Title

AI Autocomplete
Connor Brown
CSPB 4830 – Natural Language Processing
5/1/2025

Abstract

This report presents the implementation and evaluation of a GPT-2 based autocomplete system designed to accurately predict and complete user-inputted text. The study addresses key challenges in text autocompletion: maintaining contextual coherence in longer text sequences, optimizing for specific domains, and producing text consistent with contemporary language patterns. Using a subset of OpenAI's WebText corpus (10,000 documents) with minimal preprocessing to maintain authentic language patterns, I fine-tuned a GPT-2 model.. The implementation leveraged Google Firebase Studio with a Nix environment and utilized key libraries including Transformers/Hugging Face, PyTorch, NLTK, and Rouge_Score. Despite computational constraints limiting training to 1,000 samples with 1-3 epochs, the model produced grammatically sound sentences, though with limited contextual awareness. Performance metrics included a perplexity of 38.91, word-level accuracy of 1.67%, character-level accuracy of 6.60%, and ROUGE-1 F1 score of 0.0424. Future work includes scaling to larger datasets, increasing model size and training duration, implementing a GUI for interactive testing, and utilizing GPU acceleration to overcome current computational limitations.

Introduction

Accurate autocompletion of written text is increasingly relevant to the demands of the rapidly-evolving digital age. New developments in the field of Natural Language Processing, such as transformer-based models using the GPT (Generative Pre-trained Transformer) architecture, are needed to get peak performance for autocompletion tools. As a subset of text generation, autocompletion tasks face several challenges. First, generated text can get less coherent and lose contextual relevance as its length increases. Second, text needs to be optimized for specific fields, like scientific, legal, or creative writing. Lastly, autocompletion needs to produce text that is consistent with contemporary written language. This project aims to address these challenges.

I will implement a GPT-2 based autocomplete system with the intent of accurately predicting and finishing user-inputted text. Using a corpus such as Gutenberg or OpenAI’s WebText, the model will be trained to generate text that is consistent with contemporary written language. The goal will be to determine just how well the model carries out predictions and text completion. I hope to get an overall accuracy of at least 15 percent and text completions that are contextually appropriate.

Related Work

Several recent studies have explored techniques for improving text autocompletion and generation, addressing challenges like maintaining contextual coherence and optimizing using specific performance metrics, including accuracy and human evaluation.

Research by Chitnis et al. [1] investigated reinforcement learning for inline text autocomplete using a Markov Decision Process framework. Their findings revealed that cognitive load is more influenced by suggestion correctness than length, and that reinforcement learning approaches did not significantly outperform simple threshold-based methods. The study concluded that optimizing for text entry speed alone is insufficient, suggesting that accurate short-term prediction may be more valuable than long-term sentence completion.

Cruz-Benito et al. [2] evaluated multiple deep learning architectures (AWD-LSTM, AWD-QRNN, and Transformers like GPT-2, BERT, and RoBERTa) with different tokenization methods for code generation and autocompletion. Character-level tokenization with AWD-LSTM yielded highest accuracy for generation, while BERT and RoBERTa performed best at autocompletion, and GPT-2 produced more natural-looking outputs. Their work emphasized the importance of appropriate tokenization methods and architectures for specific applications.

Goyal et al. [3] surveyed 146 studies (2011-2022) on text generation, categorizing approaches into traditional, deep learning, and Transformer-based models. They compiled evaluation metrics including n-gram based (BLEU, ROUGE), distance-based, diversity-based, embedding-based, and learned metrics to assess different aspects of generated text quality. The survey highlighted challenges like ensuring coherence and handling ambiguity while providing frameworks for systematic evaluation of text generation systems.

Philip and Minhas [7] presented an overview of NLP techniques for text generation, categorizing models into Seq2Seq, GANs, and miscellaneous approaches. They detailed the standard text generation pipeline and discussed performance metrics including BLEU, ROUGE, and human evaluation. The paper highlighted advancements in hybrid models while emphasizing the need for improved evaluation metrics that better align with human judgment, providing important context for developing effective autocomplete systems.

After reviewing the literature, I saw several places for improvement. The papers provided methods and summaries, but did not provide code or practical applications. I intend to do both. I write and provide code that is thorough and well-documented, and discuss creating some kind of user interface that allows readers to play with the system on their own. I am not surprised that code and implementation details are not readily available, because autocomplete software is typically proprietary.

Data

For my first iteration of this project, I used OpenAI's WebText corpus. It is a massive collection of eight million publicly-available web documents, taking up approximately 40 gigabytes of memory. Due to computational constraints such as memory and processor capability, I chose to work with a 10,000 document subset. I wanted to create a general-use autocomplete system, so I performed no data cleaning or preprocessing in order to maintain authentic language patterns. Ideally, I would have used an 80/20 split on the 10,000 documents for training and testing, but once again, I was limited by computational constraints, and had to use smaller subsets (1000 for documents for training, 100 for testing).

Methodology

The main model I used was OpenAI's GPT-2, with GPT standing for Generative Pre-trained Transformer. I had a limited amount of time to complete this project, and GPT-2 (as mentioned in Related Works section) is excellent for encoding tasks, so I decided it was best to fine-tune the model rather than training it from scratch. It was just a matter of choosing the right data subsets to train it with.

For development, I used Google Firebase Studio with a Nix environment for dependency management. This allowed me to bypass any hardware compatibility issues of transformer libraries. The key libraries used were as follows: Datasets (data loading and processing), NLTK and Rouge_Score (evaluation metrics), Numpy and Scikit-Learn (data manipulation), PyTorch (underlying Machine Learning framework) and Transformers/Hugging Face (for model implementation).

My initial approach was to create a multifile object-oriented design. It featured a *src* folder containing class files for loading data, setting up models, performing autocomplete operations, evaluating the models, and performing error analysis. It also had two main scripts, one for training models, and the other for evaluating them. While solid, this approach quickly ran into computational constraints, as I was only able to use a CPU and limited memory. The combination of many class files and model training was a bit too much for Firebase Studio to handle.

My revised approach used a much simpler application design, featuring two core modules, a trainer and an evaluator. The trainer was used for training models and the evaluator was used for evaluating the models' performance. Any output would be placed in a JSON file.

For my model training parameters, I went with a batch size of four to eight (using two in my first trial), with one to three training epochs, and used 1000 samples from WebText. Training took about two hours using available hardware. I would have liked to do more training tests, but I ran out of time.

Results

After running a test on my trained model, I found out that the model performs well below expectations. For perplexity, I got a value of 38.91. My word-level accuracy was 1.67 percent, and my character-level accuracy was 6.60 percent. BLEU score was 0.0, and ROUGE-1 F1 was 0.0424. There were 50 semantic errors, 0 grammatical errors, and 0 style inconsistencies (Fig. 1).

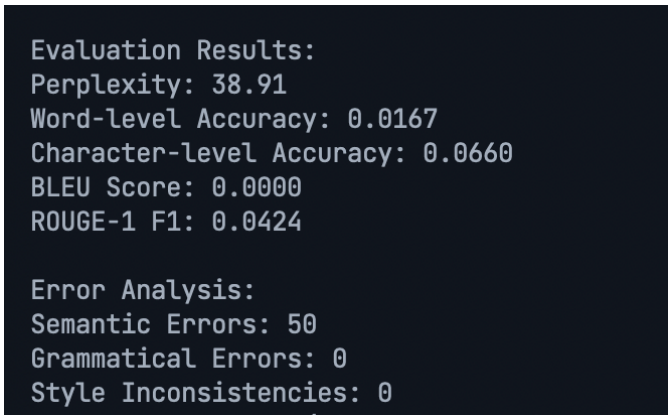

Figure 1. Initial output on model trained on 1000 samples and tested on 400 samples.

What do these mean? Perplexity, the measure of how well a model makes predictions, runs from 0 to 100, with 0 meaning a perfect prediction and 100 a completely wrong one. My value of 38.91 is on the better end of the scale, but still reflects a strong uncertainty of the model about predicting the next character or word. This is supported by my values for word-level and character-level accuracy, which are determined by comparing predicted output to a reference. The BLEU score, which is calculated by comparing n-gram overlap between reference and output, reflects *zero* n-gram overlap, indicating that the generated text is not matching the reference at all, at least at the word level. The ROUGE score is similar to BLEU, only instead of precision, it uses recall. So, it shows not just exact n-gram overlap, but also whether or not words appear together. It is also very low.

These metrics suggest that the model is not generating good text. However, there are 0 grammatical or style-related errors; they are purely semantic. That means that the sentences generated are good, but have the wrong context. This is shown in Figure 2, where the predicted sentences do not match the reference at all, but are nonetheless good sentences.

What my scores and metrics say is that my model is not correctly picking up on document context. For future iterations, I would want to increase the training and testing sets. This would hopefully solve the issue of incorrect context.

```
"examples": [
  {
    "prefix": "sorry mani",
    "prediction": " hu-hwan",
    "reference": "a i stole your forma"
  },
  {
    "prefix": "According ",
    "prediction": "\u00a0the\u00a0Gospel",
    "reference": "to a number of repor"
  },
  {
    "prefix": "The latest",
    "prediction": " in a string of high",
    "reference": " quarterback prospec"
  },
  {
    "prefix": "Get the bi",
    "prediction": "ographical details about how you",
    "reference": "ggest daily stories "
  },
```


Figure 2. Example autocomplete output.

Discussion

Using multifile OOP was a good initial choice, and definitely more scalable than the final two-module solution, but computational limitations made it unfeasible for the time being. Likewise, it would have been better to use the full WebText dataset, but in order to complete the project in a timely manner, I had to reduce it to 10,000 documents.

My low scores and metrics were pretty disappointing. I think these were cause by three factors: limited training data (a smaller 10,000 document subset), insufficient training epochs and memory and time constraints. However, the example predictions showed that my project is generating good sentences, despite being flawed. I believe that I just need more computational resources to train my model so that it can correctly grasp context.

As this course has been all about understanding how LLMs work and their applications, I took advantage of the opportunity to try using them in practice. I found out early on that coding with AI should be done the same as manually – in small, testable pieces. If code blocks get too large, they can quickly become a challenge to debug if the LLM makes a mistake: sometimes the LLM will choose an unusual implementation pattern or not fully complete the necessary code. I learned that tracing complex Machine Learning code is just as difficult and time-consuming as any other code, so moving forward I will be careful about how I use AI assistance and when.

Conclusion and Future Work

My project is on the right track, it just requires more computational resources. The solution is simple. I need to work in a GPU-enabled environment for faster model training, scale to larger datasets, e.g. the full WebText corpus, and increase the model size and training duration. A future

change might be to implement a GUI for interactive testing. Ultimately this will be a user-facing application, so it is important that it can be user-testable.

Bibliography

[1] Chitnis, R., Yang, S., & Geramifard, A. (2024). Sequential Decision-Making for Inline Text Autocomplete. *arXiv preprint arXiv:2403.15502*.

[2] Cruz-Benito, J., Vishwakarma, S., Martin-Fernandez, F., & Faro, I. (2021). Automated source code generation and auto-completion using deep learning: Comparing and discussing current language model-related approaches. *AI*, *2*(1), 1-16.

[3] Goyal, R., Kumar, P., & Singh, V. P. (2023). A Systematic survey on automated text generation tools and techniques: application, evaluation, and challenges. *Multimedia Tools and Applications*, *82*(28), 43089-43144.

[4] Musale, V., Gajjar, A., Amune, A., Singh, P., Singh, T., & Khandelwal, H. (2024, December). Comparative Assessment and Enhancement of Transformer Models for Effective Text Generation. In *2024 International Conference on Communication, Control, and Intelligent Systems (CCIS)* (pp. 1-6). IEEE.

[5] Pandey, R., Waghela, H., Rakshit, S., Rangari, A., Singh, A., Kumar, R., ... & Sen, J. (2024). Generative AI-based text generation methods using pre-trained GPT-2 model. *arXiv preprint arXiv:2404.01786*.

[6] Patriota, M. F. F. D. S. (2023). A transformer-based architecture neural network approach to email message autocomplete.

[7] Philip, P., & Minhas, S. (2022). A Brief Survey on Natural Language Processing Based Text Generation and Evaluation Techniques. *VFAST Transactions on Software Engineering*, *10*(3), 24-36.