Deep Learning Classifier for LLM identification based on “truncated text completion”

CSE 584 MidTerm Project
By
Wahid Uz Zaman

Project Group: Wahid Uz Zaman (Member-1)

Project Description:

Given a set of truncated texts, for each piece of text xi, such as “Yesterday I went”, different Large Language Models (LLMs) are asked to complete it by appending xj =”to Costco and purchased a floor cleaner.” so a complete text will be generated like “Yesterday I went to Costco and purchased a floor cleaner.” From each LLM, the same xi leads to different xj. Now I need to build a deep learning classifier to figure out, for each input (xi, xj), which LLM was used for this pair.

Data Generation/Curation:

For data points, first I need to get some truncated text. I have used the meta Llama-7B model for generating 3000 truncated text. I used some possible starters of the truncate sentence “possible_starters = ["While", "If", "However", "Although", "Since", "In order to", "Yesterday", "Once", "Because"]”. By using max_new_token, I ensured that model generates a few words to make a truncated text.

Then I used 5 different models from 5 different vendors to complete these 3000 truncated texts. I used online inferences from Gemini-1.5-flash(Google), Mistral-small, gpt-3.5-turbo-instruct through respective APIs. The others are Llama-7B (Meta), distilgpt2 (HuggingFace) that I use offline (download the model locally).

I have created 5 python scripts to complete truncated texts("llama_prefixes.csv") using each of these models. All of these are in github repo https://github.com/wzz5219/CSE-584/tree/main/MidtermProject/textcompletion .

But while generating the texts, some LLMs starts from the prefixes, some starts generating after the prefixes. So, I needed to modify the generated LLM output to separate prefix and suffix whenever needed. For some models, I changed the prompt as well so that LLM can provide better suffix output. Even if I used max_new_tokens for each of the LLM’s outputs, sometimes it generates more than a sentence. Sometimes, it generates \n multiple times in the output. So, I needed to handle these things as well.

What I did is that I replaced all \n to "," and then I tried to use stop characters (like periods) to get the suffixes. From each of the LLM models, I got suffix texts for all 3000 prefixes. Then I saved each file with the <model_name>. csv. Also, the header is “Prefix | Suffix”; I used this “|” as CSV separator as comma(,) can be easily generated in the suffix part, and it’d be hard to separate prefix and suffix in the CSD file. I have put all the data files inside ‘data/’ directory. https://github.com/wzz5219/CSE-584/tree/main/MidtermProject/data
So, my total data point is 15000 (3000 prefix text * 5 models)
For all of my experiments, i am going to use train-text-split randomly with 20% as the test size ratio. So, my training set = 12000, testing size = 3000.

In the next 2 sections, I am going to tell about my classifier and the results. I will not put any dedicated sections for “in-depth analysis” as my classifier and result section should cover detailed analysis.

Classifier

For classifier design, i used an incremental approach, and I created 3 classifiers.
SimpleDlClassifier: First, I created a simple feedforward neural network as my classifier (using Pytorch). This has 7 fully connected layers. Input Layer’s size is set to 150. The output layer’s output is mapped to the number of LLMs, which is 5. ReLU (Rectified Linear Unit) is used in each of the layer except last layer. After playing with this simple NN, I decided that I do not need to use dropout as this model is not showing overfitting at all. I am not sure, but it may be due to small training sample (12000). For optimizer, I used Pytorch’s **AdamW** [a variant of the **Adam** (Adaptive Moment Estimation)]. The learning rate is set to 2e-5. For the dataset preparation before feeding to the model, (both for training and testing), I created a TextDataset class that handles tokenization and labeling. Each sample has a prefix and a suffix part that is combined into a single input string, of max_length 150. For this simple classifier, i used simple character encoding (ASCII values) to encode the input string, and the labels are converted to tensors. Another thing is that even though the labels are LLM _names, I mapped these names into integers (0 to 5) during the train-test set split. The training process involves iterating over the training dataset in batches (batch size = 16). For each batch, training goes through **forward-pass, loss-calculation**, **backward-pass, and optimizer-step**. For each training pass, I checked train loss of current pass and accuracy

of the model (on the train data itself). This helps to understand how many passes I needed to improve model performance.

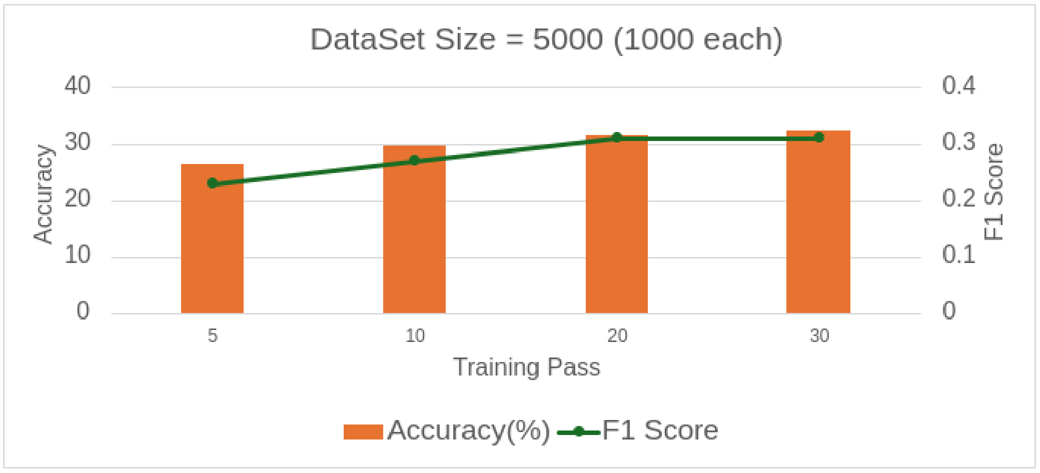
SimpleDlClassifier_withTokenizer: After i got my SimpleDL Classifier, I thought about improving the tokenizer. Previously used simple character encoding may not get context of the words, so I checked with a pretrained BERT tokenizer. It should do much better for this task, which requires understanding of language context, semantics, and so on. So, in this classifier, I just changed my tokenizer. But I didn't change the NNmodel.

DLClassifier:
Here, I used the BertForSequenceClassification model from the Hugging Face Transformers library, which is specifically designed for LLM classification based on prefix-suffix based texts. This model utilizes pretrained weights from the 'bert-base-uncased' variant, enabling it to leverage extensive language representations learned from a large corpus of text. I have configured it to operate with 5 output labels based on my classification task, and I set the hidden dropout probability to 0.1. So, the fundamental difference here is that unlike the other 2, now, this model has pretrained weights. So, training this model with training dataset is going to take much time. But its likely that this model should perform better in identifying LLMs.

Experiments and Results:

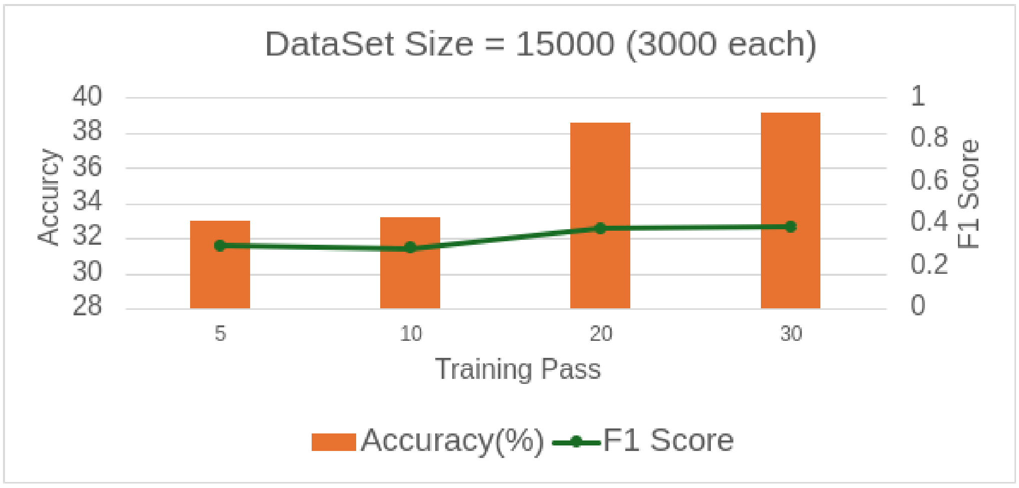
For each of the experiments, I generate train and test datasets by combining all the 5 datasets first, and then randomly select 80% data for train set and 20% data for test size. I used random_state=100 for the train-test split to make the split reproducible across all the runs. For statistics, I logged overall accuracy, per LLM accuracy, recall, precision, and F1 score. But in this result section, I mostly show accuracy and F1 score.
(**How I run this classifiers is mentioned here: https://github.com/wzz5219/CSE-584/tree/main/MidtermProject#readme)

1. First, I tried to check the impact of data samples on performance. I just ran SimpleDLClassifier with 5000 total data points.



I found maximum accuracy of 31% in this case. In the above graph, I plotted accuracy found based on number of training pass. I get 4% accuracy improvement when I feed the training sample 10 times instead of 5 times. But increasing training passes more than 10 does not help.

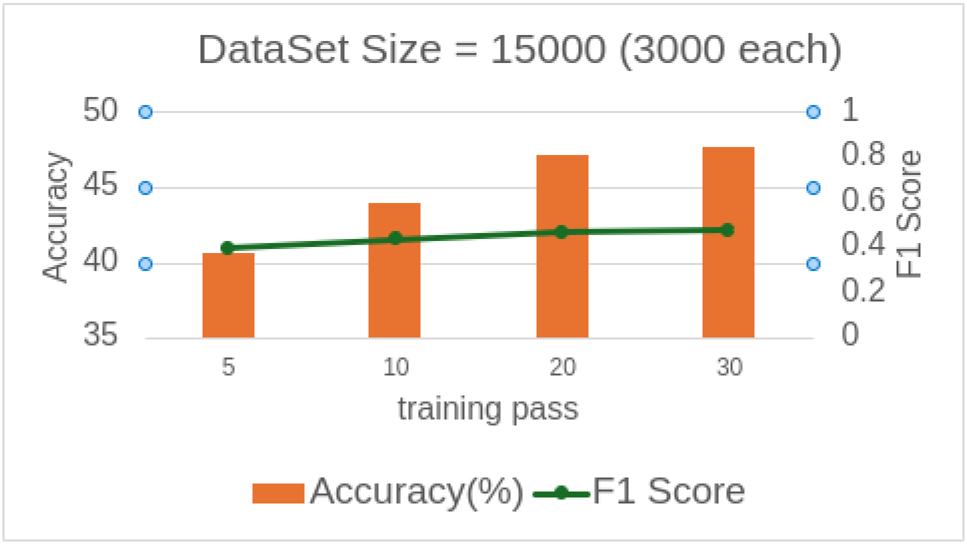
Then I increased dataset size to max 15000. The below graph shows the result



When datapoints are increased, model performance gets better. At this point, I got nearly 40% of accuracy. Also, even 5 training passes for this case produce a better result than previous experiments with fewer data points.

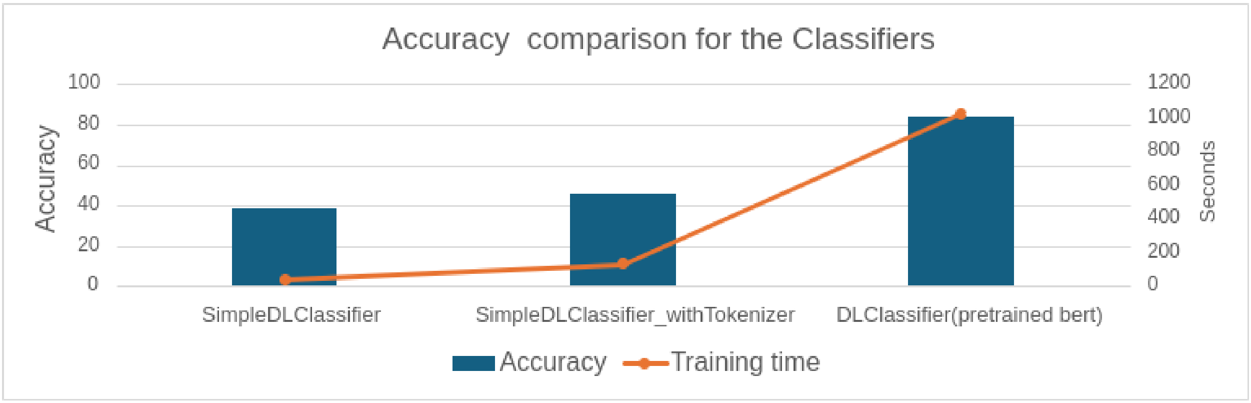
2. As i can not increase data points due to resource limitations, I then tried with improved classifier , SimpleDLClassifier_withTokoneizer, for the same 15000 datapoints. With that

classifier, I got this result shown in below graph.



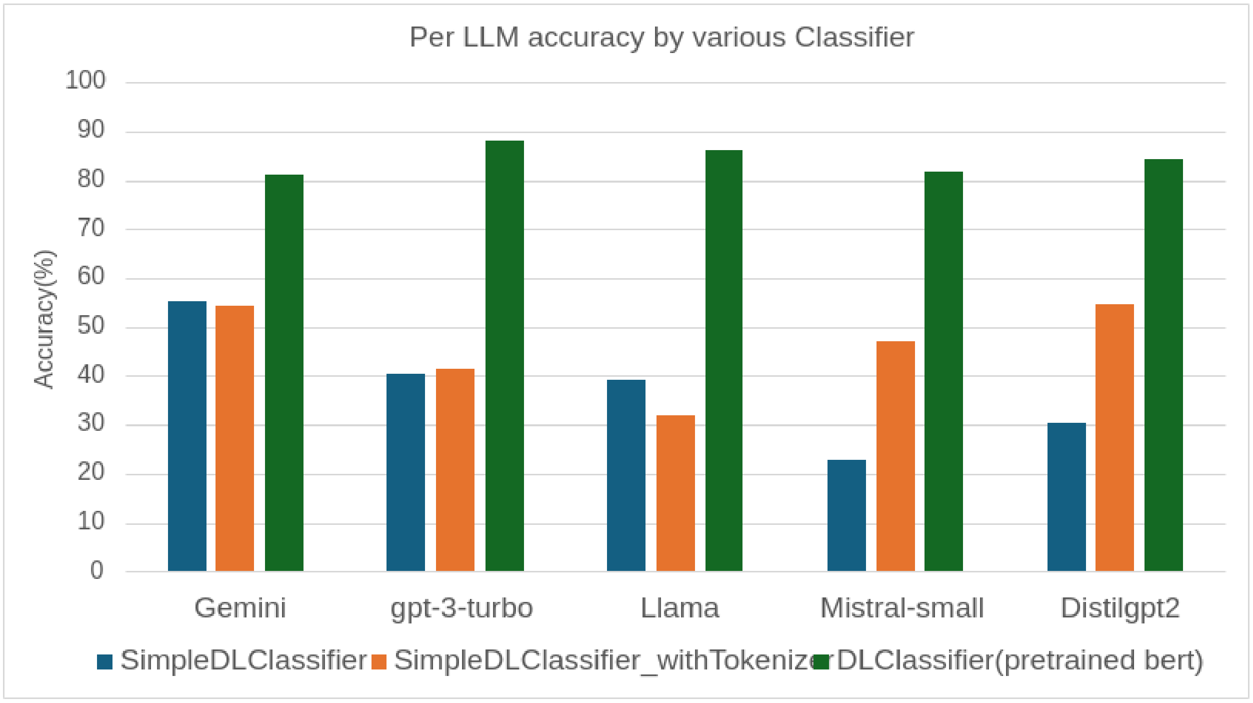
At this point, I got nearly 48% accuracy. Also, even 5 training passes for this case produce a better result than the SimpleDLClassifier experiments. Moreover, increasing training passes could not improve performance.

3. Then I used pretrained BERT model for the same 15000 datapoints. I got much improvement this time that is nearly 80%. But this time each training pass takes a lot of time. This is obvious as this model is pretrained and large (440 MB).



In the above plotted figure, the best performances of all of these classifiers are shown. Also, it shows training time as well.

4. I needed to check per-LLM accuracy by each of these classifiers.



SimpleDLClassifier_withTokenizer improved Mistral and Distillgpt2 LLM accuracy more than SimpleDLClassifier. But DLClassifier performs best for all LLM identification, as shown in the graph.

** All the run that produced these graphs are in the Readme file https://github.com/wzz5219/CSE-584/blob/main/MidtermProject/README.md

Related Works:

There are many works going on distinguishing LLM-generated texts within themselves or even from human-generated texts. Watermarking is such a technique. Here, I mention this paper, "DeepTextMark: A Deep Learning-Driven Text Watermarking Approach for Identifying Large Language Model Generated Text," authored by Travis Munyer et al. This showed a watermarking based approach to detect LLM-generated texts. It subtly inserts a watermark into the generated text during the LLM generation process. It uses Word2Vec and Sentence Encoding techniques to identify appropriate words for substitution, ensuring that the watermark remains undetectable to human readers. A transformer-based classifier is then trained to detect the watermark's presence in the text, effectively distinguishing LLM-generated text from human-generated text. Another paper that I want to mention here is “DetectGPT: Zero-Shot Machine-Generated Text Detection using Probability Curvature" (https://proceedings.mlr.press/v202/mitchell23a.html. This paper does not necessarily use any classifiers; rather, it uses a probabilistic approach to detect LLM-generated text. They observed a property of the structure of an LLM’s probability function that is useful for such detection. Leveraging this observation, they defined a new curvature-based criterion for detecting whether a passage is generated from a given LLM.