Using Codex's Edit API to Improve Codex's Python Code Generation from Natural Language

Sourya Kakarla, Derek Pham, Gabriele Tombesi
Department of Computer Science
Columbia University
New York, NY 10027

`https://github.com/ma08/codex_optimal_proj`

Abstract

Codex has become reasonably proficient generating code in several programming languages by simply parsing input natural language instructions. In this work, we investigate the use of OpenAI's new Edit and Insert APIs by performing a series of Edit/Insert operations on Python code generated by Codex from natural language prompts, and then evaluating the impact on the generated code functionality. The Completion API is first used to get preliminary Python code for a given prompt. We then aim to improve the Quality of Results (QoR) of the Codex-generated Python code by looping over it and applying to it the relevant Edit/Insert instructions using the new respective API. Throughout this process, we intend to tune key parameters for both Completion and Edit/Insert operations, and perform space exploration to look for the top-performing combinations of such parameters and Editing instructions by optimizing the evaluation metrics defined. For our experiments, we use the dataset from Automated Programming Progress Standard (APPS), which contains natural language description of questions with their corresponding metadata and Python solutions.

1 Introduction

OpenAI Codex [8] is a language model obtained by training OpenAI's base GPT-3 [3]–which is pre-trained on web and text resources–on a vast set of public code repositories retrieved from GitHub. As a result of this additional training process, Codex has become reasonably proficient generating code through simple parsing of input natural language instructions. Typical of third generation of generative transformers, Codex inherits few-shot, one-shot and zero-shot learning capabilities, intuiting a wide variety of downstream tasks just from the input settings, without need of additional instructions and with few to no examples.

As of today, Codex's performance in generating code from natural language prompts is strongly dependent on the level of complexity of the input task and the corresponding algorithms required to implement it. While the model's massive size allows to achieve state-of-the-art results for basic Python program synthesis tasks, it is still not enough to produce working solutions for more complex tasks from specific domains that might not be significantly represented in the corpra used during the model training. Several studies [4] [2] have showed that fine-tuning language models on a specific sub-domain can significantly improve the model performances at evaluation time, making smaller fine-tuned models competitive with bigger unfine-tuned ones. In particular, Chen et. al [2] report consistent gains across the examples of the HumanEval dataset, using the supervised fine-tuned models Codex-S. For these reasons, we originally aspired to work on fine-tuning Codex toward optimal solutions for sorting and searching problems, focusing more on optimal time complexity rather than functional correctness. We started from running several successful tests with the "davinci" engine following the guide "Create fine-tune" API [5]. However, as we got more familiar with the tool, we came to realize that the API only supports the text "davinci" engine, while the fine-tuning API for code engines such as "code-cushman-001" or "code-davinci-002" is not open-source yet and so not supported in the OpenAI API [10].

Starting from the conclusions drawn above, we decided to pivot our efforts to improve the average functional correctness of the code generated without using fine-tuning. In order to do so, after reviewing the existing literature, we hypothesize that the invalid Codex solutions (for a prompt) fall into two broad categories:

- **Functionally incorrect solutions**: These are the solutions generated for tasks where Codex is not able to handle some of the logical aspects or the complexity of the algorithms to infer in order to provide a correct implementation.
- **Nearly correct solutions**: These are the types of solutions that get the essential parts (algorithms, library calls, APIs etc) of the solution right yet don't pass the evaluation of correctness due to indentation issues, syntax errors such as unbalanced parentheses, minor logical issues etc.

While we recognize that the only viable approaches to handle the former category are either increasing the model size or fine-tuning, we decided to target the latter one in order to find an alternative approach for making nearly correct solutions eventually become functionally correct.

In the context of syntax errors fixes and code refactoring, it is relevant to mention that OpenAI has recently announced and released the support for a new API with Edit/Insert capabilities, which enables the user to refactor and tweak the code rather then simply complete it, or generate it from a natural language prompt [7]. We found that this new feature has huge potential in dealing with the issue of syntax errors and other similar issues affecting code functionality across a wide variety of application domains. For these reasons our core research focus in this project is to investigate the use of Codex's edit and insert capabilities, namely to explore ways to optimize the usage of these Edit and Insert APIs.

1.1 Previous Work

Chen et al. (2021) introduced Codex and evaluate its capabilities in generating Python code [2]. Codex is fine-tuned from GPT-3, an autoregressive language model that achieves strong performance across a variety of natural-language-generating tasks, including translation and question-answering [1]. A significant number of existing language models for code-related tasks developed from the GPT architecture. Among them, IntelliCode Compose leverages the autoregressive approach of GPT, applying it to the field of source code understanding, namely multilingual code completion tasks [9].

Moreover, Henraks et al. (2021) developed APPS - Automated Programming Progress Standard, a benchmark for code generation [4]. APPS measures the ability of models to take an arbitrary natural language specification and generate satisfactory Python code. This benchmark includes 10,000 problems, which range from having simple one-line solutions to presenting substantial algorithmic challenges. For our project, we emulate the APPS benchmark in our evaluation of Codex and utilize parts of the APPS dataset.

.

2 Methods

First, we searched for a dataset that suits our purposes. In particular, we wanted a batch of prompts where at least a fraction of it could already be handled by Codex without needing to deploy the

Editing API. This would enable us to observe whether there would be any improvement of pass@k metrics from adding the Editing layer to the evaluation pipeline. We identified the APPS dataset as a good candidate, since it includes problems from several open-access sites spanning in complexity, from introductory problems to competition problems.

Next, we devise an end-to-end evaluation framework based on the following pipeline, summarized in figure 1: we take natural language prompts from coding interview questions and pass them to Codex's Completion API to generate initial coding solutions with a given Temperature and k settings. At this point, a first evaluation is performed in order to collect *pass@k* metrics numbers. After that, we pass the evaluated solutions through a series of edit/insert operations for the Codex Edit and Insert APIs to refactor these solutions. Finally, we perform another evaluation of the solutions average pass rates, with the same evaluation framework used above, and compare with the previous results. We find that the novelty of this approach comes with two main factors:

- It provides a wide range of knobs to tune, enabling a vast space-explorations.
- It is based on a pipelined evaluation framework, which promotes collaboration across the group members and at the same time allows each of them to focus on each block optimizing the throughput.

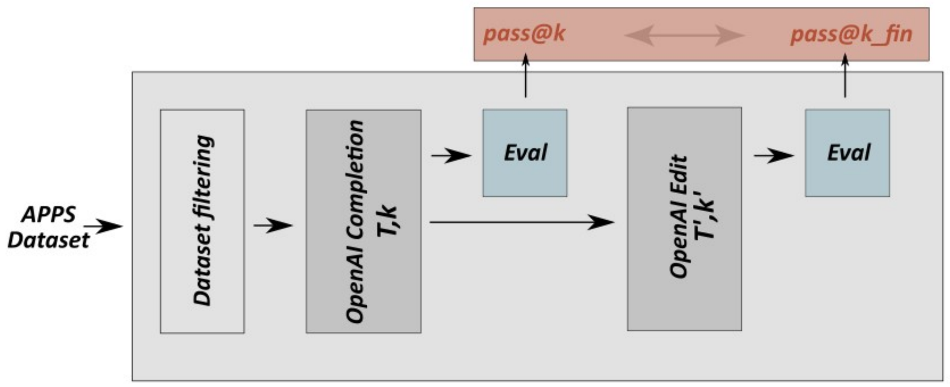

Figure 1: Evaluation Framework for the Editing API flow

2.1 Experiment 1

The main steps of the first conducted experiment are illustrated below. Each of these steps is subjected to change in future work in order to explore better solutions:

- First, we focused on filtering the dataset to obtain the introductory problems, which Chen et al. [2] reports to be the category with the most significant pass rates (namely from 4.14% for pass@1 to 25.02% for pass@10000) with the intention of extending to more complex problems later. This filtering step was done by leveraging the *metadata.json* information provided by APPS in each prompt folder.
- As an additional preprocessing step, we further filtered the dataset of introductory problems by excluding prompts that are too long for Codex to process: In order to do so, we computed the word count of each question and its longest ground truth solution as a means of estimating the feasibility of making a Completion API request, since the Codex API request has a 4000-token limit for both prompt and solution combined.
- At this point, we run the inference for Python code generation using Codex's Completion API [6] on the "code-davinci-002" engine and save the solutions obtained in the pre-defined *json* format. Tuning on temperature and number of samples is conducted during this process. The parameters used for this first attempt, that we intend to tune in future, are:

    - $k = 2$ (2 solutions per prompt).
    - $T = 0.5$.

  The temperature was chosen according to what is defined as the optimal T value for $k = 2$ in Chen et. al [2]. This task is currently being run under a different setting, and we realize this setting from Chen et. al is preliminary and may not necessarily be optimal. And so we are eager to explore other T values in future iterations.

- We then conducted the first evaluation step in order to obtain the pass@2 metric for this first experiment. The evaluation is performed using a set of dedicated scripts that perform the following operations :
    - For each prompt, take all the solutions obtained using the Completion API call
    - Test each of these solutions against all the test cases of that prompt–which are provided by the dataset–and store the results of the unit tests.
    - Check for the number of prompts that have at least one solution which passes all the unit tests. The fraction of these prompts over the total amount of prompts is returned as the test@k metric.
- Afterward, we defined (*"Fix spelling mistakes","Fix syntax errors"*) as the first sequence of test Edit Operations, and run them on the solutions obtained above, with $(k = 2, T = 0.5)$ as initial parameters of the Edit operation.
- Finally, we repeated the evaluation steps above.

  The Data filtering and Evaluation results of the illustrated steps are reported in the following section. Moreover, our code and preliminary results can be found in the following repository: `https://github.com/ma08/codex_optimal_proj`

3 Results

We obtained 2639 "introductory" level questions from the APPS dataset using the metadata and further filtered them based on combined word count length of question and estimated solution–we estimate the length of the solution for a given problem by using the length of its longest ground truth solution–to make sure the evaluated questions do not exceed the 4000-token limit as previously mentioned.

The results obtained in the first experiment are the following:

- Pass@2 for Completions (T=0.5,k=2): $5,77\%$
- Pass@2 for Completions augmented with Edit layer (T'=0.5,k'=2): $9,61\%$

The first result obtained is strongly coherent with the numbers provided by related work [2] on evaluating Codex on APPS introductory problems. In this paper, in fact, the *pass@k* values reported for these problem set with k=1,5 are respectively $4,14\%$ and $9,65\%$.
In addition, we find the second result illustrated extremely encouraging with respect to our research focus, since it shows a significant increase of the pass rate brought by the Editing layer.
In the future, we plan to extensively investigate the trends which can be obtained with additional recursive Editing calls, as well as parameters tuning for both Completions and Edits calls.

3.1 Rate limiting

During our experiments, we have been limited by the rate limits of OpenAI. We have encountered two kinds of throttling:

- Requests: `openai.error.RateLimitError: Rate limit reached for requests. Limit: 20.000000 / min. Current: 24.000000 / min. Contact support@openai.com if you continue to have issues.`

- Tokens: `openai.error.RateLimitError: Rate limit reached for tokens. Limit: 150000.000000 / min. Current: 163840.000000 / min. Contact support@openai.com if you continue to have issues.`

We are tinkering with spacing of requests by using a "sleep" function after each request to overcome the above limitations. We also reached out to `support@openai.com` and have received a response inquiring over desired rate and research areas. We are currently in discussions to get the allotted rate increased.

References

[1] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. *CoRR*, abs/2005.14165, 2020.

[2] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. *CoRR*, abs/2107.03374, 2021.

[3] Luciano Floridi and Massimo Chiriatti. Gpt-3: Its nature, scope, limits, and consequences. *Minds and Machines*, 30(4):681–694, 2020.

[4] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, et al. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938*, 2021.

[5] OpenAI. Create a fine-tuned model. `https://beta.openai.com/docs/guides/fine-tuning/create-a-fine-tuned-model`, 2022. [Online; accessed 30-March-2022].

[6] OpenAI. Natural language to python. `https://beta.openai.com/playground/p/default-english-to-python`, 2022. [Online; accessed 30-March-2022].

[7] OpenAI. New gpt-3 capabilities: Edit insert. `https://openai.com/blog/gpt-3-edit-insert/`, 2022. [Online; accessed 30-March-2022].

[8] OpenAI. Openai codex. `https://openai.com/blog/openai-codex/`, 2022. [Online; accessed 10-March-2022].

[9] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. Intellicode compose: Code generation using transformer. *CoRR*, abs/2002.08155, 2020.

[10] Team 13. Openai finetuning results. `https://github.com/ma08/codex_optimal_proj/tree/main/fine_tune_demo`, 2022. [Online; accessed 30-March-2022].