**AI ASSISTED CODE AUTO COMPLETION TOOL**

Ved Patel

University Of South Florida

COT4210 : Automata Theory

Valentina N. Korzhova

April 24, 2025

**Introduction**

Recent advances in AI have caused much excitement about the future potential for providing massive increases to the productivity of programmers. A test of the viability of this capability can be found in code completion. This has several benefits in that it outlines a task type with a strict domain that can be tested easily by using real world ultra large, open source repos. This however is problematic in that LLMs are at best just as error prone as human software developers, and oftentimes, much more. To address this we leverage Automata Theory and formal grammar to perform basic checking for unrecoverable single line syntax errors. This allows the end user to not be misled with spurious suggestions. Testing shows that the training with easily available data, open source software, and small amounts of compute produce quantitatively large improvements in loss reduction which can be further filtered by formal grammars. The end result is an AI-Assisted code autocompletion tool with full basic functionality.

**Architecture**

The first and most important choice was that of which dependencies and systems we would leverage and how we would use them together. In our original proposal we planned on using Lex and Yacc to produce a tokenized form of the prompt and predict next tokens. The issue with this is that it would only be plausible to predict what kind of token should come next, an integer, an operator, a function, etc. This is because lex would turn all actual code into classified tokens. In practice this massively increases the entropy, making precise predictions impossible. This was against the spirit of our proposal, so we rearchitected to use a traditional splitter type tokenizer to feed a fine tuned LLM, and used the unique power of Lex / Bison to verify predictions are not spurious.

**The LLM**

A primary constraint of this project is scale. Our limited resources meant that we could only fine tune with small amounts of compute on local machines. This in practice limited us to using GPT2-small. We first acquired a very large open source corpus for our selected language, C. We found the Postgres source code suited our needs well. It is considered to be a highly mature code base, it features best practices, is highly technical, and has a staggeringly large number of lines of . About 700k, after stripping away comments and non-C code. Our pre-processor strips the code base of all .c and .h files, removes inline and multiline comments, and various other tasks, finally saving the processed data to JSONL. We then used extensive trial and error to fine tune the model, often using small runs of only 500 samples to test fixes and tweaks for a surprisingly large amount of problems. This included the model learning to predict the input instead of the output, the model not knowing when to stop generating tokens, the model including comments about baseball, etc. Solving these issues was the bulk of the work done on this stage of the project. Once satisfied, we used our group’s most powerful desktop to overnight train. We found that we were only able to use 55k lines for an overnight training run instead of the full 700k, lest the run take several full days. The last portion is the prompting script, which uses many different configuration arguments to refine the output. This took extensive testing too, but because it can be tested instantly instead of requiring a retrain, it was a much faster process.

**The Lex-Bison Verifier**

Lex is a lexical tokenizer. It scans natural language, or in this case programs, and classifies words into enumerated categories. For example 14 is an INT, + an operator. This tokenization allows real world code to be flattened into categories. The tokens are then fed into GNU Bison, a so-called meta-compiler which can use context grammars to analyze the validity of constructs of tokens. An example of this is “14 +;” would be lexed into “INT OPERATOR;” which would then be classified as illegal because it is not in the language of a CFG in bison. By implementing the lex tokenizer and Bison parse-generator we were able to use the principles of Automata theory to validate the outputs of the LLM. In this project we focused on a particular subset of syntax errors which we call irrecoverable single line errors. Certain errors such as parenthesis miss match are not possible to conclusively prove in a single line, for example a for loop may have the structure “for(...){“, which has an unpaired bracket but is still potentially valid. We targeted the subset of errors which are not ambiguous. For example if the for loop was structured as “for(int i=;;{“, we need see no more to invalidate it. There is no multiline context which could recover this. In this manner we were able to eliminate an entire class of spurious error from being suggested to the user.

**UI And Data Flow**

We opted for a simple UI with an emphasis on demonstrating our underlying tools. The actual structure is an extension for IntelliJ, a Java focused IDE. This was done due to existing expertise in our group with making extensions for this IDE. The user sees a window on the right side of their environment which provides suggestions when asked. The backend consists of a collection of files for building other files, moving data to the LLM, and sending it to be verified, to then finally be shown to the user. This allows the user to conveniently access the predictions and verification in the environment without using our scripts themselves.

**Future Work**

The main limitation of this tool is the use of the particular model. GPT2 is unsophisticated. Those who were following LLMs at the time noted it for lacking coherence over more than a few sentences, not being able to even emulate reasoning, etc. We were also only to fine tune the model on an amount of compute that is achievable for a small team. This was not an enterprise project. The end result is that while the model outputs code, its predictive power is limited. We point out that this is a resource problem and that more recent LLMs and code assistance tools such as Copilot have overcome these issues. We also believe that there is good potential for tools integration at tool time using Lex-Bison as a tool to help LLMs learn to create rigorously valid code.

**Conclusion**

We have leveraged a GPT2-Bison Pipeline to produce and verify proposed completions to partial lines of code and which improves its predictions by using the previous lines as context. This project has successfully demonstrated that LLMs can complete lines of code and that they can be combined with tools to produce better results.

**References**

1. Jain, S. M. (2022). *Introduction to Transformers for NLP : With the Hugging Face Library and Models to Solve Problems* (1st ed. 2022.). Apress. <https://doi.org/10.1007/978-1-4842-8844-3>
2. Keishi, S., & Seika, A. (2014). PaGe: Portable LALR(1) Parser Generator. *Proceedings of ILC 2014 on 8th International Lisp Conference*, 8–8. <https://doi.org/10.1145/2635648.2635650>
3. Ruberg, P., Meinberg, E., & Ellervee, P. (2022). Software Parser and Analyser for Hardware Performance Estimations. *2022 International Conference on Electrical, Computer and Energy Technologies (ICECET)*, 1–6. <https://doi.org/10.1109/ICECET55527.2022.9872951>
4. *Installation*. (n.d.). Hugging Face – The AI community building the future. <https://huggingface.co/docs/transformers/en/installation>