len(category\_values)))

ax.set\_xticklabels([label for label in category\_values]) # set labels in desired order

plt.show()

metrics\_df2 = calculate\_metrics(df2)

df2

generate\_boxplots2(df2, 'completion')

import scipy.stats as stats

def perform\_ranksum(dataframe, category\_column):

# Create a list to hold the data for each category

category\_data = []

for category\_value in dataframe[category\_column].unique():

category\_data.append(dataframe[dataframe[category\_column] == category\_value])

# Extract the data from numerical columns in each category data frame

numerical\_columns = dataframe.select\_dtypes(include=['float64', 'int64']).columns

data = [[df[col] for col in numerical\_columns] for df in category\_data]

# Perform the rank sum test for each numerical column and category

results = {}

for i, col in enumerate(numerical\_columns):

groups = [cat\_data[i] for cat\_data in data]

if len(groups) == 2:

result = stats.ranksums(groups[0], groups[1])

else:

result = stats.kruskal(\*groups)

results[col] = result

return results

ranksum = perform\_ranksum(df, 'completion')

print(ranksum)

ranksum2 = perform\_ranksum(df2, 'completion')

print(ranksum2)

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import matplotlib

import seaborn as sns

import nltk

from textblob import TextBlob

df = pd.read\_excel("3class.xlsx")

df2 = pd.read\_excel("human\_ai.xlsx")

def calculate\_metrics(df):

"""

Calculates the average sentence length, average grammatical complexity, and average sentiment

score for each row of text in the input DataFrame.

Parameters:

df (Pandas DataFrame): A DataFrame with a single column of text

Returns:

Pandas DataFrame: A new DataFrame with the columns 'average\_sentence\_length', 'average\_grammatical\_complexity',

and 'average\_sentiment'

"""

# Define a function to calculate the POS tag count for a given sentence

def pos\_tag\_count(sentence):

pos\_tags = nltk.pos\_tag(nltk.word\_tokenize(sentence))

tag\_count = len(pos\_tags)

return tag\_count

# Define a function to calculate the sentiment score for a given sentence

def sentiment\_score(sentence):

blob = TextBlob(sentence)

score = blob.sentiment.polarity

return score

# Tokenize the text into sentences

sentences = df['prompt'].apply(nltk.sent\_tokenize)

# Calculate the average sentence length for each row

df['average\_sentence\_length'] = sentences.apply(lambda x: sum(len(sentence.split()) for sentence in x)/len(x))

# Calculate the average POS tag count for each row

df['average\_grammatical\_complexity'] = sentences.apply(lambda x: sum(pos\_tag\_count(sentence) for sentence in x)/len(x))

# Calculate the average sentiment score for each row

df['average\_sentiment'] = sentences.apply(lambda x: sum(sentiment\_score(sentence) for sentence in x)/len(x))

# Drop the 'sentences' column

#df = df.drop('sentences', axis=1)

return df[['average\_sentence\_length', 'average\_grammatical\_complexity', 'average\_sentiment']]

metrics\_df = calculate\_metrics(df)

df

import matplotlib.pyplot as plt

import pandas as pd

import seaborn as sns

def generate\_boxplots(dataframe, category\_column):

numeric\_cols = dataframe.select\_dtypes(include=['float64', 'int64'])

category\_values = ['6thgrade', '10thgrade', 'college'] # define desired order of labels

sns.set\_style("whitegrid")

for col in numeric\_cols.columns:

fig, ax = plt.subplots(figsize=(5, 3))

title = col.replace("\_", " ").title()

ax.set\_title(title)

boxplot\_data = []

for category\_value in category\_values:

label = category\_value.replace("grade", "") # remove "grade" from label

label = label.capitalize() # capitalize first letter of label

category\_data = dataframe[dataframe[category\_column] == category\_value][col]

boxplot\_data.append(category\_data)

bp = ax.boxplot(boxplot\_data, patch\_artist=True, positions=range(len(category\_values)))

for patch in bp['boxes']:

patch.set\_facecolor('#99ccee')

ax.set\_xticks(range(len(category\_values)))

ax.set\_xticklabels([label for label in category\_values]) # set labels in desired order

plt.show()

generate\_boxplots(df, 'completion')

def generate\_boxplots2(dataframe, category\_column):

numeric\_cols = dataframe.select\_dtypes(include=['float64', 'int64'])

category\_values = ['human', 'ai'] # define desired order of labels

sns.set\_style("whitegrid")

for col in numeric\_cols.columns:

fig, ax = plt.subplots(figsize=(5, 3))

title = col.replace("\_", " ").title()

ax.set\_title(title)

boxplot\_data = []

for category\_value in category\_values:

label = category\_value.replace("grade", "") # remove "grade" from label

label = label.capitalize() # capitalize first letter of label

category\_data = dataframe[dataframe[category\_column] == category\_value][col]

boxplot\_data.append(category\_data)

bp = ax.boxplot(boxplot\_data, patch\_artist=True, positions=range(len(category\_values)))

for patch in bp['boxes']:

patch.set\_facecolor('#99ccee')

ax.set\_xticks(range(len(category\_values)))

ax.set\_xticklabels([label for label in category\_values]) # set labels in desired order

plt.show()

metrics\_df2 = calculate\_metrics(df2)

df2

generate\_boxplots2(df2, 'completion')

import scipy.stats as stats

def perform\_ranksum(dataframe, category\_column):

# Create a list to hold the data for each category

category\_data = []

for category\_value in dataframe[category\_column].unique():

category\_data.append(dataframe[dataframe[category\_column] == category\_value])

# Extract the data from numerical columns in each category data frame

numerical\_columns = dataframe.select\_dtypes(include=['float64', 'int64']).columns

data = [[df[col] for col in numerical\_columns] for df in category\_data]

# Perform the rank sum test for each numerical column and category

results = {}

for i, col in enumerate(numerical\_columns):

groups = [cat\_data[i] for cat\_data in data]

if len(groups) == 2:

result = stats.ranksums(groups[0], groups[1])

else:

result = stats.kruskal(\*groups)

results[col] = result

return results

ranksum = perform\_ranksum(df, 'completion')

print(ranksum)

ranksum2 = perform\_ranksum(df2, 'completion')

print(ranksum2)

### list of ft models

- 'ada:ft-personal-2023-03-13-22-18-36'

- 3 class, initial attempt

- "ada:ft-personal-2023-03-17-16-36-40"

- also 3 class, but I regarded this one as better

- "ada:ft-personal-2023-03-17-17-21-10"

- labels with spaces removed

- "ada:ft-personal-2023-03-17-17-49-37"

- data shuffled now

- "curie:ft-personal-2023-03-17-18-08-07"

- trained on curie to see of results better

- "ada:ft-personal-2023-03-20-00-09-00"

- went up to 10 epochs and back to ada

ft\_model = "ada:ft-personal-2023-03-20-00-09-00"

import pandas as pd

import openai

import os

from sklearn.metrics import confusion\_matrix

import matplotlib.pyplot as plt

import numpy as np

api\_key = os.getenv("OPENAI\_API\_KEY")

print(type(api\_key))

openai.api\_key = api\_key

data = pd.read\_excel("data.xlsx")

labels = data['student']

texts = data['response']

df = pd.DataFrame(zip(texts, labels), columns = ['prompt','completion']) #[:300]

df.head()

df.to\_excel("3class.xlsx")

df.to\_json("essays.jsonl", orient='records', lines=True)

# !openai tools fine\_tunes.prepare\_data -f essays.jsonl -q

# !openai api fine\_tunes.create -t "essays\_prepared\_train.jsonl" -v "essays\_prepared\_valid.jsonl" --compute\_classification\_metrics --classification\_positive\_class " 10th grade" -m ada

# !openai api fine\_tunes.results -i ft-B29wadYuk9CqMvVvKJKQ4P5u > student\_result5.csv

results = pd.read\_csv('student\_result5.csv')

results\_filtered = results[results['classification/accuracy'].notnull()].tail(10)

print(results\_filtered)

print(results\_filtered[results\_filtered['classification/accuracy'].notnull()].tail(1))

results\_filtered[results\_filtered['classification/accuracy'].notnull()]['classification/accuracy'].plot(marker='o')

# Add x and y axis labels

plt.xlabel('Epoch')

plt.ylabel('Training Accuracy')

plt.xticks(ticks = [241, 482, 723, 963, 1204, 1445, 1685, 1925, 2166, 2400],labels= range(1, 11))

plt.show()

plt.xlabel('Epoch')

plt.ylabel('Training Loss')

test = pd.read\_json('students2\_prepared\_valid.jsonl', lines=True)

test.head()

for i in range(len(test)):

res = openai.Completion.create(model=ft\_model, prompt=test['prompt'][i] + '\n\n###\n\n', max\_tokens=1, temperature=0, logprobs=2)

test.loc[i, 'prediction'] = res['choices'][0]['text']

#test.loc[i, 'log\_probs'] = res['choices'][0]['logprobs']['top\_logprobs'][0]

print(test.head(20))

classes = list(test['prediction'].unique())

test['completion'] = test['completion'].str.replace('thgrade', '')

y\_actu = pd.Series(test['completion'].values, name='Actual')

y\_pred = pd.Series(test['prediction'].values, name='Predicted')

def plot\_confusion\_matrix(df\_confusion, title='Confusion matrix', cmap=plt.cm.gray\_r):

plt.matshow(df\_confusion, cmap=cmap) # imshow

#plt.title(title)

plt.colorbar()

tick\_marks = np.arange(len(df\_confusion.columns))

plt.xticks(tick\_marks, df\_confusion.columns, rotation=45)

plt.yticks(tick\_marks, df\_confusion.index)

#plt.tight\_layout()

plt.ylabel(df\_confusion.index.name)

plt.xlabel(df\_confusion.columns.name)

df\_confusion = pd.crosstab(y\_actu, y\_pred)

plot\_confusion\_matrix(df\_confusion)

def plot\_confusion\_matrix(df\_confusion, title='Confusion matrix', cmap=plt.cm.Blues):

plt.imshow(df\_confusion, interpolation='nearest', cmap=cmap)

plt.title(title)

plt.colorbar()

tick\_marks = np.arange(len(df\_confusion.columns))

plt.xticks(tick\_marks, df\_confusion.columns, rotation=45)

plt.yticks(tick\_marks, df\_confusion.index)

fmt = 'd'

thresh = df\_confusion.values.max() / 2.

for i, j in itertools.product(range(df\_confusion.shape[0]), range(df\_confusion.shape[1])):

plt.text(j, i, format(df\_confusion.values[i, j], fmt),

horizontalalignment="center",

color="white" if df\_confusion.values[i, j] > thresh else "black")

plt.tight\_layout()

plt.ylabel(df\_confusion.index.name)

plt.xlabel(df\_confusion.columns.name)

import itertools

classes = list(test['prediction'].unique())

test['completion'] = test['completion'].str.replace('thgrade', '')

y\_actu = pd.Series(test['completion'].values, name='Actual')

y\_pred = pd.Series(test['prediction'].values, name='Predicted')

df\_confusion = pd.crosstab(y\_actu, y\_pred)

plot\_confusion\_matrix(df\_confusion)

- "ada:ft-personal-2023-03-17-19-05-55"

- "ada:ft-personal-2023-03-21-18-34-25"

- I think this is the final model

ft\_model = "ada:ft-personal-2023-03-21-18-34-25"

import pandas as pd

import openai

import os

from sklearn.metrics import confusion\_matrix

import matplotlib.pyplot as plt

import numpy as np

api\_key = os.getenv("OPENAI\_API\_KEY")

print(type(api\_key))

openai.api\_key = api\_key

df = pd.read\_excel("human\_ai.xlsx")

df.to\_json("ai.jsonl", orient='records', lines=True)

# !openai tools fine\_tunes.prepare\_data -f ai.jsonl -q

# !openai api fine\_tunes.create -t "ai\_prepared\_train.jsonl" -v "ai\_prepared\_valid.jsonl" -m ada --compute\_classification\_metrics --classification\_positive\_class " human" --n\_epochs 10

# !openai api fine\_tunes.results -i ft-B29wadYuk9CqMvVvKJKQ4P5u > student\_result5.csv

results = pd.read\_csv('ai\_result2.csv')

print(results)

print(results[results['classification/accuracy'].notnull()].tail(1))

results[results['classification/accuracy'].notnull()]['classification/accuracy'].plot()

import matplotlib.pyplot as plt

import matplotlib.pyplot as plt

results\_filtered = results[results['classification/accuracy'].notnull()].tail(10)

print(results\_filtered)

print(results\_filtered[results\_filtered['classification/accuracy'].notnull()].tail(1))

results\_filtered[results\_filtered['classification/accuracy'].notnull()]['classification/accuracy'].plot(marker='o')

# Add x and y axis labels

plt.xlabel('Epoch')

plt.ylabel('Training Accuracy')

plt.xticks(ticks = [355, 711, 1067, 1422, 1778, 2134, 2489, 2845, 3200, 3550],labels= range(1, 11))

plt.show()# Add x and y axis labels

plt.xlabel('Epoch')

plt.ylabel('Training Accuracy')

plt.show()

results[results['training\_loss'].notnull()]['training\_loss'].plot()

plt.xlabel('Epoch')

plt.ylabel('Training Loss')

test = pd.read\_json('ai\_prepared\_valid.jsonl', lines=True)

test.head()

classes = list(test['completion'].unique())

print(classes)

for i in range(len(test)):

res = openai.Completion.create(model=ft\_model, prompt=test['prompt'][i] + '\n\n###\n\n', max\_tokens=1, temperature=0, logprobs=2)

test.loc[i, 'prediction'] = res['choices'][0]['text']

#test.loc[i, 'log\_probs'] = res['choices'][0]['logprobs']['top\_logprobs'][0]

print(test.head(20))

classes = list(test['prediction'].unique())

classes

test['prediction'] = test['prediction'].str.replace('a', 'ai')

y\_actu = pd.Series(test['completion'].values, name='Actual')

y\_pred = pd.Series(test['prediction'].values, name='Predicted')

def plot\_confusion\_matrix(df\_confusion, title='Confusion matrix', cmap=plt.cm.gray\_r):

plt.matshow(df\_confusion, cmap=cmap) # imshow

#plt.title(title)

plt.colorbar()

tick\_marks = np.arange(len(df\_confusion.columns))

plt.xticks(tick\_marks, df\_confusion.columns, rotation=45)

plt.yticks(tick\_marks, df\_confusion.index)

#plt.tight\_layout()

plt.ylabel(df\_confusion.index.name)

plt.xlabel(df\_confusion.columns.name)

df\_confusion = pd.crosstab(y\_actu, y\_pred)

print(df\_confusion)

plot\_confusion\_matrix(df\_confusion)

import pandas as pd

import nltk

from textblob import TextBlob

import tensorflow as tf

from tensorflow.keras.layers import Input, Embedding, Dense, Concatenate, GlobalAveragePooling1D

from tensorflow.keras.models import Model

from tensorflow.keras.preprocessing.text import Tokenizer

from tensorflow.keras.preprocessing.sequence import pad\_sequences

from sklearn.preprocessing import LabelEncoder

from sklearn.model\_selection import train\_test\_split

import numpy as np

from tensorflow.keras.layers import Input, Embedding, LSTM, Dense, Concatenate, Dropout

from tensorflow.keras import regularizers

from tensorflow.keras.optimizers import Adam

from tensorflow.keras.callbacks import TensorBoard

import datetime

nltk.download('averaged\_perceptron\_tagger')

def calculate\_metrics(df):

"""

Calculates the average sentence length, average grammatical complexity, and average sentiment

score for each row of text in the input DataFrame.

Parameters:

df (Pandas DataFrame): A DataFrame with a single column of text

Returns:

Pandas DataFrame: A new DataFrame with the columns 'average\_sentence\_length', 'average\_grammatical\_complexity',

and 'average\_sentiment'

"""

# Define a function to calculate the POS tag count for a given sentence

def pos\_tag\_count(sentence):

pos\_tags = nltk.pos\_tag(nltk.word\_tokenize(sentence))

tag\_count = len(pos\_tags)

return tag\_count

# Define a function to calculate the sentiment score for a given sentence

def sentiment\_score(sentence):

blob = TextBlob(sentence)

score = blob.sentiment.polarity

return score

# Tokenize the text into sentences

sentences = df['prompt'].apply(nltk.sent\_tokenize)

# Calculate the average sentence length for each row

df['average\_sentence\_length'] = sentences.apply(lambda x: sum(len(sentence.split()) for sentence in x)/len(x))

# Calculate the average POS tag count for each row

df['average\_grammatical\_complexity'] = sentences.apply(lambda x: sum(pos\_tag\_count(sentence) for sentence in x)/len(x))

# Calculate the average sentiment score for each row

df['average\_sentiment'] = sentences.apply(lambda x: sum(sentiment\_score(sentence) for sentence in x)/len(x))

# Drop the 'sentences' column

#df = df.drop('sentences', axis=1)

return df[['average\_sentence\_length', 'average\_grammatical\_complexity', 'average\_sentiment']]

df2 = pd.read\_excel("3class.xlsx")

df2

# Calculate the metrics

metrics\_df2 = calculate\_metrics(df2)

df2

print(list(df2.columns))

## all ai text model, 6th, 10th, college

# Load the data from the CSV file

# data = pd.read\_csv("path/to/csv")

data = df2.copy()

# Split the data into training and testing sets

train\_data, test\_data = train\_test\_split(data, test\_size=0.2, random\_state=42)

# Encode the labels as integers

label\_encoder = LabelEncoder()

train\_labels = label\_encoder.fit\_transform(train\_data["completion"])

test\_labels = label\_encoder.transform(test\_data["completion"])

# Convert the data to numpy arrays

train\_prompt = np.array(train\_data["prompt"])

train\_numerical = np.array(train\_data[["average\_sentence\_length", "average\_grammatical\_complexity", "average\_sentiment"]])

test\_prompt = np.array(test\_data["prompt"])

test\_numerical = np.array(test\_data[["average\_sentence\_length", "average\_grammatical\_complexity", "average\_sentiment"]])

print(train\_data['completion'])

for elem in train\_labels:

print(elem)

# Tokenize the text data

tokenizer = Tokenizer(num\_words=10000, oov\_token="<OOV>")

tokenizer.fit\_on\_texts(train\_prompt)

train\_sequences = tokenizer.texts\_to\_sequences(train\_prompt)

test\_sequences = tokenizer.texts\_to\_sequences(test\_prompt)

# Pad the sequences to be the same length

max\_length = 100

train\_padded = pad\_sequences(train\_sequences, maxlen=max\_length, padding="post", truncating="post")

test\_padded = pad\_sequences(test\_sequences, maxlen=max\_length, padding="post", truncating="post")

# Define the inputs for the model

input\_text = Input(shape=(max\_length,))

input\_numerical = Input(shape=(3,))

# Embed the text data

embedding = Embedding(input\_dim=10000, output\_dim=128)(input\_text)

lstm1 = LSTM(64, kernel\_regularizer=regularizers.l1(0.055))(embedding)

lstm2 = LSTM(16)(embedding)

# Combine the text

combined = Concatenate()([lstm1, input\_numerical])

# combined = Concatenate()([lstm1])

# Add some dropout for regularization

dropout = Dropout(0.0)(combined)

# Add a dense output layer

output = Dense(len(label\_encoder.classes\_), activation="softmax")(dropout)

# Define the model

model = Model(inputs=[input\_text, input\_numerical], outputs=output)

# model = Model(inputs=[input\_text], outputs=output)

learning\_rate = 0.001

batch\_size = 132

optimizer = Adam(learning\_rate=learning\_rate)

# Compile the model

model.compile(loss="sparse\_categorical\_crossentropy", optimizer=optimizer, metrics=["accuracy"])

log\_dir = "logs/" + datetime.datetime.now().strftime("%Y%m%d-%H%M%S")

tensorboard\_callback = TensorBoard(log\_dir=log\_dir, histogram\_freq=1)

# Train the model

history = model.fit([train\_padded, train\_numerical], train\_labels, validation\_data=([test\_padded, test\_numerical], test\_labels), epochs=500, batch\_size=batch\_size, callbacks=[tensorboard\_callback])

# history = model.fit([train\_padded], train\_labels, validation\_data=([test\_padded], test\_labels), epochs=500, batch\_size=batch\_size, callbacks=[tensorboard\_callback])

import matplotlib.pyplot as plt

import pandas as pd

# convert the accuracy data to a Pandas Series

train\_acc = pd.Series(history.history["accuracy"])

val\_acc = pd.Series(history.history["val\_accuracy"])

# compute a rolling average with a window size of 10

window\_size = 20

train\_acc\_smooth = train\_acc.rolling(window\_size).mean()

val\_acc\_smooth = val\_acc.rolling(window\_size).mean()

# plot the smoothed data

plt.plot(train\_acc\_smooth)

plt.plot(val\_acc\_smooth)

plt.title("Model accuracy")

plt.xlabel("Epoch")

plt.ylabel("Accuracy")

plt.legend(["Train", "Validation"], loc="upper left")

plt.show()

# Plot the training and validation loss

plt.plot(history.history["loss"])

plt.plot(history.history["val\_loss"])

plt.title("Model loss")

plt.xlabel("Epoch")

plt.ylabel("Loss")

plt.legend(["Train", "Validation"], loc="upper left")

plt.show()

import numpy as np

from sklearn.metrics import confusion\_matrix

# Make predictions on the training set

train\_pred = model.predict([train\_padded, train\_numerical])

train\_pred\_classes = np.argmax(train\_pred, axis=1)

# Create the confusion matrix for the training set

train\_cm = confusion\_matrix(train\_labels, train\_pred\_classes)

print("Training confusion matrix:")

print(train\_cm)

# Make predictions on the test set

test\_pred = model.predict([test\_padded, test\_numerical])

test\_pred\_classes = np.argmax(test\_pred, axis=1)

# Create the confusion matrix for the test set

test\_cm = confusion\_matrix(test\_labels, test\_pred\_classes)

print("Test confusion matrix:")

print(test\_cm)

# Evaluate the model on the testing data

test\_loss, test\_acc = model.evaluate([test\_padded, test\_numerical], test\_labels)

# Print the testing accuracy

print("Test accuracy:", test\_acc)

import numpy as np

from sklearn.metrics import confusion\_matrix

import matplotlib.pyplot as plt

import seaborn as sns

class\_labels = ['10th', '6th', 'college']

# Define function to create and style confusion matrix

def plot\_confusion\_matrix(cm, class\_labels):

plt.figure(figsize=(5,5))

sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')

plt.title('Confusion matrix')

plt.ylabel('True label')

plt.xlabel('Predicted label')

tick\_marks = np.arange(len(class\_labels))

plt.xticks(tick\_marks, class\_labels)

plt.yticks(tick\_marks, class\_labels)

plt.show()

# Make predictions on the training set

train\_pred = model.predict([train\_padded, train\_numerical])

train\_pred\_classes = np.argmax(train\_pred, axis=1)

# Create the confusion matrix for the training set

train\_cm = confusion\_matrix(train\_labels, train\_pred\_classes)

print("Training confusion matrix:")

print(train\_cm)

# Plot styled confusion matrix for training set

plot\_confusion\_matrix(train\_cm, class\_labels)

# Make predictions on the test set

test\_pred = model.predict([test\_padded, test\_numerical])

test\_pred\_classes = np.argmax(test\_pred, axis=1)

# Create the confusion matrix for the test set

test\_cm = confusion\_matrix(test\_labels, test\_pred\_classes)

print("Test confusion matrix:")

print(test\_cm)

# Plot styled confusion matrix for test set

plot\_confusion\_matrix(test\_cm, class\_labels)

import numpy as np

from sklearn.metrics import confusion\_matrix

import matplotlib.pyplot as plt

import seaborn as sns

# Define function to create and style confusion matrix

def plot\_confusion\_matrix(cm, labels):

plt.figure(figsize=(5,5))

sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')

plt.title('Confusion matrix')

plt.ylabel('True label')

plt.xlabel('Predicted label')

tick\_marks = np.arange(len(labels))

plt.xticks(tick\_marks, labels, rotation=45)

plt.yticks(tick\_marks, labels)

plt.show()

# Get unique labels in your data

labels = np.unique(train\_data['completion'])

# Make predictions on the training set

train\_pred = model.predict([train\_padded, train\_numerical])

train\_pred\_classes = np.argmax(train\_pred, axis=1)

# Create the confusion matrix for the training set

train\_cm = confusion\_matrix(train\_labels, train\_pred\_classes)

print("Training confusion matrix:")

print(train\_cm)

# Plot styled confusion matrix for training set

plot\_confusion\_matrix(train\_cm, labels)

# Make predictions on the test set

test\_pred = model.predict([test\_padded, test\_numerical])

test\_pred\_classes = np.argmax(test\_pred, axis=1)

# Create the confusion matrix for the test set

test\_cm = confusion\_matrix(test\_labels, test\_pred\_classes)

print("Test confusion matrix:")

print(test\_cm)

# Plot styled confusion matrix for test set

plot\_confusion\_matrix(test\_cm, labels)

import pandas as pd

import nltk

import numpy as np

from textblob import TextBlob

import tensorflow as tf

from tensorflow.keras.layers import Input, Embedding, Dense, Concatenate, GlobalAveragePooling1D, LSTM, Dropout

from tensorflow.keras.models import Model

from tensorflow.keras.preprocessing.text import Tokenizer

from tensorflow.keras.preprocessing.sequence import pad\_sequences

from tensorflow.keras import regularizers

from tensorflow.keras.optimizers import Adam

from sklearn.preprocessing import LabelEncoder

from sklearn.model\_selection import train\_test\_split

from tensorflow.keras.callbacks import TensorBoard

import datetime

def calculate\_metrics(df):

"""

Calculates the average sentence length, average grammatical complexity, and average sentiment

score for each row of text in the input DataFrame.

Parameters:

df (Pandas DataFrame): A DataFrame with a single column of text

Returns:

Pandas DataFrame: A new DataFrame with the columns 'average\_sentence\_length', 'average\_grammatical\_complexity',

and 'average\_sentiment'

"""

# Define a function to calculate the POS tag count for a given sentence

def pos\_tag\_count(sentence):

pos\_tags = nltk.pos\_tag(nltk.word\_tokenize(sentence))

tag\_count = len(pos\_tags)

return tag\_count

# Define a function to calculate the sentiment score for a given sentence

def sentiment\_score(sentence):

blob = TextBlob(sentence)

score = blob.sentiment.polarity

return score

# Tokenize the text into sentences

sentences = df['prompt'].apply(nltk.sent\_tokenize)

# Calculate the average sentence length for each row

df['average\_sentence\_length'] = sentences.apply(lambda x: sum(len(sentence.split()) for sentence in x)/len(x))

# Calculate the average POS tag count for each row

df['average\_grammatical\_complexity'] = sentences.apply(lambda x: sum(pos\_tag\_count(sentence) for sentence in x)/len(x))

# Calculate the average sentiment score for each row

df['average\_sentiment'] = sentences.apply(lambda x: sum(sentiment\_score(sentence) for sentence in x)/len(x))

# Drop the 'sentences' column

#df = df.drop('sentences', axis=1)

return df[['average\_sentence\_length', 'average\_grammatical\_complexity', 'average\_sentiment']]

df = pd.read\_excel("human\_ai.xlsx")

metrics\_df = calculate\_metrics(df)

df

train\_labels

train\_data

# Load the data from the CSV file

# data = pd.read\_csv("path/to/csv")

data = df.copy()

# Split the data into training and testing sets

train\_data, test\_data = train\_test\_split(data, test\_size=0.2, random\_state=42)

# Encode the labels as integers

label\_encoder = LabelEncoder()

train\_labels = label\_encoder.fit\_transform(train\_data["completion"])

test\_labels = label\_encoder.transform(test\_data["completion"])

# Convert the data to numpy arrays

train\_prompt = np.array(train\_data["prompt"])

train\_numerical = np.array(train\_data[["average\_sentence\_length", "average\_grammatical\_complexity", "average\_sentiment"]])

test\_prompt = np.array(test\_data["prompt"])

test\_numerical = np.array(test\_data[["average\_sentence\_length", "average\_grammatical\_complexity", "average\_sentiment"]])

# Tokenize the text data

tokenizer = Tokenizer(num\_words=10000, oov\_token="<OOV>")

tokenizer.fit\_on\_texts(train\_prompt)

train\_sequences = tokenizer.texts\_to\_sequences(train\_prompt)

test\_sequences = tokenizer.texts\_to\_sequences(test\_prompt)

# Pad the sequences to be the same length

max\_length = 100

train\_padded = pad\_sequences(train\_sequences, maxlen=max\_length, padding="post", truncating="post")

test\_padded = pad\_sequences(test\_sequences, maxlen=max\_length, padding="post", truncating="post")

# Define the inputs for the model

input\_text = Input(shape=(max\_length,))

input\_numerical = Input(shape=(3,))

# Embed the text data

embedding = Embedding(input\_dim=10000, output\_dim=128)(input\_text)

lstm1 = LSTM(64, kernel\_regularizer=regularizers.l1(0.01))(embedding)

lstm2 = LSTM(16)(embedding)

# Combine the text

combined = Concatenate()([lstm1, input\_numerical])

# Add some dropout for regularization

dropout = Dropout(0.0)(combined)

# Add a dense output layer

output = Dense(len(label\_encoder.classes\_), activation="softmax")(dropout)

# output = Dense(len(label\_encoder.classes\_), activation="softmax")

# Define the model

model = Model(inputs=[input\_text, input\_numerical], outputs=output)

learning\_rate = 0.001

batch\_size = 32

optimizer = Adam(learning\_rate=learning\_rate)

# Compile the model

model.compile(loss="sparse\_categorical\_crossentropy", optimizer=optimizer, metrics=["accuracy"])

log\_dir = "logs/" + datetime.datetime.now().strftime("%Y%m%d-%H%M%S")

tensorboard\_callback = TensorBoard(log\_dir=log\_dir, histogram\_freq=1)

# Train the model

history = model.fit([train\_padded, train\_numerical], train\_labels, validation\_data=([test\_padded, test\_numerical], test\_labels), epochs=500, batch\_size=batch\_size, callbacks=[tensorboard\_callback])

import matplotlib.pyplot as plt

import pandas as pd

# convert the accuracy data to a Pandas Series

train\_acc = pd.Series(history.history["accuracy"])

val\_acc = pd.Series(history.history["val\_accuracy"])

# compute a rolling average with a window size of 10

window\_size = 20

train\_acc\_smooth = train\_acc.rolling(window\_size).mean()

val\_acc\_smooth = val\_acc.rolling(window\_size).mean()

# plot the smoothed data

plt.plot(train\_acc\_smooth)

plt.plot(val\_acc\_smooth)

plt.title("Model accuracy")

plt.xlabel("Epoch")

plt.ylabel("Accuracy")

plt.legend(["Train", "Validation"], loc="upper left")

plt.show()

# Plot the training and validation loss

plt.plot(history.history["loss"])

plt.plot(history.history["val\_loss"])

plt.title("Model loss")

plt.xlabel("Epoch")

plt.ylabel("Loss")

plt.legend(["Train", "Validation"], loc="upper left")

plt.show()

import numpy as np

from sklearn.metrics import confusion\_matrix

# Make predictions on the training set

train\_pred = model.predict([train\_padded, train\_numerical])

train\_pred\_classes = np.argmax(train\_pred, axis=1)

# Create the confusion matrix for the training set

train\_cm = confusion\_matrix(train\_labels, train\_pred\_classes)

print("Training confusion matrix:")

print(train\_cm)

# Make predictions on the test set

test\_pred = model.predict([test\_padded, test\_numerical])

test\_pred\_classes = np.argmax(test\_pred, axis=1)

# Create the confusion matrix for the test set

test\_cm = confusion\_matrix(test\_labels, test\_pred\_classes)

print("Test confusion matrix:")

print(test\_cm)

import numpy as np

from sklearn.metrics import confusion\_matrix

import matplotlib.pyplot as plt

import seaborn as sns

class\_labels = ['ai', 'human']

def plot\_confusion\_matrix(cm, class\_labels):

plt.figure(figsize=(5,5))

ax = sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', annot\_kws={"ha": 'center',"va": 'center'})

plt.title('Confusion matrix')

plt.ylabel('True label')

plt.xlabel('Predicted label')

tick\_marks = np.arange(len(class\_labels))

plt.xticks(tick\_marks, class\_labels)

ax.set\_yticklabels(class\_labels, va='center')

ax.set\_xticklabels(class\_labels, ha='center')

plt.show()

# Make predictions on the training set

train\_pred = model.predict([train\_padded, train\_numerical])

train\_pred\_classes = np.argmax(train\_pred, axis=1)

# Create the confusion matrix for the training set

train\_cm = confusion\_matrix(train\_labels, train\_pred\_classes)

print("Training confusion matrix:")

print(train\_cm)

# Plot styled confusion matrix for training set

plot\_confusion\_matrix(train\_cm, class\_labels)

# Make predictions on the test set

test\_pred = model.predict([test\_padded, test\_numerical])

test\_pred\_classes = np.argmax(test\_pred, axis=1)

# Create the confusion matrix for the test set

test\_cm = confusion\_matrix(test\_labels, test\_pred\_classes)

print("Test confusion matrix:")

print(test\_cm)

# Plot styled confusion matrix for test set

plot\_confusion\_matrix(test\_cm, class\_labels)

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import matplotlib

import seaborn as sns

import nltk

from textblob import TextBlob

import scipy.stats as stats

from statsmodels.stats.anova import AnovaRM

import random

def calculate\_metrics(df):

"""

Calculates the average sentence length, average grammatical complexity, and average sentiment

score for each row of text in the input DataFrame.

Parameters:

df (Pandas DataFrame): A DataFrame with a single column of text

Returns:

Pandas DataFrame: A new DataFrame with the columns 'average\_sentence\_length', 'average\_grammatical\_complexity',

and 'average\_sentiment'

"""

# Define a function to calculate the POS tag count for a given sentence

def pos\_tag\_count(sentence):

pos\_tags = nltk.pos\_tag(nltk.word\_tokenize(sentence))

tag\_count = len(pos\_tags)

return tag\_count

# Define a function to calculate the sentiment score for a given sentence

def sentiment\_score(sentence):

blob = TextBlob(sentence)

score = blob.sentiment.polarity

return score

# Tokenize the text into sentences

sentences = df['prompt'].apply(nltk.sent\_tokenize)

# Calculate the average sentence length for each row

df['average\_sentence\_length'] = sentences.apply(lambda x: sum(len(sentence.split()) for sentence in x)/len(x))

# Calculate the average POS tag count for each row

df['average\_grammatical\_complexity'] = sentences.apply(lambda x: sum(pos\_tag\_count(sentence) for sentence in x)/len(x))

# Calculate the average sentiment score for each row

df['average\_sentiment'] = sentences.apply(lambda x: sum(sentiment\_score(sentence) for sentence in x)/len(x))

# Drop the 'sentences' column

#df = df.drop('sentences', axis=1)

return df[['average\_sentence\_length', 'average\_grammatical\_complexity', 'average\_sentiment']]

df = pd.read\_excel("3class.xlsx")

metrics = calculate\_metrics(df)

df = df.drop(columns=['prompt'])

df

import pandas as pd

from scipy.stats import friedmanchisquare

# Load your data into a DataFrame

#df = pd.read\_csv('data.csv')

# Define your variables

completion = 'completion'

metrics = ['average\_sentence\_length', 'average\_grammatical\_complexity', 'average\_sentiment']

# Perform the Friedman test for each metric

for metric in metrics:

data = []

for level in df[completion].unique():

data.append(df[df[completion]==level][metric].values)

stat, p = friedmanchisquare(\*data)

print(f'Friedman test for {metric}: chi-squared={stat:.3f}, p={p:.10f}')

df2 = pd.read\_excel("human\_ai.xlsx")

metrics2 = calculate\_metrics(df2)

df2

# Get 50 random entries from each category

human\_data = df2[df2['completion'] == 'human'].sample(n=50, random\_state=42)

ai\_data = df2[df2['completion'] == 'ai'].sample(n=50, random\_state=42)

# Combine the two dataframes

df\_new = pd.concat([human\_data, ai\_data])

# Perform dependent samples t-test between categories 1 and 2

t\_stat1, p\_val1 = stats.ttest\_rel(df\_new[df\_new['completion'] == 'human']['average\_sentence\_length'],

df\_new[df\_new['completion'] == 'ai']['average\_sentence\_length'])

# Perform dependent samples t-test between categories 1 and 3

t\_stat2, p\_val2 = stats.ttest\_rel(df\_new[df\_new['completion'] == 'human']['average\_grammatical\_complexity'],

df\_new[df\_new['completion'] == 'ai']['average\_grammatical\_complexity'])

# Perform dependent samples t-test between categories 2 and 3

t\_stat3, p\_val3 = stats.ttest\_rel(df\_new[df\_new['completion'] == 'human']['average\_sentiment'],

df\_new[df\_new['completion'] == 'ai']['average\_sentiment'])

# Print results

print('T-statistic for average\_sentence\_length:', t\_stat1)

print('P-value for average\_sentence\_length:', p\_val1)

print('T-statistic for caverage\_grammatical\_complexity:', t\_stat2)

print('P-value for average\_grammatical\_complexity:', p\_val2)

print('T-statistic for average\_sentiment:', t\_stat3)

print('P-value for average\_sentiment:', p\_val3)