CS-584 Final Project: Period Placement Model

Wyatt Blair

Stevens Institute of Technology, Hoboken, NJ, wblair@stevens.edu

December 16, 2024

Abstract

An open issue with Natural Language Processing is the dynamic addition of punctuation to sentences. In general, this problem has been solved by Large Language Models; however, Large Language Models cost either money or significant computational power to utilize properly, making it a difficult solution to scale with. This paper takes a hybrid approach through which a specific behavior of a Large Language Model can be learned from a lighter-weight Natural Language Processing model.

1 Annotating the Transcript

1.1 Using ChatGPT

In order to build a model capable of predicting where periods should go in a transcript, the project begins first by building a training data set. One could achieve this by hand, but ChatGPT is designed specifically to handle problems such as this.

Using the Python module LangChain, developers can create chains of functions which include specific prompts to a Large Language Model. In my case, I opted to use OpenAI's "gpt-4o-mini" model to accomplish my goal of automatically adding periods to the provided transcript; due to it's fast inference time and relatively cheap cost. First, each section of the transcript was chunked into strings of length 1000 with an overlap of 100. Then these chunks were added to a prewritten prompt, which asks the LLM to add periods to the text without altering any other content in the text. There were several redundancies added to this process which would reject any response from the LLM that altered the text in any other way besides adding periods. I read through a good deal of these predicted sentences to ensure that ChatGPT did not add erroneous periods and I was unable to find a single one. Through an asynchronous call to the OpenAI API, a complete annotated transcript was prepared in 10 minutes and cost about $4.22 to generate. The whole process used 8,977,379 prompt tokens and 4,788,061 completion tokens.

1.2 Unpacking the OpenAI Results

The results of the asynchronous process is a list of dictionaries with two keys: "original_text" and "annotated_text". Using these dictionaries, the original transcript is updated via a simple string ".replace(original_text, annotated_text)" call. The end result is a ground truth off of which a smaller, lighter-weight model can adapt the period placing behavior.

2 Processing the Data

2.1 Build the DataFrame

Using the newly annotated transcript, each individual sentence is isolated and added to a Pandas DataFrame. This step includes additional steps such as removing commas, ellipses, hyphens, and other symbols to simplify the problem. The transcript contains a good deal of references to websites so phrases such as "www.patreon.com" were modified to "www dot patreon dot com" in order to avoid period ambiguity issues. Additionally, question marks and exclamation marks were replaced with periods. Finally, the whole transcript is cast to lower-case to further simplify the problem. The Dataframe at this step has tens of thousands of rows, with one full sentence per row.

2.2 Build the Vocabulary

The DataFrame is analyzed by a function called "build_vocabulary" which finds every unique word in every sentence and adds it to a set called vocab. Every word in the vocabulary is enumerated such that a one-to-one mapping of words and "ids" is produced. This step is essential to inform the design of the model's architecture later on.

2.3 Vectorize Sentences

Using these mappings, each sentence in the DataFrame is transformed into a vector. The vector is padded with trailing zeroes so that every sentence is the same length. This length is determined based on the longest sentence in the dataset. These vectors will be the input into the model during training.

2.4 Build the DataLoader

The last data preprocessing step is to create the custom PyTorch DataLoader. The DataLoader loads each sentence vector, and then randomly decides whether to keep the sentence whole or to cut it off. The random decision is made based on a "ratio_sentence_to_non_sentence" float which is between zero and one. In the event that the sentence cut-off decision is made, the end of the sentence is "cut-off" at a random point achieved with a mask of zeros. The resulting vector is labeled as a 0 and represents an incomplete sentence. In the other case, the sentence is not masked or altered in any way and is instead labeled as a 1. In this way, we have a set of labeled "complete sentences in need of a period" and a set of labeled "incomplete sentence that do not need a period". Going forward, the matrix X is the full table of vectorized sentences (some cut-off, some not) and the vector y contains the labels of 0 and 1, depending on if the sentence is cut-off or not. Finally, X and y are randomly split into train and test sets with a ratio of 80% train and 20% test.

3 Model Architecture and Training

3.1 Architecture

The Neural Network contains three layers. The first is an embedding layer which converts the sentence vector into an embedded dimension. The idea being that as the model learns, it will use this layer to associate similar words with one another in this embedded space. Then, the tensor is passed into a Long-Short Term Memory (LSTM) layer. Commonly, Recurrent Neural Networks (RNNs) are employed for situations such as this problem: where the order of the input matters both on "long" and "short" term time scales. Finally, a Linear layer is used to produce a tensor of shape $(\text{batch size}, 2)$. In order to obtain a prediction from this tensor, the "argmax" function is used to determine whether the model is more confident in the 0 or 1 label.

As for hyper-parameters, the embedding dimension is set to 100 and the hidden dimension is set to 64. The ratio of sentence to non-sentence data points is an even 50%.

3.2 Training

The training loop was fairly standard. For 15 epochs with a batch size of 128, the model was optimized using a Adam Optimizer with a learning rate of 0.01. PyTorch's CrossEntropyLoss was used for the loss function. After the full training loop was completed, the model was tested on the holdout test dataset.

4 Results

Across only 15 epochs of training the model was able to correctly predict whether it was looking at a full sentence in need of a period or a cut-off sentence 92% of the time on the training dataset and 87% of the time on the test dataset. While it certainly isn't perfect, it's a lot better than 50-50 guessing which means the model found a meaningful gradient. In this way, the model has distilled one of the many complex behaviors of Large Language Models into a single light-weight model.

5 Further Work

I am certain that with more time, the accuracy of the model could be improved through tuning of the hyper parameters. A simple GridSearchCV process could be used to find the optimal sentence to non-sentence ratio, embedding dimension, and hidden dimension; since the parameters stated in this report were simply educated guesses that worked sufficiently well. Additionally, this model is trained only on the words present in the transcript– if instead the vocabulary was more all-encompassing of the English language it would be more generalizable.

6 Conclusion

Large Language Models are incredibly powerful tools which, if you're willing to pay a small fee, can create robust training datasets. What this project demonstrates is that, if the problem is simple enough, simply distilling one of the many behaviors of Large Language Models into a smaller less computationally intensive model is completely viable and is, in many cases, a more scalable solution. An LLM like GPT-4o-mini has 8 billion parameters compared to my model which has 5.2 million. This means my model is more easily deployed and will on average have a faster inference time allowing it to be used in real-time during initial transcript creation.

In an actual use-case of such a model, the input vectors would be constructed one token at a time until a period is predicted. At which point a new empty input vector will be created and have one token added to it at a time until the next period is predicted, and so on.