RAR: Retrieval Augmented Retrieval for Code Generation in Low Resource Languages

Anonymous ACL submission

Abstract

Language models struggle in generating correct code for low resource programming languages, since these are underrepresented in training data. Popular approaches use either examples or documentation to improve the performance of these models. Instead of considering the independent retrieval of this information, we introduce retrieval augmented retrieval (RAR) as a two-step retrieval method for selecting relevant examples and documentation. Extensive experiments on two low resource languages (Power Query M and OfficeScript) show that RAR outperforms example or grammar retrieval techniques (2.81–26.14%).

1 Introduction

Large language models (LLMs) struggle to generate low-resource programming languages from natural language due to limited pre-training knowledge (Luo et al., 2023; Wang et al., 2023b; Singh et al., 2023). Previous work improves code generation with language models with retrieval augmented generation with relevant examples (Poesia et al., 2022; Khatry et al., 2023b) and code snippets (Nijkamp et al., 2023), programmatic reasoning paths like tree-of-thought (Rosa et al., 2024) and program-of-thought (Chen et al., 2023), and documentation (Zhou et al., 2022).

There are several challenges in using documentation as context for code generation. First, documentation often does not include how the components are actually pieced together in the form of real code, which makes it difficult for models to understand the syntax and usage for new languages. Second, documentation is weakly correlated to the natural language utterance and specific parts of documentation might not be related at all to the utterance but crucial for the code generation. For example, the query, *"Highlight top 5 projects based on sales"* requires a flag `OrderByDescending` to be set, but this is not related to the query. Third, documentation is often dense and flat, and selecting the right subset of documentation is both challenging and crucial to the success of these systems.

To address these challenges, we propose Retrieval Augmented Retrieval (RAR) for code generation. This approach enhances the retrieval process by leveraging the outputs of an initial retriever, the *driver retriever*, to guide a secondary retriever, the *influenced retriever*. This sequential retrieval mechanism aims to improve the quality of the retrieved examples and grammar entities.

RAR addresses several challenges in code generation for low-resource languages. First, it addresses data scarcity by utilizing publicly available documentation and examples, and shows that these are often sufficient. Second, it leverages the strength of few-shot learning by providing high-quality, relevant examples that the model can learn from. Finally, it emphasizes the importance of leveraging linked grammatical structure and examples in the prompts, ensuring that the LLM generates correct and syntactically sound programs.

We evaluate RAR on two low resource languages, (Power Query) M and OfficeScript (OS). We compare individual context selection of RAR to multiple existing documentation and example retrieval techniques, showing improvements of +25% (OS) and +3% (M) for documentation and +1.28% (OS) and +3.5% (M) for examples. When combining examples and grammar, RAR shows improvements of +4% (OS) and +2% (M) over independent retrieval. We also analyze the impact of using two-step retrieval, including only relevant and irrelevant context items in the prompt, and the token length.

We make the following contributions:

- We use a two-step retrieval method, where the influenced retriever learns from the findings and mistakes of the driver retriever.
- We demonstrate that publicly available documentation is sufficient for NL-to-code genera-

tion tasks, even for low-resource languages.

- We perform thorough experiments to highlight the critical role of both documentation and example prompts, to evaluate the impact of token length, and to evaluate the embeddings used for the driver retriever.

2 Related Work

Multiple techniques have been developed to improve code generation from natural language with LLMs, including (1) retrieval augmented generation for adding contextually relevant examples to the prompt (Khatry et al., 2023b; Poesia et al., 2022); (2) execution-guided refinement (Kroening et al., 2004; Chen et al., 2019); and (3) reasoning involving chain-of-thought variants adapted to programming tasks (Li et al., 2023; Le et al., 2024).

These techniques often struggle in generating accurate generations for low-resource programming languages. Recent work focused on code generation for low resource languages has leveraged documentation as context instead of examples (Bareiß et al., 2022), (Zhou et al., 2022). CAPIR is one such popular technique, which uses contextually relevant parts of the documentation as inputs to code models. Grammar prompting (Wang et al., 2023a) also follows this paradigm. One drawback of these techniques is that documentation, even though complete, often does not provide the same signals to the models as examples. Documentation nodes that do not seem semantically aligned with the task also tend to be ignored.

3 Documentation and examples

Documentation is the most comprehensive and structured resource (Roehm et al., 2012) publicly available (Forward and Lethbridge, 2002) for most programming languages. The documentation consists of a grammar (D) that describes how code is built over entities (classes, methods, properties) and examples (E) that depicts how to use and combine elements from D. An example of grammar and examples from a page of the OfficeScript documentation is shown in Figure 1.

The **grammar** (D) serves as a bank for grammar elements over which retrieval is performed. We consider each grammar element g_i to be one standalone function or class method. We can use the path of each node in an abstract syntax tree (AST) to extract elements $\in D$ from a snippet of code. For example, even if multiple classes have a method `getFormat`, then following a path up the AST allows us to disambiguate which class this method is from. A full example of code and the associated grammar entities is shown in Figure 2.

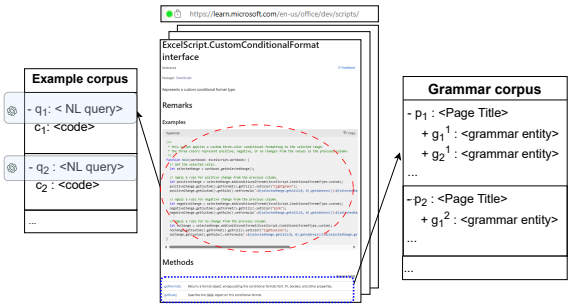


Figure 1: Illustrates how we extract the grammar (blue marker) and examples (red marker) from the publicly available documentation to build their respective corpora for retrieval.

The **example corpus** is composed of description (query) and code pairs (q_i, c_i). We only consider examples present in the documentation, which consists of sample code illustrating the usage of grammar elements. If a textual description of an example is not available, we use an LLM (`gpt-4`) to generate it. More information on this augmentation can be found in Appendix A.

4 Retrieval Augmented Retrieval

RAR uses a two-step retrieval where the driver retriever (R_D) influences the influenced retriever (R_I). There are two possible scenarios:

1. **Example → Grammar**: R_D retrieves from E and R_I retrieves from D.

2. **Grammar → Example**: R_D retrieves from D and R_I retrieves from E.

We first describe how to use and fine-tune embeddings for retrieval, and then show how this is applied in R_D and R_I in both these scenarios.

4.1 Embeddings for retrieval

Retrieval commonly relies on a transformer embedding cosine similarity

$$S(\mathcal{M}, q_1, q_2) = cos(\mathcal{M}(q_1), \mathcal{M}(q_2))$$

with $\mathcal{M}$ the embedding model used to compute the similarity between strings q_1 and q_2. We write $\mathcal{M}_{PT}$ for a pre-trained embedding and $\mathcal{M}_{FT}$ for a fine-tuned embedding.

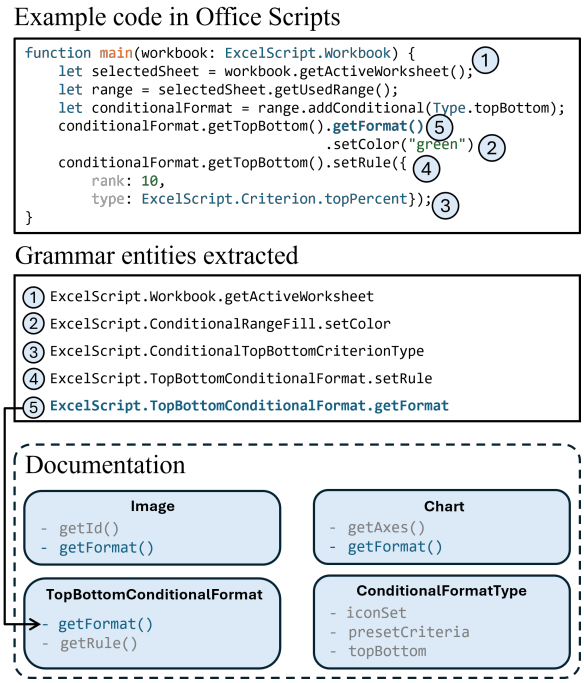


Figure 2: Example code entities (1 to 5) extracted from a sample OfficeScript program. The extracted entities are mapped to grammar nodes using the abstract syntax tree of the node. (5) in figure is mapped to `TopBottomConditionalFormat` despite the same property being present in `Image` and `Chart`.

Off-the-shelf embedding models struggle to generate accurate representations of code for low resource languages. To counter this, we fine tune the embedding model. We use a Siamese network architecture (Reimers and Gurevych, 2019a) with triplets $(q, J(g), L(c, g))$ forming the training set. $J(g)$ maps the grammar entity to a textual representation (see Appendix B). $L(c, g)$ evaluates to 1 if g is used in code c and -1 otherwise. To select negative labels, we find grammar entities g that are not used in code c, but which are closer to the decision boundary according to similarity $S(\mathcal{M}_{PT}, q, J(g))$ when compared with the test query, as well as an equal number of grammar entities which have the lowest similarities. More details on fine-tuning can be found in Appendix C.

4.2 Example → Grammar

In this setup, we retrieve examples first and then use it to retrieve the grammar elements.

First, R_D extracts top-k examples (E_k) based on $S(\mathcal{M}_{PT}, q_t, q_i)$ with q_t the target query.

Second, R_I uses E_k to select relevant n grammar entities D_n. We extract grammar entities from each code snippet c_i in an example $(q_i, c_i) \in E_k$, along with other similar entities from their respective documentation pages using $S(\mathcal{M}_{FT}, q_t, J(g))$. This set extracted forms the *good* grammar entities D_{good}. We also consider the possibility that irrelevant examples were retrieved. We thus want grammar element g that are similar to q_t ($S_t = S(\mathcal{M}_{FT}, q_t, J(g))$ is high) but dissimilar to the selected examples ($S_i = S(\mathcal{M}_{FT}, q_i, J(g))$ is low for $(q_i, c_i) \in E_k$). We combine this in a single score $(1 - S_i) + \lambda_{E\to D}(1 + S_t)$ that we want to maximize for each example i, where $\lambda_{E\to D}$ is a hyper-parameter that marks the relative importance of similarity to the query compared to dissimilarity between the example. This set is called D_{bad}. The final set of grammar entities is $D = D_{good} + D_{bad}$.

4.3 Grammar → Example

In this setup, we first retrieve the grammar entities and then use them to select examples.

First, R_D extracts top-n grammar entities (D_n) using $S(\mathcal{M}_{FT}, q_t, J(g))$ with q_t the target query.

Similar to R_I for $E \to G$, we consider successful (E_{good}) and unsuccessful (E_{bad}) retrieval of grammar. To ensure that we pay more attention to *unique* grammar entities, we compute the inverse-document frequency of such an entity as $idf(g)$. Let G_i be the grammar entities extracted from code c_i. The score of a good example is then computed as the average similarity of its entities, weighted by their idf

$$\frac{1}{|G_i|} \sum_{g \in G_i \cap D_n} idf(g) \cdot S(\mathcal{M}_{FT}, q_t, J(g)). \quad (1)$$

For building E_{bad}, we assume that D_n does not contain the grammar entities which would be relevant to answer q_t and compute the above score over $D \setminus D_n$ as S^i_{bad}. Similar to before, the bad examples are selected according to $S^i_{bad} + \lambda_{D\to E} S(\mathcal{M}_{PT}, q_t, q_i)$ with $\lambda_{D\to E}$ the importance factor of relevance of query to examples versus irrelevance of query to selected grammar.

5 Experimental Setup

We describe the experimental setup and the conditions set for a fair comparison between our approach and the baseline.

5.1 Datasets and metrics

We focus our experiments on two sets of programming languages: **OfficeScript** and **(Power Query) M**. We use *sketch* and *execution* match as metrics

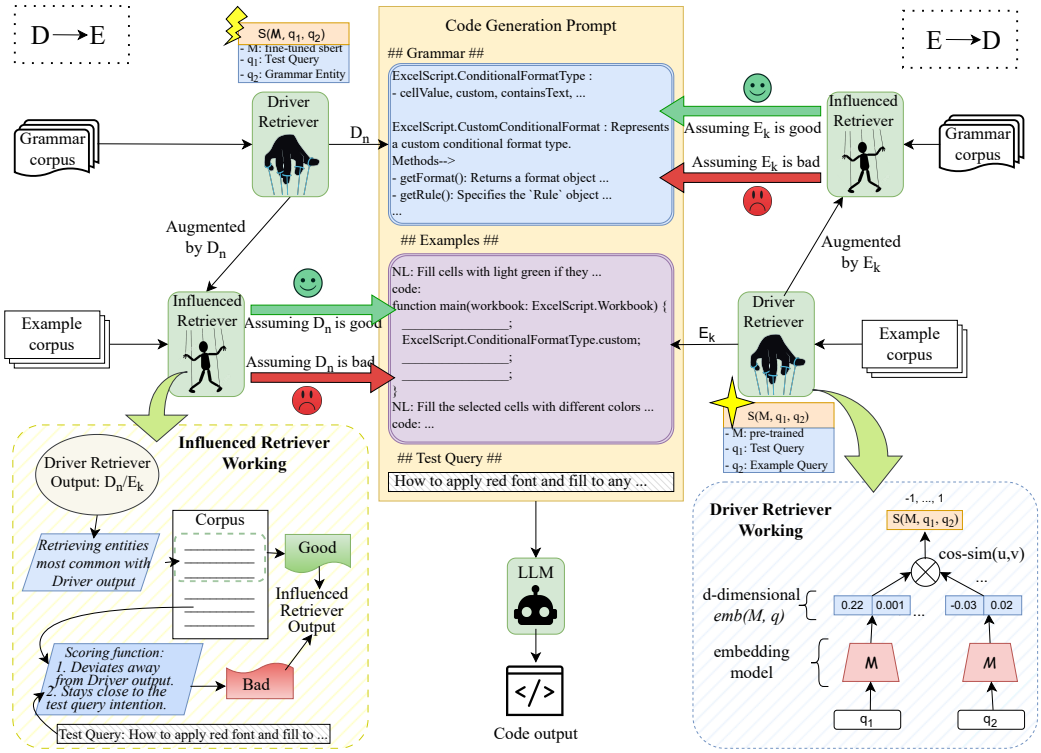


Figure 3: The top-left section demonstrates the scenarios where *Driver* retriever operates on the Grammar corpus and *influenced* retriever operates on the Example corpus. Similar for the top-right section where *Driver* operates on Examples and *Influenced* operates on Grammar. Towards the bottom right we demonstrate the working of *Driver* retriever which uses a simple transformer embedding similarity for extracting context. To the bottom left we define the *Influenced* retriever working which takes input from the *Driver* output and passes through a *good* and *bad* assumption of extracted context to generate its own retrieved context. The extracted context is fed into code generation prompt to pass onto the LLM.

for both datasets, details of which have been outlined in Appendix E.

OfficeScript We obtain (q, c) pairs from the InstructExcel benchmark (Payan et al., 2023) and filter them for conditional formatting specific tasks, as we can compute execution match for them (Singh et al., 2022). Examples and grammar are scraped from its documentation.[1]

Power Query M We use the test split of the benchmark release in (Khatry et al., 2023a). Besides q and c, each test contains a table to execute the code over, which is also provided in the prompt. We scrape the examples and grammar from the official documentation.[2]

[1] https://github.com/OfficeDev/office-scripts-docs-reference

[2] https://github.com/OfficeDev/office-js-docs-reference

Dataset	n	$\|E\|$	$\|D\|$
Office Scripts	589	17	275
Power Query M	77	144	746

Table 1: Summary of the datasets: n *implies* dataset size, $|E|$ *implies* #examples, $|D|$ *implies* #doc pages. We extract E and D from documentation which forms the corpora for our approach.

5.2 Baselines and Versions

We define the symbols representing the specifications of RAR and baselines.

- Ret$_D$: uses $\mathcal{M}_{FT}$ embeddings to retrieve D_n from D. Only D_n is included in the prompt.
- Ret$_E$: uses $\mathcal{M}_{PT}$ embeddings to retrieve E_k from E. Only E_k is included in the prompt.
- Ret$_{E\perp D}$: uses $\mathcal{M}_{FT}$ embedding to retrieve D_n from D and $\mathcal{M}_{PT}$ embedding to retrieve

E_k from E. Both D_n and E_k are included in the prompt.

- RAR$_D$: R_D operates on E to give E_k and R_I on D to give D_n. Only D_n is included in the prompt.
- RAR$_E$: R_D operates on D to give D_n and R_I on E to give E_k. Only E_k is included in the prompt.
- RAR$_{E\rightarrow D}$: R_D operates on E to give E_k and R_I on D to give D_n. Both E_k and D_n are included in the prompt.
- RAR$_{D\rightarrow E}$: R_D operates on D to give D_n and R_I on E to give E_k. Both D_n and E_k are included in the prompt.

5.3 Models

We use `text-embedding-ada-002` as the pre-trained embedding model $\mathcal{M}_{PT}$ and SentenceBERT (Reimers and Gurevych, 2019b) for $\mathcal{M}_{FT}$. We use GPT-4 (Brown et al., 2020) as the base LLM.

6 Evaluation

We aim to answer the following research questions:

RQ1 How does RAR compare against existing grammar and example retrieval methods?

RQ2 Does a two-step dependent approach extract better context than stand-alone retrieval techniques independent of one another?

RQ3 Does the adaptive strategy of including *Bad* context entities, along with *Good*, help in increasing the performance?

RQ4 How does the performance vary as a function of increasing context token length?

RQ5 Is RAR reliant on the *Driver* retriever for its performance gain over independent retrievers?

6.1 Compared with other SOTA (RQ1)

We evaluate RAR against other state-of-the-art retrieval methods over both examples and grammar. We use a fixed number of retrieved examples and grammar for each task, which are eventually prompted to GPT-4 for code generation. For OfficeScript, we extract 3 examples and 66 grammar entities. For M, we extract 10 examples and 20 grammar entities.

6.1.1 Baselines

For grammar retrieval, we consider (1) retrieval by calculating cosine similarity of pre-trained embedding model ($\mathcal{M}_{PT}$), (2) retrieval by calculating cosine similarity of fine-tuned embedding model ($\mathcal{M}_{PT}$), (3) **DocPrompting** (Zhou et al., 2022), which uses BM25 retriever, (4) **CAPIR** (Ma et al., 2024), which is a divide-and-conquer and re-ranking based strategy for retrieval.

For example retrieval, we consider (1) $\mathcal{M}_{PT}$, (2) **TST** (Poesia et al., 2022) and **TST**^{R} (Khatry et al., 2023b), which fine-tunes SentenceBERT and a small dense network on top of $\mathcal{M}_{PT}$ to make NL intents reflect their respective code similarities.

Model	Office Scripts		M	
	Sketch	Exec	Sketch	Exec
$\mathcal{M}_{PT}$	52.28	40.81	65.34	45.28
$\mathcal{M}_{FT}$	55.99	44.35	56.43	43.40
DocPrompting	50.17	38.68	73.64	50.94
CAPIR	51.69	41.06	71.07	55.68
RAR$_D$	**86.68**	**70.49**	**74.27**	**58.49**

Table 2: Comparing RAR against other *Grammar* retrieval techniques.

Models	Office Scripts		M	
	Sketch	Exec	Sketch	Exec
$\mathcal{M}_{PT}$	83.42	69.04	74.24	50.94
TST	64.76	52.95	70.21	51.16
TST^R	73.86	60.37	69.90	45.35
RAR$_E$	**85.67**	**70.32**	**76.29**	**54.72**

Table 3: Comparing RAR against other *Example* retrieval techniques.

6.1.2 Results

Table 2 and Table 3 show that RAR outperform the baselines for both grammar and example retrieval. RAR$_D$ shows significant gain in grammar extraction for OfficeScript. It has an execution match gain of 26.14% against the best performing baseline ($\mathcal{M}_{PT}$ with SentenceBERT). For M, we see an execution match gain of 2.81% over CAPIR.

We find RAR$_E$ to retrieve better examples to aid code generation. The respective baselines cover both pre-trained and fine-tuned (TST and TST^R) versions of retrieval. Our dependent retrieval strategy performs better than either case. For M, we find the improvement in both sketch and execution match to be marginal. This implies that the grammar entities retrieved by R_D is able to guide

the extraction of relevant examples for those NL queries, which were difficult to extract using direct similarity of embeddings.

We note that the fine-tuned models TST and TST^R perform worse than the unsupervised embedding model $\mathcal{M}_{PT}$. We attribute this to the fact that our training set is only scraped from documentation and thus smaller with low variations of the same function, and fine-tuning can more easily overfit.

6.2 Dependence vs Independence (RQ2)

To answer this question, we compare our dependent approach with baselines which operate independently on the two corpora.

6.2.1 Setup

For OfficeScript, we extract $n \approx 66$ grammar entities and $k = 3$ examples. For RAR$_{E\to D}$, we extract $k = 3$ examples first using R_D (which matches the output from Ret$_E$). We extract D_{good} and D_{bad} with equal proportion such that the final average count across all tests ≈ 66. We tune hyper-parameter $\lambda_{E\to D} = 20$ (see Appendix F).

RAR$_D$ is composed of the same D_n retrieved by *influenced* above. RAR$_{D\to E}$ uses the D_n extracted by *driver* (same as Ret$_D$) to augment the retrieval on E. We again choose E_{good} to be in the same proportion as E_{bad}. We set $\lambda_{D\to E} = 10$. This retrieved E_k is the same set used by RAR$_E$ in its prompt. Ret$_{E\perp D}$ uses both example retrieved by Ret$_E$ and grammar retrieved by Ret$_D$ in its prompt.

For M, we extract $n \approx 20$ grammar entities and $k = 10$ examples. RAR$_{E\to D}$ uses the same 10 examples retrieved by Ret$_E$ through R_D. D_{good} is obtained by taking the grammars extracted from each code snippet retrieved. To get D_{bad}, we set $\lambda_E = 100$ and extract 10 grammar for each example. The final de-duplicated version yields $|D_n| \approx 20$. For RAR$_{D\to E}$, we extract E_k using D_n obtained from *driver* (same as Ret$_D$). We get $|E_{good}| = 5$ and set $\lambda_{D\to E}$=10 to obtain E_{bad}.

6.2.2 Results

Table 4 shows that dependent retrieval (RAR) consistently performs better than independent retrieval (Ret) even if only a single type of context is provided.

Grammar Independent retrieval of grammar performs significantly worse than retrieving grammar through relevant examples (-25% for OfficeScript and -15% for M). This shows that RAR is able to pick more relevant documentation, without requiring examples to show how they should be used in the context of a program.

Context	Method	Office Scripts		M	
		Sk.	Ex.	Sk.	Ex.
G	Ret$_D$	55.99	44.35	56.43	43.40
	RAR$_D$	86.68	70.49	74.27	58.49
E	Ret$_E$	83.42	69.04	74.24	50.94
	RAR$_E$	85.67	70.32	**76.29**	54.72
G + E	Ret$_{E\perp D}$	87.18	72.34	73.40	58.49
	RAR$_{E\to D}$	**92.36**	**76.40**	72.87	58.49
	RAR$_{D\to E}$	90.71	76.01	74.86	**60.38**

Table 4: Comparison of RAR with independent retrieval techniques. Context implies whether only grammar (G), or examples (E), or both (G+E) have been included in the prompt for LLM. Methods with *Ret* are the independent retrievers with the subscript defining their corpus. The values denote match accuracy in %. RAR outperforms its Ret counterpart for all context scenarios.

Examples When independently retrieving examples, the difference between RAR and independent retrieval is smaller. Still, RAR consistently performs better. On M, retrieving only examples using RAR achieves the highest sketch match, indicating the similarity of the retrieved examples.

Grammar and examples Grammar + examples together yields better results than separate (+6% for OfficeScript and +2% for M). The examples help the model in figuring out the general program structure, and the documentation helps in figuring out how to adapt these examples (see Appendix G). This is highlighted in M where sketch match is highest when only using examples (+1.5% over RAR$_{D\to E}$), but execution match is significantly higher for the latter (+5%).

Recall in grammar Table 5 reports the recall of retrieving relevant grammar entities for Ret$_D$ and RAR$_D$. RAR beats independent retrieval again with a considerable margin (+25% for OfficeScript and +46% for M). The relevance of grammar extracted using RAR$_D$ further explains the jump in performance in Table 4.

6.3 Ablation (RQ3)

We evaluate whether the *good* and *bad* assumption of the driver retriever output actually helps the LLM to obtain relevant context or not. In this setting, for *influenced* retriever, we consider including only *good* or only *bad* extracted entities in the

Method	OfficeScript	Power Query M
Ret_D	50.04	24.83
RAR_D	**76.36**	**67.16**

Table 5: Comparison of retrieval quality when grammar is extracted either independently or with RAR. We evaluate quality by taking average of recall Rate (in %) for the occurrence of the retrieved grammar entity in the actual code for comparison.

prompt. We compare them with our proposed approach where we use an equivalent count of *good* and *bad*. The number of examples and grammar used in the prompt is kept constant across all scenarios for a fair comparison. Table 6 depicts the results for the ablation study.

RAR	R_I	Office Scripts		M	
		Sk.	Ex.	Sk.	Ex.
$E \rightarrow D$	D_{good}	79.80	64.01	72.38	58.49
	D_{bad}	83.47	69.98	74.22	52.83
	D_{good}+D_{bad}	92.36	76.40	72.87	58.49
$D \rightarrow E$	E_{good}	88.87	73.19	72.72	64.15
	E_{bad}	72.51	58.68	75.70	58.49
	E_{good}+E_{bad}	90.71	76.01	74.86	60.38

Table 6: Ablation to show importance of assuming R_D output to be both *Good* and *Bad* while retrieving for R_I. We find clear majority for Office Scripts. For PQ, we need both *Good* and *Bad* to attain a balanced performance improvement in both metrics.

We find that combining *good* and *bad* examples based on the result of the *driver* retriever output helps in obtaining better context. The retrieval of D_{bad} or E_{bad} is able to catch some important context which get missed when we trust the *driver* output to be *good*. For OfficeScript, we see a clear improvement in performance. However, for M, we find that sketch match is better for D_{bad} and E_{bad}, while execution is better for D_{good} and E_{good}. Using both in equal proportion helps us attain a balance when trying to improve both metrics.

6.4 Variation with token size (RQ4)

We vary the token size by changing the number of retrieved examples and grammar entities in the prompt. We use independent retrievers with the same context count as RAR as baseline.

Figure 4 shows that RAR performs better at most token counts. The only exception remains with Ret_E for M, where we find both sketch and execution below baseline for larger token sizes This happens because M has a larger example corpus

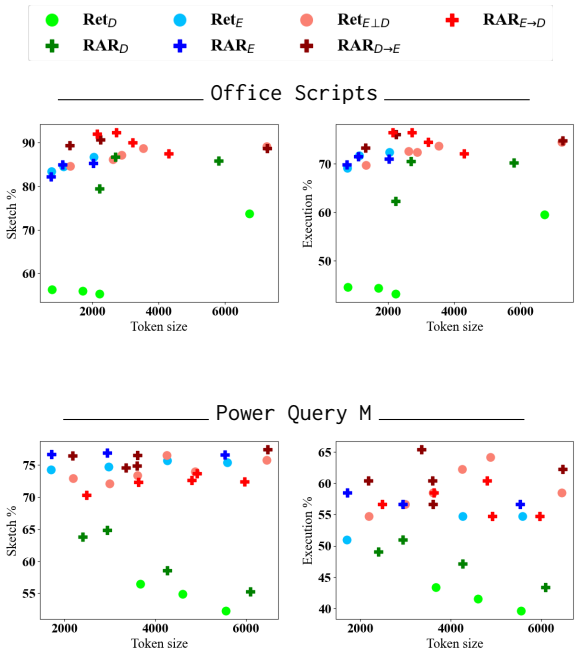


Figure 4: Shows a detailed comparison of RAR with the baseline independent retrievers as a function of increasing prompt token length. Plots on the left show *sketch* match accuracy and on the right show *execution* match accuracy. RAR outperforms its baseline even for large token sizes. We find lower token lengths are enough for accurate code generation.

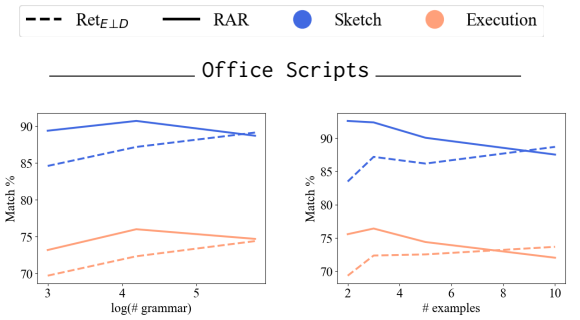
compared to OfficeScript. The *influenced*, even while considering *retriever* output to be *bad*, might be extracting functions from the same pool of incorrect intent. As a result, the performance is low. Moreover, even though the context size increases, the performance remains steady and does not increase further. This removes the notion of models trying to populate the prompt with more content rather than including only the relevant ones. We find that the best context is achieved around the $\sim$3000 token size mark. Further additions simply confuse (can be seen by a slight drop) or play no role in improving the quality of generation.

6.5 Reliance on Driver (RQ5)

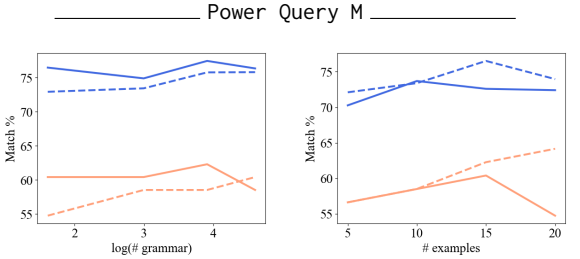
In this section, we consider evaluating code based on a prompt containing both documentation and example. We compare independent retrievers $\text{Ret}_{E \perp D}$ with $\text{RAR}_{D \rightarrow E}$ and $\text{RAR}_{E \rightarrow D}$, by altering the *driver* retrieval (1) output size, and (2) method. This helps us understand how R_I behaves as R_D changes. Including R_D's output in the prompt also enables us to understand the impact of R_I alone as we compare with the baseline, containing the same R_D output. This provides a clear

view on the impact R_I has towards performance improvement.

Increasing Driver output We find in Figure 5 that RAR is better than its baseline when both example and grammar retrieved from the *driver* is increased. There is a general trend of the match accuracy declining as we increase the output size. This implies that R_I is unable to infer a specific topic from R_D's output to make a *Good* or *Bad* assumption. So the retrieval becomes free and randomized, and it fails to converge to a particular topic for a candidate solution. We also find the match accuracy for RAR going below its baseline for M in $E \rightarrow D$ setting. The reason is again



(a) Influence of $n \uparrow (D_n)$ on RAR$_{D\rightarrow E}$ and Ret$_{D\perp E}$ (b) Influence of $k \uparrow (E_k)$ on RAR$_E$ and Ret$_{D\perp E}$



(c) Influence of $n \uparrow (D_n)$ on RAR$_{D\rightarrow E}$ and Ret$_{D\perp E}$ (d) Influence of $k \uparrow (E_k)$ on RAR$_E$ and Ret$_{D\perp E}$

Figure 5: Shows an impact on performance compared with baseline when the retrieved context size from *driver* is increased. Both the baseline and RAR in each setting have the same R_D output. The only thing which brings a performance difference is the output from R_I. Through this we show that R_I is not entirely reliant on R_D. It adapts itself to keep the performance above baseline with increasing context length.

similar to what we discussed in section 6.4. Additionally, when R_I tries to retrieve grammar from the extracted examples, the LLM receives some very relevant entities. It now has to decide a grammar from a list which has function descriptions very similar to the query, which causes confusion while choosing the exact grammar. On the other hand, independent retrieval extract grammar which is diverse. This makes identifying the right grammar from the available lot easy and hence results in a better match numbers when compared with RAR.

Altering retrieval method We compare Ret$_{E\perp D}$ with RAR$_{D\rightarrow E}$ for two different settings. One where we use pre-trained embeddings $\mathcal{M}_{PT}$ and the other where we use fine-tuned embeddings $\mathcal{M}_{FT}$ (SentenceBERT) for retrieval. Table 7 depicts that RAR still holds its ground and performs better than the baselines even when the retrieval style for R_D is changed. The *Good* and *Bad* retrieval assumption helps *influenced* retriever to adapt to the changing *driver*, and eventually fetches context which is relevant to the solution. This proves that our approach is not stringent in terms of the retrieval being used and it can adapt and perform well even with other retrieval techniques.

Method	Office Scripts		M	
	Sk.	Ex.	Sk.	Ex.
Pre-trained embed. model				
Ret$_{E\perp D}$	87.86	71.67	74.29	54.72
RAR$_{D\rightarrow E}$	**88.17**	**73.48**	**75.52**	**54.72**
Fine-tuned embed. model				
Ret$_{E\perp D}$	87.18	72.34	73.4	58.49
RAR$_{D\rightarrow E}$	**90.71**	**76.01**	**74.86**	**60.38**

Table 7: Shows that our approach performs better than the baseline even when different embeddings for retrieval is used. This further consolidates that R_I is independent of R_D and can even improve performance with other retrieval styles.

7 Conclusion

We introduce RAR, a two-step retrieval technique used to extract relevant context for code generation over low-resource programming languages. Our approach claims that off-the-shelf documentation for a language is enough to help an LLM generate syntactically and semantically correct programs. We also show how grammar and example work better together. Our approach establishes a working relationship between the two, capable of generating sound and reliable programs. The results we outline opens gates for future research, where grammar and example complement each other to formulate unseen programming languages.

8 Limitations and Ethical Considerations

Despite showing that RAR performs best at different token counts, combining both grammar and examples significantly increases the number of tokens and thus cost. Our method relies on extensive documentation, which might not be available for all low-resource languages.

We only scrape public documentation that is openly accessible. We do not use any unethical methods to extract data from sources that are protected by privacy policies.

References

Patrick Bareiß, Beatriz Souza, Marcelo d'Amorim, and Michael Pradel. 2022. Code generation tools (almost) for free? a study of few-shot, pre-trained language models on code. *Preprint*, arXiv:2206.01335.

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. 2020. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc.

Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W. Cohen. 2023. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *Preprint*, arXiv:2211.12588.

Xinyun Chen, Chang Liu, and Dawn Song. 2019. Execution-guided neural program synthesis. In *International Conference on Learning Representations*.

Andrew Forward and Timothy C. Lethbridge. 2002. The relevance of software documentation, tools and technologies: a survey. In *Proceedings of the 2002 ACM Symposium on Document Engineering*, DocEng '02, page 26–33, New York, NY, USA. Association for Computing Machinery.

Anirudh Khatry, Joyce Cahoon, Jordan Henkel, Shaleen Deep, Venkatesh Emani, Avrilia Floratou, Sumit Gulwani, Vu Le, Mohammad Raza, Sherry Shi, Mukul Singh, and Ashish Tiwari. 2023a. From words to code: Harnessing data for program synthesis from natural language. *Preprint*, arXiv:2305.01598.

Anirudh Khatry, Sumit Gulwani, Priyanshu Gupta, Vu Le, Ananya Singha, Mukul Singh, and Gust Verbruggen. 2023b. Tstr: Target similarity tuning meets the real world. In *Findings of EMNLP 2023*. Association for Computational Linguistics.

Daniel Kroening, Alex Groce, and Edmund Clarke. 2004. Counterexample guided abstraction refinement via program execution. In *Formal Methods and Software Engineering*, pages 224–238, Berlin, Heidelberg. Springer Berlin Heidelberg.

Hung Le, Hailin Chen, Amrita Saha, Akash Gokul, Doyen Sahoo, and Shafiq Joty. 2024. Codechain: Towards modular code generation through chain of self-revisions with representative sub-modules. *Preprint*, arXiv:2310.08992.

Jia Li, Ge Li, Yongmin Li, and Zhi Jin. 2023. Structured chain-of-thought prompting for code generation. *arXiv preprint arXiv:2305.06599*.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. Wizardcoder: Empowering code large language models with evol-instruct. *Preprint*, arXiv:2306.08568.

Zexiong Ma, Shengnan An, Bing Xie, and Zeqi Lin. 2024. Compositional api recommendation for library-oriented code generation. *ArXiv*, abs/2402.19431.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. Codegen: An open large language model for code with multi-turn program synthesis. *Preprint*, arXiv:2203.13474.

Justin Payan, Swaroop Mishra, Mukul Singh, Carina Negreanu, Christian Poelitz, Chitta Baral, Subhro Roy, Rasika Chakravarthy, Benjamin Van Durme, and Elnaz Nouri. 2023. Instructexcel: A benchmark for natural language instruction in excel. *Preprint*, arXiv:2310.14495.

Gabriel Poesia, Oleksandr Polozov, Vu Le, Ashish Tiwari, Gustavo Soares, Christopher Meek, and Sumit Gulwani. 2022. Synchromesh: Reliable code generation from pre-trained language models. *ArXiv*, abs/2201.11227.

Nils Reimers and Iryna Gurevych. 2019a. Sentence-bert: Sentence embeddings using siamese bert-networks. *Preprint*, arXiv:1908.10084.

Nils Reimers and Iryna Gurevych. 2019b. Sentence-BERT: Sentence embeddings using Siamese BERT-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 3982–3992, Hong Kong, China. Association for Computational Linguistics.

Tobias Roehm, Rebecca Tiarks, Rainer Koschke, and Walid Maalej. 2012. How do professional developers comprehend software? In *2012 34th International Conference on Software Engineering (ICSE)*, pages 255–265.

Ricardo La Rosa, Corey Hulse, and Bangdi Liu. 2024. Can github issues be solved with tree of thoughts? *Preprint*, arXiv:2405.13057.

Mukul Singh, José Cambronero, Sumit Gulwani, Vu Le, Carina Negreanu, Mohammad Raza, and Gust Verbruggen. 2022. Cornet: A neurosymbolic approach to learning conditional table formatting rules by example. *arXiv preprint arXiv:2208.06032*.

Mukul Singh, José Cambronero, Sumit Gulwani, Vu Le, Carina Negreanu, and Gust Verbruggen. 2023. Codefusion: A pre-trained diffusion model for code generation. *Preprint*, arXiv:2310.17680.

Bailin Wang, Zi Wang, Xuezhi Wang, Yuan Cao, Rif A. Saurous, and Yoon Kim. 2023a. Grammar prompting for domain-specific language generation with large language models. In *Advances in Neural Information Processing Systems*, volume 36, pages 65030–65055. Curran Associates, Inc.

Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi D. Q. Bui, Junnan Li, and Steven C. H. Hoi. 2023b. Codet5+: Open code large language models for code understanding and generation. *Preprint*, arXiv:2305.07922.

Shuyan Zhou, Uri Alon, Frank F Xu, Zhiruo Wang, Zhengbao Jiang, and Graham Neubig. 2022. Docprompting: Generating code by retrieving the docs. *arXiv preprint arXiv:2207.05987*.

A Generating NL queries for code examples

We obtain the code examples from the documentation which is easily scraped using regex expressions. However, NL queries accompanying the code examples are not provided for Office Scripts. Hence we generate them using an LLM (GPT-4). We use the prompt in Figure 6 to obtain queries which resemble a human as much as possible. For PQ, the NL queries are already available along with the code examples in the documentation and we directly use them to build the (q, c) pair for E.

B Grammar Representation used during retrieval

To calculate the transformer embeddings, we need to convert grammar entities to a suitable form. We use the representation function $J(.)$ for this conversion. We use the form:

```
<page>.<grammar>: <description>
```

An example for the representation used for grammar entities in office scripts:

```
ExcelScript.Range.setFormulaLocal: set the
formula in local A1 style...
```

An example for grammar entities in PQ:

```
Table.FromColumns: creates a table of type
`columns` from a list `lists` containing...
```

We use this representation when using transformer embedding for any form of retrieval done in the paper.

C Sentence-BERT Fine-tuning

For both the datasets, we build the training set as described in section 4.1. The train set has a ratio of 1:2 for positive and negative labels. The validation set is built by sampling data from the benchmark itself. We maintain a train-validation split of 80%-20%. Refer to Table 8 for more details on training and evaluation after training. We also include relevant graphs for further analysing the performance of our fine-tuning. Refer to Figure 7 for office scripts and Figure 8 for PQ.

Parameter	Office Scripts	PQ
training size	3390	3522
epochs	3	5
batch size	32	32
warmup steps	50	50
eval accuracy	54.08%	64.00%
ROC AUC	0.084	0.025

Table 8: Parameter details for fine-tuning S-BERT on Office scripts and PQ. We also include the final validation set accuracy and ROC AUC value as a measure for evaluating the training process.

D Code Generation Prompt

We provide sample prompt structures used for generating code in Office Scripts and PQ, for ease in replication of our results. Refer to the figures 9 and 10 for an example prompt we generate using RAR. More details about the representation is provided in the respective captions of the figures.

***Note** The sample data, table names and column header names used in the prompt for PQ have been parsed and extracted from the documentation itself. All examples in the documentation contain the `Table.From` function which enlists data on which the operation is performed. We extract data from these function arguments and replace them with the table or column names in the final version of the example program.

E Evaluation metric

In this section, we delve into more details about the techniques used for performing sketch and execution match on the code generated for the two datasets.

Office Scripts For *Sketch* match, we compare the LLM generated code with the ground truth to see if their functionalities match. We build a customized parser which is able to map functions in the code to create a symbolic mapping between them and the conditional format type (whose count is finite). This type of comparison ignores constants, variable values or function arguments. However, when doing *execution* match, we consider comparing all of the above fields. We compare the generated code with what we call revision records of the actual code. These records are Json representations which are obtained as a result of executing the ground truth code in Microsoft Excel. They contain information about the changes that happen in an Excel workbook after a code gets executed (in terms of target and type of impact). We draw heuristics to check if the code matches the changes in the revision records for execution accuracy.

PQ We perform *sketch* match by masking the constants, identifiers and other user-defined entities in the code and calculate the `SequenceMatcher`

```
<|im_start|>system
You are an expert in writing user queries. For a code given in Typescript (Office Scripts), write a query in natural language which
is similar in style to a query written by a human. Do not mention all the details as is present in the code and consciously try to miss
some information to make it appear more human-like. Keep the queries short and subtle and the content should be based on the
impact you want and not how it is accomplished using the code.
<|im_end|>
<|im_start|>user
function main(workbook: ExcelScript.Workbook) {
    let selectedSheet = workbook.getActiveWorksheet();
    selectedSheet.getRange("14:14").insert(ExcelScript.InsertShiftDirection.down);
    selectedSheet.getRange("C14").setFormulaLocal("=average(C9:C13)");
}
<|im_end|>
<|im_start|>assistant
Add row and calculate the Average.
<|im_end|>
<|im_start|>user
function main(workbook: ExcelScript.Workbook) {
    let conditionalFormatting: ExcelScript.ConditionalFormat;
    let selectedSheet = workbook.getActiveWorksheet();
    conditionalFormatting = selectedSheet.getRange("F3:F71").addConditionalFormat(ExcelScript.ConditionalFormatType.cellValue);
    conditionalFormatting.getCellValue().getFormat().getFont().setColor("#9C0006");
    conditionalFormatting.getCellValue().getFormat().getFill().setColor("#FFC7CE");
    conditionalFormatting.getCellValue().setRule({formula1:"=1",
        formula2:undefined,operator:ExcelScript.ConditionalCellValueOperator.greaterThan,});
}
<|im_end|>
<|im_start|>assistant
Highlight values greater than one in the range F3-F71.
<|im_end|>
<|im_start|>user
<code_example>
<|im_end|>
<|im_start|>assistant
```

Figure 6: Prompt used for generation of human-like NL queries for Office Scripts. The <code_example> highlighted in green is where the code examples extracted from the documentation goes for which the NL query needs to be generated. We design a few-shot setting for more similarity with the test query.

ratio of the two programs. For execution match, we obtain results by actually executing the programs in the data table that is already provided. We then compare the output to check for equality.

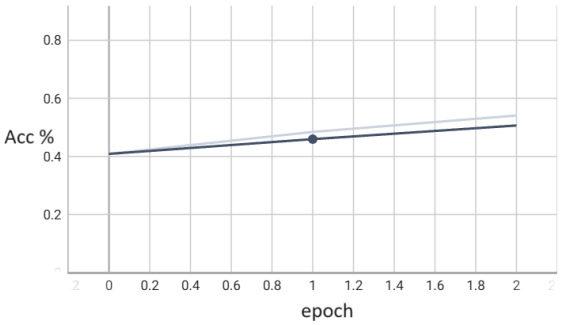
F Hyperparameter Tuning

We run different simulations of the experiment by altering the hyperparameter $\lambda_{E \to \mathcal{D}}$ and $\lambda_{\mathcal{D} \to E}$. For the sake of brevity, we show the variation in performance for only $E \to \mathcal{D}$ scenario for Office Scripts. For other, we do a similar sweep across different values and record that which gives us the best execution match.

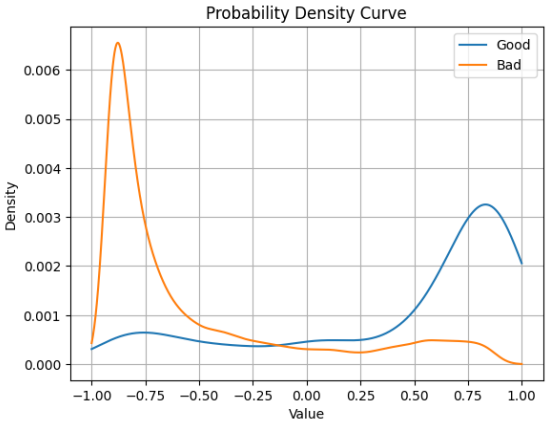
Figure 11 shows the increase in code match accuracy as we increase $\lambda_{E \to \mathcal{D}}$ and the drop and final stability for larger values. Lower values of impact factor implies that the deviation from the examples retrieved by R_D is favoured more. Hence grammar elements we include in the prompt are completely opposite in intent to the retrieved examples. As we increase $\lambda_{E \to \mathcal{D}}$, the impact of query intent comes in and the search becomes more organized towards looking for relevant grammar, rather than deviating blindly from the examples. A sweet-spot is reached near 20 which we use in our experiments. Further rise of $\lambda_{E \to \mathcal{D}}$ results in more impact from the query intent and the retrieved examples from R_D plays no role here. Hence we find the performance stabilising as we go higher. This shows that a balance between both search technique is required to identify the best and most relevant grammar with the help of *driver* retriever.

G Motivating example

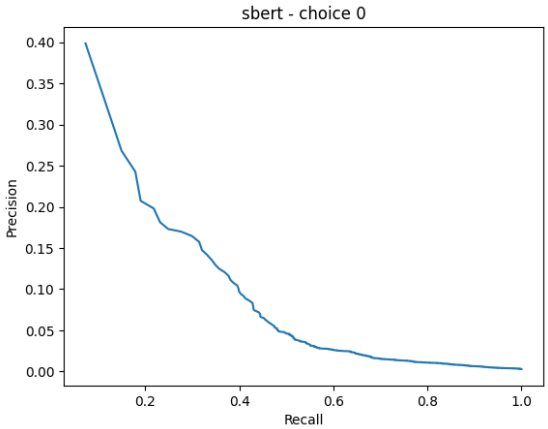
We showcase one motivating example which argues in favour towards including grammar along with examples in the prompt for code generation.



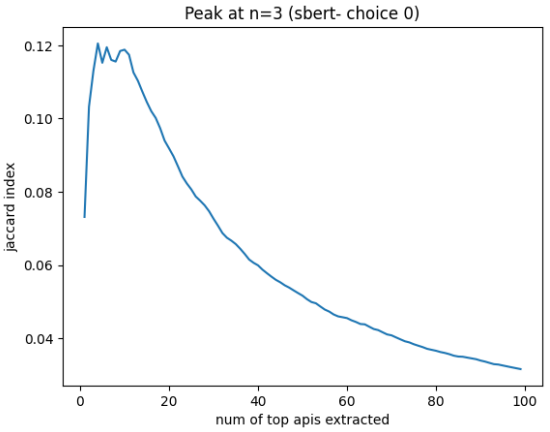
(a) Evaluation accuracy computed on the validation set as a function of increasing epoch. The training was terminated as we achieved maximum eval accuracy.



(b) The distribution of actual good and bad labels after training along the similarity score with their query in x-axis. This shows the demarcation on cosine-similarity score we are able to attain after fine-tuning s-bert.

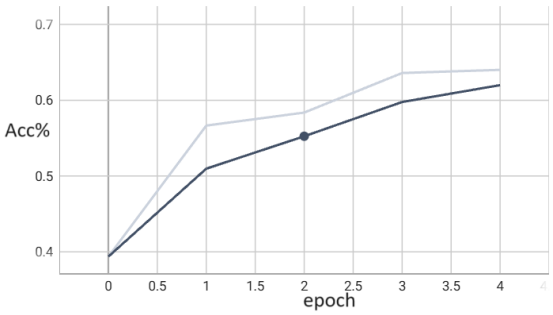


(c) Precision Recall curve obtained by varying the retrieval of top-n grammars from the documentation and calculating precision and recall by comparing with the grammar in the code. Each point in the plot corresponds to the average precision and recall across the entire benchmark.

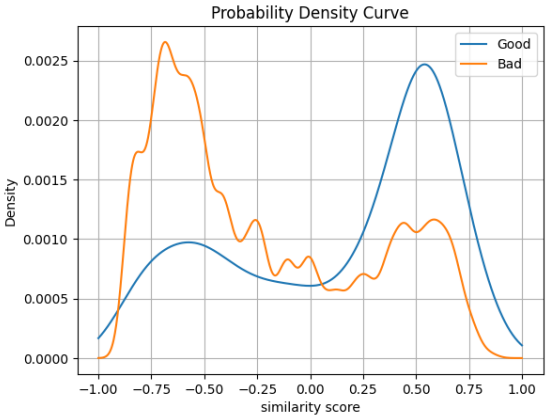


(d) Plot of jaccard index calculated against increasing number of grammar n retrieved from documentation, in order of decreasing cosine-similariy score with test query. Each point denotes the average value calculated across the entire benchmark. The best and condensed grammar is shown to be obtained for lower retrieval size.

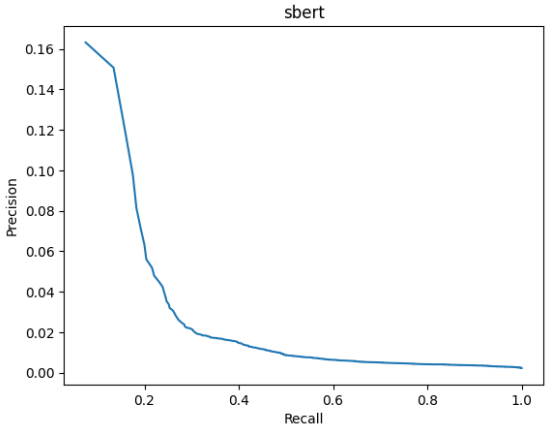
Figure 7: Analytical significance and performance upon fine-tuning s-bert on **Office Scripts**. The similarity scores are computed by taking the cosine of the trained embeddings of the query and the grammar elements.



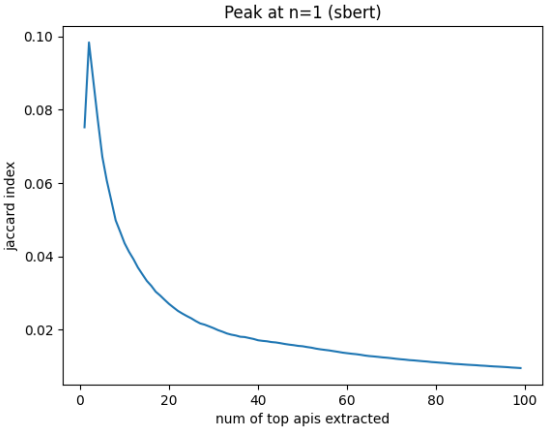
(a) Evaluation accuracy computed on the validation set as a function of increasing epoch. The training was terminated as we achieved maximum eval accuracy.



(b) The distribution of actual good and bad labels after training along the similarity score with their query in x-axis. This shows the demarcation on cosine-similarity score we are able to attain after fine-tuning s-bert.



(c) Precision Recall curve obtained by varying the retrieval of top-n grammars from the documentation and calculating precision and recall by comparing with the grammar in the code. Each point in the plot corresponds to the average precision and recall across the entire benchmark.



(d) Plot of jaccard index calculated against increasing number of grammar n retrieved from documentation, in order of decreasing cosine-similariy score with test query. Each point denotes the average value calculated across the entire benchmark. The best quality grammar is shown to be obtained for lower retrieved size.

Figure 8: Analytical significance and performance upon fine-tuning s-bert on **Power Query M**. The similarity scores are computed by taking the cosine of the trained embeddings of the query and the grammar elements.

```
<|im_start|>system
You are an assistant that generates programs in typescripts using Office Scripts APIs. The program you generated will help solve a query provided by the user. The Office Script you generate will be run directly in Microsoft Excel and the effect it has on the workbook will be checked to confirm if the code you generated matches the user content or not. If target where the change is applied on the sheet is not mentioned in the user query, assume a target range or column of your own and complete the program. Your code should be complete and not have any placeholders. In case of ambiguity in terms of which color or formatting style to use, use your best judgement and make assumptions at the same time mentioning your assumptions as comments. Your code must be syntactically and semantically correct. To help you generate correct and quality programs, a list of methods, properties and enums are provided from the official Office Scripts documentation along with some examples of their usage.
<|im_end|>
```

```
# Documentation #

ExcelScript.IconSetConditionalFormat: Represents an icon set criteria for conditional formatting.
Methods-->
- getCriteria() : ConditionalIconCriterion[]
- setCriteria(criteria: ConditionalIconCriterion[]) : void

ExcelScript.ConditionalIconCriterion: Represents an icon criterion which contains a type, value, an operator, and an optional custom icon ...
Properties-->
- customIcon (ExcelScript.Icon): The custom icon for the current criterion, if different ...
- type (ExcelScript.ConditionalFormatIconRuleType): What the icon conditional format should be based on.

ExcelScript.ConditionalFormatIconRuleType: Represents the types of icon conditional format.
- formula, invalid, number, percent, percentile
```

```
# Examples #
<|im_start|>user
Apply traffic light icons to the range A1-A5 based on percentage values.
<|im_end|>
<|im_start|>assistant
function main(workbook: ExcelScript.Workbook) {
    const sheet = workbook.getActiveWorksheet();
  const range = sheet.getRange("A1:A5");

    const conditionalFormatting = range.addConditionalFormat(ExcelScript.ConditionalFormatType.iconSet);

    conditionalFormatting.getIconSet().setStyle(ExcelScript.IconSet.threeTrafficLights1);

    const criteria: ExcelScript.ConditionalIconCriterion[] = [
    {
      formula: '=0', operator: ExcelScript.ConditionalIconCriterionOperator.greaterThanOrEqual,
      type: ExcelScript.ConditionalFormatIconRuleType.percent
    },
    {
      formula: '=33', operator: ExcelScript.ConditionalIconCriterionOperator.greaterThanOrEqual,
      type: ExcelScript.ConditionalFormatIconRuleType.percent
    },
    {
      formula: '=67', operator: ExcelScript.ConditionalIconCriterionOperator.greaterThanOrEqual,
      type: ExcelScript.ConditionalFormatIconRuleType.percent
    }];
  conditionalFormatting.getIconSet().setCriteria(criteria);
}
<|im_end|>

 ...
```

```
<|im_start|>user
How do I add a three-arrow icon set to the range M8-M14 on the 7a sheet based on percentage criteria?
<|im_end|>
<|im_start|>assistant
```

Figure 9: Shows the following prompt stucture which is passed to the LLM for code generation. The above prompt example is for **Office Scripts** generation. We highlight the representation used for grammar in the prompt. The green and yellow highlighted represents the match between the code and grammar and on how the return type of a grammar element is mapped to its definition in the same prompt. This helps the LLM establish an understanding of the function chaining strategy used in the example program so that it is able to use the right function in the right place where it's return type matches.

```
<|im_start|>system
You are an assistant that answers questions from a table by converting them to Power Query M queries.
For your reference, you are also provided relevant functions form documentation to assist you in generating the queries.
<|im_end|>

# Documentation #

Replacer.ReplaceText : Replaces the `old` text in the original `text` with the `new` text. This replacer function can be used in
`List.ReplaceValue` and `Table.ReplaceValue`.

Replacer.ReplaceValue : Replaces the `old` value in the original `value` with the `new` value. This replacer function can be used in
`List.ReplaceValue` and `Table.ReplaceValue`.

Table.ReplaceErrorValues : Replaces the error values in the specified columns of the `table` with the new values in the `errorReplacement`
list. The format of the list is {{column1, value1}, ...}. There may only be one replacement value per column, specifying the column more
than once will result in an error.

Table.ReplaceValue : Replaces `oldValue` with `newValue` in the specified columns of the `table`.
...

# Examples #

<|im_start|>user
Columns: A, B
Sample Data: [[1, "hello"], [2, "wurld"]]
Table Name: Table1
Question: Replace the text "ur" with "or" in column B, matching any part of the value.
<|im_end|>
<|im_start|>assistant
M: Table.ReplaceValue(Table1,"ur", "or", Replacer.ReplaceText, {"B"})
<|im_end|>

<|im_start|>user
Columns: A, B
Sample Data: [[Error, Error], [1, 2]]
Table Name: Table1
Question: Replace the error value in column A with the text "hello" and in column B with the text "world" in the table.
<|im_end|>
<|im_start|>assistant
M: Table.ReplaceErrorValues(
    Table1,
    {{"A", "hello"}, {"B", "world"}}
)
<|im_end|>
...

<|im_start|>user
Columns: Column1
Sample Data: [['aaa bbb ccc ddd eee fff'], ['aaa bbb ccc ddd'], ['aaa ccc ddd eee fff'], ['aaa bbb ccc fff'], ['aaa bbb ddd fff'],
['aaa bbb ccc ddd eee fff'], ['aaa bbb ccc ddd eee fff'], ['aaa ccc ddd eee fff'], ['aaa bbb ccc ddd fff'], ['bbb ccc ddd eee fff']]
Table Name: Source
Question: replace only "eee" with "xxx-eee", in Column1 in table Source
<|im_start|>assistant
M:
```

Figure 10: Shows the prompt structure used with grammar and examples for **PQ** generation. The code requires he sample data from table, the table name and its column headers to write specific code, which is executable and also easy to match with the ground truth. We also provide this table metadata for the test query for complete code generation without any assumptions. The documentation structure is uniform throughout and contains only the grammar name and its description. The same colored highlights represent the common functionalities between the grammar and the example which gets retrieved during RAR.

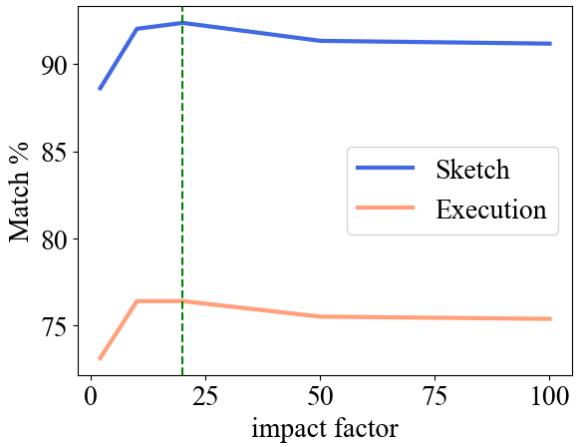


Figure 11: Shows the variation of performance (sketch and execution) as a function of increasing impact factor $\lambda_{E \to \mathcal{D}}$ for *influenced* retriever. We find a rise and a drop beyond the green marker where impact factor is 20. We use this value in our experiments for RAR. This shows that the factor needs to strike a balance between the deviation from retrieved examples and while remaining close to the test query intent.

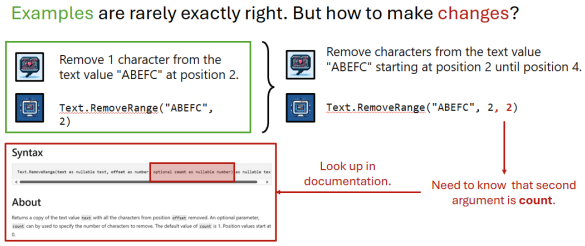


Figure 12: Provides an explanation on how adding grammar to the prompt helps an LLM understand variations to code structure better. A subtle difference in the NL query like "until position 4" does not confuse the LLM to look for a new function or hallucinate something which is incorrect. It is able to understand from the function description that using the third argument value of the same function in the example will generate the correct solution.