Dynamic Retrieval-Augmented Generation

Anonymous ACL submission

Abstract

Current state-of-the-art large language models are effective in generating high-quality text and encapsulating a broad spectrum of world knowledge. These models, however, often hallucinate and lack locally relevant factual data. Retrieval-augmented approaches were introduced to overcome these problems and provide more accurate responses. Typically, the retrieved information is simply appended to the main request, restricting the context window size of the model. We propose a novel approach for the Dynamic Retrieval-Augmented Generation (DRAG), based on the entity-augmented generation, which injects compressed embeddings of the retrieved entities into the generative model. The proposed pipeline was developed for code-generation tasks, yet can be transferred to some domains of natural language processing. To train the model, we collect and publish a new project-level code generation dataset. We use it for the evaluation along with publicly available datasets. Our approach achieves several targets: (1) lifting the length limitations of the context window, saving on the prompt size; (2) allowing huge expansion of the number of retrieval entities available for the context; (3) alleviating the problem of misspelling or failing to find relevant entity names. This allows the model to beat all baselines (except GPT-3.5) with a strong margin.

1 Introduction

In the area of natural language and code generation tasks, large-scale pre-trained language models (LLMs) such as T5 (Raffel et al., 2019), GPT-3 (Brown et al., 2020a), and LLaMA (Touvron et al., 2023) have made significant strides. However, the process of tuning these models is tedious and resource-demanding. These models have no knowledge on something outside of their training data and are difficult to update. The scientific community adopted the practice of prompting to fix this problem (Brown et al., 2020b). In this paradigm, the prompt includes the relevant information that helps the model to solve the task. To automate this process of knowledge grounding, the pertinent information is selected from a knowledge base by some similarity measure. These chunks of information are typically referred to as documents, and the method is referred to as the retrieval-augmented generation (RAG). For more information on such approaches, we recommend the survey by Li et al. (2022).

RAG methods, however, suffer from some issues. First, the performance of the generation is limited by the performance of the retrieval model — if the retrieval model fails to retrieve a relevant document, the generation model will be left with the information available during the training time or even distracted by irrelevant context, providing wrong answers. Second, tuning and inference of such models are resource-intensive due to the sheer size of the documents in the knowledge base, typically being prepared for humans to read, not for models to scrape.

To solve these problems, the community proposed a multitude of methods and solutions. In order to improve retrieval accuracy, researchers proposed to use the unaugmented model output as a query for the retriever (Zemlyanskiy et al., 2022) or to expand the query with the additional context generated by a language model (Mao et al., 2020). To ease the computational burden, the Memorizing Transformers method proposed by Wu et al. (2022) added a memory bank to the architecture and allowed the generator model to attend to entities from the memory bank, saving computations by caching.

In this work, we target a specific case of RAG that we call *generation with named entities*. For this problem, models can use names associated with the documents during generation. This setup appears in many tasks: for code generation, models can call methods defined in a project (Zhang et al.,

2023), for SQL generation (Yu et al., 2018) they can use table and column names, for Bash generation (Lin et al., 2018) – command flags. For all the mentioned tasks, knowledge of the available entity names is crucial for generation quality but cannot be captured during the model's initial pre-training. Moreover, the number of available project methods or SQL columns can be tremendous, which makes it infeasible to add them all to the context.

To account for named entities in the code generation task, the work on Repository-Level Prompt Generation (RLPG) by Shrivastava et al. (2022) and RepoCoder by Zhang et al. (2023) proposed specific ways of advanced context gathering.

In this work, we propose DRAG (Dynamic Retrieval Augmented Generation), a novel way to both retrieve named entities and use them during generation. DRAG first encodes the named entities and their contexts (*e.g.,* methods available in the project) to the latent space. Then, it integrates the embeddings into the generator's vocabulary so that during the token-by-token generation, it can also predict an entity name as a single token. The dynamic nature of retrieval comes from the fact that with DRAG, the generator has access to retrieval data embeddings at each token generation step and is aware of the current context. An important feature of DRAG is its ability to enhance existing language models, even though by further fine-tuning. We evaluate our method in Python, SQL, and Bash generation setups. For Python generation, we also collect and publish a large-scale dataset targeting project-level retrieval.

With this work, we make the following contributions:

- DRAG, a novel method for generation with named entities that can be used with existing language models to retrieve from a large set of entities on the fly and directly predict entity names during generation.
- A large-scale dataset for repository-level code generation that focuses on the evaluation of models' ability to identify and correctly use methods from the project at hand.
- Evaluation that demonstrates consistent performance improvements by extending the existing models with DRAG across three different domains on both publicly available datasets and our newly gathered one: Python generation, text-to-SQL task, and Bash generation.

2 Method

To define our method more precisely, we start with the notation. We denote one separate chunk of context as a document if it represents a description of a single named entity. For the sake of simplicity, we consider cases where extracting the entity name is trivial. For example, in the case of the code generation task, one document will be the function declaration, and the corresponding entity name will be the function name.

To improve the entity-augmented generation, we propose to reimagine the stages of the typical retrieval-augmented generation pipeline. Typically, the retrieval is first performed by a separate model, whether sparse, *e.g.*, BM25 (Robertson and Zaragoza, 2009), or dense, *e.g.*, DPR (Karpukhin et al., 2020). Then, the retrieved documents are fed into the generation model, either as an embedded representation, *e.g.*, in Memorizing Transformers (Wu et al., 2022), or as a sequence of tokens, *e.g.*, in RepoCoder (Zhang et al., 2023).

Following the findings by Morris et al. (2023), which show how much information is packed inside the embeddings provided by LLMs, we hypothesize that modern LLMs may not need the whole text of entities to gain useful information from the document. On the other hand, we note that there is an entire class of tasks where the retrieved documents coincide with the entities that models need to use during generation. For example, the Question-Answering task requires extracting information from the documents, but in the API call prediction in coding tasks, one type of the used documents is function code, while the entity to be used during generation is the function name. Current approaches, as discussed, rely on a retriever model to find relevant documents and then on a generator model to extract and use the entity name from the document.

We redefine the components of the retrieval-augmented generation pipeline as the *embedder* and *generator* models as shown in Figure 1. The proposed method goes through the following steps to generate one sample. First, all documents that can theoretically be used for the generation are considered available, *e.g.*, for the code generation — all code in the same repository will be considered available. The embedder compresses representa-

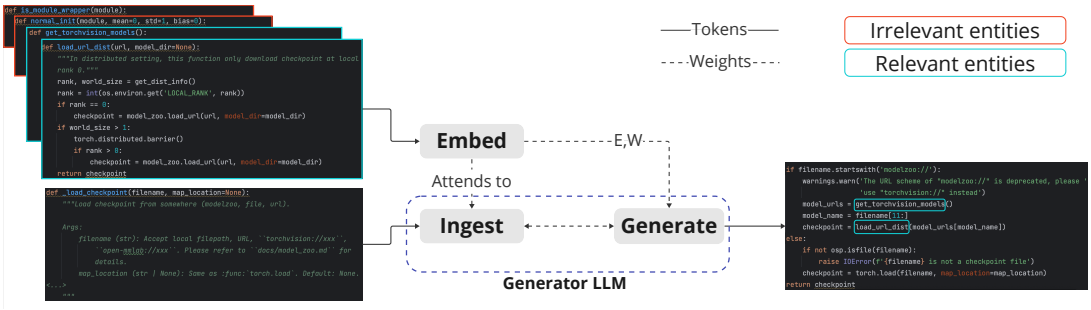

Figure 1: The proposed method architecture. The embedder model produces embeddings of all the documents associated with the sample using cross-attention with the input representation from the generator model. These embeddings extend the generator vocabulary, allowing the generator to decide at each step whether to generate a regular token or to insert an entity.

tions of all available documents into a set of embeddings. These embeddings are added (after some transformation) to the generator model's vocabulary as a single token for each entity name. The generator then performs the prediction in a usual token-by-token manner, while expanded vocabulary allows predicting the embedded representation of a document instead of a regular token. During the decoding process, the predicted embeddings are replaced with entity names directly extracted from the documents via extended vocabulary.

The change in the architecture serves several purposes. On the one hand, by incorporating the documents as the extension of the generator's vocabulary instead of the textual context in the prompt, we lift the length limitations of the context window. This, in turn, greatly expands the possible number of documents available for the model, given a good embedder model. On the other hand, by allowing the generator to refer to entities as a whole instead of generating their name token-by-token, we alleviate the problem of misspelling the entity name when copying (Rumbelow and Watkins, 2023) or failing to find them in the context if it is too long (Kamradt and Ashimine, 2023).

The common thread among the recent works in this area is leveraging pre-trained LLMs, without fine-tuning them. However, state-of-the-art LLMs require significant computational resources to run. We focus on small and medium-size models (300M–2B parameters), which we fine-tune for retrieval tasks. For that, we collect and publish a new large-scale dataset for project-level code generation.

The rest of this section describes particular elements of the proposed architecture. First, we describe the embedder and the generator models. Then, we address the training process.

2.1 Embedder

For this work, we consider dense embeddings provided by Transformer-based models. This way, we can keep the embedder consistent with the generator model. In particular, we consider tuning encoders from T5 (60M) (Raffel et al., 2019) and CodeT5 (60M) (Wang et al., 2021) for the architecture simplicity and demonstrated performance. In this work, the embedder model is also referred to as f_e.

To improve the document summarization and make it dependent on the input, we experiment with adding cross-attention between the embedder and the generator part responsible for the input ingestion,

$$softmax\left(\frac{Q_g^T K_e}{\sqrt{d}}\right) V_e,$$

where Q_g is the query matrix from the last layer of the generator model g, K_e and V_e are key and value matrices of the retriever e, and d is the hidden size. If the generator has an encoder-decoder architecture, we use the encoder. If the generator has the decoder-only architecture, we use a slice of the query matrix $Q_{g;1:T}$, where T is the length of the input in tokens. In Figure 1, this part of the generator is labeled as "Ingest".

If the inference efficiency is a larger concern than the quality, our method can offer two possibilities. The first one is to follow the Memorizing Transformers (Wu et al., 2022) and cache the K_e and V_e matrices. The second possibility is to remove the cross-attention between the embedder and the generator and cache the embeddings for the entire set of documents.

2.2 Generator

In this work, we consider the generator to be a Transformer with an encoder-decoder or decoder-only architecture. In the experiments, we use T5 (738M) (Raffel et al., 2019), OpenLLaMA (3B) (Geng and Liu, 2023), and CodeGen (350M, 2B) (Nijkamp et al., 2022) models for this purpose. We will refer to this model as f_g.

To allow the generator to select an entity from the set of available entities during generation, we extend the model's vocabulary with the entity embeddings provided by the embedder model. For a given input, we perform the vocabulary extension in the following way. First, all available documents $\{\mathcal{D}_i\}_{i=0}^{M}$ are converted into a set of embeddings $\mathcal{E}_i = f_e(\mathcal{D}_i)$. Next, we use two separate Multilayer Perceptrons (MLPs) f_ε, f_w to convert those embeddings into the columns of the generator's embedding matrix $E'_i = f_\varepsilon(\mathcal{E}_i)$ and the output linear layer $W'_i = f_w(\mathcal{E}_i)$, respectively. The input embeddings E and output linear layer W of the generator model f_g are then replaced by their extended versions $E \mapsto E\|E'$ and $W \mapsto W\|W'$, where $\|$ denotes concatenation. After this set of operations, we can treat the generator model as if we just extended its vocabulary with a set of special tokens. Note that the vocabulary is re-extended for each sample.

2.3 Training and Decoding

We propose two possible modes of the model training. The first is an end-to-end training of both the generator and embedder. The second is training only the generator, together with the MLPs f_ε, f_w. In both cases, we use the classical log-likelihood loss for the task of next token prediction, used for the language modeling.

We hypothesize that if the generator and the embedder are initialized with the same weights, there could be a way to fine-tune only the generator. Should this prove to be the case, apart from a usual generator model fine-tuning, we need to tune only the MLP connectors, converting the representations of the entities from the embedder's latent space to the space of the generative model's token embeddings. We test the hypothesis in Section 4.5 and for other experiments stick to training both the generator and embedder.

3 Related Work

Like the R2-D2 model by Fajcik et al. (2021), who composed a pipeline of a retriever, reranker, extractive reader, generative reader, and an aggregation mechanism, we aim to shorten the documents and provide the model with easier access to the entities. However, in contrast to them, we do not use the sequence-of-tokens representation at all, and we aim to solve the generation task.

Matching the PICARD model proposed by Scholak et al. (2021), which uses constrained beam search for execution-guided decoding to enforce syntax correctness, we interfere with the model's generation procedure, forcing it to produce correct entities. Yet, unlike this method, we do not change the sampling procedure but allow the model to select desired entities in the same way it selects usual tokens. The same difference lies between our work and ReCode (Hayati et al., 2018).

Similar to the Memorizing Transformers approach (Wu et al., 2022), which stores the documents in a large cached database, we transform the documents into their representations before adding them to the model. However, we do form the document set dynamically and do not explicitly retrieve the relevant documents as a separate step.

Finally, like REALM (Guu et al., 2020), RetGen (Zhang et al., 2022), RAT-SQL (Wang et al., 2020), ReACC (Lu et al., 2022), RepoCoder (Zhang et al., 2023), CoCoMIC (Ding et al., 2023) and many others, discussed in the survey by Li et al. (2022), we work on the approach of retrieval-augmented code generation. However, we combine it with entity-augmented generation and also combine the retriever and the generator in one model, relaxing some of the persistent constraints.

We would especially like to stress our connection with the RLPG and RepoCoder (Shrivastava et al., 2022; Zhang et al., 2023). RLPG proposed a heuristic set of rules to select a relevant context from the current repository and a model to predict which set of rules will work best for a specific case. RepoCoder iteratively alternates two stages: (i) retrieval based on the model prediction and (ii) prediction based on the retrieved information. RepoCoder is the standing SoTA for repository-level code generation. Therefore, we use it for experiments as a smart way to retrieve the context. However, contrary to RepoCoder, we do not modify the initial entity selection step and focus on how the

model ingests the context.

We acknowledge the concurrent method ToolkenGPT by Hao et al. (2023), who introduced tunable tokens for the tool-augmented LLMs. However, our key difference is the dynamic nature of the vocabulary extension proposed in our work.

4 Experiments

We evaluate the performance of different approaches across three datasets where the result needs to be generated using entities from the corresponding knowledge base: the novel dataset for project-level code generation, the NL2Bash dataset (Lin et al., 2018), and the Spider SQL dataset (Yu et al., 2018). For each dataset, we report task-specific metrics along with the *recall* for the generator's entity prediction. Recall is computed as a ratio of the entity names in the ground truth solution that were also predicted by the model.

The rest of this section is organized as follows. First, we discuss the baselines we will use to compare our model to. Then, for each task, we briefly describe the dataset, metrics, baselines, and results.

4.1 Baselines

We follow RepoCoder (Zhang et al., 2023) and RLPG (Shrivastava et al., 2022) when selecting baselines. We consider each method from two points of view: how it filters the context and how it utilizes the filtered context. For reference, the RepoCoder method provides a novel perspective on context filtering, retrieving the APIs from the repository in a multi-step procedure. However, it uses the classical way of context utilization, appending the text of the retrieved snippets to the prompt.

Context filtering methods

No designates the case where the context is completely absent. The only knowledge available to the generation model is stored in its weights and request.

All designates the case where the context comprises all possible knowledge, *e.g.*, the entire code repository or a knowledge base.

k-NN stands for the k Nearest Neighbours algorithm run on the distances in the embedded space. The relevant context is found by selecting 5 documents from the knowledge base that have the closest embeddings to the input. To produce the embeddings, we used BERT sentence embeddings (Devlin et al., 2018).

Oracle describes the context containing only documents to be used in the sample. This baseline tests the model's ability to utilize the provided context, given its highest quality.

RepoCoder (Zhang et al., 2023) proposed a novel way of the multi-step context collection. First, the unfinished code is utilized for the initial retrieval. Then, the sparse retrieval is used to obtain code snippets that are relevant to the current prediction from the whole repository. The retrieved snippets are fed into the generator model to complete the function body. After this initial step, an alternation of retrieval based on previous prediction and generation based on the new retrieval is used to improve the quality.

Context utilization methods

Prompt describes the situation when the generator model is given access to the context by appending it in textual form to the input query. This is the leading way to give additional context to the generator model, adopted by Zhang et al. (2023, 2022); Shrivastava et al. (2022) and others.

DRAG is the proposed method of the context utilization. We extend the model's vocabulary with special tokens representing the entities from the knowledge base. The model can then utilize these entities during the generation.

4.2 Project-level code generation task

Dataset. The dataset (see Section 5 for details on the dataset collection) consists of around 16,000 samples derived from around 1,000 GitHub repositories in Python. In this task, a model should generate a function body given the function signature and the docstring while using a set of functions implemented in the project. Functions available in the repository comprise a set of entities. We take function names as entity names and the whole function as the entity content. We follow the work by Evtikhiev et al. (2023) and utilize the ChrF (Popović, 2015) as the dataset-specific metric to assess code quality.

Models and Implementations. We implement this experiment's baselines and DRAG with the CodeGen-Mono-350M and CodeGen-Mono-2B models. Additionally, we report the results of the RepoCoder and *Oracle* context filtering with the GPT-3.5 model. The embedder model for the

Context	Utilization	chrF	recall
`CodeGen-Mono-350M`			
No	Prompt	23.1	0.07
RepoCoder	Prompt	25.2	0.27
Oracle	Prompt	27.9	0.22
All	**DRAG**	38.5	0.35
RepoCoder	**DRAG**	40.0	0.48
Oracle	**DRAG**	42.1	0.74
`CodeGen-Mono-2B`			
No	Prompt	33.7	0.14
RepoCoder	Prompt	29.1	0.32
`GPT-3.5`			
RepoCoder	Prompt	42.8	0.39
Oracle	Prompt	42.5	0.93

Table 1: Performance study of the proposed approach on the newly collected repository-level code generation dataset. The task is code generation given the function signature.

DRAG method was T5-small and was tuned together with the generator (w/o cross-attention).

Results Discussion. We report the collected metrics in Table 1. We draw three conclusions here: **(i)** DRAG consistently outperforms prompting methods given the same filtering and model size, **(ii)** DRAG enables feeding the model with all available entities at once, **(iii)** being supported with a good filtering strategy, a small model equipped with DRAG outperforms larger models without it.

4.3 NL2Bash task

Dataset. The dataset (Agarwal et al., 2021) consists of around 8,000 training samples and 600 samples for validation and testing. In this task, a model should generate Bash commands based on a natural language description, given the `man` page documentation. File names and the order of flags are not taken into account during evaluation.

Bash commands can form a chain or have nested commands. The distribution of utility usage across samples is far from uniform, with just several utilities adding up to 60% of samples. For this reason, the main challenge is to predict utility flags correctly. In this dataset, we use flag names as entity names and utility flag documentation as entity descriptions.

To measure the quality, we follow the advice of the dataset authors and measure the Exact Match (EM) together with the *recall* for the generator's entity prediction.

Context	Utilization	EM	recall
`LLaMa-3B`			
No	Prompt	7.1	0.06
k-NN	Prompt	12.2	0.34
All	**DRAG**	12.8	0.36
Oracle	**DRAG**	55.2	0.75
`GPT-3.5`			
All	Prompt	28.4	0.60

Table 2: Performance study on the NL2Bash dataset by Lin et al. (2018). The model used for this experiment is LLaMa-3B for the generator and T5-small for the embedder.

Models and Implementation. For this experiment, we implement the *No*-context and *k-NN* filtering baselines with Prompt utilization strategy and compare it with *All*-entities and *Oracle* filtering with DRAG. The model architecture is LLaMa-3B, with T5-small used as the embedder model for DRAG (with cross-attention). Additionally, as a reference point of the accessible results, we report the GPT-3.5 results prompted with all flags available for the relevant commands.

Results Discussion. From Table 2, we see that the proposed model outperforms all the baselines apart from the *Oracle*. We hypothesize that the small margin could be caused by the fact that the grammar allows combining flags and having several versions of the same flags. For example, the `--verbose -a` flag can be replaced by `-Va`. The inability to modify the entities, even when the model can learn the rules of such modification, is a notable disadvantage of the DRAG method. As an argument towards this hypothesis, we point out the performance of the `GPT-3.5` model prompted with all relevant flags.

4.4 Spider SQL task

Dataset. The dataset (Yu et al., 2018) consists of around 7,000 samples in the train set and 1,000 samples in the dev set. In this task, a model should generate an SQL query from a natural language input, given the database schema. Each sample from the dataset refers to its own database.

We use a popular data preprocessing technique from the Spider leaderboard, simplifying SQL queries using NatSQL (Gan et al., 2021). The NatSQL simplification essentially makes outer join on the tables in the database, providing table aliases to the columns in the `table.column` form. We

Context	Utilization	EX	recall
	T5		
No	Prompt	16.3	0.32
k-NN	Prompt	41.9	0.63
All	**DRAG**	63.9	0.78
Oracle	**DRAG**	80.8	0.97
	GPT-3.5		
All	Prompt	74.1	0.86

Table 3: Performance study of the model on the Spider dataset by Yu et al. (2018). The task is text-to-SQL. The models used for the test are T5 as the generator and T5-small as the embedder.

concatenate the column name with the corresponding values from the database records to form documents while using the `table.column` as entity name. The model trains to generate NatSQL queries, which are subsequently converted to the full SQL form. The dataset metrics are evaluated on the full SQL form.

We follow the dataset authors in using the Exact Match (EM) and Execution Accuracy (EX) to measure model performance. EM evaluates how much the generated SQL is comparable to the ground truth, while EX measures how well it matches the output of the execution of the generated SQL.

Models and Implementation. For this experiment, we used the T5 generator model, with T5-small as the embedder model (with cross-attention). Same as in Section 4.3, we implement the *No-context* and *k-NN* filtering baselines with Prompt utilization strategy and compare it with All-entities and *Oracle* filtering with DRAG. As a reference point, we report the results of GPT-3.5 prompted with all column names of the database.

Results Discussion. As we demonstrate in Table 3, the proposed method outperforms all baselines but the *Oracle* by a solid margin. The high score of the approach using all the available context together with the DRAG suggests that for cases with a high diversity of entities and the density of their appearance, the entity filtering step can be excluded altogether.

4.5 Practitioner Questions

Finally, we do an ablation study to evaluate parts of the DRAG method proposed in this work. We evaluate whether the entity embedder should be trained in order to improve the overall quality. Then, we compare our method of retrieving entity names from vocabulary to the conventional retrieval from the prompt. We conduct both experiments on the newly proposed dataset for code generation described in detail in Section 5.

Parts Tuned	chrF	recall	precision
E+G	38.5	0.35	0.27
G	39.0	0.34	0.30

Table 4: Ablation study on the joint training of the embedder (E) and the generator (G) models.

Is embedder training required? We hypothesize that given the adaptive abilities of the generative models, the joint training of the embedder and the generator models may not be needed. To check this, we additionally tuned one instance of DRAG without optimizing the embedder. The results are presented in Table 4. We conclude that training of the embedder model is not a strict requirement.

Is retrieving from the vocabulary beneficial compared to prompting? Our method introduced the novel entity retrieval approach, where the model can select an entity from the vocabulary instead of copying the entity name from the prompt. The performance experiments presented in Section 4 demonstrate that this is generally beneficial. To add to this comparison, we note that (i) DRAG outperforms Prompt when both are fed with the *Oracle* entities, and (ii) when fitting the full context in smaller models is impossible, DRAG is not only able to fit the full context in, but makes a step in quality towards the `GPT-3.5` level.

5 DRAG Code Generation Dataset

Existing datasets for code generation either do not have a repository-level context or are not tailored towards training or fine-tuning language models, comprising small sets of examples. For instance, the dataset collected by RLPG (Shrivastava et al., 2022) consists of 47 repositories. While sufficient to train a small classifier model, as proposed in their method, it will not satisfy the requirements for the generator model fine-tuning of our approach. The same appears to be true for the datasets from RepoCoder (Zhang et al., 2023) and PragmaticCode introduced by Agrawal et al. (2023), with 14 and 100 repositories, respectively.

Hence, to train our models for repository-level code generation, we collected a **repository-level code generation** dataset of around 16,000 samples

derived from around 1,000 GitHub repositories in Python, divided into train/dev/test sets in the ratio of 80/10/10 without an overlap of repositories between the sets. Each sample includes a method signature, the repository-wide context given as input, and the respective method body as target output.

We considered the following criteria for data mining to make baseline evaluation feasible. Firstly, the repository should be created later than 01.01.2022 to avoid data leakage, and have at least 10 stars and 5 watchers to select good-quality code (Kalliamvakou et al., 2014). Secondly, the target function with intra-project calls should (a) contain from 30 to 400 tokens to reduce the hallucination effect during sequence generation and (b) have a docstring. In addition, we paid careful attention to the licensing, and selected only repositories that have one of the following licenses: Apache-2.0, MIT, BSD-3-Clause, and BSD-3-Clause-Clear, which are the most prevalent permissive licenses for Python projects. This ensures that our dataset can be used for further studies.

On average, the collected repositories have 55 functions in the context (median 60), the target function has a length of 70 tokens (median 60), and there are 1.45 calls of project functions (median 1) per sample. We will publish the dataset alongside the paper upon acceptance. We attach a sample of data and source code to this submission.

To assess the models' quality on our dataset, we propose to focus on the quality of code generation as well as the entity retrieval quality, thus measuring how well the entity was retrieved and then utilized for the generation. We propose to use ChrF (Popović, 2015) to assess the quality of code generation following the study on metrics' quality by Evtikhiev et al. (2023). To assess the entity retrieval quality, we suggest computing recall and precision of entity prediction, comparing entity names in the reference solution and the ones predicted by the generator (without accounting for their positions).

6 Limitations

We consider the following limitations of our work.

The size of the models used for comparison is relatively low. While we consider our research substantial for small-scale models, larger models are lacking in this paper. We argue that the models of larger size are not likely to demonstrate worse results, and we conduct a comparison with baselines of larger size.

The predictions of the DRAG can not modify the predicted entities to make them match the context in the grammatical sense. This either requires a multi-pass approach or changing the model to accommodate flexibility. However, we argue that in some tasks—including code generation—such modifications are not needed.

7 Conclusion

In this paper, we proposed a novel pipeline for the dynamic retrieval-augmented generation (DRAG). We evaluated it on publicly available datasets, comparing it to both widely used baselines and state-of-the-art methods.

The new architecture achieves several targets: (1) increasing the length limitations of the context window, saving on the prompt size; (2) allowing a huge expansion of the number of retrieval documents available for the context; (3) alleviating the problem of misspelling or failing to find entity name when copying them from context. This allows the proposed approach to improve the results twofold compared to the baseline prompting.

For our experiments with repository-level code generation, we collected a novel large-scale repository-level code generation dataset. We will publish the dataset and the code to reproduce our method upon acceptance. We attach a sample of data and source code to this submission.

While we test our method in the Python, SQL, and Bash generation settings, we hypothesize that it can be beneficial in other generation cases where the model is required to use the pre-defined entities, *e.g.*, in legal or medical texts, documentation and question-answering systems.

Computational Resources and Libraries Used

We implement our code in PyTorch and use Transformer implementations from the HuggingFace library (Wolf et al., 2019). We also take pre-trained model weights from the HuggingFace hub. We conduct all experiments on eight NVIDIA A10 and one A100 GPUs using Ubuntu 22.04 LTS with 256 GB RAM. In all experiments, we optimize model weights with the Adam optimizer (Kingma and Ba, 2014) with `1e-4` learning rate and cosine learning rate schedule (Loshchilov and Hutter, 2016) for 20 epochs. The generation is performed with beam search decoding with a beam size of 5.

References

Mayank Agarwal, Tathagata Chakraborti, Quchen Fu, David Gros, Xi Victoria Lin, Jaron Maene, Kartik Talamadupula, Zhongwei Teng, and Jules White. 2021. Neurips 2020 NLC2CMD competition: Translating natural language to bash commands. *CoRR*, abs/2103.02523.

Lakshya A Agrawal, Aditya Kanade, Navin Goyal, Shuvendu K. Lahiri, and Sriram K. Rajamani. 2023. Guiding Language Models of Code with Global Context using Monitors.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D 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 Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020a. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc.

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. 2020b. Language Models are Few-Shot Learners.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805.

Yangruibo Ding, Zijian Wang, Wasi Uddin Ahmad, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. 2023. Cocomic: Code completion by jointly modeling in-file and cross-file context.

Mikhail Evtikhiev, Egor Bogomolov, Yaroslav Sokolov, and Timofey Bryksin. 2023. Out of the BLEU: How should we assess quality of the Code Generation models? *Journal of Systems and Software*, 203:111741.

Martin Fajcik, Martin Docekal, Karel Ondrej, and Pavel Smrz. 2021. R2-D2: A modular baseline for open-domain question answering. In *Findings of the Association for Computational Linguistics: EMNLP 2021*, pages 854–870, Punta Cana, Dominican Republic. Association for Computational Linguistics.

Yujian Gan, Xinyun Chen, Jinxia Xie, Matthew Purver, John R. Woodward, John Drake, and Qiaofu Zhang. 2021. Natural SQL: Making SQL easier to infer from natural language specifications. In *Findings of the Association for Computational Linguistics: EMNLP 2021*, pages 2030–2042, Punta Cana, Dominican Republic. Association for Computational Linguistics.

Xinyang Geng and Hao Liu. 2023. Openllama: An open reproduction of llama.

Kelvin Guu, Kenton Lee, Zora Tung, Panupong Pasupat, and Ming-Wei Chang. 2020. REALM: retrieval-augmented language model pre-training. *CoRR*, abs/2002.08909.

Shibo Hao, Tianyang Liu, Zhen Wang, and Zhiting Hu. 2023. ToolkenGPT: Augmenting Frozen Language Models with Massive Tools via Tool Embeddings.

Shirley Anugrah Hayati, Raphael Olivier, Pravalika Avvaru, Pengcheng Yin, Anthony Tomasic, and Graham Neubig. 2018. Retrieval-based neural code generation. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 925–930, Brussels, Belgium. Association for Computational Linguistics.

Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. German, and Daniela Damian. 2014. The promises and perils of mining github. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 92–101, New York, NY, USA. ACM.

Greg Kamradt and Ikko Eltociear Ashimine. 2023. Needle In A Haystack - Pressure Testing LLMs.

Vladimir Karpukhin, Barlas Oguz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. 2020. Dense Passage Retrieval for Open-Domain Question Answering. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 6769–6781, Stroudsburg, PA, USA. Association for Computational Linguistics.

Diederik Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *International Conference on Learning Representations*.

Huayang Li, Yixuan Su, Deng Cai, Yan Wang, and Lemao Liu. 2022. A survey on retrieval-augmented text generation. *CoRR*, abs/2202.01110.

Xi Victoria Lin, Chenglong Wang, Luke Zettlemoyer, and Michael D. Ernst. 2018. NL2Bash: A corpus and semantic parser for natural language interface to the linux operating system. In *Proceedings of the Eleventh International Conference on Language Resources and Evaluation (LREC 2018)*, Miyazaki, Japan. European Language Resources Association (ELRA).

Ilya Loshchilov and Frank Hutter. 2016. SGDR: stochastic gradient descent with restarts. *CoRR*, abs/1608.03983.

Shuai Lu, Nan Duan, Hojae Han, Daya Guo, Seungwon Hwang, and Alexey Svyatkovskiy. 2022. Reacc: A retrieval-augmented code completion framework. *arXiv preprint arXiv:2203.07722*.

Yuning Mao, Pengcheng He, Xiaodong Liu, Yelong Shen, Jianfeng Gao, Jiawei Han, and Weizhu Chen. 2020. Generation-Augmented Retrieval for Open-domain Question Answering.

John X. Morris, Volodymyr Kuleshov, Vitaly Shmatikov, and Alexander M. Rush. 2023. Text Embeddings Reveal (Almost) As Much As Text.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis.

Maja Popović. 2015. chrF: character n-gram F-score for automatic MT evaluation. In *Proceedings of the Tenth Workshop on Statistical Machine Translation*, pages 392–395, Lisbon, Portugal. Association for Computational Linguistics.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2019. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer.

Stephen Robertson and Hugo Zaragoza. 2009. The Probabilistic Relevance Framework: BM25 and Beyond. *Foundations and Trends® in Information Retrieval*, 3(4):333–389.

Jessica Rumbelow and Matthew Watkins. 2023. SolidGoldMagikarp (plus, prompt generation).

Torsten Scholak, Nathan Schucher, and Dzmitry Bahdanau. 2021. PICARD: Parsing incrementally for constrained auto-regressive decoding from language models. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 9895–9901, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

Disha Shrivastava, Hugo Larochelle, and Daniel Tarlow. 2022. Repository-Level Prompt Generation for Large Language Models of Code.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. Llama: Open and efficient foundation language models.

Bailin Wang, Richard Shin, Xiaodong Liu, Oleksandr Polozov, and Matthew Richardson. 2020. RAT-SQL: Relation-aware schema encoding and linking for text-to-SQL parsers. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 7567–7578, Online. Association for Computational Linguistics.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven C. H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. 2019. HuggingFace's Transformers: State-of-the-art Natural Language Processing.

Yuhuai Wu, Markus N. Rabe, DeLesley Hutchins, and Christian Szegedy. 2022. Memorizing Transformers.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. 2018. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 3911–3921, Brussels, Belgium. Association for Computational Linguistics.

Yury Zemlyanskiy, Michiel de Jong, Joshua Ainslie, Panupong Pasupat, Peter Shaw, Linlu Qiu, Sumit Sanghai, and Fei Sha. 2022. Generate-and-Retrieve: use your predictions to improve retrieval for semantic parsing.

Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023. RepoCoder: Repository-Level Code Completion Through Iterative Retrieval and Generation.

Yizhe Zhang, Siqi Sun, Xiang Gao, Yuwei Fang, Chris Brockett, Michel Galley, Jianfeng Gao, and Bill Dolan. 2022. Retgen: A joint framework for retrieval and grounded text generation modeling. *Proceedings of the AAAI Conference on Artificial Intelligence*, 36(10):11739–11747.