AI Can Look Up StackOverflow too: Retrieval-Augmented Code Generation

Stanford CS224N Custom Project

Shreyas Vinayakumar
Stanford University
`vshreyas@stanford.edu`

Swagata Ashwani
Stanford University
`swag1506@stanford.edu`

Minh-Tue Vo-Thanh
Stanford University
`minhtuev@stanford.edu`

Abstract

This research paper investigates the effectiveness of retrieval-augmented generation for generating code from natural language queries for specific domains, without the need for fine-tuning a model on a domain-specific corpus. Code generation from natural language is a challenging task with numerous applications in software engineering and data science. Retraining or fine-tuning Large Language Models (LLMs) is an expensive undertaking, and LLMs have been shown to often effectively use in-context examples to arrive at the correct answer. The paper specifically explores using retrieval from a bimodal corpus with snippets consisting of description and code, to enhance prompting for code generation for data science problems in the Python programming language. Retrieval is performed against a corpus of StackOverflow posts using a weighted ensemble of 2 models, one for ranking query-code similarity and the other for query-description similarity. We compare our approach to 3 baselines using 2 widely used metrics on the DS-1000 data set. Experimental results show that (1) Our method is able to generate more correct programs, improving the $pass@5$ score by up to 10% compared to the baseline(sans retrieval) score of generative models such as codex-davinci-002 from OpenAI, achieving the new state-of-the-art. (2) Our method enhances robustness to semantic perturbation of the query text.

1 Introduction

The global software development market size was valued at 429.59 billion in 2021 [1] and is expected to grow at a compound annual growth rate (CAGR) of 17% until 2030. Software developers spend a lot of time on repetitive tasks such as boilerplate code and routine calls, which can be automated using code generation tools, freeing them up to work on higher-level logic and improving the industry's productivity. This motivates our research into natural language-based code synthesis, with the aim of taking a natural language description and generating a program in the target programming language(eg. Python) that implements the desired functionality. For instance, the system would be expected to generate Python code that calculates the sum of three numbers when presented with a natural language description: "I have 3 numbers a,b and c. Write a program that adds the numbers and returns the result". In this project, we focus on autoregressive(left-to-right completion) tasks, rather than fill-in-the-middle(insertion) tasks, although the same methods could conceivably be applied to both. We specifically focus on Python problems related to data science applications, which is a growing area with a number of specialized libraries. Few publications have explored subdomains within a programming language.

2 Related Work

Various approaches have been explored in the literature to facilitate code generation (Yin and Neubig, 2017; Gu et al., 2016). Currently, state-of-the-art performance is achieved by deep-learning-based sequence-to-sequence models, including well-known examples such as the InCoder model [2].

Despite recent advances made by these models, the quality of generated code is still poor in many cases, suffers from bugs and vulnerabilities [3], and uses deprecated APIs. Due to the high cost of training or even fine-tuning these models, it is hard to keep them up-to-date with the latest APIs. Additionally, since the training corpora for these models contain fewer examples from specialized domains such as graphics, web development and data science, performance is lower for these domains [4]. Since large-language models are few-shot learners that can use context provided during inference to produce better output [5], retrieval-based approaches aim to bridge this gap by allowing a general-purpose model to perform well on various domains without the need for specific fine-tuning.

Parvez et al [6] proposes a method that incorporates retrieval-based methods to augment the generation process. The authors demonstrate that this approach results in more effective code generation and summarization. They evaluated their method against generic Python and Java code generation benchmarks such as HumanEval and CoNaLa, but did not explore subdomains.

On another hand, in SkCoder, Li et al [7], presents a sketch-based approach to automatic code generation. The authors propose a method that allows users to specify the intent of the code they want to generate using a sketch, which is then transformed into code using deep learning models. They chose to use a simplistic approach for retrieval based on the BM25 dense-retrieval model [8]. This approach is demonstrated to be effective in generating correct code for various programming tasks, with some limitations when the retrieved code needs a lot of editing.

Chandel et. al [9] trained a model JupyT5 on all publicly available Jupyter notebooks and reported 77% accuracy on the Data Science Problems dataset. They have not made the details of their implementation publicly available for comparison, and it needs substantial effort in fine-tuning. The DSP dataset has simplistic problems meant for teaching students and is significantly less challenging than DS-1000 based on reported numbers.

Our approach is inspired by Parvez et al. We aim to enhance the code generation's effectiveness by retrieving relevant examples using similarity measure between the query and the description as well as between the query and the code, and including them in the prompt. In contrast with approaches such as RAG [10] which jointly train the retriever and generator on the end-to-end loss metric, we decouple the retrieval layer from the generation and allow them to be trained independently.

3 Our Approach

To select examples close to the prompt, we use the methods from Heyman et al [11]. We trained a Neural Bag-of-Words model to predict the similarity between query text and code snippets. The starter code for this model came from an open-source project codesearch.

In this model, the query representation is computed as a sum of the query word embeddings: $q = sum_{j=1}\mathrm{e2}(q_j)$. Similarly, an embedding for the code is computed by summing the code token embeddings weighted with their respective IDF scores: $c = \sum_{j=1} \mathrm{idf}(c_j)\mathrm{e2}(c_j)$, where $\mathrm{e2}(\cdot)$ is the embedding model that maps code tokens and query words to their respective embeddings. The embedding model $\mathrm{e2}(\cdot)$ is trained with fastText on a multi-modal text corpus containing both source code and natural language code description.

We also experimented with the BM25 retrieval algorithm but found that it wasn't effective in retrieving relevant snippets. While there are other more sophisticated models [12] [13] that can yield higher accuracy at the cost of more compute time (eg. dense retrieval for code), we believe this method offers a good tradeoff between accuracy and inference time which is critical for tasks like the generation of real-time code suggestions.

After the examples are retrieved, they are prepended in reverse order of their rank to the code generation prompt which is then passed into a code generation model such as the GPT-3 based OpenAPI codex[Figure 1].

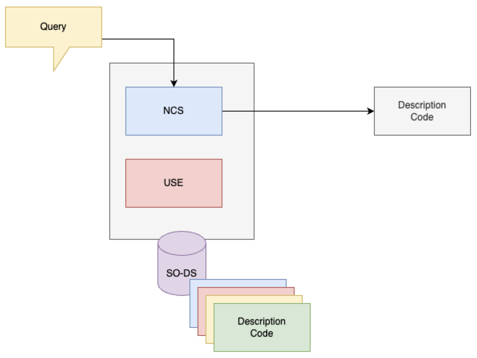


Figure 1: Architecture diagram of our retrieval system to generate description and code from query

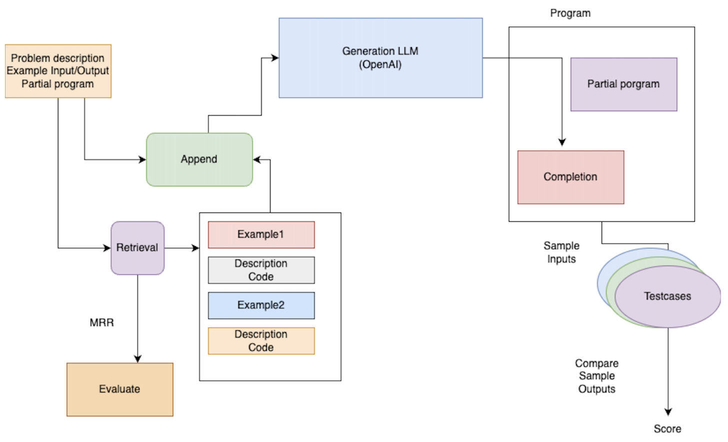


Figure 2: Our proposed system to enhance query prompts with retrieved descriptions and code

4 Experiments

4.1 Data

For building the retrieval database, and for training the retrieval models, we employ the SO-DS-20 dataset. The dataset was mined from StackOverflow posts from 2012-2020 with data science-related tags. Related pairs of text and code snippets were obtained by correlating duplicate posts.

For evaluating the whole end-to-end system, we employ the DS-1000 dataset. The DS-1000 dataset is designed to reflect diverse, realistic, and practical use cases, and is constructed with a reliable metric and perturbed questions to defend against overfitting. The DS-1000 dataset contains seven Python libraries (Numpy, Matplotlib, Sklearn, Tensorflow, Pandas, Pytorch and Scipy). In each library, there are two types of tasks: Completion and Insertion.

For Completion, the model must complete the program after the "<code>" token in the prompt. The resulting code will be executed to pass the multi-criteria automatic evaluation, which includes test cases and surface form constraints. For Insertion, the model must fill in the code in place of the "[insert]" token in the prompt, and the resulting code will be executed. The authors ensure that this data set is not seen by LLMs during pretraining, by not uploading the data in text format to public repositories and only distributing it as an archive. For our project, we are only interested in the Completion task in six relevant libraries (Numpy, Sklearn, Tensorflow, Pandas, Pytorch, and Scipy).

While there might be some overlap between the DS-1000 prompts and the retrieval database which were also mined from StackOverflow, the perturbation and rewrite done by the DS-1000 authors make the retrieval task challenging.

4.2 Evaluation method

For the evaluation of the retrieval system, we use *Mean Reciprocal Rank* (MRR) and *recall @ 3*,2 metrics commonly used in information retrieval tasks. recall@k measures the proportion of instances where the correct result (e.g., a relevant document or item) is found among the top k items returned by the system, while MRR measures the average of the reciprocals of the highest ranked answer among the returned results. While MRR is an order-aware metric, recall@3 only measures the percentages of relevant results and is relevant for cases where more examples are utilized in the prompt. The metrics were calculated on the so-ds-test dataset, which serves as a good proxy for the actual performance of DS-1000 since they are drawn from similar distributions. We lack ground truth for the DS-1000 samples, and evaluation of retrieval was done manually by annotating 60 sample queries as relevant or not.

For the evaluation of the end-to-end system, we use the pass@K metric. Pass@k for code generation systems [14] measures the number of problems where there is at least one correct sample among the k samples generated by the system. We choose the pass@k metrics because it is easy to measure and has consistent behavior(when compared with distance-based metrics like CodeBLEU) with respect to the high sensitivity of code to minor edits. The authors of the DS-1000 dataset used s slightly different metric, average pass rate of the generated samples, as the benchmark. We report the results for average pass rate as well in the Appendix, however pass@k appears more widely used and is suitable for most code generation tasks where only one sample is used in the end.

We have evaluated our approach extensively against the following code generation models that are available to us: - Codex-Cushman (max token budget = 2049) - Codex-DaVinci (max token budget = 4097) - Incoder 1B (max token budget = 1600 based on GPU memory constraints)

Owing to budget constraints, we were unable to evaluate more complex models that might provide better answers for code generation. However, this is a reasonable trade-off since we are interested in how to improve the performance of the available models using prompt augmentation.

4.3 Experimental details

For all the models we have chosen the following hyperparameters: max temperature = 0.2, top-p = 0.95. These are the same values as the parameters chosen in the DS-1000 paper.

For each request we fetch a batch of five samples; this is to ensure that even though each sample is randomly generated, we can generate a good representative distribution that can remain consistent over different runs. For our purpose, we generally restrict the max token length to 500 to 600 tokens as there is a strict OpenAI token budget limit and we are interested in the generation of short to medium-length completions.

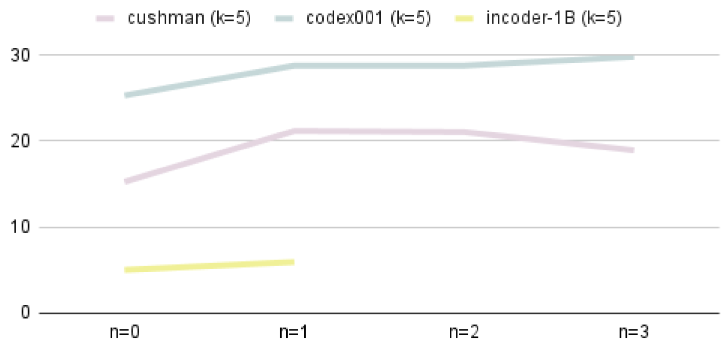
For the retrieval ensemble model, we have experimented with the following hyperparameter value: 0.3, 0.4, and 0.5. We trained the model on so-ds-train which has around 8000 snippets, split further into 80-20 for validation of hyperparameters. The training was run for 12000 steps with the recommended initial learning rate of 1e-5, although the loss on the validation set reached a trough at 5300 steps. The training process took 6 hours on a moderately sized AWS G4.large VM with a Tesla T4 GPU, which is a small fraction of the cost of fine-tuning.

For each prompt, we experimented with a different number of retrieval examples (1, 2, and 3). It is difficult for us to include a higher number of examples as it will most likely exceed the token budget of the model. To mitigate the problem with the token budget, we implemented logic to trim part of the retrieved description and code, in that order, to make sure that the total token budget for each request met the constraint. In rare cases where the prompt itself exceeds the token budget constraint, we tried to remove a minimal number of tokens from the prompt.

4.4 Results

Retrieval system evaluation: The ensemble was able to achieve an MRR of 0.29 on the SO-DS-TEST dataset and a recall@3 of 0.33. In comparison, BM25 was only able to achieve an MRR of 0.12, and using only query-code matching yielded an MRR of 0.08. On the DS-1000 dataset, out of 60 queries(prompts), only around 12 of the top retrieved results were found to be relevant. This number doesn't reflect the accuracy of the retrieval since DS-1000 may contain problems that are not represented in the corpus. We also measured the time to perform retrieval on various database sizes. This number did not vary significantly between database sizes of 5k and 10k snippets. The model was run with half-precision to allow execution on the available platform, an AWS G5dn.2xlarge instance(Intel Cascade CPU, Tesla A10G GPU). On average, a query took 0.084s to return, with an outlier being the first call that loaded the model onto the GPU which took 2s.

Figure 3: Pass@k value Retrieval Sample = n

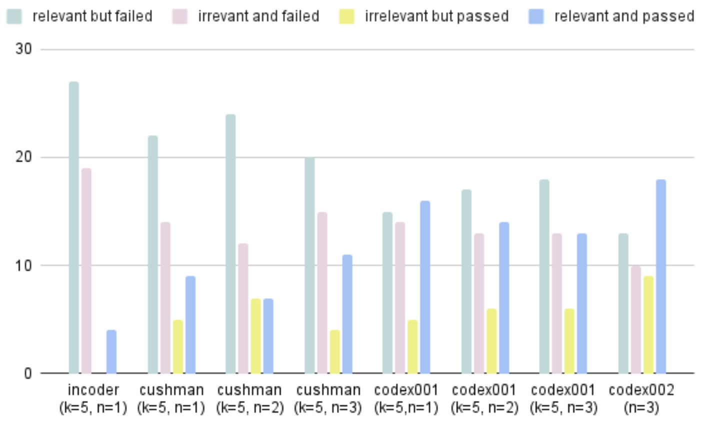


Retrieval-only performance: We measured the number of times the retrieval system actually returned a program that passed all the test cases for each question. We got a pass@1 score of 0.23% for the top ranked results, and 0.35% for the top 2 ranked results, indicating that retrieval alone is not sufficient to solve the problems and they can't be trivially looked up in the database.

Comparison of retrieval-augmented generation performance with generation-only performance: We experimented with various generative models including state-of-the-art decoder-only architectures(Codex models) as well as an encoder-decoder archicture(Incoder-1B), and were able to get performance improvements of up to 7% as shown in Fig. 3. We observed an improvement with more examples included in the context for the larger models. Additionally, we observed that even when we retrieve irrelevant results, in most cases it doesn't affect the performance(Fig. 5).

Robustness to perturbation of the problem text: The dataset includes several problems that were constructed by modifying the original text, adding confounding statements or rewriting the text in a way that makes it challenging for the generative model to memorize the results. We observe that our method increases robustness to certain types of perturbation as shown in Fig. 4.

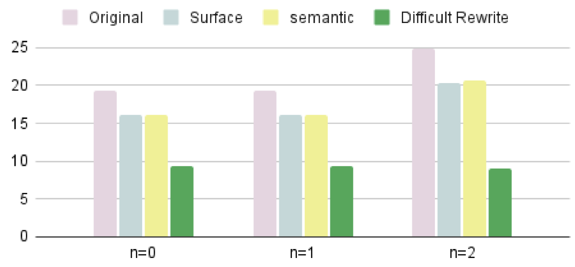
Figure 4: Retrieval Evaluation for Incoder,Cushman, Codex001 and Codex002



5 Analysis

From our manual inspection of a number of examples of failure from codex-cushman model, we see that a significant number of failures (40%) are near-miss, meaning that the generated code is almost correct and just suffers from minor syntactic errors or missing an argument. For most of the failures(90 %), the retrieval returned irrelevant results which likely led to a degradation in the quality of generative sampling. Around 20% of these problems had no close match in the corpus, probably because they used recent features introduced by the library that were not represented on StackOverflow at the time of collection of the corpus. In a real-world setting, the retrieved database could be updated frequently from various sources including online forums and API documentation to ensure that the examples are up-to-date. We have included some of the near miss examples in the Appendix section.

Figure 5: Accuracy drop by Perturbation type for Codex001



We also investigated the effectiveness of the token truncation algorithm, as it is necessary to remove tokens to meet strict token constraints. From our repeated experimentation, we found that truncating the description has few negative impacts on the quality of the results; however, truncating the actual problem statement leads to further degradation of the results.

6 Conclusion

In this project, we have designed and implemented a retrieval system that enhances the effectiveness of code generation for data science problems. Our system works well under restrictive token constraints. In particular, we have demonstrated that our method improves the pass@K metric for models and does not degrade the effectiveness even if retrieval is not relevant. We have also analyzed the output from our system through manual inspection and identified that we can further enhance the system by making the generative model avoid syntactic errors, and by improving quality of retrieved results by increasing the size of the corpus and training a better model. In the future, we are interested to evaluate our system against a range of available language models including general-purpose models such as BLOOM, coupled with other retrieval models such as ColbertV2. We also plan to do a deeper analysis of the code generated in both the cases, to see if there was qualitative improvement that was not captured by all the test cases passing. For example, we could categorize the failures based on compilation errors, the number of test cases passed and the CodeBLEU score compared to a few reference implementations. We plan to extend this approach to other domains and data sets as well.

References

[1] Grand View Research. Software development industry market research report 2021.

[2] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Scott Yih, Luke Zettlemoyer, and Mike Lewis. Incoder: A generative model for code infilling and synthesis. In *The Eleventh International Conference on Learning Representations*, 2023.

[3] Hossein Hajipour, Thorsten Holz, Lea Schönherr, and Mario Fritz. Systematically finding security vulnerabilities in black-box code generation models, 2023.

[4] Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Scott Wen tau Yih, Daniel Fried, Sida Wang, and Tao Yu. Ds-1000: A natural and reliable benchmark for data science code generation. *ArXiv*, abs/2211.11501, 2022.

[5] Mark Chen et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

[6] Md. Rizwan Parvez, Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. Retrieval augmented code generation and summarization. *CoRR*, abs/2108.11601, 2021.

[7] Jia Li, Yongmin Li, Ge Li, Zhi Jin, Yiyang Hao, and Xing Hu. Skcoder: A sketch-based approach for automatic code generation, 2023.

[8] Stephen Robertson and Hugo Zaragoza. The probabilistic relevance framework: Bm25 and beyond. *Foundations and Trends in Information Retrieval*, 3:333–389, 01 2009.

[9] Shubham Chandel, Colin B. Clement, Guillermo Serrato, and Neel Sundaresan. Training and evaluating a jupyter notebook data science assistant, 2022.

[10] Patrick Lewis, Ethan Perez, Aleksandara Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive nlp tasks, 05 2020.

[11] Georgi Heyman and Tom Van Cutsem. Neural code search revisited: Enhancing code snippet retrieval through natural language intent. *arXiv preprint arXiv:2008.12193*, 2020.

[12] Jose Cambronero, Hongyu Li, Seohyun Kim, Koushik Sen, and Satish Chandra. When deep learning met code search, 2019.

[13] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. Deep code search. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, page 933–944, New York, NY, USA, 2018. Association for Computing Machinery.

[14] Mark Chen et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

A Appendix (optional)

Here is an example of a near-miss (Numpy library, question 92):

Wrong code (with enhanced prompt):

```
result = np.argsort(a)[-N:]
```

Correct code:

```
result = np.argsort(a)[-N:][::-1]
```

Reference code:

```
result = np.argsort(a)[::-1][:N]
```

Since a significant number of failure cases result from near-miss, we expect that as the quality of the generation model improves we should see fewer such cases.