July 2023

Using Code Review Repositories and Changelists to Train Large Language Models for Code Generation

Joseph Johnson Jr

Shiblee Hasan

Emmanouil Koukoumidis

Using Code Review Repositories and Changelists to Train Large Language Models for Code Generation

ABSTRACT

While large language models (LLMs) can generate code, training of such models has not made use of data generated during the collaborative code review process that is a standard part of software development. This disclosure describes techniques that utilize historical code review data (including reviewer comments and corresponding code edits) available within organization internal code repositories to train LLMs to generate code. The historical code review data can be used for model tuning, to train an LLM via reinforcement learning from human feedback (RLHF), and/or via prompt engineering. The trained model can be utilized to generate code starting from code description provided using a prompt template. The prompt template can incorporate organization specific factors such as developer guidelines, developer or team style, etc. Code generated by the LLM can be iteratively refined via human review as well as from analytical tools that ensure style compliance, code coverage, test success rate, comment conventions, etc.

KEYWORDS

- Large language model (LLM)
- Alignment problem
- Code generation
- Code refinement
- Code repository
- Code coverage
- Model tuning
- Changelist
- Prompt engineering
- Reinforcement learning
- Conversational dialogue

BACKGROUND

Developers can leverage a large language model (LLM) to generate or modify code based on relevant input, such as a question, prompt, or description. Many software development tools and Integrated Development Environments (IDEs) include LLM-driven code generation features that are similar to text auto completion but for coding tasks. For instance, after a user enters some code, such tools can automatically generate small amounts of additional code or can suggest edits to the typed code.

Since the training data for LLMs typically includes source code repositories, the quality of the generated code is often high enough to be comparable to production code. In many cases, the code generated using an LLM compiles successfully without any modifications. However, in several cases, especially ones that involve complexity and nuance, developers may need to modify the initial LLM-generated code by filling in and/or refining various details. Alternatively, or in addition, developers can engage in an interactive dialogue with the LLM to iteratively refine the initially provided or generated code based on feedback and replies within the dialogue. In most cases, the LLM has a reasonable amount of memory to refine the code via dialogue.

Developers need several conversational turns and/or prompt refinement to solve what is referred to as the alignment problem. The alignment problem is for an LLM to choose a reply among many potential choices where the LLM needs help to determine the optimal way in which the choices are to be ranked for the task at hand. In many cases, a generic ranker cannot be trained on its own since making the optimal choice for most problems requires even humans (e.g., developers) to ask questions and receive feedback in conversation with other humans (e.g., code reviewers).

Reinforcement Learning from Human Feedback (RLHF) is a popular technique to refine a machine learning model for better handling the general case of selecting the best response from a set of model-generated potential responses. In essence, RLHF can serve as an answer ranker refined on human feedback. RLHF can involve human actors on both sides of the conversation, where one or more humans act as the user by simulating the questions and prompts input to the model. The same or different humans can act as the model by selecting the best reply from the set of choices the model generates. The questions and answers are both used to refine the ranking ability of the LLM and train it to select the best answers from a generated set.

Apart from using RLHF to train LLMs to rank possible answer choices, LLMs can also be enhanced to be better at specific tasks, such as generating code. Such enhancements can be achieved in various ways, such as fine tuning the model by retraining a pretrained model by unfreezing the model weights and training for a further few iterations or epochs on data that is specific to the task. Alternatively, or in addition, the RLHF process described above can be similarly fine-tuned with question-answer pairs that match the task. Further, input prompts to an LLM can be fine-tuned via prompt engineering to create better task-specific prompts that lead to better answers being generated by the LLM.

Currently, developer use of LLMs for generating code typically follows the normal coding process of starting with initial, developer-written code and then creating a corresponding description. Coding is a collaborative task that involves code reviews by other developers. Reviewers often suggest edits to code to achieve goals such as improved performance, bug fixes, enhanced code readability, etc. In many source code management platforms, reviewers can submit change requests with suggested code edits and corresponding descriptive notes that the original author of the code can access to converse with the reviewer, if needed, and to update the original code as appropriate. The use of LLMs for coding purposes does not currently incorporate the collaborative code review process that is a standard part of the software development process.

DESCRIPTION

This disclosure describes techniques that enable the incorporation of collaborative code review practices (including historical code review data, changelists or patches, etc.) to train LLMs to generate code. Application of the techniques can enable LLMs to automatically generate code starting from a description in the style of a code review. The code generated initially in response to a prompt can be iteratively refined with dialogue that includes relevant additional mini prompts. The iterative review dialogue can in turn be employed to improve the future performance of the code generation model.

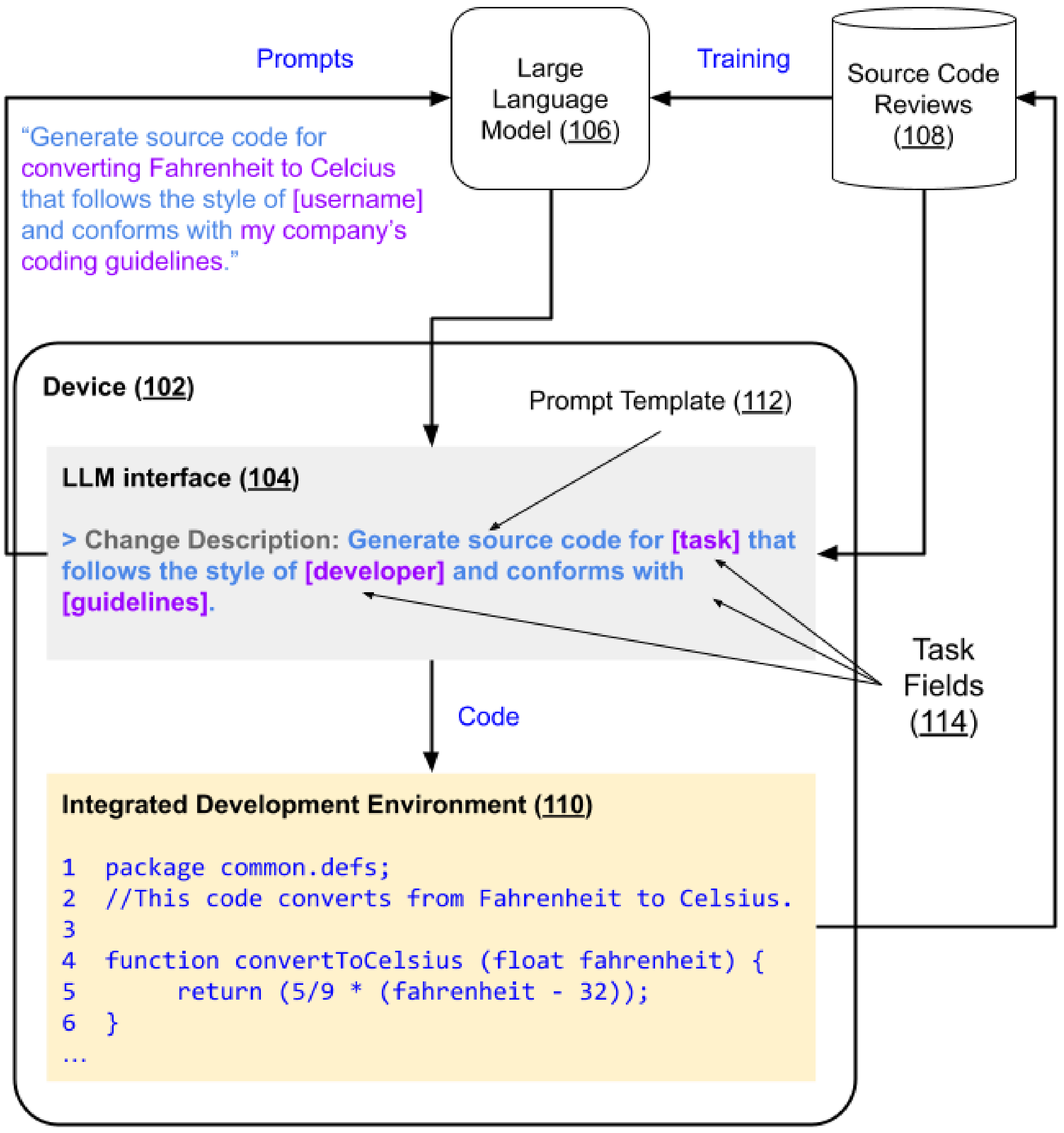
The above capabilities can be realized by enhancing standard LLMs for the specific purposes of generating code from code review descriptions. The enhancements involve one or more of the following approaches:

1. **Model tuning based on historical code review data:** By unfreezing their weights, existing LLMs can be trained for a few iterations on existing repositories code reviews and corresponding edits and developer conversations. The training and validation loss can be validated against a holdout set of data. On reasonably high-powered machines, such training can be done relatively quickly.
2. **Task-specific RLHF:** The model can be trained by using the dialogue between the author of the original code and various code reviewers as task-specific question-answer pairs, with the code review descriptions serving as the question and the code review comments and/or the corresponding code edits (changelist) as the answer. The approach

can employ any suitable reinforcement learning technique, such as Q-Learning, Deep Reinforcement Learning (Deep Q-Learning), Value Iteration, or Proximal Policy Optimization (PPO).

3. **Prompt Engineering:** A generic prompt (e.g., "Generate code for…") can be used as the starting review template for each code generation task. A developer can modify the template prompt by filling in details specific to the task at hand to generate initial source code from an off-the-shelf LLM. The initial code is iteratively refined via code review comments that suggest changes. Initial comments can be from the developer commenting as a reviewer of code authored by the LLM. Subsequently, the developer can send the code for review by other human developers and/or automated code inspection and testing tools. A task-specific LLM tuned on historical code review data is employed to refine the code automatically in response to code reviews received from developers and/or automated tools.

Fig. 1 shows an example operational implementation of the techniques described in this disclosure. A developer using an IDE (110) on a device (102) employs an LLM (106), which is tuned by training on existing source code reviews (108) to automatically generate code. In the example of Fig. 1, the developer's goal is to obtain code to convert temperatures from the Fahrenheit to Celsius scale.


Prompts
Large Language Model (106)
Training
Source Code Reviews (108)
"Generate source code for converting Fahrenheit to Celcius that follows the style of [username] and conforms with my company's coding guidelines."
Device (102)
Prompt Template (112)
LLM interface (104)
> Change Description: Generate source code for [task] that follows the style of [developer] and conforms with [guidelines].
Task Fields (114)
Code
Integrated Development Environment (110)

```
package common.defs;
//This code converts from Fahrenheit to Celsius.

function convertToCelsius (float fahrenheit) {
     return (5/9 * (fahrenheit - 32));
}
...
```

Fig. 1: Automatically generating and refining code from code review descriptions using LLMs

The developer starts with a template prompt (112) within the LLM interface (104) and specifies the coding task by appropriately modifying the task-specific fields (114) within the template. In the example of Fig. 1, the developer completes the template prompt with task specific parameters to specify the prompt "Generate source code for converting Fahrenheit to Celsius that follows the style of [username] and conforms with my company's coding guidelines." The LLM-generated source code is displayed within the IDE and can be iteratively refined by subsequent human and/or automated code reviews and interaction with the LLM as described above. Automated code review can include the use of analytical tools to ensure style compliance, code coverage, test success rate, comment conventions, etc. The code review history can serve as data for further refinement of the model.

During the initial prompt specification and iterative code refinement, fields within the prompt templates can be filled in automatically with relevant information as appropriate. Alternatively, or in addition, the human developer can edit the prompts as necessary. Code reviews from automated code inspection tools that result in corresponding LLM-generated code refinements can take place behind the scenes, with the conversation and edit history associated with these reviews hidden by default. However, developers may be provided options to view and edit such reviews and code edits on demand, e.g., if needed for debugging and refinement. At any point in the iterative code development process, the human developer seeking auto-generated code from the LLM can choose to make code refinements in addition to those generated automatically. Alternatively, the human developer can choose to stop seeking code via the LLM and continue editing the code manually.

Tuning LLMs on the repositories of reviews of source code available within an organization can enable developers to employ LLMs to obtain automatically generated code that conforms more closely from the outset with organizational coding norms and practices (e.g., using the "guidelines" field in the template prompt of Fig. 1) as well as with individual developer/team styles (e.g., using the "developer" field in the template prompt of Fig. 1). Since the tuning based on code repositories of an organization adds to the existing training of the model, the existing knowledge of codebases outside the organization encoded within the LLM is retained.

The techniques described in this disclosure can be implemented to train any LLM and can utilize historical data within any source code and code review repository. The techniques can support any automated or manual code review tools and any IDE. LLMs trained using the described techniques can be implemented to operate within the code development platforms and processes of the organization. The LLM interface can be embedded within current tools and/or provided in any suitable form, such as standalone product or via an application programming interface (API). Implementation of the techniques can enhance the quality and speed of generating and reviewing code generated with the aid of LLMs, helping boost developer productivity and transform software development practices.

CONCLUSION

This disclosure describes techniques that utilize historical code review data (including reviewer comments and corresponding code edits) available within organization internal code repositories to train LLMs to generate code. The historical code review data can be used for model tuning, to train an LLM via reinforcement learning from human feedback (RLHF), and/or via prompt engineering. The trained model can be utilized to generate code starting from code description provided using a prompt template. The prompt template can incorporate organization specific factors such as developer guidelines, developer or team style, etc. Code generated by the LLM can be iteratively refined via human review as well as from analytical tools that ensure style compliance, code coverage, test success rate, comment conventions, etc.