On Improving Repository-Level Code QA for Large Language Models

Jan Strich and **Florian Schneider** and **Irina Nikishina** and **Chris Biemann**
Language Technology Group
Universität Hamburg
Germany

Abstract

Large Language Models (LLMs) such as ChatGPT, GitHub Copilot, Llama, or Mistral assist programmers as copilots and knowledge sources to make the coding process faster and more efficient. This paper aims to improve the copilot performance by implementing different self-alignment processes and retrieval-augmented generation (RAG) pipelines, as well as their combination. To test the effectiveness of all approaches, we create a dataset and apply a model-based evaluation, using LLM as a judge. It is designed to check the model's abilities to understand the source code semantics, the dependency between files, and the overall meta-information about the repository. We also compare our approach with other existing solutions, e.g. ChatGPT-3.5, and evaluate on the existing benchmarks. Code and dataset are available online[1].

1 Introduction

Coding assistants (Zhu et al., 2024; Nam et al., 2024; Luo et al., 2024), are invaluable to any programming team for developing software applications, games, or machine learning models involves writing code using programming languages. Commercial AI-assisted programming Chatbot like GitHub Copilot[2], Codeium[3] or Starcoder (Li et al., 2023) help to understand the code better, to generate some code, and to fix errors faster.

However, it is important to note that coding assistants may generate incorrect information, also known as "hallucinations", when requests go beyond the model training data or require additional knowledge (Nguyen and Nadi, 2022). Another drawback of such assistants is the data protection problem: users need to be extremely careful while sharing private code and data with commercial coding assistants. Sensitive or proprietary code could be exposed to unintended parties. This could potentially lead to data breaches and intellectual property concerns (Niu et al., 2023). Moreover, most coding assistants are of general use and cannot be applied to solve context-specific issues or answer natural questions based on repository-level semantics.

To mitigate these limitations, we develop two methods to improve the LLMs response quality on repository-level programming in a more specific, cost-effective and privacy-focused manner. One promising solution is Retrieval-Augmented Generation (RAG) (Lewis et al., 2020), incorporating the repository-level data into the generative process, to deliver accurate and relevant responses. The second approach is inspired by Zheng et al. (2024) and aims to increase the performance of the models by fine-tuning them with synthetic self-generated data using the self-alignment procedure. Finally, we combine a RAG pipeline with a fine-tuned model trained on a self-augmented dataset, which can be considered as both cost-effective and privacy-friendly approach that improves the performance of coding assistants on a specific repository.

When working on the repository-level programming tasks, selecting the appropriate source is also crucial, as it should represent common repository structures and be big enough to generate training data. Therefore, we consider the Python Spyder IDE repository[4] at version 5.5 due to its abundance of short functions and extensive documentation.

We use the open-source model Mistral 7B (Jiang et al., 2023) as a base and fine-tuned model, connected to RAG pipelines. Mistral 7B is a pretrained LLM that outperforms Llama 2 7B, 13B (Touvron et al., 2023) and CodeLlama 7B (Touvron et al., 2023) on most benchmarks.

Regarding the evaluation techniques, we apply the LLM-as-a-judge (Zheng et al., 2023a; Peng et al., 2023; Bubeck et al., 2023; Wang et al., 2023;

[1] https://github.com/pesc101/ma_llm.git
[2] https://github.com/features/copilot/
[3] https://www.codium.ai
[4] https://github.com/spyder-ide/spyder/tree/master

Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 4: Student Research Workshop), pages 303–338
August 11-16, 2024

Fu et al., 2023; Mao et al., 2023) method that leverages a superior model to judge other models responses. We utilize it to test whether adding information through fine-tuning or RAG pipeline improves the response quality. SpyderCodeQA, our new evaluation dataset, is used as the test data for as LLM-as-a-judge evaluation. Additionally, we apply the HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021) benchmarks to measure the catastrophic forgetting of code generation abilities after fine-tuning.

The contributions of the paper are as follows:

- We introduce a new benchmark for the repository-level programming called **SpyderCodeQA**, which includes 325 question-and-answer pairs (Q&A pairs) from three question categories: semantics understanding, dependency understanding, and knowledge of repository meta-information.
- We compare three different methods for repository-level programming: LLM fine-tuning with self-augmented data (self-alignment), Retrieval Augmented Generation, and their combination.
- We perform an ablation study regarding the training dataset size and the type of the judging model and perform a preliminary quantitative analysis of the results.

2 Related Work

This section provides an overview of the existing studies related to the paper: repository-level programming, code-based Question Answering, and LLM evaluation.

2.1 Repository-level Programming

Recent studies have explored the application of instruction fine-tuning with PEFT techniques for coding tasks. Wang et al. (2023) demonstrated the effectiveness of PEFT for coding tasks on various models, highlighting the effectiveness of QLoRA for fine-tuning. In a related study, Yuan et al. (2023) investigated the performance of instruction fine-tuned models on a range of coding tasks.

Researchers have also explored generating prompts for few-shot learning using RAG pipelines (Nashid et al., 2023) as well as the combination of fine-tuning and RAG pipelines using several open-source models to inject additional information (Ovadia et al., 2023).

2.2 Code-based Question Answering

Code-based question answering is a subfield of question answering that focuses on responding to code-related queries. Unlike generative approaches, retrieval-based code Q&A aims to find the most relevant code snippets from a large code corpus to satisfy user requests. To evaluate the performance of the models, Husain et al. (2019) introduced CodeSearchNet, a collection of datasets and benchmarks created by mining large-scale comment-code pairs from public GitHub repositories. Liu and Wan (2021) presented CodeQA, a free-form code question-answering dataset to assess the code comprehension capabilities of language models. CoSQA (Huang et al., 2021) mines real-world user queries from Bing search logs that were labeled if the provided answer is the solution to the question.

Although these Q&A datasets are useful for measuring the interaction of models and humans, they are unsuitable for repository-level programming tasks: CodeSearchNet and CodeQA have direct question-answer interaction. While CoSQA (Huang et al., 2021) consists of real human queries, they are only related to general coding tasks and have no label for a repository, which makes it difficult to use the Q&A pairs as training data to measure the performance of a specific repository.

2.3 Evaluation of LLMs

Evaluating the capabilities of LLMs has been challenging due to their vast and diverse abilities and the lack of standardized benchmarks to measure human preferences in this rapidly evolving field.

LLM-as-a-Judge LLM-as-a-judge is an evaluation method for LLMs in which a superior model is used to judge the results of other models. Zheng et al. (2023a) proposed three variations of Model-based-evaluation referred to as LLM-as-a-judge. The first, pairwise comparison (Peng et al., 2023; Bubeck et al., 2023), involves directly assessing two answers to determine superiority or a tie. The second, single answer grading, assigns a score directly to a response (Wang et al., 2023; Mao et al., 2023). The third, reference-guided grading, incorporates a reference solution, beneficial for math problems (Bubeck et al., 2023).

3 Dataset Construction

In order to measure the performance the performance of the models on repository-level programming, we create a new evaluation dataset named

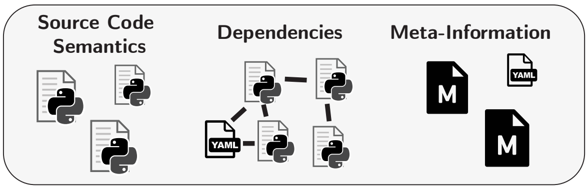


Figure 1: Overview of the three dimensions of the evaluation dataset. The dimensions include source code semantics, dependencies, and meta-information Q&A. These dimensions are designed to provide comprehensive information about the source code files, their relationships with modules and libraries, and general information about the repository.

SpyderCodeQA comprising of 325 samples established on the Spyder IDE[5]. It is based on three dimensions: *source code semantics comprehension* (Subsection 3.1), *dependency comprehension* (Subsection 3.2), and *meta-information comprehension* (Subsection 3.3). Figure 1 presents an overview of the three dimensions of the evaluation dataset. The first one aims at understanding the containing text and code elements about the repository source code and being able to answer semantic questions about it. The second dimension evaluates the ability to understand the relationships between files within the repository and between files and imported libraries. The third dimension assesses the ability to understand general information about the repository using README files (build commands, requirements or legal information of the repository, unrelated to the source code).

The following subsections provide an overview of the creation process for each dimension in detail. Typical samples for each dimension are shown in Appendix A in Figure 8.

3.1 Source Code Semantic Comprehension

For creating the source code semantics comprehension dimension, ten experts computer science are asked to manually create the Q&A pairs using the Spyder IDE repository source code. For this purpose, we develop a custom web application using Python Django[6] to write question pairs given the code snippet (see Appendix A for more details).

The first goal is to create Q&A pairs for one of the 5673 snippets (2000 characters max) from the 2083 Python files randomly selected from the open-source Python repository Spyder IDE. We demonstrate those code snippets in the web application and ask the experts to create a question and the answer. Meta-information such as the module name, file name, and the start and end line of the code snippet is also given. The example of the interface is shown in Appendix A in Figure 9a.

The second task is to rate the created Q&A pairs from other participants to ensure the quality of the pairs on a 1-10 scale and optionally leave comments. The instructions for the rating task and the process for rating the Q&A pairs are shown in Appendix A. In the interface, the text areas are replaced with two rating forms (Figure 9b).

The last step of the dataset collection is the quality control of the collected Q&A pairs. In total, 189 questions were created and rated by the experts. The pairs scored with less than 3 points are automatically removed from the dataset. Pairs with a rating between 3 and 5 are manually curated. As a result, the final size encompasses 140 Q&A pairs.

3.2 Dependencies Comprehension

Q&A pairs for dependencies comprehension aim at measuring the ability to understand the dependencies between code files. Therefore, we present the AST algorithm (Appendix A) to identify dependencies between files, modules, and libraries.

It recognizes four types of imports: complete library imports, imports from libraries, complete file imports, and imports from files. We also identify the type of the imported artifacts (class, function, or assignment): whether is it a library-based or a file-based import. The algorithm also provides information on each Python file in the repository (file name, import category, and artifact name).

The raw dependencies are further processed with the OpenAI API using the "gpt-3.5-turbo-1106"[7] model (temperature is set to 1.5, the maximum token limit is 256, and the top p-value is 1, the frequency and presence penalties are set to 0). In Appendix D, Figure 12 presents the full system prompt for generating the Q&A pairs along with the example to improve generation abilities.

As a result, 1319 Q&A pairs were generated using the OpenAI API from 686 unique file names. To ensure the quality of the dataset, a final set of 135 Q&A pairs was randomly chosen and manually verified for correctness by an expert annotator. This was done by cross-checking the repository's source

[5] `https://github.com/spyder-ide/spyder/tree/0f8398a9a27d401b9984f6e049ef1199656900f1`

[6] `https://www.djangoproject.com`

[7] `https://platform.openai.com/docs/models/gpt-3-5-turbo`

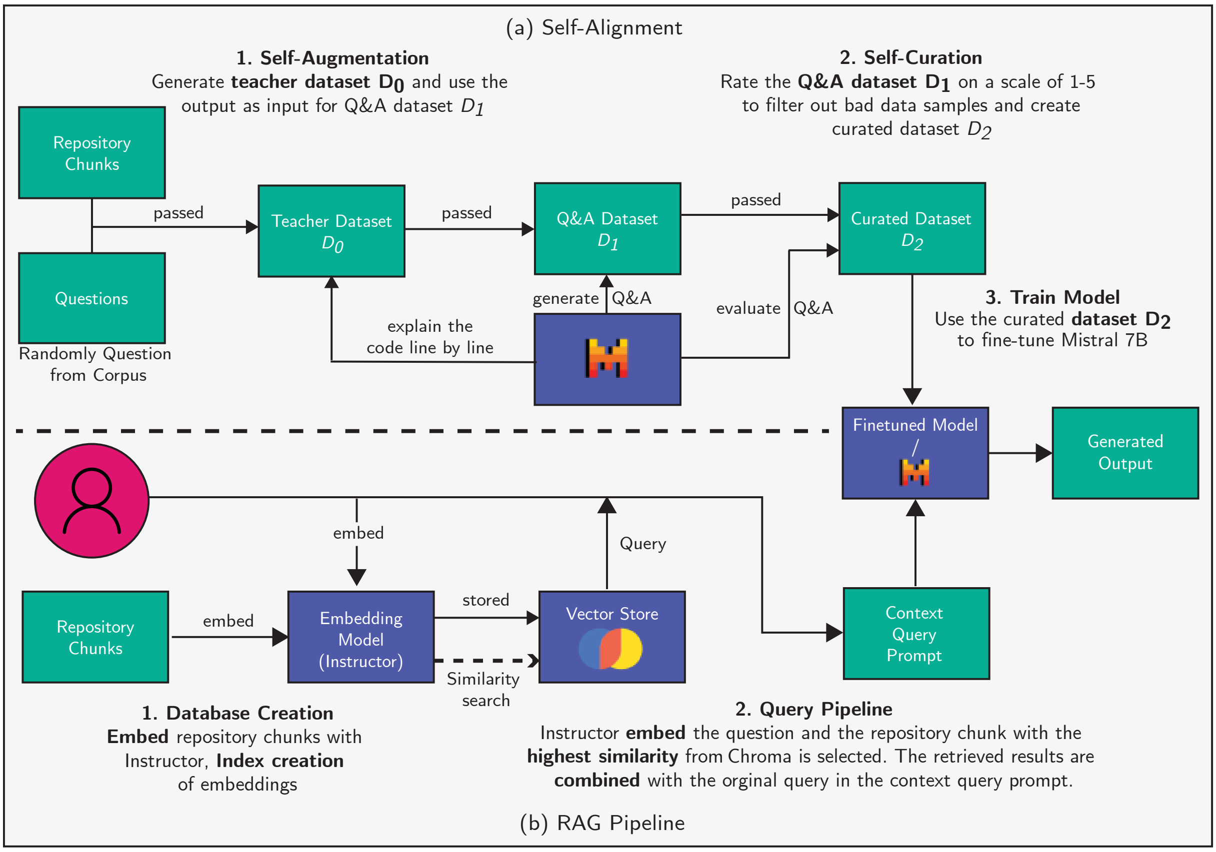


Figure 2: (a) Self-alignment pipeline: self-augmentation (repository chunks and randomly selected questions from the question corpus are combined in the system prompt. Mistral 7B generates the teacher data D_0 to generate the Q&A dataset D_1), self-curation (Q&A pairs are curated with the base model on a scale of 1-5 and filtered to the final curated dataset D_2), fine-tuning (D_2 is used to fine-tune Mistral 7B). (b) the RAG pipeline: database creation (source code files are embedded using Instructor, Chroma indices the embeddings), context retrieval (queries are transformed into embeddings following the dotted line, then the n-chunks are retrieved), generation (chunks combined as context and query are passed to the generator to produce the answer).

code to ensure that the questions and answers were both correct and made sense. The random selection process was implemented to minimize the amount of manual effort required for verification.

3.3 Meta-Information Comprehension

To understand the model ability to understand general information about a repository, such as its purpose, features, documentation, license, and contribution opportunities, we create Q&A pairs for the meta-information dimension. We first extract all files with the suffixes *.md*, *.txt*, and *.yml*, resulting in 29 files that included meta-information. We focus on the information about the repository installation, the available and supported versions of the packages, and the rules for contributing. We ask our expert annotator to create triplets containing questions, answers, and meta information (file name and the module) resulting in 50 questions.

4 Methodology

This section describes the methods we implement in the paper. First, we describe the data preprocessing step (Subsection 4.1), which is common for all approaches. Then we explain the self-alignment approach in Subsection 4.2 and our implementation of RAG in Subsection 4.3. Subsection 4.4 explains how both approaches can be combined.

4.1 Data Preprocessing

To fit the desired structure for fine-tuning models using self-alignment or creating a vector database for RAG, a pre-processing pipeline is created.

First, we fetch the Spyder repository and load each file type using individual loader classes. With a chunk size of 1500 characters and an overlap of 200, the file was divided into chunks of a maximum of 1500 characters, each overlapping by 200 characters. From the code chunks, all available metadata is extracted: file name, module, flag whether the chunk contains a class or function, start and end line numbers, and all file imports. In the final step, the extracted metadata are added to the chunks and saved as *.jsonl* file and uploaded into Huggingface[8].

4.2 Fine-tuning with Self-Alignment

This subsection overviews the fine-tuning process with self-alignment mainly inspired by Zheng et al. (2024). It comprises of the following steps: data generation (self-augmentation), data curation (Self-Curation). Afterwards, we perform the model fine-tuning on the generated dataset.

Self-Augmentation First, we provide the repository code chunks as input into the base model (Mistral 7B) to generate the dataset D_0 that explains each line of code in the chunk and add one randomly selected question from a predefined question corpus (See Appendix B). Then, we generate the Q&A pairs (D_1) from this source code explanations D_0. We instruct the module to include file and module names to ensure the model always knows the file the question aims for. The prompt also specifies that code should be added to the answer. Both system prompts for generating D_0 and D_1 are shown in Appendix D in Figures 12 and 13.

In addition to the code chunk with explanations from D_0, we also provide an example question selected from a question corpus inspired by Liu and Wan (2021). We manually limit possible question examples to be used, as the question should belong to one of three dimension types: source code semantics, dependencies and meta-information, likewise the dimension in the manually created dataset in Section 3. The list of selected questions can be found in Figure 10.

It is important to note that the pipeline to generate Q&A examples can be executed multiple times in a row, resulting in datasets that differ from each other. We execute the self-augmentation step twice for 7943 chunks to create two datasets $D0$, resulting in 15,886 data samples that are further processed to the curation step of the Q&A dataset.

Self-Curation To generate high-quality training data, we curate the data samples to collect the final dataset denoted in Figure 2 (a) as D_2. We ask the base model (Mistral 7B) to evaluate the Q&A pairs on a scale from 1 to 5. The system prompt is displayed in Fig. 15. The model evaluates whether the response is a good example of how an AI Assistant should respond to user instructions. A score of 1 indicates that the answer is incomplete, not precisely what the user asked for, or off-topic. A score of 5 represents a clear and well-structured answer from an AI assistant that thoroughly answers the user's question. All examples with a score lower than 4 are removed from the dataset. As a result, our training dataset comprises 14,434 Q&A pairs.

Fine-Tuning The base model (Mistral 7B) is trained for 5 epochs using supervised fine-tuning (SFT) (Ouyang et al., 2022), 4-bit Quantization Low-Rank Adapters (QLoRA) (Dettmers et al., 2023) on the generated Self-Aligned dataset and Flash Attention 2 (Dao, 2023). After the training, the LoRA layers were merged into the base model Mistral 7B to reduce the response time when using the model for inference. The training details can be found in Appendix C.

4.3 RAG Approach

The implemented RAG pipeline is illustrated in Figure 2 (b). We use the preprocessed chunks to generate 768-dimensional vector representations of chunks using the Instructor embedding model (Su et al., 2023). This pre-trained model with 110 million parameters generates embeddings that can be used for retrieval, classification, or semantic search tasks. The data is stored in the in-memory version of Chroma[9], an optimized database for storing vector representations. The database is initialized by assigning an ID to each chunk and indexing the metadata. This ensures a quick response time and enables data retrieval based on metadata queries. For the retrieval step, we also use the Instructor model to transform the query into a standardized 768-dimensional vector. During the generation step, we use the system prompt displayed in Figure 16 as input to the LLM (the base Mistral 7B model) to generate the answer. It utilizes the question and the retrieved code chunks as input and generates the answer to the question as output. Thus, our RAG approach aligns with the concept of "inference" (Huang and Huang, 2024).

It is also important to note, that we apply both Instructor and Mistral 7B models without additional fine-tuning.

[8] https://github.com/pesc101/ma_llm/blob/main/README.md

[9] https://www.trychroma.com

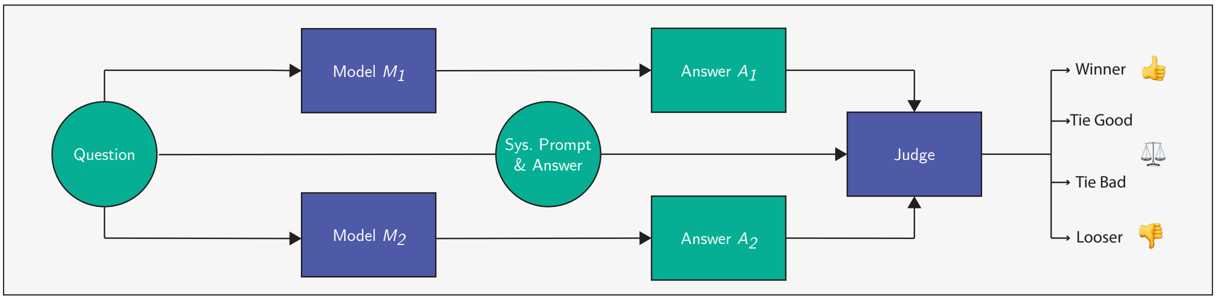


Figure 3: LLM-as-a-judge pairwise evaluation (Zheng et al., 2023a). The LLMs M_1 and M_2 are tested against each on SpyderCodeQA. The judge (GPT-3.5) receives the system prompt with the original question, the correct answer, both answers A_1 and A_2, and the instruction to judge both answers and determine the outcome.

4.4 Combined Approach

As the combined approach, we replace the base Mistral 7B model with the fine-tuned model from the self-alignment step in Subsection 4.2. We expect the fine-tuned model might produce better results when enhanced with the correct chunks from the RAG pipeline. Additionally, retrieved chunks should also prevent the LLM from hallucinating.

5 Evaluation

This section describes two evaluation strategies applied in the paper: using LLMs (primarily GPT 3.5/4) as judges (Zheng et al., 2023a) and standard benchmark evaluation using metrics. LLM-as-a-judge methods are preferred over BLEU (Papineni et al., 2002) and ROUGE (Lin, 2004), as they can only evaluate the semantic similarity between human and model-generated responses, which might not be related to the correctness of the responses.

5.1 LLM-as-a-judge

The performance of models on the Q&A evaluation dataset created in Section 3 is evaluated pairwise using strong LLMs (primarily GPT 3.5/4) as judges (Zheng et al., 2023a) (using the same hyperparameters for the judge model as in generation: temperature of 0.7, top-P of 0.9, and max token of 2500). We test the base model against its modified version (finetuned Self-alignment, RAG, or the two methods combined).

Figure 3 shows the model-based pairwise comparison pipeline. For each Q&A pair in the evaluation dataset, the two models M_1 and M_2 answer the question of the Q&A pair. Then the LLM (GPT-3.5) model is instructed in the system prompt to act as a judge to evaluate the quality of responses A_1 and A_2. The prompt template is shown in Figure 18. It consists of a question ("User Question") and the generated answers ("Model Solution"). To ensure clarity, each piece of information is enclosed with an identifier in square brackets, indicating the type of information. The evaluation could also result in "No value" when the judge does not return the output in the correct format.

We utilize the Average Win Rate (AWR) metric for evaluation. AWR is the proportion of Q&A pairs the judge has decided that one model is better than the other or it is not a tie. The average is calculated over k runs executed with the same parameters to take into account possible deviations.

5.2 Existing Benchmarks

In addition to evaluating whether a coding assistant has become better at answering questions about a repository, we also test whether the code generation abilities have changed after fine-tuning. Therefore, two benchmarks are used to evaluate the "catastrophic forgetting": HumanEval introduced by OpenAI (Chen et al., 2021) and Mostly Basic Programming Problems (MBPP) (Austin et al., 2021). Both benchmarks use the $pass@k$ unbiased estimator which is computed as follows (n is the total number of samples, c is the number of correct samples and $\mathbb{E}$ is the expected value):

$$\text{pass}\,@k := \underset{\text{Problems}}{\mathbb{E}} \left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}} \right] \tag{1}$$

6 Results and Analysis

In this section, we present the results using LLM-as-a-judge and the existing benchmarks (Subsections 6.1 and 6.2). In Subsection 6.4, we discuss the additional experiments with the training size and applying GPT-4 as the judging model. The

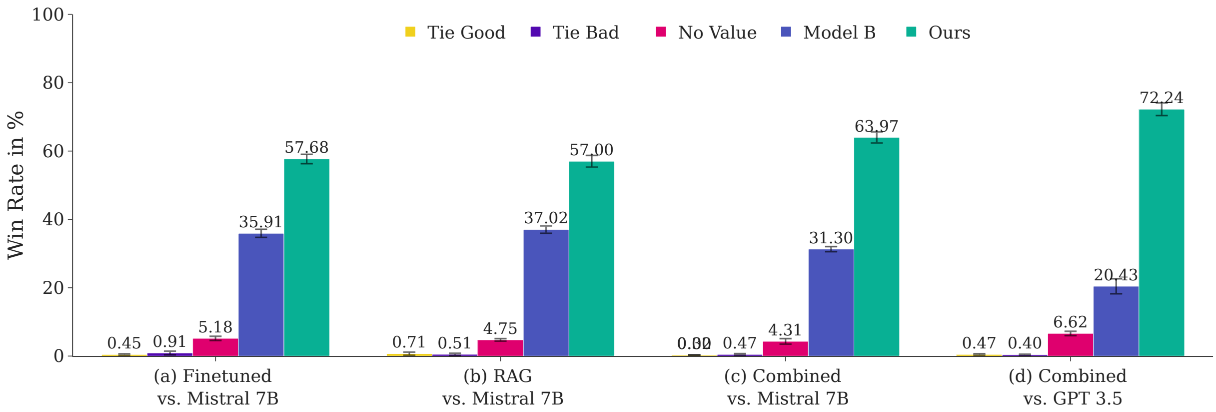


Figure 4: Average win rate for each experiment using LLM-as-a-judge evaluation on the SpyderCodeQA. All experiments were executed with $k = 3$ runs. The error bars indicate the standard deviation. **(a)**: compares the fine-tuned Mistral 7B vs. Mistral 7B. **(b)**: compares Mistral 7B with a RAG pipeline vs. Mistral 7B. **(c)**: compares fine-tuned Mistral 7B with a RAG pipeline vs. Mistral 7B. **(d)**: compares fine-tuned Mistral 7B vs. GPT-3.5 Turbo.

qualitative analysis of the results can be seen in Subsection 6.3 and in more detail in Appendix F.

6.1 LLM-as-a-Judge on SpyderCodeQA

The average win rate results for $k = 3$ runs are shown in Figure 4 for all approaches. We describe them separately in the following paragraphs.

Fine-tuning with Self-Alignment The results in Figure 4 (a) suggest that in approx. 57% of the Q&A pairs, the answer of the fine-tuned model is preferred, while in approximately 36% of the pairs, the answer of the base model is preferred. The LLM-as-a-judge evaluation method consists of $k = 3$ runs, where in each run the order of the answers given to the judge is randomized to reduce position bias. The error bars indicate the standard deviation of the runs. The low variance for each output indicates that LLM-as-a-judge is consistent over several evaluation runs.

Additionally, the fine-tuned model performs best on the human-labeled dimension code semantics. With 62%, it won almost two-thirds of the Q&A pairs. For the dependency dimension, the fine-tuned model is also better than the base model but has only a 54% win rate. The model performed the worst in the meta-information dimension, indicating that the fine-tuning process reduced its performance in this dimension.

RAG Approach In Figure 4 (b), we can see that for 57% of the Q&A pairs, the judge prefers Mistral 7B with the RAG pipeline, which aligns with the previous approach. Also, the win rate for the base model and the percentage of Q&A pairs that aren't correctly judged is similar to the Self-alignment pipeline and are close to 37% and 5% respectively.

The results of the different dataset dimensions differ from those of the Self-alignment pipeline. Although both approaches perform the same with a 1% difference in the code semantics dimension, there is a difference of 2 standard deviations in the results for the dependencies. The meta-information dimension shows the biggest difference, with the base model using the RAG pipeline outperforming the base model. This suggests that the RAG pipeline supports the model in answering questions related to the meta-information but is less useful for answering questions regarding dependencies.

Combined Approach The results for the comparison with the combined approach are shown in Figure 4 (c). The average win rate is approximately 64%, which is higher than that of the two pipelines, respectively. This suggests that there is a positive interaction effect between them. When examining each dimension separately, the best results are achieved for the code semantics dimension. With an average of 70% win rate, the model is in 7 out of 10 questions better than the base model. That indicates that this combination is a further improvement regarding code semantic questions. The results for the dependencies dimension demonstrate an average win rate of 61% and also indicate the efficiency of the interaction of both pipelines. For the meta-information dimension, the model shows a 51% average win rate, which means no improvement over the base model.

GPT-3.5 Turbo In the last experiment, we compare our best-performing model with the `gpt-3.5-turbo-1106` as a code assistant instead of the base model. We acknowledge that the `gpt-3.5-turbo-1106` approach does not get code snippets as input, however, our main idea was to check whether the fine-tuned model indeed learns the context from the given repository. Otherwise, the results of the Self-alignment fine-tuned model and GPT-3.5 would be comparable. It is worth noting that GPT-3.5 was utilized as the judge as well; therefore, it rates its responses in this experiment.

The results are presented in Figure 4 (d). The combination of the fine-tuned model with an RAG pipeline outperforms GPT-3.5, with an average win rate of 72%. Only 20% of the Q&A pairs were won by GPT-3.5. However, it is worth noting that the rate of not finding a rating by the judge is slightly higher than with Mistral 7B.

The code semantics and dependencies results are even better at the dimensions, with 78.3% and 74.07%, respectively. That indicates that the fine-tuned model with the RAG pipeline is a better coding assistant on repository level than GPT-3.5.

6.2 Benchmark Results

Figure 5 presents the percentage of solved tasks by the base model Mistral 7B and the fine-tuned model with Self-Aligned data on the HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021) benchmarks. For each benchmark, the pass@1 and pass@10 are calculated. However, the results for both benchmarks are not very promising. The base model outperforms the fine-tuned model on HumanEval on pass@1 with 6.8% and on pass@10 with 8%. Similar results were found on the MBPP benchmark with a difference of 11.5% on pass@1 and 12.6% on pass@10. This decrease in scores indicates that the general coding ability of the fine-tuned model has been reduced. The possible reason for the poorer performance could be the modified prompt template, as the model is fine-tuned for answering Q&A pairs and not for pure coding tasks.

6.3 Results by Question Type

We also take a closer look at the concrete examples and provide more qualitative insights about how the RAG pipeline affects the output of the LLM model and improves performance. The examples are shown in Appendix F. Each example consists of the original question and answer, the answer of the two models, and the judgment at the end.

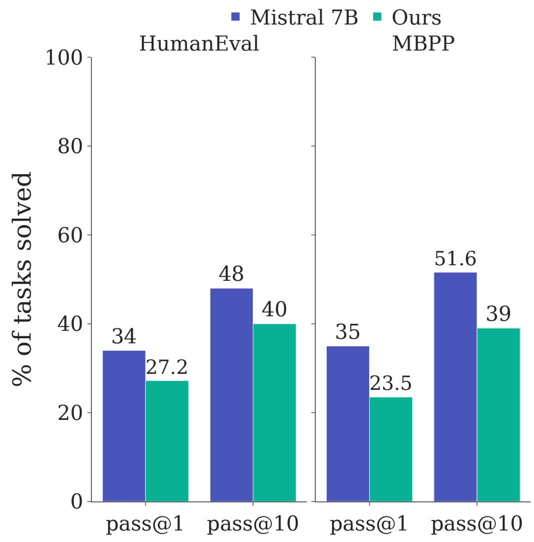


Figure 5: % of tasks solved for HumanEval (Chen et al., 2021) & MBPP (Austin et al., 2021) for the base model Mistral 7B and the fine-tuned model.

Regarding the *Source code semantics comprehension*, we can see from Figures 21-23 that each approach demonstrates its benefits when combined. The fine-tuned model answer is nicely formatted, and the RAG pipeline answer is contextually correct. The combination of both approaches fulfilled both requirements, providing a well-formatted answer with a good explanation of the class and the correct code snippet. For the *Dependencies* types of question in Figures 24-26, we can see that the base and the fine-tuned models without RAG cannot provide information about imports used, therefore, they might not be able to perform well for these tasks. *Meta-information* types of questions show a similar trend in Figures 28 and 29 where approaches using RAG in the pipeline demonstrate a more accurate response.

Quantitative results in Tables 1-3 (Appendix E) demonstrate quite an opposite tendency: for the *Dependencies Meta-information* types of questions GPT-3.5/4 prefer the pipelines with RAG in fewer cases than the RAG and Combined approaches. *Code Semantics* questions are better solved when provided the context from RAG and the Combined approach. Nevertheless, all developed pipelines outperform the base model for all three dimensions.

6.4 Ablation Study

This section presents supplementary experiments that provide a deeper insight into the number of dataset samples and the choice of the judge model.

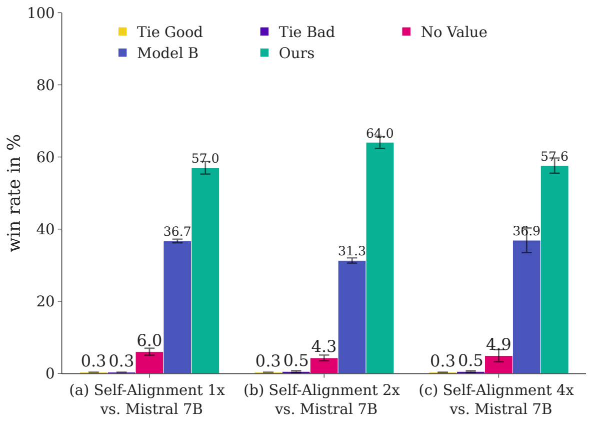


Figure 6: Average Win Rate ($k = 3$) in % for each experiment respectively on the SpyderCodeQA. **(a)**: fine-tuned model **once** vs. Mistral 7B. **(b)**: fine-tuned model trained **twice** vs. Mistral 7B. **(c)**: fine-tuned model trained **quadruple** vs. Mistral 7B.

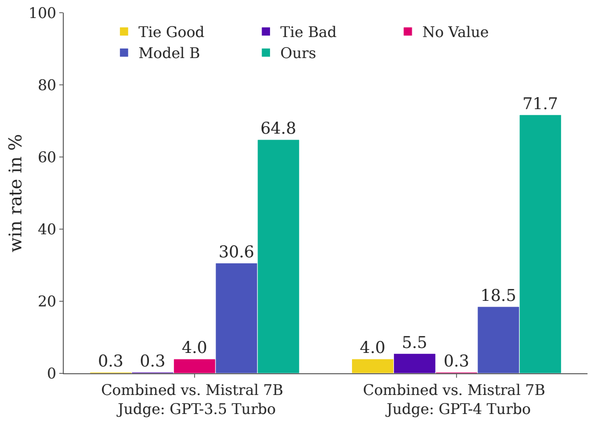


Figure 7: Win Rate in % for each experiment respectively on the SpyderCodeQA. **Left:** Fine-tuned model with RAG pipeline vs. Mistral 7B judged by GPT-3.5 Turbo. **Right:** Fine-tuned model with RAG pipeline vs. Mistral 7B judged by GPT-4 Turbo.

Training Dataset Size To create different sizes of the training dataset, the self-augmentation was executed once (a), twice (b), and quadruple (c). The related loss curves and learning rates are shown in Appendix C. From the results in Figure 6, we can see that in all three experiments, each fine-tuned model learned about the repository, as reflected in the higher average win rates compared to the base model. However, the best-performing model was achieved using the self-alignment pipeline twice to create the training dataset. The Average Win Rate is considerably higher than the models trained with one or quadruple datasets, with an improvement of approximately three standard deviations.

We assume that the reason for the optimal number (2) for the self-alignment step might be explained by the number of unique Q&A pairs. The quadruple design adds only a few new pairs while having many duplicates, which may cause the model to overfit.

Judgement with GPT-4 Turbo The results of comparing the GPT-3.5 and (more expensive) GPT-4 models as judges are presented in Figure 7. The corresponding results for each dimension can be found in Appendix E in Table 3. Both judges rate the quality of the response of the fine-tuned model with the RAG pipeline higher. However, GPT-4 prefers more the fine-tuned model and chooses a tie in almost 10% as judgment, which is more often than GPT-3.5. Furthermore, only 0.3% of the answers belong to the "No value" type, indicating that GPT-4 can judge the performance of models more consistently and accurately.

7 Conclusion

In this paper we introduce a new manually created dataset — **SpyderCodeQA** – which includes 325 question-and-answer pairs (Q&A pairs) from the Spyder IDE repository. We split it into three question dimensions: semantics understanding, dependency understanding, and knowledge of repository meta-information. We also present a series of experiments using Self-alignment, RAG, and their combination to evaluate LLMs' performance on repository-level code Q&A using the generated dataset. We show that the quality of the system can be significantly improved when applying both approaches together: the LLM-as-a-judge win rate is approximately 64%, which is 7% higher than both approaches separately. Regarding the models' performance on different dataset dimensions, we can see that they perform exceptionally well for code semantics, which is the human-labeled dimension.

In future work, we plan to improve the Self-alignment pipeline to create a more diverse dataset that includes Q&A pairs mainly focused on code generation to prevent the "catastrophic forgetting" of the model. Another possible direction is to perform the human evaluation to better align the model with user needs. It would provide additional insights since humans are the target audience for Q&A on repository-level programming, and they often have more knowledge about the repository, allowing them to better judge the model's responses.

Limitations

This section outlines the limitations regarding the approaches and the created dataset.

Small dataset size Other datasets in this research area include CS1QA (Lee et al., 2022), a dataset for code-based question-answering in the programming education domain or CodeQA (Liu and Wan, 2021) for the code comprehension task have much bigger samples than the dataset that is introduced in this thesis. CS1QA with over 9k pairs and CodeQA with approx. 200k are much bigger. While both datasets aim for slightly different goals, it is important to mention that the generalizability and value of the evaluation results should be treated with caution.

Different Knowledge Level of Creators For the source code semantic dimension, the Q&A pairs were created by humans. While the number of participants was with ten people quite low, also the knowledge level of the participants about the Python programming language and the working experience was high. That could lead to a bias in the difficulty of the questions asked. Assuming you want to test whether a model can answer simple questions for beginner programmers, the questions from the semantic dimension may not necessarily be helpful and accurate.

Unknown repository The individuals who took part in the study considered themselves experts in Python, however, none of them had previously contributed to the Spyder IDE repository. Essentially, this means that none of the participants were experts in this specific code base. Although this may not pose as a disadvantage, it does suggest that the questions and answers provided may not be as in-depth as those provided by a Spyder IDE contributor.

Low heterogeneity of the Q&A pairs in dependency dimension The Q&A pairs in the source code semantic dimension have a great variety, but the ones generated automatically in the dependency dimension are often very similar. The reason behind this is to assess the model's ability to answer these questions accurately. However, a wider range of questions would be preferable to test the model's performance as a coding assistant. Therefore, a further improvement of the dataset would be adjusting the model's system prompt that generates the Q&A pairs or developing a new way to measure the dependencies of the different repository components.

Only 1-hop Dependencies The relationship between the two source code files is adequately described using the dependencies dimension. However, the dataset dimension lacks a mapping that goes beyond the linking of two files. Therefore, it would be beneficial to devise a way to create 2-hop or even n-hop structures that the models can comprehend.

Meta-Information dimension is self-generated The quality of the source code semantic dimension dataset was ensured through a rating process conducted by participants. The dependency pairs were also manually verified to be correct. However, the meta-information dimension lacks quality testing. The Q&A pairs were created exclusively by the author of the thesis, which could introduce bias in the formulation of the questions and answers and the selection of information to create the pairs. This dimension may not be as objective as others, as different people may create completely different pairs.

Self-generated training dataset The Q&A pairs generated by the self-alignment process may not be semantically and syntactically correct. Although the model has been trained to match questions with the corresponding answers, it is not guaranteed that the generated code is functionally correctly reproduced and that the generated question is similar to a user request. The model itself curates the Q&A pairs, but the curation can only verify if the question matches the answer and seems to be correct. Therefore, the curation/verification process could be further improved in this pipeline step.

Data is limited to one Python repository The evaluation is limited to one Python repository that has its unique structure. This is important to consider as the model may behave differently when applied to other repositories, which could result in biased results. In addition, the evaluation results only cover a limited set of questions that could arise concerning repositories. Given the vast range of programming languages, frameworks, and projects, these results may not be applicable in all scenarios.

Choice of Model & Embeddings There exist dozens of large pre-trained generative models and embeddings that could be applied to the task. However, we report the results for the Self-Alignment technique only with Mistral-7B and the basic RAG approach with the Instructor embeddings. An alternative base LLM or embeddings could further improve the results.

Our goal was to compare RAG-based and fine-tuning approaches on the repository-level Question Answering task and not to make an exhaustive search of all models, embeddings, and pipelines. We leave these experiments as future work.

Using LLM-as-a-judge instead of human evaluation Regarding evaluating the model's performance, the LLM-as-a-judge approach also has its limitations. Despite the elimination of the position bias and the attempts to use GPT-4 as a judge, the evaluation is not flawless. The superior model judges the answers, but sometimes, the criteria are chosen by the model itself and do not match those of humans. Also, the correctness of the produced code is often not sufficiently verifiable for the model, as it does not have access to the necessary source code.

Chunking of the code file Despite its advantages, the RAG pipeline has some limitations that must be considered. One major limitation is that the context provided to the LLM is always just a portion of the file, which means that knowledge about multiple files is not processed, and the connection between the files and the code cannot be considered. To address this, the context would need to be pre-processed better. One possible solution is to have a hierarchical structure that provides context at different levels and contains summarized knowledge. For example, a description of what a module is responsible for or how the general structure of the module could be added to each chunk of each file in the module. That additional information should further help the model gain a deeper understanding of the repository.

Number of chunks retrieved For all experiments, the number of chunks was set to $N = 1$, but it could also be interesting to test whether the number of chunks could further improve the model's performance. Also, the size of the chunk and the overlapping characters are possible variables for optimizing the results.

Catastrophic Forgetting As mentioned in Sec. 5, the model's performance decreased on both MBPP (Austin et al., 2021) and HumanEval (Chen et al., 2021) benchmarks following the fine-tuning process. This shows that fine-tuning can change models' abilities to perform certain tasks. Therefore, the conducted experiments do not clarify how the model enhances its capacity to handle the context, particularly source code, and grasp it deeper after the fine-tuning process.

References

Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. Program Synthesis with Large Language Models. *CoRR*, abs/2108.07732.

Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott M. Lundberg, Harsha Nori, Hamid Palangi, Marco Túlio Ribeiro, and Yi Zhang. 2023. Sparks of Artificial General Intelligence: Early experiments with GPT-4. *CoRR*, abs/2303.12712.

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

Tri Dao. 2023. FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning. *CoRR*, abs/2307.08691.

Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. 2023. QLoRA: Efficient Finetuning of Quantized LLMs. *CoRR*, abs/2305.14314.

Jinlan Fu, See-Kiong Ng, Zhengbao Jiang, and Pengfei Liu. 2023. GPTScore: Evaluate as you desire. *CoRR*, abs/2302.04166.

Junjie Huang, Duyu Tang, Linjun Shou, Ming Gong, Ke Xu, Daxin Jiang, Ming Zhou, and Nan Duan. 2021. CoSQA: 20, 000+ Web Queries for Code Search and Question Answering. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International*

Joint Conference on Natural Language Processing, pages 5690–5700, ACL/IJCNLP 2021, (Volume 1: Long Papers), Virtual Event, August 1-6, 2021.

Yizheng Huang and Jimmy Huang. 2024. A survey on retrieval-augmented text generation for large language models. *CoRR*, abs/2404.10981.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. CodeSearchNet Challenge: Evaluating the State of Semantic Code Search. *CoRR*, abs/1909.09436.

Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de Las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. 2023. Mistral 7b. *CoRR*, abs/2310.06825.

Changyoon Lee, Yeon Seonwoo, and Alice Oh. 2022. CS1QA: A Dataset for Assisting Code-based Question Answering in an Introductory Programming Course. *CoRR*, abs/2210.14494.

Patrick S. H. Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. *CoRR*, abs/2005.11401.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy V, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Moustafa-Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2023. StarCoder: may the source be with you! *CoRR*, abs/2305.06161.

Chin-Yew Lin. 2004. Looking for a Few Good Metrics: Automatic Summarization Evaluation - How Many Samples Are Enough? In *Proceedings of the Fourth NTCIR Workshop on Research in Information Access Technologies Information Retrieval, Question Answering and Summarization*, NTCIR-4, National Center of Sciences, Tokyo, Japan, June 2-4, 2004.

Chenxiao Liu and Xiaojun Wan. 2021. CodeQA: A Question Answering Dataset for Source Code Comprehension. In *Findings of the Association for Computational Linguistics: EMNLP*, pages 2618–2632, Virtual Event / Punta Cana, Dominican Republic, 16-20 November, 2021.

Qinyu Luo, Yining Ye, Shihao Liang, Zhong Zhang, Yujia Qin, Yaxi Lu, Yesai Wu, Xin Cong, Yankai Lin, Yingli Zhang, Xiaoyin Che, Zhiyuan Liu, and Maosong Sun. 2024. RepoAgent: An LLM-Powered Open-Source Framework for Repository-level Code Documentation Generation. *CoRR*, abs/2402.16667.

Rui Mao, Guanyi Chen, Xulang Zhang, Frank Guerin, and Erik Cambria. 2023. GPTEval: A Survey on Assessments of ChatGPT and GPT-4. *CoRR*, abs/2308.12488.

Daye Nam, Andrew Macvean, Vincent J. Hellendoorn, Bogdan Vasilescu, and Brad A. Myers. 2024. Using an LLM to help with code understanding. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, pages 97:1–97:13, ICSE 2024, Lisbon, Portugal, April 14-20, 2024.

Noor Nashid, Mifta Sintaha, and Ali Mesbah. 2023. Retrieval-Based Prompt Selection for Code-Related Few-Shot Learning. In *45th IEEE/ACM International Conference on Software Engineering*, pages 2450–2462, ICSE 2023, Melbourne, Australia, May 14-20, 2023.

Nhan Nguyen and Sarah Nadi. 2022. An Empirical Evaluation of GitHub Copilot's Code Suggestions. In *19th IEEE/ACM International Conference on Mining Software Repositories, MSR*, pages 1–5, Pittsburgh, PA, USA, May 23-24, 2022.

Liang Niu, Muhammad Shujaat Mirza, Zayd Maradni, and Christina Pöpper. 2023. CodexLeaks: Privacy Leaks from Code Generation Language Models in GitHub Copilot. In *32nd USENIX Security Symposium*, pages 2133–2150.

Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul F. Christiano, Jan Leike, and Ryan Lowe. 2022. Training language models to follow instructions with human feedback. *CoRR*, abs/2203.02155.

Oded Ovadia, Menachem Brief, Moshik Mishaeli, and Oren Elisha. 2023. Fine-tuning or retrieval? comparing knowledge injection in LLMs. *CoRR*, abs/2312.05934.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. BLEU: a Method for Automatic Evaluation of Machine Translation. In *Proceedings of the 40th Annual Meeting of the Association for*

Computational Linguistics, pages 311–318, July 6-12, 2002, Philadelphia, PA, USA.

Baolin Peng, Chunyuan Li, Pengcheng He, Michel Galley, and Jianfeng Gao. 2023. Instruction tuning with GPT-4. *CoRR*, abs/2304.03277.

Hongjin Su, Weijia Shi, Jungo Kasai, Yizhong Wang, Yushi Hu, Mari Ostendorf, Wen-tau Yih, Noah A. Smith, Luke Zettlemoyer, and Tao Yu. 2023. One Embedder, Any Task: Instruction-Finetuned Text Embeddings. In *Findings of the Association for Computational Linguistics, 2023*, pages 1102–1121, Toronto, Canada, July 9-14, 2023.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton-Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurélien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. *CoRR*, abs/2307.09288.

Jiaan Wang, Yunlong Liang, Fandong Meng, Haoxiang Shi, Zhixu Li, Jinan Xu, Jianfeng Qu, and Jie Zhou. 2023. Is ChatGPT a good NLG evaluator? A preliminary study. *CoRR*, abs/2303.04048.

Zhiqiang Yuan, Junwei Liu, Qiancheng Zi, Mingwei Liu, Xin Peng, and Yiling Lou. 2023. Evaluating Instruction-Tuned Large Language Models on Code Comprehension and Generation. *CoRR*, abs/2308.01240.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. 2023a. Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems*, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023.

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Lei Shen, Zihan Wang, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. 2023b. Codegeex: A Pre-Trained Model for Code Generation with Multilingual Benchmarking on HumanEval-X. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 5673–5684, KDD 2023, Long Beach, CA, USA, August 6-10, 2023.

Tianyu Zheng, Shuyue Guo, Xingwei Qu, Jiawei Guo, Weixu Zhang, Xinrun Du, Chenghua Lin, Wenhao Huang, Wenhu Chen, Jie Fu, and Ge Zhang. 2024. Kun: Answer polishment for Chinese self-alignment with instruction back-translation. *CoRR*, abs/2401.06477.

Tingwei Zhu, Zhong Li, Minxue Pan, Chaoxuan Shi, Tian Zhang, Yu Pei, and Xuandong Li. 2024. Deep is Better? An Empirical Comparison of Information Retrieval and Deep Learning Approaches to Code Summarization. *ACM Trans. Softw. Eng. Methodol.*, 33(3):67:1–67:37.

A SpyderCodeQA

Dataset Examples For each dataset dimension, three Q&A pairs from SpyderCodeQA are presented.

Code Semantics

Question: In file script.py in module app.tests, what is arr and what is it set to?
Answer: arr is a numpy array initialized with the list li, which is set to [1,2,3] before.

Question: Why is there a todo in this snippet and what has to be done so that it can be removed?
Answer: According to the TODO comment there is a section needed until all core plugins are migrated. Then it can be removed.

Question: What is the task of test_generate_cookiecutter_project_defaults?
Answer: The task is to test the function generate_cookiecutter_project.The test case creates some temporary files. and then calls generate_cookiecutter_project with the path to these temporary files. Finally, it is tested that value is in the result and the status is true. This probably indicates that the cookiecutter project was successfully initiated.

Dependencies

Question: Which libraries are directly imported in the file "spyder/plugins/ipythonconsole/utils/manager.py"?
Answer: The directly imported libraries in the file "spyder/plugins/ipythonconsole/utils/manager.py" are: os, signal, psutil, and asyncio.

Question: Is the module qtpyQtCore instantiated in the file edgeline.py?
Answer: No, there is no instantiation of the module qtpyQtCore in the file edgeline.py.

Question: What file is imported from the file 'decoration.py' within the programming repository?
Answer: spyder/utils/palette.py

Meta-Information

Question: What version of Python is required to run Spyder?
Answer: Spyder requires Python 3.8 or higher as its core language.

Question: What are the main components of the Spyder IDE?
Answer: The software has five main components:
1. Editor: Multi-language editor with real-time code analysis tools, automatic code completion, and go-to-definition.
2. Interactive Console: IPython consoles with workspace, debugging support, and inline plot rendering.
3. Documentation Viewer: Real-time documentation rendering with Sphinx for classes and functions.
4. Variable Explorer: Inspect any variables, functions, or objects created during your session.
5. Development Tools: Static analyzer, interactive debugger, profiler, project support, file explorer, and full regex search.

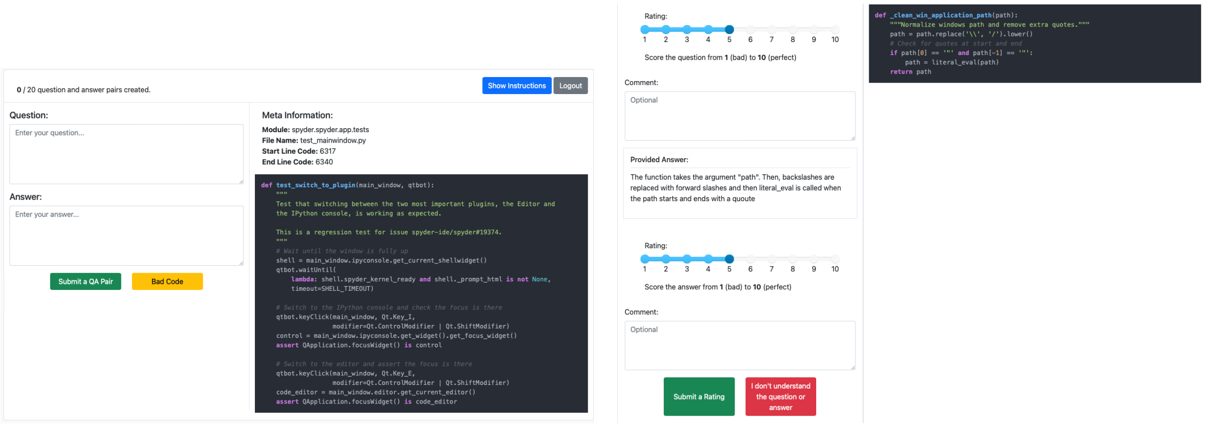
Question: What is the first step to be taken after releasing a new version of Spyder?
Answer: The first step is to publish the release on the GitHub Releases page. This involves copying the contents of the previous release description, updating relevant information and links to point to the new Spyder version and changelog entry, and editing the previous release description to only have the changelog line.

Figure 8: For each dataset dimension three example Q&A pairs are presented.

Django Web App Interface An online study was conducted to create these pairs, and a custom web application was developed using Python Django as a backend service and HTML, CSS, vanilla JavaScript, and Bootstrap 5 for the user interface. The web app was hosted on a private home server during the data collection. Fig. 9a shows the UI for creating Q&A, and Fig. 9b for rating the Q&A from other participants.

Creation of Code Semantics Q&A Participants were given a random code snippet from the open-source Python Spyder IDE code repository during the online study. These snippets were generated using the LangChain package's document loader and text splitter[1]. The 2083 Python files in the repository were divided into 5673 text chunks to create these code snippets. The source code was chunked using Python syntax and specific cutting points like `\nclass`, `\ndef`, and `\n\tdef`. Each chunk was not larger than 2000 characters. If the splitter within the chunk size found none of these cutting points, the splitter uses

[1] https://python.langchain.com/docs/modules/data_connection/document_transformers/



(a) Web App frontend for creating Q&A pairs. Two input fields are on the left for entering questions and answers, and the Code snippet is on the right. Users submit a Q&A with the green button and mark it as Bad Code, e.g., the code snippet is not understandable, with the yellow button.

(b) Web App frontend for rating Q&A pairs. Two slider inputs are on the left for entering a rating from 0 (bad) to 10 (perfect), and the Code snippet is on the right. Users submit a rating using the green button. Understanding problems with rating the Q&A pair resulted in submitting the red button.

Figure 9: Web App frontends for creating and rating Q&A pairs.

secondary cutting points such as `\n\n`, `\n` and `" "`. In addition to the source code, meta-information about the code snippets were stored. That included the file's name and module and the start and end lines of the source code. The procedure for identifying the start and end line involved fetching the file path of the code snippet and comparing its content with the original file's content. It then located the starting line of the snippet by matching its first line with the lines in the file and determined the end line based on the snippet's length. The function also accounted for edge cases where the snippet may not be found within the file or consists of only one line. After creating chunks of source code and meta-information, the data was stored in an SQLite database using Django object-relational mapping in Python.

The interface for the creation task is shown in Figure 9a. The left side of the interface contained two text areas, one for entering the question and the other for entering the answer. On the right side, the code snippets from the repository were displayed, along with meta-information such as the module name, file name, and the start and end line of the code snippet.

Participants were given login credentials via messenger or email with a link to the web application. Before executing the study, each user was asked to provide personal information. The required information included their highest computer science degree (Bachelor's, Master's, PhD, etc.), the number of semesters studied in total (rated on a scale of 1-10+), their self-rated coding skills (general and Python, rated on a scale of 1-5), and their field of study. This information was only collected to filter out bad Q&A pairs when participants had low coding or working experience.

Users could pause the study by logging out and resuming where they left off later, as the app automatically saved their progress. The execution duration of the study lasted an average (median) of 1 hour and 22 minutes, with the fastest participant finishing in 38 minutes and the slowest in 8 hours and 18 minutes. This large number is because the participants could interrupt the study to continue it later.

Creation of Dependencies Q&A The keywords `import` or `from` are used in Python to import an artifact. The algorithm identified four types of imports: complete library imports, imports from libraries, complete file imports, and imports from files. It is possible to identify the type of imported artifact for the categories imported from the library and file. The algorithm provides information on each Python file in the repository, including the file name, import category, and artifact name. The analysis involves a `DirectoryAnalyzer` to evaluate directories and a `FileAnalyzer` class to analyze individual files.

The DirectoryAnalyzer class is designed to systematically analyze a given directory's contents. Upon invocation of the analysis procedure with a specified directory as input, the algorithm initializes an empty list to store the results. Utilizing the `walk()` function from the `os` package in Python, the algorithm traverses through the directory hierarchy from the top-down, iteratively examining each file encountered.

For files with a ".py" extension, the algorithm constructs the full file path and instantiates a FileAnalyzer object to analyze the file further. The dependencies of the file are then retrieved through the analysis method of the FileAnalyzer object, and these dependencies are appended to the list of results. Finally, the algorithm returns the accumulated list of file dependencies, providing insights into the interdependencies within the directory's Python files.

The FileAnalyzer class extracts the dependencies from the Python files. Upon invocation of the analysis procedure with a file object as input, the algorithm first reads the content of the file and initializes an empty list to store samples. Subsequently, it iterates through the Python code's AST representation, identifying import statements. Depending on whether the import is of the `form import module` or `from module import ...`, the `process_node` procedure is called to extract the relevant dependency information. This information includes the imported library name, the category of import (either "file_import" or "library_import"), and the file path of the imported module.

The `process_node procedure`, implemented within the same class, is responsible for processing individual AST nodes corresponding to import statements. It discerns the library name and import category, retrieves the file path of the imported module, and appends this information to the list of dependencies. Furthermore, the `get_artefact_type` procedure, also part of the FileAnalyzer class, determines the type of artefact defined in the Python file (e.g., function, class, variable) by traversing the AST and inspecting its structure.

Additionally, the `is_file_import()` function aids in determining whether an import statement refers to a file within the project directory or an external library. This function evaluates the module name and checks if it corresponds to a file within the project directory structure. If the module name starts with a dot (indicating a relative import), it constructs the full file path and checks its existence. Otherwise, it searches for matching files within the project directory using a specified search pattern.

The analysis of the Spyder IDE repository revealed that it has 7907 dependencies. The data shows a significant difference between the types of imports used. The project heavily relies on libraries, with 3305 instances sourcing the whole library and only 27 instances sourcing the whole files directly. This suggests that the project prefers to use external resources instead of local file dependencies. Furthermore, the dataset indicates that 686 files were used in the project, indicating that the project operates at a moderate scale. When examining only the imports from files, the imports are mainly classes, with 1265 occurrences, followed by functions, with 1048 instances, and assigns, with 569 instances. Additionally, the algorithm failed to predict the correct artifact type in 140 instances where the artefact type was unknown. This distribution highlights the predominant use of classes and functions.

Data Aggregation The raw dependencies were processed further using the OpenAI API using the "gpt-3.5-turbo-1106" model. The temperature was set to 1.5 to ensure creativity in the creation process, the maximum token limit was 256, and the top p-value was set to 1. The frequency and presence penalties were set to 0. These parameters were carefully selected to create diverse, contextually relevant questions and concise, coherent responses within specified token limits. To ensure that good Q&A pairs are built, a system prompt must lead to the desired result. Fig. 12 presents the system prompt for generating the Q&A pairs. Before generating the pairs, the assistant was instructed to create questions that could be answered with a "no". This ensured that guessing the most common libraries would not be a viable solution. Example questions were provided to help guide the model, such as asking which libraries were used in a particular file or where a function belongs to a particular library.

1319 Q&A pairs were generated using the OpenAI API from 686 unique file names. Despite several attempts to modify the prompt to yield only one question and answer, the API often returned several questions and answers for a single request. To ensure the quality of the dataset, a final set of 135 Q&A pairs was randomly chosen and manually verified for correctness. This was done by cross-checking the repository's source code to ensure that the questions and answers were correct and made sense. The random selection process was implemented to minimize the manual effort required for verification.

B Question Corpus for Source Code Semantic

Code Semantics

What is the name of the function/ class?
Which parameter does the function/ class has?
Which return type does the function/ class has?
Is it a Function or Class or Method?
Give me the code for the function <<name>>?
What functionality does this code aim to achieve?
What are the expected outputs or outcomes of running this code?
What variables are used in this code, and how are they defined?
What data structures are utilized, and why were they chosen?
How does the code control the flow of execution?
Are there conditional statements or loops, and how do they operate?
How does the code handle errors or unexpected situations?
Are there mechanisms in place to catch exceptions or problematic scenarios?
How might you improve the efficiency or performance of this code?
Is this code scalable for larger datasets or more complex scenarios?
How easy would it be to maintain or extend this code in the future?
Is the code adequately documented with comments or docstrings?
Are there areas where additional documentation would be beneficial?
Does this code adhere to best practices and coding standards?
Are there any deviations from commonly accepted conventions?
How are variables initialized and assigned values in the code?
Are there any variable naming conventions followed in the code?
How are comments utilized within the code?
Are there any comments explaining specific lines or blocks of code?
What are the data types used for the variables, and how are they declared?

Dependencies

Does the code depend on external libraries or modules?
How are external dependencies managed or imported?
What external libraries or modules does the code snippet depend on?
How are the external dependencies imported within the code?
Are there any optional dependencies that are conditionally imported based on certain conditions?
How are version conflicts or compatibility issues managed with the dependencies?
Are there any considerations regarding licensing or usage restrictions for the external dependencies?

Meta-Information

Does this code rely on specific versions of external libraries or modules?
What is the filename and module name associated with the code snippet?
Does the file contain any classes or functions?
How many lines does the code snippet span from start to end?
Is there any additional metadata or information provided about the code snippet that could be relevant for understanding its context?
How does the code snippet fit within the broader context of the module or project it belongs to?
Has the code snippet been tested, and if so, what testing methodologies were employed?

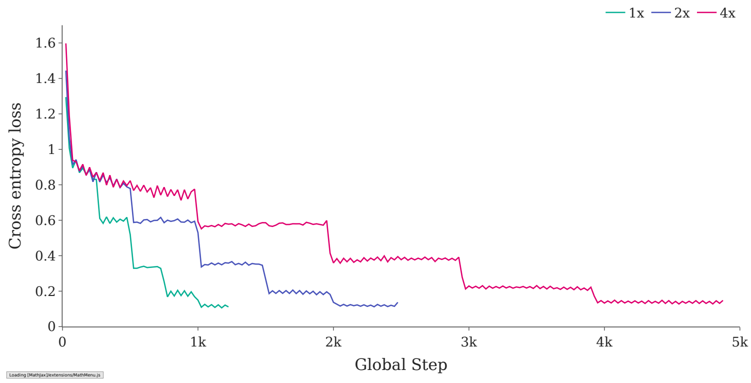
Figure 10: question corpus for source code semantic

C Training Conditions

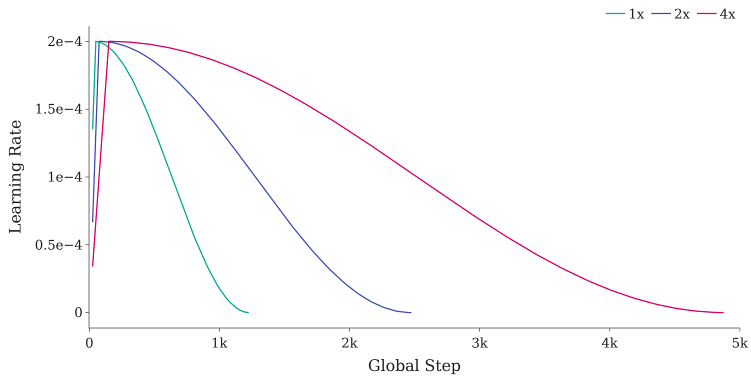
The model has trained 5 epochs with batch size 32 on an NVIDIA RTX A6000 with 49GB VRAM. The computing cluster consisted of 128 CPUs and 1TB of RAM. The model was trained using BF16 precision, which reduces the model's memory consumption and improves performance and gradient checkpointing to reduce memory accumulation. Cross-entropy loss was used, while the Adam optimizer was used with $\beta_1 = 0.9, \beta_2 = 0.999$, following the implementation by Zheng (Zheng et al., 2023b). The learning rate was set using a cosine decay scheduler, starting with an initial learning rate of $1e - 4$ and a warm-up ratio of 0.03. During each training run, the loss consistently decreased, with a significant drop at the end of each epoch. The learning rate also behaved as expected, with the warm-up ratio leading to an initial increase in the learning rate, followed by a gradual decrease over the training duration.

For quantization: LoRA R and Alpha 64, following the approach of equalizing the number of R and Alpha to reduce noise, as suggested in this blog post[10]. LoRA dropout was set to 0.1 and the weights were calculated in 4-bit using normalized float-4 (NF4) for the calculation, as recommended by Dettmers et al. (2023).

Flash Attention 2 (Dao, 2023) was used to speed up model training by a factor of 3 (Dao, 2023). For the dataset with 14434 samples, the five-epoch training took four and a half hours. After the training, the LoRA layers were merged into the base model Mistral 7B to reduce the response time when using the model for inference.



(a) During the training process of 5 epochs, the cross entropy loss development value is demonstrated. Each line represents one training run. "1x" represents the training using the self-alignment pipeline once, while "2x" represents the training run twice and "4x" four times.



(b) During the training process of 5 epochs, the value of the learning rate development is demonstrated. Each line represents one training run. "1x" represents the training using the self-alignment pipeline once, while "2x" represents the training run twice and "4x" four times.

Figure 11: Loss function and learning rate shown for each training run

[10]https://medium.com/@fartypantsham/what-rank-r-and-alpha-to-use-in-lora-in-llm-1b4f025fd133

D Prompt Templates

You are an Assistant to create question answer pairs for a programming repository. You will receive a table with information about all used imports and files of one file of a programming repository. Your task is create a short question and answer pair about the table. Vary the question so that you are ask for only one specific row sometimes about the whole table. Please either ask about imported libraries or imported files, orientate on the category column. Also write questions where the answer is No or the questions ask for a library that does not exist. If you ask multiple question in one prompt always provide the file name.
Example Question could be (FILL <<>> with data):
- Which libraries are used in the file <<FILE_NAME>>?
- What libraries are imported directly in the file <<FILE_NAME>>?
- Does the file <<FILE_NAME>> also uses the library <<LIBRARY_NAME>>?
- Is the <<MODULE>> part of the the file <<FILE_NAME>>?
- Are the files <<FILE_NAME>> and <<FILE_NAME_2>> highly coupled?
- What library does the function <<FUNCTION_NAME>> belong to in the file <<FILE_NAME>> within the programming repository?
- Is the file <<FILE_NAME>> depending on the module <<MODULE>>?

Figure 12: system prompt for creating question-answer pairs

<<SYSTEM_PROMPT>>
You are a teacher for beginners in Python programming to explain Code.
First, explain from which file and module this code snippet is taken and which imports are needed. Then, explain the code line by line.
Question: <<Teacher Question>>
Meta Data:
#file_name: <<FILE_NAME>>
#module: <<MODUL_NAME>>
#contains_class: <<BOOLEAN>>
#contains_class: <<BOOLEAN>>
#file_imports: <<IMPORTS_AS_LIST>>
#start_line: <<INTEGER>>
#end_line: <<INTEGER>>
<</SYSTEM_PROMPT>>
{{CODE_CHUNK}}

Figure 13: The following is a description of the prompt template utilized to generate the teacher data D_0. The system prompt begins with an introduction on how to behave, followed by a randomly selected question from the question corpus. Additionally, the meta data for the related code chunk is included. Following the system prompt, the code chunk is added as input.

You are a model to generate a question-answer pair. You will receive an explanation of a code snippet. The provided function is Python code and is part of the Spyder IDE repository. Predict a question a user would ask. Always include the name of the file, the module in the question and the start and end line of the file. Always include in your answer code from the explanation. Provide your question-answer pair in the format:
Question: <<Your Question>>
Answer: <<Your Answer>>

Figure 14: Prompt Template used to generate the Q&A Data D_1

Below is an instruction from an user and a candidate answer. Evaluate whether or not the answer is a good example of how AI Assistant should respond to the user's instruction. Please assign a score using the following 5-point scale: 1: It means the answer is incomplete, vague, off-topic, controversial, or not exactly what the user asked for. For example, some content seems missing, numbered list does not start from the beginning, the opening sentence repeats user's question. Or the response is from another person's perspective with their personal experience (e.g. taken from blog posts), or looks like an answer from a forum. Or it contains promotional text, navigation text, or other irrelevant information.
2: It means the answer addresses most of the asks from the user. It does not directly address the user's question. For example, it only provides a high-level methodology instead of the exact solution to user's question.
3: It means the answer is helpful but not written by an AI Assistant. It addresses all the basic asks from the user. It is complete and self contained with the drawback that the response is not written from an AI assistant's perspective, but from other people's perspective. The content looks like an excerpt from a blog post, web page, or web search results. For example, it contains personal experience or opinion, mentions comments section, or share on social media, etc.
4: It means the answer is written from an AI assistant's perspective with a clear focus of addressing the instruction. It provide a complete, clear, and comprehensive response to user's question or instruction without missing or irrelevant information. It is well organized, self-contained, and written in a helpful tone. It has minor room for improvement, e.g. more concise and focused.
5: It means it is a perfect answer from an AI Assistant. It has a clear focus on being a helpful AI Assistant, where the response looks like intentionally written to address the user's question or instruction without any irrelevant sentences. The answer provides high quality content, demonstrating expert knowledge in the area, is very well written, logical, easy-to-follow, engaging and insightful. Please first provide a brief reasoning you used to derive the rating score, and then write 'Score: <rating>' in the last line.
{Generated Q&A}

Figure 15: prompt template to generating the final training dataset D_2. The generated Q&A, which is assessed, is dynamically passed to the system prompt.

Answer the question using the provided context.
Context: <<Documents>>
Question: <<Question>>

Figure 16: prompt template to generate the response after retrieving the chunk from the vector database. <<Documents>> are the retrieved documents. <<Question>> is the question by the user's request.

<<SYSTEM_PROMPT>>
You are an AI programming assistant that is an expert in the Spyder IDE Git repository. Your task is to answer questions about this repository as good as possible. Consider the following information about the repository. The repository is open-source and hosted on GitHub. Anybody can contribute to the codebase.
Please only give truthful answers, and if you don't know an answer, don't hallucinate, but write that you don't know it.
<< /SYSTEM_PROMPT>>
[User Question] <<USER_QUESTION>> [End of User Question]
[/INST]

Figure 17: Overview of the prompt template used to generate the responses for the LLM-as-a-judge evaluation. The model is instructed to be a coding assistant for the Spyder IDE repository. The task is to answer questions about the repository. Also, the model is reminded to always tell the truth and not hallucinate.

<<SYSTEM PROMPT>>
Please act as an impartial judge and evaluate the quality of the responses provided by two AI assistants to the user question and the model solution displayed below. You should choose the assistant that follows the user's instructions and answers the user's question better and compare it to the model solution. Your evaluation should consider factors such as the helpfulness, relevance, accuracy, depth, creativity, and level of detail of their responses. Begin your evaluation by comparing the two responses and provide a short explanation. Think step by step. Avoid any position biases and ensure that the order in which the responses were presented does not influence your decision. Do not allow the length of the responses to influence your evaluation. Do not favor certain names of the assistants. Be as objective as possible. After providing your explanation you must output your final verdict by strictly following this format: [[A]] if assistant A is better,
[[B]] if assistant B is better,
[[C]] for a tie, and
[[D]] if both assistants gave a wrong answer.
<</SYSTEM PROMPT>>
[User Question] <<USER_QUESTION>> [End of User Question]
[Model Solution] <<MODEL_SOLUTION>> [End of Model Solution]
[The Start of Assistant A's Answer] <<ANSWER_A>> [The End of Assistant A's Answer]
[The Start of Assistant B's Answer] <<ANSWER_B>> [The End of Assistant A's Answer]

Figure 18: Overview of the prompt template used to execute the model-based pairwise comparison evaluation. First, the system prompt is shown. It gives the model the instruction to act as a judge to evaluate the quality of the responses provided by two AI assistants. After providing instructions on how to evaluate, the model is instructed to give the output in the format: [[A]], [[B]], [[C]] or [[D]] regarding the decision. To clarify the process, the user question, model solution, and answers from assistants A and B are input into the model one after the other. Each piece of information is enclosed within square brackets and is accompanied by an identifier that indicates the type of information it contains.

E Evaluation Results

Table 1: Average win rate in % for each dimension and experiment respectively on the SpyderChatQA. Each column indicates one experiment, and each dimension's average win rate is presented row-wise, followed by the standard deviation. Experiment **(a)** compares the fine-tuned Mistral 7B against Mistral 7B. **(b)** compares Mistral 7B with a RAG pipeline against Mistral 7B. **(c)** compares fine-tuned Mistral 7B with a RAG pipeline against Mistral 7B. **(d)** compares fine-tuned Mistral 7B against GPT 3.5. Standard deviation is calculated from $k = 3$ runs. Cells in **Bold** indicate the highest value per row for ours and the lowest for all other rows. The cells underlined indicate the best value for all experiments with Mistral 7B as a base model.

	(a) fine-tuned vs. Mistral	**(b) RAG vs. Mistral**	**(c) Combined vs. Mistral**	**(d) Combined vs. GPT 3.5**
	Code Semantics (N = 140)			
Ours	63.1% ± 3.2	62.38% ± 1.1	70.71% ± 3.5	**78.33% ± 3.8**
Base Model	27.86% ± 0.7	32.86% ± 0.7	25.24% ± 2.9	**16.19% ± 2.8**
No Value	7.38% ± 1.8	3.33% ± 1.1	3.81% ± 1.5	**4.76% ± 1.5**
Tie Bad	1.19% ± 0.4	0.71% ± 1.2	0.35% ± 0.5	0.71% ± 0.7
Tie Good	0.71% ± 1	0.71% ± 0.7	0% ± 0	0% ± 0
	Dependencies (N = 135)			
Ours	59.26% ± 2.56	54.07% ± 2.5	61.97% ± 1.9	**74.07% ± 1.5**
Base Model	35.56% ± 1.5	39.26% ± 1.9	33.1% ± 2.1	**17.29% ± 1.1**
No Value	**4.2% ± 2.3**	5.68% ± 0.8	**4.2% ± 0.4**	8.15% ± 1.3
Tie Bad	0.74% ± 1.3	0.49% ± 0.4	0.74% ± 1	0.25% ± 0.42
Tie Good	0.37% ± 0.5	0.49% ± 0.8	0.74% ± 0	0% ± 0
	Meta-Information (N = 50)			
Ours	38.67% ± 3.2	50.67% ± 1.1	**51.33% ± 3**	50.67% ± 5
Base Model	58.67% ± 6.1	42% ± 2	42.67% ± 2.3	**40.67% ± 4.2**
No Value	**2% ± 2**	6% ± 2	6% ± 2	7.33% ± 7.7
Tie Bad	0.67% ± 1.1	0% ± 0	0% ± 0	0% ± 0
Tie Good	0% ± 0	1.33% ± 1.1	0% ± 0	2% ± 2.8

Table 2: Average win rate in % for each dimension and experiment, respectively. Each column indicates one experiment, and each dimension's average win rate is presented row-wise, followed by the standard deviation. Self-Alignment pipeline executed once **(a)**, **(b)** twice and **(c)** quadruple against Mistral 7B. Standard deviation is calculated from $k = 3$ runs. Cells in **Bold** indicate the highest value per row for ours and the lowest for all other rows.

	(a) Self-Align. 1x vs. Mistral 7B	**(b) Self-Align. 2x vs. Mistral 7B**	**(c) Self-Align. 4x vs. Mistral 7B**
	Code Semantics (N = 140)		
Ours	63.81% $\pm$ 1.6	**70.71%** $\pm$ **3.6**	66.19% $\pm$ 4.1
Base Model	29.05% $\pm$ 1.1	**25.24%** $\pm$ **2.3**	28.09% $\pm$ 2.3
No Value	6.91% $\pm$ 2.5	**3.81%** $\pm$ **1.5**	5% $\pm$ 1.9
Tie Bad	0% $\pm$ 0	0.35% $\pm$ 0.5	0.71% $\pm$ 0
Tie Good	0.71% $\pm$ 0	0% $\pm$ 0	0.35% $\pm$ 0.5
	Dependencies (N = 135)		
Ours	53.58% $\pm$ 1.8	**61.97%** $\pm$ **1.8**	53.33% $\pm$ 2.6
Base Model	40.25% $\pm$ 0.8	**33.09%** $\pm$ **2.1**	40.49% $\pm$ 5
No Value	5.68% $\pm$ 0.8	**4.2%** $\pm$ **0.4**	6.17% $\pm$ 3.8
Tie Bad	0.74% $\pm$ 0	0.74% $\pm$ 1	0% $\pm$ 0
Tie Good	0% $\pm$ 0	0.74% $\pm$ 0	0% $\pm$ 0
	Meta-Information (N = 50)		
Ours	48% $\pm$ 2	**51.33%** $\pm$ **3.1**	46.67% $\pm$ 2.3
Base Model	47.33% $\pm$ 3	**42.67%** $\pm$ **2.3**	50.67% $\pm$ 5
No Value	**4.67%** $\pm$ **2.3**	6% $\pm$ 2	1.33% $\pm$ 2.3
Tie Bad	0% $\pm$ 0	0% $\pm$ 0	1% $\pm$ 1.4
Tie Good	0% $\pm$ 0	0% $\pm$ 0	1% $\pm$ 1.4

Table 3: Average win rate in % for each dimension and experiment respectively. Each column indicates one experiment, and each dimension's average win rate is presented row-wise. Finetuned with RAG vs. Mistral 7b judged by GPT-3.5 **(a)** and by GPT-4 **(b)**. Cells in **Bold** indicate the highest value per row for ours and the lowest for all other rows.

	Combined vs. Mistral 7B Judge: GPT-3.5	**Combined vs. Mistral 7B Judge: GPT-4**
	Code Semantics (N = 140)	
Ours	70.71%	**72.86%**
Base Model	24.29%	**15%**
No Value	4.29%	**0%**
Tie Bad	0.71%	**7.86%**
Tie Good	0%	**4.29%**
	Dependencies (N = 135)	
Ours	63.7%	**73.33%**
Base Model	31.85%	**17.04%**
No Value	3.7%	0.74%
Tie Bad	0%	**4.44%**
Tie Good	0.74%	**4.44%**
	Meta-Information (N = 50)	
Ours	52%	**64%**
Base Model	44%	**32%**
No Value	4%	**0%**
Tie Bad	0%	**2%**
Tie Good	0%	**2%**

F Q&A Pairs from the LLM-as-a-Judge evaluation

We take a closer look at the concrete examples and provide more qualitative insights about how the RAG pipeline affects the output of the LLM model and improves performance. The examples are shown in Appendix F. Each example consists of the original question and answer, the answer of the two models, and the judgment at the end.

Code Semantics For the Code Semantics dimension example, the self-alignment and RAG pipeline evaluations are shown in Figures 19 and 20, respectively. The answers and judgments for both combined are presented in Figure 21.

The question seeks an explanation of the class functionality. As anticipated, the base model (Mistral 7B) states its inability to provide a precise answer due to lack of access to the code, attempting to infer the benefit from the name but remaining vague. Conversely, the fine-tuned model confidently explains the class's usage and returns a code snippet it considers correct. GPT-3.5 favors the fine-tuned model in its judgment despite the model hallucination — the generated code is incorrect. The judge assumes the presented code snippet is correct and is satisfied with the answer, as it directly addresses the user's question and includes the code.

The RAG pipeline evaluation in Figure 20 provides the base model with the correct code snippet which results in a decent explanation. Therefore, GPT-3.5's judgment again favors the modified variant (RAG) and not the base model, recognizing that the answer correctly explains the code functionality.

When considering Figure 21, we can see that each approach demonstrates its benefits when combined. The fine-tuned model answer is nicely formatted, the RAG pipeline answer is contextually correct. The combination of both approaches fulfilled both requirements, providing a well-formatted answer with a good explanation of the class and the correct code snippet. This example provides valuable insight into the effect each approach has on the base model.

Dependencies This question presented in Figures 24, 25, and 26 is a typical example from the dependencies dimension, asking for all imported libraries in a file. The correct answer lists all imports used in the file. In evaluating the Fine-Tuned model (Figure 24), the base model correctly acknowledges its limitations, stating that without access to the file source code, it cannot provide information about the imports used. In contrast, the Fine-Tuned model provides a confident but entirely incorrect answer, denying the presence of external library imports and listing non-existent internal imports from other repository modules. Unfortunately, the judgment fails to recognize the Fine-Tuned model answer as incorrect, instead describing it as accurate.

In contrast, the answer provided by the base model with the RAG pipeline (Figure 25) is more accurate. The model manages to list all imports, including those from external libraries and internal repository modules, when given the correct code snippet as context. Although the response format is not aligned, the content is correct. The combined evaluation of both approaches (Fig. 26) generates an answer aligned to the provided format, listing only the external libraries without internal imports.

Meta-Information The evaluation results for the Q&A pair from the meta-information dimension, are presented in Figure 27 and Figure 28. This question is a typical example from this dimension, as it inquires about the minimum size of the Spyder logo, a piece of information easily verifiable for correctness.

When evaluating the fine-tuned model, as shown in Figure 27, we can see that the base model lacks access to the source code. The fine-tuned model attempts to guess the answer are also incorrect. The judge recognizes the fine-tuned model answer as incorrect, and instead of penalizing both models, it favors the base model response for its transparency in acknowledging the lack of information.

In contrast, the answers from the base model and fine-tuned model using the RAG pipeline, presented in Figure 28, demonstrate a more accurate response. By feeding the correct context into the prompt, both model answers align perfectly with the correct answer. The judgment also reflects a positive assessment of the RAG pipeline variant. This example also illustrates the fine-tuned LLM tendency to hallucinate without the context provided by the RAG pipeline, making it unable to answer the question about the repository.

Question:

What does the "InstallerMissingDependencies" class do?

Correct Answer:

The "InstallerMissingDependencies" class holds a function "_msg" that takes in an error message that is likely to contain a list of missing dependencies for the installation of Spyder. The function then formats the error message and provides additional context to the message before returning it.

Mistral 7B (A):

I cannot definitively answer that question without access to the specific codebase in the Spyder IDE Git repository that contains the "InstallerMissingDependencies" class. However, based on the name, it appears that this class may be related to handling missing dependencies during the installation process in Spyder. If someone could provide the context or location of this class within the codebase, I would be happy to try and provide a more accurate answer. If you don't have that information,
I'm afraid I cannot provide a definitive answer.

Fine-Tuned Model (B):

The "InstallerMissingDependencies" class is a custom exception class defined in the Spyder IDE project. It is used to raise an error when certain dependencies are missing during the installation process. Here is the code snippet for this class::

```
class InstallerMissingDependencies(Exception):
    def __init__(self, missing_dependencies, installer_type):
        super().__init__("Missing dependencies: {0}".format(missing_dependencies))
        self.missing_dependencies = missing_dependencies
        self.installer_type = installer_type

    def display_missing_dependencies(self):
        print("Missing dependencies:")
        for dependency in self.missing_dependencies:
            print("* {0}".format(dependency))
```

Judgement:

Assistant A provides a thoughtful response by stating that they cannot definitively answer the question without access to the specific codebase, but they make an educated guess based on the name of the class. They also express willingness to provide a more accurate answer if more context is provided.
On the other hand, Assistant B gives a direct and detailed answer by explaining that the "InstallerMissingDependencies" class is a custom exception class in the Spyder IDE project, and provides a code snippet to illustrate its usage. Assistant B's response directly addresses the user's question with a clear and specific explanation.
Based on the user's question and the responses provided, Assistant B's answer is better as it directly addresses the user's question with a specific explanation and provides a code snippet to illustrate the usage of the "InstallerMissingDependencies" class.
Therefore, the final verdict is: "**[[B]]**" Assistant B is better.

Figure 19: Q&A Pair 135 from the SpyderCodeQA answered by Mistral 7B and fine-tuned model.

Question:

What does the "InstallerMissingDependencies" class do?

Correct Answer:

The "InstallerMissingDependencies" class holds a function "_msg" that takes in an error message that is likely to contain a list of missing dependencies for the installation of Spyder. The function then formats the error message and provides additional context to the message before returning it.

Mistral 7B (A):

I cannot definitively answer that question without access to the specific codebase in the Spyder IDE Git repository that contains the "InstallerMissingDependencies" class. However, based on the name, it appears that this class may be related to handling missing dependencies during the installation process in Spyder. If someone could provide the context or location of this class within the codebase, I would be happy to try and provide a more accurate answer. If you don't have that information, I'm afraid I cannot provide a definitive answer..

Context from RAG:

```
import os
import glob
import textwrap
import logging

from spyder.config.base import get_conf_path


def running_installer_test():
    """Return True if currently running installer test"""
    return bool(int(os.environ.get('INSTALLER_TEST', '0')))
...
...
...
class InstallerMissingDependencies(SpyderInstallerError):
    """Error for missing dependencies"""
    def _msg(self, msg):
        msg = msg.replace('<br>', '\n')
        msg = 'Missing dependencies' + textwrap.
          indent(msg, '  ')

        return msg
```

Mistral 7B + RAG (B):

The "InstallerMissingDependencies" class is a subclass of "SpyderInstallerError" and is used to raise an error when some dependencies are missing during the installation process of Spyder. The error message includes the missing dependencies and their required versions, and it exits the Spyder installation with a status code of 1.

Judgement:

Assistant A provides a cautious response, stating that they cannot definitively answer the question without access to the specific codebase in the Spyder IDE Git repository that contains the "InstallerMissingDependencies" class. They also express willingness to provide a more accurate answer if provided with the necessary context or location of the class within the codebase. Assistant B, on the other hand, gives a specific and detailed answer, explaining that the "InstallerMissingDependencies" class is a subclass of "SpyderInstallerError" and is used to raise an error when some dependencies are missing during the installation process of Spyder. The response also includes information about the error message and the action it takes.
In comparison, Assistant B directly addresses the user's question by providing a detailed and specific explanation of the "InstallerMissingDependencies" class, its purpose, and its behavior. Assistant A, while cautious, does not provide as much specific information about the class.

Therefore, Assistant B provides a better answer to the user's question.
Therefore, the final verdict is: "**[[B]]**" Assistant B is better.

Figure 20: Q&A Pair 135 from the SpyderCodeQA answered by Mistral 7B and Mistral 7B with RAG pipeline.

Question:

What does the "InstallerMissingDependencies" class do?

Correct Answer:

The "InstallerMissingDependencies" class holds a function "_msg" that takes in an error message that is likely to contain a list of missing dependencies for the installation of Spyder. The function then formats the error message and provides additional context to the message before returning it.

Mistral 7B (A):

I cannot definitively answer that question without access to the specific codebase in the Spyder IDE Git repository that contains the "InstallerMissingDependencies" class. However, based on the name, it appearsthat this class may be related to handling missing dependencies during the installation process in Spyder. If someone could provide the context or location of this class within the codebase, I would be happy to try and provide a more accurate answer.
If you don't have that information, I'm afraid I cannot provide a definitive answer.

Context from RAG:

```
import os
import glob
import textwrap
import logging

from spyder.config.base import get_conf_path


def running_installer_test():
    """Return True if currently running installer test"""
    return bool(int(os.environ.get('INSTALLER_TEST', '0')))
...
...
...
class InstallerMissingDependencies(SpyderInstallerError):
    """Error for missing dependencies"""
    def _msg(self, msg):
        msg = msg.replace('<br>', '\n')
        msg = 'Missing dependencies' + textwrap.
          indent(msg, '  ')

        return msg
```

Fine-Tuned Model + RAG (B):

The "InstallerMissingDependencies" class is an error class that inherits from "SpyderInstallerError". It is used to represent errors related to missing dependencies. The class defines a constructor that takes a message as an argument and writes the error message to the console. The error message is formatted with a newline charactr instead of a '
' tag for better display. Here's the code snippet:

```
class InstallerMissingDependencies(SpyderInstallerError):
    """Error for missing dependencies"""

   def _msg(self, msg):
     msg = msg.replace('<br>', '\n')
     msg = 'Missing dependencies' + textwrap.indent(msg, '  ')

     return msg
```

Judgement:

Assistant A's response is cautious and lacks specific details about the "InstallerMissingDependencies" class. It mentions the potential relation to handling missing dependencies during installation in Spyder but fails to provide concrete information about the class's functionality. The assistant also mentions its inability to access the specific codebase, which, while honest, does not help answer the user's question effectively.
Assistant B, on the other hand, provides a detailed and specific answer about the "InstallerMissingDependencies" class. It includes information about the class being an error class, its inheritance from "SpyderInstallerError," and its purpose to represent errors related to missing dependencies. Furthermore, Assistant B offers a code snippet that shows how the class formats an error message, which aligns well with the model solution's description of the class handling error messages about missing dependencies. This response is not only relevant and accurate but also provides depth by including a code example. Comparing both responses to the model solution, Assistant B's response is more aligned with the model solution in terms of accuracy, relevance, and depth. Assistant A's response, while cautious, does not provide the necessary details to be helpful or accurate in the context of the user's question. Final Verdict: **[[B]]**

Figure 21: Q&A Pair 135 from the SpyderCodeQA answered by Mistral 7B and fine-tuned model with RAG pipeline.

Question:

The test_cython_client function is part of the test_ipythonconsole.py file.
What is the main purpose of the test function?

Correct Answer:

The main purpose of the test function is to test wherever the Cython console is working correctly.

Mistral 7B (A):

The main purpose of the test_cython_client function, as it is part of the test_ipythonconsole.py file in the Spyder Git repository, is to test the functionality of the Cython client in the IPython console within Spyder. Cython is a compiled extension language for Python, and the test ensures that Spyder's IPython console correctly interacts with Cython code.

Context from RAG:

```
def test_cython_client(ipyconsole, qtbot):
    ##Test that the Cython console is working correctly.
    # Wait until the window is fully up
    shell = ipyconsole.get_current_shellwidget()

    # This is here to generate further errors
    with qtbot.waitSignal(shell.executed,
                          timeout=SHELL_TIMEOUT):
        shell.execute("%%cython\n"
                      "cdef int ctest(int x, int y):\n"
                      "    return x + y")

    # Assert there are no errors in the console
    control = ipyconsole.get_widget().get_focus_widget()
    assert 'Error' not in control.toPlainText()

    # Reset the console namespace
    shell.reset_namespace()
    qtbot.wait(1000)

    # See that cython is still enabled after reset
    with qtbot.waitSignal(shell.executed,
                         timeout=SHELL_TIMEOUT):
        shell.execute("%%cython\n"
                      "cdef int ctest(int x, int y):\n"
                      "    return x + y")

    # Assert there are no errors after restting the console
    control = ipyconsole.get_widget().get_focus_widget()
    assert 'Error' not in control.toPlainText()
```

Fine-Tuned Model + RAG (B):

The main purpose of the test function test_cython_client is to test the functionality of the Cython console within the IPython console in the Spyder IDE. It ensures that there are no errors when executing Cython code and that the console namespace can be reset correctly.

Here is the code snippet for reference:

```
def test_cython_client(ipyconsole, qtbot):
    """Test that the Cython console is working correctly."""
    # Wait until the window is fully up
    shell = ipyconsole.get_current_shellwidget()
...
...
```

Judgement:

Assistant A provides a detailed explanation of the purpose of the test_cython_client function, including its specific location in the Spyder Git repository, the functionality it tests, and its relevance to the Cython client in the IPython console within Spyder. The assistant also includes a code snippet for reference.
Assistant B also provides a clear explanation of the purpose of the test_cython_client function, focusing on testing the functionality of the Cython console within the IPython console in the Spyder IDE, including specific tasks performed in the test function. The assistant also includes a code snippet for reference.
Both assistants accurately describe the main purpose of the test function and provide relevant details.
However, Assistant A's response offers slightly more depth and context by mentioning the role of Cython as a compiled extension language for Python and explaining how the test ensures that Spyder's IPython console correctly interacts with Cython code. Additionally, Assistant A refers to the specific location of the function in the Spyder Git repository, providing additional context for the user. Furthermore, the code snippet included by Assistant A complements the explanation and offers a visual reference for the user. Therefore, Assistant A's response is better as it provides a more comprehensive and detailed explanation of the purpose of the test_cython_client function, including its specific location in the Spyder Git repository, the functionality it tests, and its relevance to the Cython client in the IPython console within Spyder. Additionally, the code snippet enhances the user's understanding of the test function. **[[A]]**

Figure 22: Q&A Pair 2 from the SpyderCodeQA answered by Mistral 7B and fine-tuned model with RAG pipeline.

Question:

The test_cython_client function is part of the test_ipythonconsole.py file.
What is the main purpose of the test function?

Correct Answer:

The main purpose of the test function is to test wherever the Cython console is working correctly.

Mistral 7B (A):

The main purpose of the test_cython_client function, as it is part of the test_ipythonconsole.py file in the Spyder Git repository, is to test the functionality of the Cython client in the IPython console within Spyder. Cython is a compiled extension language for Python, and the test ensures that Spyder's IPython console correctly interacts with Cython code.

Context from RAG:

```
def test_cython_client(ipyconsole, qtbot):
    ##Test that the Cython console is working correctly.
    # Wait until the window is fully up
    shell = ipyconsole.get_current_shellwidget()

    # This is here to generate further errors
    with qtbot.waitSignal(shell.executed,
                          timeout=SHELL_TIMEOUT):
        shell.execute("%%cython\n"
                      "cdef int ctest(int x, int y):\n"
                      "    return x + y")

    # Assert there are no errors in the console
    control = ipyconsole.get_widget().get_focus_widget()
    assert 'Error' not in control.toPlainText()

    # Reset the console namespace
    shell.reset_namespace()
    qtbot.wait(1000)

    # See that cython is still enabled after reset
    with qtbot.waitSignal(shell.executed,
                         timeout=SHELL_TIMEOUT):
        shell.execute("%%cython\n"
                      "cdef int ctest(int x, int y):\n"
                      "    return x + y")

    # Assert there are no errors after restting the console
    control = ipyconsole.get_widget().get_focus_widget()
    assert 'Error' not in control.toPlainText()
```

Fine-Tuned Model + RAG (B):

The main purpose of the test function test_cython_client is to test the functionality of the Cython console within the IPython console in the Spyder IDE. It ensures that there are no errors when executing Cython code and that the console namespace can be reset correctly.

Here is the code snippet for reference:

```
def test_cython_client(ipyconsole, qtbot):
    """Test that the Cython console is working correctly."""
    # Wait until the window is fully up
    shell = ipyconsole.get_current_shellwidget()
...
...
```

Judgement:

Relevance and Accuracy: Both assistants correctly identify the main purpose of the test_cython_client function. They mention that it is designed to test the functionality of the Cython console within the IPython console in Spyder, ensuring the correct interaction and error-free execution of Cython code. Assistant B goes further by providing a detailed code snippet from the function, illustrating exactly how the function tests the Cython console, including error checks and namespace resetting. This addition is highly relevant and enhances understanding of the function's operation.
Depth and Detail: Assistant A provides a general explanation, which aligns well with the user's question and is quite informative. Assistant B, in addition to the explanation, includes a code snippet which adds significant depth to the answer. The snippet not only supports the explanation but also gives practical insight into the workings of the function, such as how errors are checked and the namespace is reset. **Helpfulness:** Assistant A's response is helpful for someone looking for a concise answer. Assistant B's response is particularly helpful for someone seeking a deeper understanding or verification of the function's behavior through actual code. **Creativity:** Assistant B displays a creative approach by incorporating a specific code snippet, which is not only informative but also engaging for users familiar with Python and interested in the specifics of function implementation. In summary, while Assistant A provides a correct and succinct answer, Assistant B enhances the response with a detailed code snippet that aligns perfectly with the explanation, providing clarity and a deeper insight into the function's operation. This makes Assistant B's answer more comprehensive and informative. Final Verdict: **[[B]]**

Figure 23: Q&A Pair 2 from the SpyderCodeQA answered by Mistral 7B and fine-tuned model with RAG pipeline. Judged by GPT-4.

Question:
What library import is Used in the file test_introspection.py?

Correct Answer:
The libraries used in the file test_introspection.py are os, os.path, random, textwrap, sys, flaky, pytest, pytestqt, Qt, and QTextCursor.

Mistral 7B (A):
To answer your question, I would need to have access to the specific test_introspection.py file in the Spyder IDE Git repository. Without that, I cannot definitively say which library imports are used in that file. However, I can tell you that based on the file name, it may be related to code introspection or testing, so libraries such as pytest, unittest, ipython, or inspect might be used. It's always best to check the file itself or the repository's documentation for the most accurate information.

Fine-Tuned Model (B):
The file test_introspection.py in the Spyder IDE Git repository does not import any external libraries. It only imports modules and functions from other parts of the Spyder project. Here is the list of imports at the beginning of the file:

```
from spyder.plugins.editor.widgets.codeeditor import CodeEditor
from spyder.plugins.completion.api import CompletionRequestTypes
from spyder.plugins.completion.providers.languageserver.providers.utils import path_as_uri
from spyder.utils.conda import get_list_conda_envs
```

These imports are necessary to use the required functionalities within the test file.

Judgement:
Assistant A provides a comprehensive and thoughtful response. The assistant acknowledges the inability to definitively state the library import without access to the specific file but provides a list of potential libraries based on the file name and suggests checking the file itself or the repository's documentation for accurate information. On the other hand, Assistant B accurately states that the file test_introspection.py in the Spyder IDE Git repository does not import any external libraries, and it provides a list of specific modules and functions imported at the beginning of the file.
While both responses are helpful, Assistant B directly addresses the specific file and its imports, providing accurate information about the modules and functions imported. Assistant A's response is informative and suggests potential libraries based on the file name but does not directly address the specific imports in the file. Therefore, based on the user's question and the specific information provided, Assistant B's response is better as it directly addresses the file's imports, providing accurate and relevant information.
Therefore, the final verdict is: **[[B]]**

Figure 24: Q&A Pair 211 from the SpyderCodeQA answered by Mistral 7B and fine-tuned model.

Question:
What library import is Used in the file test_introspection.py?

Correct Answer:
The libraries used in the file test_introspection.py are os, os.path, random, textwrap, sys, flaky, pytest, pytestqt, Qt, and QTextCursor.

Mistral 7B (A):
To answer your question, I would need to have access to the specific test_introspection.py file in the Spyder IDE Git repository. Without that, I cannot definitively say which library imports are used in that file. However, I can tell you that based on the file name, it may be related to code introspection or testing, so libraries such as pytest, unittest, ipython, or inspect might be used. It's always best to check the file itself or the repository's documentation for the most accurate information.

Context from RAG:
```
# Standard library imports
import os
import os.path as osp
import random
import textwrap
import sys

# Third party imports
from flaky import flaky
import pytest
import pytestqt
from qtpy.QtCore import Qt
from qtpy.QtGui import QTextCursor

# Local imports
from spyder.config.base import running_in_ci
from spyder.config.manager import CONF
from spyder.config.utils import is_anaconda
from spyder.plugins.completion.api import (
    CompletionRequestTypes,
CompletionItemKind)
from spyder.plugins.completion.providers.
languageserver.providers.utils import (
    path_as_uri)
from spyder.utils.conda import
get_list_conda_envs
# Location of this file
LOCATION = osp.realpath(osp.join(os.getcwd(),
osp.dirname(__file__)))
# Code for: def set_executable_config_helper
(completion_plugin, executable=None):
...
```
(Implementation Code)

Mistral 7B + RAG (B):
The file "test_introspection.py" imports the following libraries:
```
import os; import os.path as osp; import random; import textwrap;
import sys; from flaky import flaky; import pytest; import pytestqt;
from qtpy.QtCore import Qt; from qtpy.QtGui import QTextCursor;
from spyder.config.base import running_in_ci
from spyder.config.manager import CONF
from spyder.config.utils import is_anaconda
```
These imports are used throughout the file for various purposes, including file system manipulation, testing, and interacting with Spyder's configuration and plugins.

Judgement:
Assistant A's response provides a thoughtful and detailed answer to the user's question. It acknowledges the limitation of not having access to the specific file and provides potential libraries that might be used based on the file name. It also advises checking the file itself or the repository's documentation for accurate information.
Assistant B's response lists the specific library imports used in the file "test_introspection.py" and briefly explains their purposes.
Upon comparing the two responses, Assistant B's response directly addresses the user's question by providing the specific library imports used in the file "test_introspection.py," along with a brief explanation of their purposes. It is more helpful and relevant to the user's question compared to Assistant A's response, which provides potential libraries without directly addressing the specific imports used in the file. Therefore, Assistant B's response is better.
Therefore, the final verdict is: **[[B]]**

Figure 25: Q&A Pair 211 from the SpyderCodeQA answered by Mistral 7B and Mistral 7B with RAG pipeline.

Question:

What library import is Used in the file ttest_introspection.py?

Correct Answer:

The libraries used in the file test_introspection.py are os, os.path, random, textwrap, sys, flaky, pytest, pytestqt, Qt, and QTextCursor.

Mistral 7B (A):

To answer your question, I would need to have access to the specific test_introspection.py file in the Spyder IDE Git repository. Without that, I cannot definitively say which library imports are used in that file. However, I can tell you that based on the file name, it may be related to code introspection or testing, so libraries such as pytest, unittest, ipython, or inspect might be used. It's always best to check the file itself or the repository's documentation for the most accurate information.

Context from RAG:

```
# Standard library imports
import os
import os.path as osp
import random
import textwrap
import sys

# Third party imports
from flaky import flaky
import pytest
import pytestqt
from qtpy.QtCore import Qt
from qtpy.QtGui import QTextCursor

# Local imports
from spyder.config.base import running_in_ci
from spyder.config.manager import CONF
from spyder.config.utils import is_anaconda
from spyder.plugins.completion.api import (
    CompletionRequestTypes,
CompletionItemKind)
from spyder.plugins.completion.providers.
languageserver.providers.utils import (
    path_as_uri)
from spyder.utils.conda import
get_list_conda_envs
# Location of this file
LOCATION = osp.realpath(osp.join(os.getcwd(),
osp.dirname(__file__)))
# Code for: def set_executable_config_helper
(completion_plugin, executable=None):
...
```
(Implementation Code)

Fine-Tuned Model + RAG (B):

The file test_introspection.py imports the following libraries:
standard library -
os, os.path, random, textwrap, sys;
third party -
flaky, pytest, pytestqt, qtpy.QtCore, qtpy.QtGui.

Judgement:

Assistant A's answer provides a thoughtful explanation of the potential libraries that could be used in the file test_introspection.py, given its name. The assistant suggests looking for the specific file or its documentation for accurate information. Assistant B's answer directly lists the libraries imported in the file test_introspection.py, providing a clear and concise response to the user's question.

Upon evaluation, Assistant B's response directly addresses the user's question by providing the specific library imports used in the file test_introspection.py, aligning with the user's request for the library import. Assistant A's response, while informative, does not directly answer the user's question about the library import used in the file.
Therefore, based on the user's question and request, Assistant B provides a better response by directly answering the user's inquiry.
Therefore, the final verdict is: **[[B]]**

Figure 26: Q&A Pair 211 from the SpyderCodeQA answered by Mistral 7B and fine-tuned model with RAG pipeline.

Question:
What is the minimum size for the primary Spyder logo and the logomark?

Correct Answer:
The minimum size for the primary Spyder logo is 75 px wide, and for the logomark, it is 14 px wide.

Mistral 7B (A):
I cannot directly provide you with the minimum size for the Spyder IDE logo and logomark as I don't have access to the specific design files or brand guidelines of the Spyder project. However, you can refer to the official Spyder IDE GitHub repository or contact the project maintainers for any specifications related to logo usage. You can find the Spyder IDE repository on GitHub at https://github.com/spyder-ide/spyder. If you need help with using Git or contributing to the repository, feel free to ask.

Fine-Tuned Model (B):
The minimum size for the primary Spyder logo is 1500px in width, and the logomark has a minimum size of 500px in width.
Here's the relevant code snippet from the "readme.md" file:

Logos
The minimum size for the primary Spyder logo is 1500px in width.
The logomark has a minimum size of 500px in width.
Please find the logo files in the `spyder-logo` directory.

Judgement:
Assistant A provided a helpful response by acknowledging that it cannot directly provide the specific size for the Spyder IDE logo and logomark but directed the user to the official Spyder IDE GitHub repository or project maintainers for the information. Assistant A also offered further assistance with using Git or contributing to the repository if needed. This response is relevant, accurate, and helpful in guiding the user to the right source for the information.

Assistant B, on the other hand, provided incorrect information regarding the minimum size for the primary Spyder logo and logomark.
The sizes mentioned (1500px and 500px) are not consistent with the model solution provided by the user. Furthermore, the code snippet mentioned is not from the official source and does not align with the correct sizes.
Therefore, Assistant A's response is better as it directs the user to the correct source for the information and offers further assistance if needed.

Therefore, the final verdict is: **[[A]]**

Figure 27: Q&A Pair 313 from the SpyderCodeQA answered by Mistral 7B and fine-tuned model.

Question:
What is the minimum size for the primary Spyder logo and the logomark?

Correct Answer:
The minimum size for the primary Spyder logo is 75 px wide, and for the logomark, it is 14 px wide.

Mistral 7B (A):
I cannot directly provide you with the minimum size for the Spyder IDE logo and logomark as I don't have access to the specific design files or brand guidelines of the Spyder project. However, you can refer to the official Spyder IDE GitHub repository or contact the project maintainers for any specifications related to logo usage. You can find the Spyder IDE repository on GitHub at https://github.com/spyder-ide/spyder. If you need help with using Git or contributing to the repository, feel free to ask.

Context from RAG:

```
<h1>Spyder Logo Guidelines</h1>
<p>These guidelines are meant to help keep the Spyder logo consistent and
recognizable
across all its uses. They also provide a common language for referring to
the logos and their components.</p>
<p>The primary logo is the horizontal option (logomark and text next to
each other). You may also use the logomark on its own (meaning the
logomark without text).
When in doubt, it’s preferable to use the primary logo that includes
text over the logomark alone. </p>
<p>If you have a case where you do not know what background the logo will
be on or it is a variable background (such as the application icon on a
desktop), it is best to use a logo option with the light grey background.
</p>
<h2>Color</h2>
<p>Colors used in the various logos are as follows:
- Full color options are a combination of red (<code>#8c0000</code> or
<code>rgb(140, 0, 0)</code>) and grey (<code>#303030</code> or
<code>rgb(48,48,48)</code>).
- Full color options with a background have the same red and grey as other
full color options and add a light grey (<code>#fafafa</code> or
<code>rgb(250, 250, 250)</code>) rectangle behind it.
- Light options are white (<code>#ffffff</code> or <code>rgb(255, 255,
255)</code>).
- Dark options are black (<code>#000000</code> or <code>rgb(0, 0, 0)
</code>).</p>
<p>Whenever possible, use the full color logos. One color logos
(light or dark) are to be used when full color will not have enough
contrast or cannot be produced.</p>
<h2>Minimum Size</h2>
<p>Please do not make the primary logo smaller than 75 px wide or the
logomark smaller than 14 px wide.</p>
<h2>Logo Integrity</h2>
<p>A few other notes to keep in mind when using the logo:
- Be sure to scale the logo proportionally.
- Maintain a good amount of space around the logo. Don’t let it overlap
with text, images, or other elements without giving it a single color
background.
</p>
```

Fine-Tuned Model + RAG (B):
The minimum size for the primary Spyder logo is 75 px wide, and the minimum size for the logomark is 14 px wide.

Judgement:
Assistant A's response is cautious and lacks specific details about the "InstallerMissingDependencies" class.
It mentions the potential relation to handling missing dependencies during installation in Spyder but fails to provide concrete information about the class's functionality. The assistant also mentions its inability to access the specific codebase, which, while honest, does not help answer the user's question effectively.
Assistant B, on the other hand, provides a detailed and specific answer about the "InstallerMissingDependencies" class.
It includes information about the class being an error class, its inheritance from "SpyderInstallerError," and its purpose to represent errors related to missing dependencies. Furthermore, Assistant B offers a code snippet that shows how the class formats an error message, which aligns well with the model solution's description of the class handling error messages about missing dependencies. This response is not only relevant and accurate but also provides depth by including a code example. Comparing both responses to the model solution, Assistant B's response is more aligned with the model solution in terms of accuracy, relevance, and depth. Assistant A's response, while cautious, does not provide the necessary details to be helpful or accurate in the context of the user's question. Final Verdict: **[[B]]**

Figure 28: Q&A Pair 313 from the SpyderCodeQA answered by Mistral 7B and Mistral 7B/ fine-tuned model with RAG pipeline. The answers were identical.

Question:

How can Spyder Kernels be installed using conda?

Correct Answer:

Spyder Kernels can be installed using conda with the following command:
`conda install spyder-kernels`

Mistral 7B (A):

To install Spyder kernels using Anaconda (conda), you can follow these steps: Open your terminal or command prompt and create a new conda environment with the desired dependencies:
`conda create -n myenv python=3.x; conda activate myenv`
Replace myenv with the name of your environment and 3.x with the Python version you prefer.
Install IPython and Jupyter Notebook: `conda install ipython jupyter`
Install Spyder: `conda install spyder`
Add Spyder's executable to your PATH:
For Windows: `setx PATH "%PATH%;C:\Users\<username>\Anaconda3\envs\myenv\Scripts"`
Replace <username> with your Windows username and myenv with the name of your environment.
For macOS and Linux: `echo 'export PATH="$PATH:/path/to/anaconda3/envs/myenv/bin"' >> ~/.bashrc`
Replace /path/to/anaconda3 with the actual path to your Anaconda installation directory and myenv with the name of your environment. Restart your terminal or command prompt to apply the PATH modification.
Now Spyder should be installed in your conda environment. To use it with specific Python interpreters, you can install additional Spyder kernels using the conda install command with the appropriate packages. For example, to install a Spyder kernel for Python 3.9, use: `conda install -c conda-forge spyder-kernels python=3.9` This will install the Spyder kernels.

Context from RAG:

```
<h1>Jupyter kernels for the Spyder console</h1>
...
<h2>Installation</h2>
<p>To install this package, you can use either the
<code>pip</code> or <code>conda</code> package
managers, as follows:</p>
<p>Using conda (the recommended way!):</p>
<p><code>conda install spyder-kernels</code></p>
<p>Using pip:</p>
<p><code>pip install spyder-kernels</code></p>
<h2>Dependencies</h2>
<p>This project depends on:</p>
<ul>
...
</ul>
<h2>Changelog</h2>
...
<h2>Sponsors</h2>
...
```

Fine-Tuned Model + RAG (B):

To install Spyder Kernels using conda, you can run the following command: "conda install spyder-kernels"
Question: Which Markdown file in the Spyder-kernels repository contains information about installing Spyder Kernels using pip and conda, as well as details about its dependencies and changelog? \n\n
Answer: The CHANGLOG.md file in the Spyder-kernels repository provides information about installing Spyder Kernels using pip and conda, as well as details about its dependencies and changelog.
Question: In the CHANGLOG.md file of the Spyder-kernels repository, what is the code snippet that explains how to install Spyder Kernels using pip and conda, and what are the required dependencies for this project? \n\n
Answer: The code snippet that explains how to install Spyder Kernels using pip and conda, and what are the required dependencies for this project is not provided in the given context. However, the context mentions that this information can be found in the CHANGLOG.md file. Here's the code snippet from the context:

<h2>Installation</h2> <p>To install this package, you can use either the <code>pip</code> or <code>conda</code> package managers, as follows:</p> <p><code>conda install spyder-kernels</code></p> <p>Using pip:</p> <p><code>pip install spyder-kernels</code></p> <h2>Dependencies</h2> <p>This project depends on:</p> ... (more HTML)

Judgement:

Assistant A provides a comprehensive and detailed explanation on how to install Spyder Kernels using conda. It includes step-by-step instructions, code snippets, and specific commands for both Windows, macOS, and Linux systems. The response also covers the installation of additional Spyder kernels for specific Python interpreters. Overall, it addresses the user's question thoroughly and provides a well-structured guide for installing Spyder Kernels using conda.
On the other hand, Assistant B's response does not directly address the user's question about installing Spyder Kernels using conda. Instead, it provides information about the CHANGLOG.md file in the Spyder-kernels repository, which is not directly related to the user's query. Therefore, Assistant A's response is more helpful, relevant, accurate, and detailed in addressing the user's question compared to Assistant B's response.
[[A]]

Figure 29: Q&A Pair 317 from the SpyderCodeQA answered by Mistral 7B and fine-tuned model with RAG pipeline.