Retrieval Guided Code Generation Leveraging the Planning Capabilities of LLMs

Lakshmipathi Balaji
2021114007
Team String
lakshmipathi.balaji

Advaith Malladi
2021114005
Team String
advaith.malladi

Marri Abhinav
2021112015
Team String
abhinav.marri

1 Introduciton

Recent advancements in language models have led to improvements in automated code generation and completion tasks for enhancing programmer productivity and reducing repetitive coding tasks. Despite these advancements, current models still face challenges such as low code quality, compilation errors, and inability to meet intended functionalities effectivel. One of the emerging solutions to these challenges is the integration of retrieval-based augmentation techniques. In this project, we adopt a similar approach to improve code generation and completion tasks.

Building on the success of retrieval-based techniques, our study leverages a hybrid framework that combines TF-IDF-based retrieval with Large Language Models to enhance code generation and completion. We focus on two core tasks: code generation based on a given textual description and code completion using augmented or partial code segments. We utilize datasets like MBPP (Austin et al., 2021), OOPS benchmark (Wang et al., 2024) and methodologies that incorporate code embeddings and semantic understanding to outperform traditional models. We aim to leverage the planning capabilities of LLMs for code generation. We have made our code publicly available at https://github.com/kolubex/IRE_2024/.

2 Literature

2.1 Code Generation

Code generation refers to the task of automatic generation of code based on the provided input text instructions. In recent times, with the advances in generation models, automatic code generation started receiving renewed attention due to it's potential in increasing programmer's productivity and reducing the heavy and repetitive workload of a software developer.

Despite its initial success, much of the generated code still suffers from poor quality, failure to pass test cases, compilation errors, and inability to fulfill intended functionality. This might be because most of the code generated from language models is still a variation of next token prediction which is not how code is usually written. To overcome this problem, (Parvez et al., 2021) proposed a retrieval augmented code generation framework that retrieves the relevant code and provides them as a supplement to the code generation model. By guiding the code generation process with a retrieval pipeline, we see two added benefits. First, the performance of the overall code generation piepline can be iteratively increased by actively updating the database. Second, state of the art retrieval techniques can be used to increase performance of the retrieval pipeline which in turn increases the performance of the entire pipeline. Another similar task is code completion, it involves predicting the next segment of code based on a given code context. We see how to tackle that too. Hence, in this project, we try to make use of this retrieval-augmented code generation framework.

2.2 Code Generation with Planning

We build upon retrieval-augmented methodologies explained above, leveraging the planning capabilities of large language models (LLMs) to address challenges in code generation. Recent advanced in NLP have demonstrated the planning and reasoning capabilities of LLMs (Sun et al., 2024). Given a long task like code generation, we believe the LLM could be kept from hallucinating by coming up with a plan and making sure the LLM stays faithful to the plan across all the generation steps. To this end, we plan on utilize the planning capabilities of LLMs to come up with a plan or pseudo code beforehand and start generation based on the generated plan.

2.3 Datasets and Benchmarks

The MBPP (Many Bugs and Problems in Python) (Austin et al., 2021) dataset is a benchmark for code generation and completion tasks, focusing on generation of Python-based codes for a given description. CodeSearchNet (Husain et al., 2019) is a large-scale dataset designed for code retrieval, containing millions of code snippets across multiple languages. CodeXGLUE (Lu et al., 2021) is a comprehensive benchmark suite for code-related tasks. In this study, we utilized the MBPP dataset for both code generation and code completion experiments.

The Object-Oriented Programming (OOPs) benchmark (Wang et al., 2024) is a dataset of 170 challenging OOP-related questions, complete with test cases. We extended this dataset to suit our retrieval-augmented framework by generating gold-standard codes and high-level descriptions for the provided problems.

3 Interim Submission

For the interim submission, we worked on both the above tasks, by exploiting MBPP dataset (Austin et al., 2021). We used metods like TF-IDF (Aizawa, 2003), Retrieval Augmented Generation which are described below. 3.1 and 3.2 describe our methodology, experiments and results for both the tasks above respectively. We use Mistral-7B (Jiang et al., 2023) for both the tasks.

3.1 Code Generation using TF-IDF

For the code generation task, the input is a short, one-line overview of the expected output from the MBPP dataset. We decided to employ a TF-IDF-based retrieval system to retrieve the top 3 codes, using the top 1000 tokens with the highest TF-IDF scores. Since TF-IDF retrieval cannot be applied directly to the code, we generated LLM-based descriptions for all codes in the retrieval database. We then retrieve the top three codes by calculating cosine similarity or another similarity metric between the input task description and the descriptions of all codes in the retrieval database.

We decided to run the following experiments:

- **Experiment 1:** Zero-shot code generation without any retrieval.
- **Experiment 2:** Code generation with the top 3 codes retrieved using TF-IDF and the input task description.
- **Experiment 3:** Code generation with the top 3 codes and descriptions retrieved using TF-IDF and the input task description. We pass the descriptions along with the code snippets because we believe a detailed description of the task at hand would help the language model generate better code when compared to passing only the retrieved relevant code snippets.

The results can be found in Table 1.

Experiment	Score
Zero Shot Code Generation	0.413
TF-IDF retrieval (only code)	0.438
TF-IDF retrieval (code + descriptions)	**0.484**

Table 1: Results of Code Generation Experiments

3.1.1 Analysis

- From the results in Table 1, we can observe that retrieval-guided code generation using both code and descriptions significantly outperforms zero-shot code generation.
- Code retrieval-guided code generation also outperforms zero-shot code generation, but not by a large margin. This suggests that TF-IDF retrieval is not effective for retrieving relevant code snippets based on the LLM-based descriptions.
- We notice a spike in performance when the retrieved description is passed to the generation model along with the retrieved code snippet, which leads us to hypothesize that **a detailed task description is more helpful to an LLM than similar code snippets for generating new code.**
- These results suggest that we need to develop a better retrieval technique, as TF-IDF is not robust enough. We will be exploring semantic similarity-based description and code retrieval for our final retrieval pipeline.

3.2 Dense Passage Retrieval

For code completion, we used a dataset that was augmented with various types of corruptions. The corruptions were applied to simulate realistic scenarios where parts of the code might be missing, have syntactical errors, or contain logical flaws. We make controlled augmentations over the Test set

Figure 1: Example of an original data item.

Ground-Truth code of MBPP dataset by changing variable names, removing parts of code (cut_ratio) and adding some dead code in between. We use cut_ratio of 0.5 for our experiments explained below.

- **Experiment 1:** Evaluated the model's code completion ability using corrupted code snippets without additional context, relying solely on internal knowledge and coding patterns.
- **Experiment 2:** Assessed code completion with descriptions accompanying corrupted snippets, providing high-level functionality insights for more accurate completions.
- **Experiment 3:** Incorporated a retrieval step using a DPR-based system trained on CodeSearchNet (Husain et al., 2019) to fetch the top 3 relevant code examples and descriptions, enhancing the model's context and accuracy through external references.

Results for these experiments can be found in Table 2.

Experiment	Accuracy
Augmented Code (AG)	0.178
AC + Description	0.416
Top-3-DPR + AC + Desc.	**0.492**

Table 2: Code Completion Experiment Results

3.2.1 Analysis

The results presented in Table 2 highlight the following key observations.

- The baseline accuracy achieved using only augmented code (AG) is relatively low at 0.178. This indicates that without additional context or descriptions, the model struggles to accurately complete the corrupted code.
- Introducing descriptions alongside the augmented code (AC + Description) significantly improves the accuracy to 0.416. This improvement demonstrates that the additional contextual information helps the model better understand the intended functionality of the code.
- Finally, incorporating the top-3 retrieved codes (Top-3-DPR) alongside the descriptions (Top-3-DPR + AC + Desc.) leads to the highest accuracy of 0.492. This suggests that providing relevant code examples along with descriptions enables the model to make more informed and accurate predictions.

4 Code Generation with Planning

In this section we explain about our approach of utilizing planning capabilities of LLMs along with retrieval to perform code generation.

4.1 Dataset

We adapt OOPS benchmark dataset for this task. An example of the dataset entry is given in fig 1. It includes a unique "task-id" for identification and a "question" detailing the problem statement. Additionally, it contains a "test-list" with predefined assertions to validate the solution, a "test-function" to define the execution flow, and an "entry-point" indicating the starting function. A "test-matching" attribute specifies the required structure of the implemented classes and methods using string-based assertions, while the "test-match-function" evaluates whether the solution aligns with the required structure by iterating over the expected content. To utilize this dataset, we required more additions, which will be discussed in the next section.

```
class MPT:
    def __init__(self, difficulty):
        self.difficulty = difficulty

class SN_MPT(MPT):
    def __init__(self, difficulty, profit, worker):
        super().__init__(difficulty)
        self.profit = profit
        self.worker = worker

    def Maximum_profit(self, m, n):
        jobs = sorted(range(n), key=lambda x: self.difficulty[x])
        workers = sorted(range(m), key=lambda x: self.worker[x])
        assigned = [0] * m
        profit = 0

        for j in jobs:
            for i in range(m - 1, -1, -1):
                if assigned[i] < self.difficulty[j] and self.worker[workers[i]] >= self.difficulty[j]:
                    assigned[i] = self.difficulty[j]
                    profit += self.profit[j]
                    break

        return profit
```

Figure 2: A qualitative example from our adapted dataset of OOPS (Wang et al., 2024).

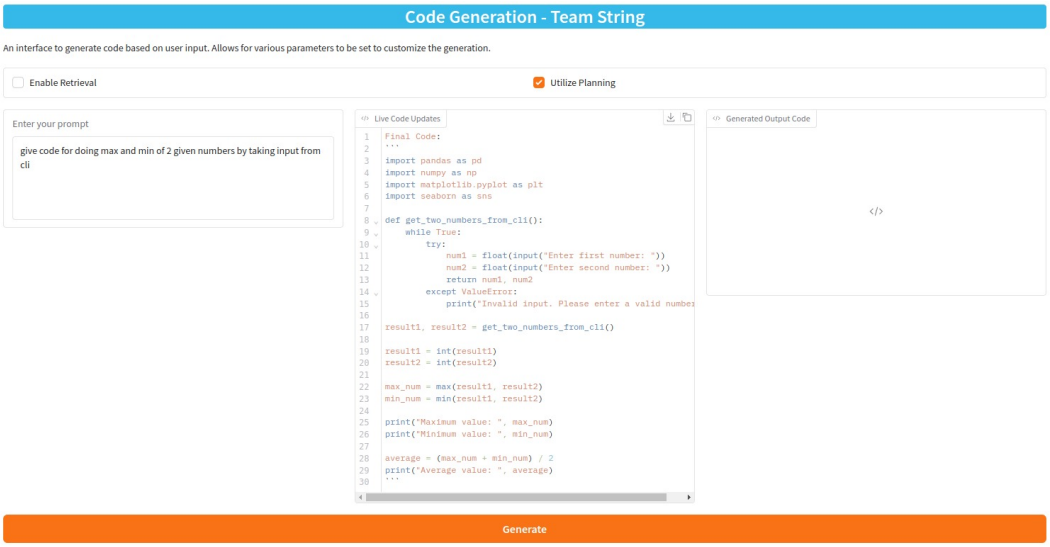

Figure 3: The web application developed for intuitive UX.

4.2 Dataset Expansion

We first generating Gold Standard codes for given queries. This helps while performing code generation using RAG as these codes can be appended to the question. We made use of Mixtral 8x7b (Jiang et al., 2023) to generate gold standard codes. We also generated brief high-level descriptions for these gold standard codes using llama-3.1-8b (Touvron et al., 2023) that enable retrieval. Figure 2 shows an example of the generated gold code.

4.2.1 Retrieval

We implemented a retrieval system to identify relevant code snippets and descriptions for a given query. By leveraging pooled hidden state representations from the *bert_base_uncased* (Devlin, 2018) model and cosine similarity measures, we retrieved the top-k (three in below experiments) relevant examples. This retrieval step significantly improved the performance of our pipeline.

4.3 Code Generation

We developed a web app (Fig. **??**] to allow for a user friendly intuitive experience while performing code generation using our pipeline. The web app allows for one to toggle retrieval and planning to observe generation with multiple features.

- **Without Retrieval, Without Planning.** This baseline approach generates code directly from the input query using an LLM, without any additional context or planning.

- **With Retrieval, Without Planning.** In this method, the LLM generates code using the input query and retrieved examples but without a structured plan.

- **Without Retrieval, With Planning** Here, the LLM generates code based on a detailed plan formulated for the task, but without any external examples or retrieval steps.

- **With Retrieval, With Planning** This approach combines both retrieval and planning, where the LLM uses retrieved examples and a step-by-step plan to generate code. It represents the most robust and reliable method in our pipeline.

  Planning Framework. We used the Llama-2 70B model to generate stepwise plans for each task. The process involves:

  1. Generating a plan for the overall task using a predefined prompt.
  2. Breaking the task into modular steps and generating code for each step.
  3. Combining the modular code snippets to form the final solution.

  Modular Code Generation. The modular approach not only simplifies code generation but also enhances error isolation, allowing for targeted debugging and modification of individual components.

```
Step: 1, plan: Create a class SID with an attribute A
class SID:
    def __init__(self):
        self.A = None
--------------------------------------------------------------------------------
Step: 2, plan:
Initialize the attribute A in the constructor of class SID
class SID:
    def __init__(self, A):
        self.A = A
--------------------------------------------------------------------------------
Step: 3, plan:
Create a class SN_SID that inherits from class SID
class SID:
    def __init__(self, A):
        self.A = A

class SN_SID(SID):
    def __init__(self, A):
        super().__init__(A)
--------------------------------------------------------------------------------
```

Figure 4: An example of planning taking place during code generation.

```
--------------------------------------------------------------------------------
Step: 12, plan:
Return the count of good starting indexes at the end of the function
class SID:
    def __init__(self, A):
        self.A = A

class SN_SID(SID):
    def __init__(self, A):
        super().__init__(A)

    def start_index(self):
        good_starting_indexes_count = 0
        smallest_even_jump_index = float('inf')
        smallest_odd_jump_index = float('inf')
        for i in range(len(self.A)):
            good = True
            max_reach = i
            for j in range(i, len(self.A)):
                if self.A[j] < self.A[i]:
                    good = False
                    break
                max_reach = max(max_reach, j + self.A[j])
                if max_reach >= len(self.A) - 1:
                    good_starting_indexes_count += 1
                    break
            if good:
                if i % 2 == 0 and i < smallest_even_jump_index:
                    smallest_even_jump_index = i
                if i % 2 != 0 and i < smallest_odd_jump_index:
                    smallest_odd_jump_index = i
        return good_starting_indexes_count
```

Figure 5: Final steps of planning taking place during code generation.

Accuracy	With Planning	w/o Planning
With Retrieval	0.437	0.165
w/o Retrieval	0.313	0.088

Table 3: Accuracy on OOPs Benchmark

Accuracy	With Planning	w/o Planning
With Retrieval	0.5662	0.499
w/o Retrieval	0.5424	0.436

Table 4: Accuracy of Llama 2 70B on MBPP Benchmark

4.3.1 Evaluation

Evaluation was done by running the given test cases on the generated code. The average accuracy was computed based on this. All numerical values presented in the tables are the obtained average accuracy.

4.3.2 Analysis

Retrieval's Impact. The retrieval step improved code quality and contextual relevance by providing the LLM with real-world examples. This was particularly evident in the OOPs benchmark, where gold-standard codes and descriptions guided the generation process.

Planning's Impact Incorporating planning significantly enhanced the coherence and logical flow of the generated code. By breaking tasks into smaller, manageable steps, the LLM was better able to handle complex problems.

5 Challenges

- **Limited Retrieval Quality:** With our implementation and TF-IDF retrieval method, we found it very hard to retrieve high-relevant code snip-pets, and this indicates the need for stronger retrieval methods to enhance the overall performance of code generation.
- **Dataset Augmentation Issues:** With our attempts to augment the MBPP dataset using corrupted code, it was tough to contain the level of realistic representation of errors in coding within a controlled bound that influenced the efficiency of our code completion experiments.
- **Balancing Code and Descriptions:** Adapting a balance between the amount of retrieved code generation was challenging because the effectiveness of additional descriptions depended on other codes.
- **Dataset Limitations:** Extending the OOPs benchmark required significant manual effort to ensure the quality of the generated codes and descriptions.
- **Scalability:** Integrating retrieval and planning at scale posed computational challenges, particularly when handling large datasets and models.

6 Contributions

The following are the contributions of the team members:

- **Advaith Malladi**: Retrieval with BERT, Code generation using planning in LLMs, Presentaion.
- **Lakshmipathi Balaji**: Literature Survey, Dense Passage Retrieval, Analysis, Report.
- **Abhinav Marri**: TF-IDF based RAG in code Generation, Dataset Expansion, Web Application, Report.

7 Conclusion

We demonstrate advancements in automated code generation and completion by leveraging retrieval-based augmentation techniques alongside the planning capabilities of Large Language Models (LLMs). We compared classical methods like TF-IDF and semantic retrieval methods in code generation. We provided an approach of semantic retrieval methods with modular and stepwise planning approaches for code generation. The experimental results validate the effectiveness of integrating retrieval and planning, with notable improvements in accuracy and contextual relevance. While challenges such as retrieval limitations and scalability remain, the methodologies developed here lay a strong foundation for future enhancements in code generation frameworks.

References

Akiko Aizawa. 2003. An information-theoretic perspective of tf–idf measures. *Information Processing & Management*, 39(1):45–65.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

Jacob Devlin. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*.

Albert Q Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, et al. 2023. Mistral 7b. *arXiv preprint arXiv:2310.06825*.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*.

Md Rizwan Parvez, Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Retrieval augmented code generation and summarization. *arXiv preprint arXiv:2108.11601*.

Simeng Sun, Yang Liu, Shuohang Wang, Dan Iter, Chenguang Zhu, and Mohit Iyyer. 2024. PEARL: Prompting large language models to plan and execute actions over long documents. In *Proceedings of the 18th Conference of the European Chapter of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 469–486, St. Julian's, Malta. Association for Computational Linguistics.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*.

Shuai Wang, Liang Ding, Li Shen, Yong Luo, Bo Du, and Dacheng Tao. 2024. Oop: Object-oriented programming evaluation benchmark for large language models. *arXiv preprint arXiv:2401.06628*.