RepoHyper: Hybrid Retrieval for Repository-level Code Completion

Anonymous ACL submission

Abstract

Code Large Language Models (CodeLLMs) have demonstrated impressive proficiency in code completion tasks. However, they often fall short of fully understanding the extensive context of a project repository, such as the intricacies of relevant files and class hierarchies, which can result in less precise completions. To overcome these limitations, we present REPOHYPER, a multifaceted framework designed to address the complex challenges associated with repository-level code completion. Central to REPOHYPER is the *Repo-level Semantic Graph* (RSG), a novel semantic graph structure that encapsulates the vast context of code repositories. Furthermore, REPOHYPER leverages *Expand and Refine* retrieval method, including a graph expansion and a link prediction algorithm applied to the RSG, enabling the effective retrieval and prioritization of relevant code snippets. Our evaluations show that REPOHYPER markedly outperforms existing techniques in repository-level code completion, showcasing enhanced accuracy across various datasets when compared to several strong baselines. Our implementation is published at. [1]

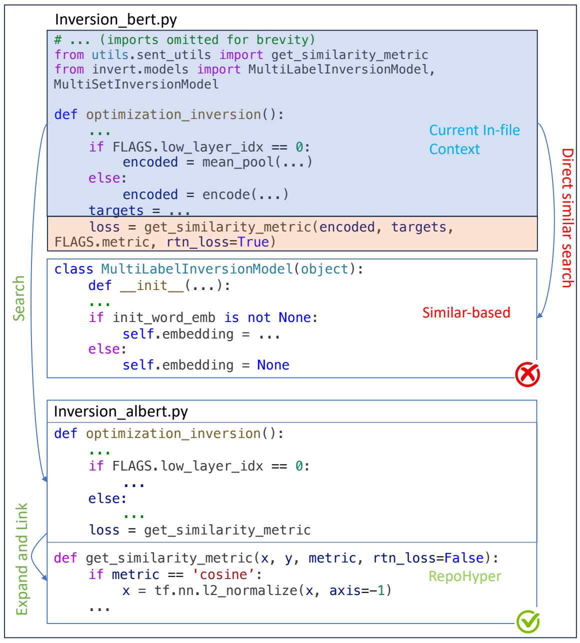

Figure 1: Illustration of graph-based semantic search versus similarity-based search. The orange block indicates the ground-truth line that needs to complete to call the function *get_similarity_metric*. Similarity-based methods mistakenly focus on *MultiLabelInversionModel* class due to its similarity in form with current in-file context, leading to incorrect completions. Conversely, REPOHYPER successfully identifies the correct context via first identify the most similar code snippet in the codebase then expand and link.

1 Introduction

The advent of AI-assisted code completion tools, such as GitHub Copilot, marks a significant milestone in software development. These tools, while adept at interpreting the immediate context of the code being written, often do not fully exploit the broader context available within the entire code repository. This oversight results in suggestions that might not be optimally aligned with the project's architecture or intended functionality, as these tools tend to overlook the rich information embedded in related files, class hierarchies, dependencies, and more.

To overcome these shortcomings, a direct but complex solution involves enhancing the context length of language models by applying efficient attention techniques (Dao et al., 2022; Dao, 2023; Press et al., 2021; Chen et al., 2023). Nonetheless, increasing the context length significantly raises costs and is not feasible indefinitely, especially with respect to the voluminous number of files in a given repository. Hence, there is a crucial need for more refined strategies that accurately identify relevant contexts rather than indiscriminately ana-

[1] https://anonymous.4open.science/r/RepoHyper-3836/README.md

lyzing every file within a repository. In response to this challenge, the concept of repository-level code completion has gained traction. It aims to incorporate the full context of a project, including inter-file relationships, imported modules, and the overarching project structure (Liu et al., 2023b; Liao et al., 2023; Shrivastava et al., 2023a; Agrawal et al., 2023; Shrivastava et al., 2023b). These methodologies generally employ a similarity-based approach to retrieve contexts for completing a given code snippet, drawing either from raw source code or a pre-constructed database with essential metadata. However, this strategy exhibits significant limitations. It often fails to consider that diverse contexts within the current repository, not necessarily involving similar code, can provide valuable insights for code completion. This includes the intricate network of dependencies, shared utility functions, inter-module method calls, class hierarchies, inter-class dependencies, and encapsulation patterns—all of which are fundamental to program semantics.

To address these shortcomings, we introduce REPOHYPER, a novel repository-level code completion approach that considers global, repository-level contexts including not only similarity code but also program-semantic related contextual information within the current repository. Specifically, REPOHYPER incorporates three following components: (1) **Repo-level Semantic Graph (RSG)**, which is a graph-structure designed to encapsulate the core elements of a repository's global context and their dependencies pertaining to code completion, serve as a reliable knowledge source to retrieve accurate contexts instead of raw codebase; (2) **Expand and Refine** retrieval method which consists of two steps: (2.1) **Search-then-Expand Strategies**,which broaden the exploration of contexts by identifying semantically similar ones and then expanding the search to include contexts semantically linked, utilizing the RSG for intelligent navigation and (2.2) **Link Predictor** which is a mechanism that refines the broad set of contexts obtained from the *Search-then-Expand Strategies* and prioritizes them in a smaller, highly relevant subset for code completion. This is accomplished by formulating the re-ranking problem as a link prediction within the RSG, thereby reducing distractions for the LLM.

We conduct comprehensive evaluations of REPOHYPER on both **context retrieval (CR)** and **end-to-end code completion tasks (EECC)** using the RepoBench benchmark (Liu et al., 2023a), demonstrating significant improvements over existing state-of-the-art methods. In CR, REPOHYPER outperforms similarity-based approaches by an average of 49% in retrieval accuracy, utilizing the same encoder (Wang et al., 2023; Guo et al., 2022). For EECC, our method surpasses RepoCoder and other RepoBench baselines, achieving an improvement of +4.1 in Exact Match (EM) and +5.6 in CodeBLEU scores.

To summarize, our main contributions are:

1. We introduce **REPOHYPER**, a novel framework featuring three novel modules designed to address the multifaceted challenges of repository-level end-to-end code completion.

2. We develop the **RSG**, a novel graph representation that captures the global context of a repository, expanding to include non-similar but yet relevant contexts for repo-level code completion. This innovation significantly improves the accuracy and relevance of context retrieval, surpassing conventional methods.

3. We implement an **Expand and Refine** retrieval method via **Search-then-Expand Strategies** and **Link Prediction algorithm** within the RSG, optimizing the retrieval of the most relevant and program-semantic related contexts.

4. We perform extensive evaluation of REPOHYPER in both repository-level code retrieval and code completion tasks demonstrates a significant improvement over the state-of-the-art approaches. Through a series of analytical and ablation studies, we confirm the vital role of each component of REPOHYPER.

2 Related Work

2.1 Code LLMs for code generation & understanding

Recent research has introduced a plethora of Large Language Models (LLMs) tailored for code-related tasks (Bui et al., 2023; Chowdhery et al., 2023; Chen et al., 2021a; Austin et al., 2021; Hendrycks et al., 2021; Nijkamp et al., 2023b,a; Zheng et al., 2023; Wang et al., 2023; Bui and Jiang, 2018; Jayasundara et al., 2019; Bui et al., 2023; Guo et al., 2024; Li et al., 2023; Roziere et al., 2023), aiming to enhance code understanding and generation. These models are categorized into closed-source and open-source variants. Initially, closed-source models like Codex (Chen et al., 2021a), Code-Davinci (Chen et al., 2021a), and PaLM-Coder (Chowdhery et al., 2023) demonstrated exceptional performance on well-known code completion benchmarks, including HumanEval (Chen et al., 2021a), MBPP (Austin et al., 2021), and APPS (Hendrycks et al., 2021). Subsequently, the emergence of open-source models such as the CodeGen series (Nijkamp et al., 2023b,a), CodeT5 (Wang et al., 2021a), CodeT5+ (Wang et al., 2023), CodeGeeX (Zheng et al., 2023), StarCoder (Li et al., 2023), Wizard Coder (Luo et al., 2023), CodeLlama (Roziere et al., 2023), and DeepSeek-Coder (Guo et al., 2024) began to rival the closed-source models in terms of benchmark performance. Despite their purported efficacy across a broad spectrum of code intelligence tasks, code generation and completion emerge as their most notable and widely utilized applications.

2.2 Repository-level Code Completion

Repository-level code completion has seen notable advancements through works like RLPG (Wang et al., 2021b), CoCoMIC (Ding et al., 2022), RepoCoder (Liu et al., 2023b), CodePlan (Bairi et al., 2023) and A3-Codegen (Liao et al., 2023). These studies highlight the critical role of leveraging both within-file and cross-file contexts to improve code completion accuracy. Conversely, RepoFusion (Shrivastava et al., 2023a), RepoPrompts (Shrivastava et al., 2023b), and MGD (Agrawal et al., 2023) propose methodologies for effectively integrating these contexts, assuming their availability from external sources. RepoBench (Liu et al., 2023a) and CrossCodeEval (Ding et al., 2023) emphasize the need for end-to-end benchmarks designed to evaluate code completion systems within the broader, repository-level contexts. Furthermore, CodeAgent (Zhang et al., 2024) incorporates documentation, contexts, runtime environments, and a pipeline for interacting with repositories through multi-agent systems.

3 Methodology

Figure 2 illustrates the overall architecture of our approach, REPOHYPER. Given an existing, incomplete code snippet Q, we first encode it into a semantic vector using an encoder function. Our objective is to retrieve relevant semantic contexts T from the repository R. Consequently, these retrieved contexts are subsequently utilized by a Large Language Model (LLM) to generate the final code prediction:

$$C = \text{LLM}(Q, T).$$

We present a *Repo-level Semantic Graph* (RSG) for global context representation (Section 3.1) and a *Expand and Refine* retrieval algorithm to re-rank and select relevant snippets from RSG. This includes two key steps: **Search-then-Expand** which tries to find the most similar and program-semantic related contexts, and **Re-ranking as Link Prediction** which aims to refine the contexts set found by the prior step. (Section 3.2 and Section 3.3).

3.1 Repo-level Semantic Graph (RSG): Representation for Global Contexts

We denotes Repo-level Semantic Graph (RSG) as $\mathcal{G} = (V, E)$, where V denotes a set of nodes and E denotes a set of relations, aims to capture the fundamental units of a project's global context and the intricate relationships among them. We consider (1) *function/method* and (2) *class* to be fundamental units due to their crucial role in program structure. Each node contains the name, parameters, and body of the corresponding function/method. This allows for precise context access and clear separation of function calls and class-method relationships, which is crucial for repository-level code completion (Shrivastava et al., 2022). This also ensures precise context segmentation, eliminating the need for manual chunking and size tuning. After extracting functions and classes, the remaining file content, such as import statements and non-functional code, is encapsulated in a (3) *Script* node.

The nodes in an RSG are interconnected based on their types and relations, which are categorized as follows: (1) *Import Relations*: These relations (Imports and Imported By) exist between script nodes and the imported modules identified from the script's import statements. This excludes external modules not within the project's scope; (2) *Invoke Relations*: These relations (Caller and Callee) exist between functions (or methods) when one node invokes another; (3) *Ownership Relations*: These relations (Owns and Owned By) exist between methods and the classes that contain them; (4) *Encapsulate Relations*: These relations (Enclose and Enclosed By) exist between script nodes and other nodes that have code snippets contained within the file represented by the script node; and (5) *Class Hierarchy Relations*: These relations (Inherits and Inherited By) exist between classes.

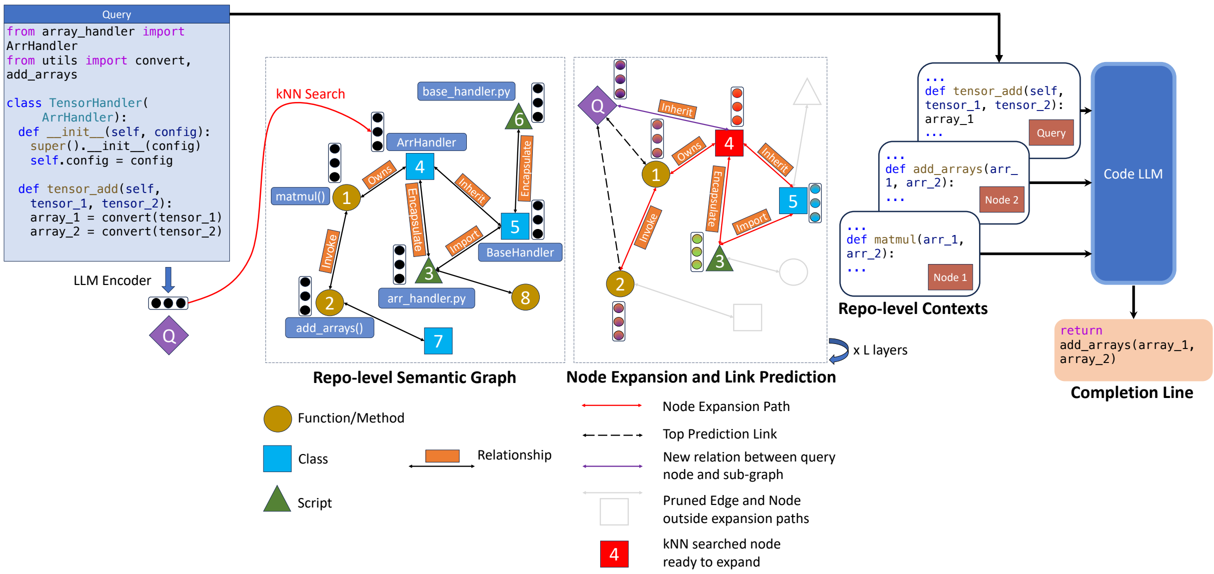

Figure 2: Overall Architecture of RepoHyper. Here we use $K = 1$.

While our implementation is for Python, RSG can be adapted for using in other high-level programming languages, e.g., Java or C++. This adaptation involves utilizing different program elements as context units, and their relations in an RSG.

3.2 Search-then-Expand Strategies

Our methodology employs a search-then-expand approach to identify the most suitable file for the decoding process within a repository context. We aim to broaden the search space beyond similarity-based candidates to encompass more relevant files, as suggested by Shrivastava et al. (2022); Liu et al. (2023b,a). These studies indicate that files with similar imports, names, or code snippets are typically the correct contexts for retrieval, and that semantic search using k-Nearest Neighbor (kNN) with encoders like UniXCoder (Guo et al., 2022) or CodeT5+ (Wang et al., 2023) is effective. Additionally, they highlight the importance of structured context sources such as *Sibling* files or *Import of Parent Class* in providing relevant contexts.

Based on these insights, REPOHYPER initially performs a kNN search with a small K to identify a set of anchor nodes in the RSG. These nodes are then expanded using strategy $\mathcal{F}$:

$$A_{\text{exp}} = \mathcal{F}(A), \quad A = \{V_i | i \in \text{kNN}(\mathcal{G}, Z_Q)\}$$

Here, $\text{kNN}(\mathcal{G}, Z_Q)$ represents the kNN search for the K nodes most similar to the query vector Z_Q in the graph $\mathcal{G}$. We experiment different strategies to find the most optimal nodes in the graph for decoding process. Our two proposed strategies are:

(1) *Exhausted Search*: Beginning from a node $V_j \in A$, we utilize a straightforward Breadth First Search algorithm (BFS) with a maximum depth of D. Theoretically, D should reach 4 to encompass the complete relationship between two contexts for repo-level code completion. However, in practice, setting $D \geq 3$ may result in the BFS covering nearly 50% of the graph. Hence, we introduce another parameter alongside D to constrain the number of BFS expanded nodes: the maximum number of nodes per BFS denoted as M.

(2) *Pattern Search*: We found that exhausted search on all the directions and paths can include too many irrelevant contexts to the query. Thus, we conduct kNN search for all the queries in the training set, then expand using exhausted strategy from kNN searched node V_j to target node V_{target}, then collect the most frequent type paths into path set $\mathcal{P}$, then add them as filters for later BFS. This is called as pattern search since it will eliminates non-frequent paths during exhausted exploration saving the walking nodes.

$$A_{\text{exp}} = \{V_i | V_i \in \mathcal{F}_{\text{exh}}(A), \text{PATH}(V_j, V_i) \in \mathcal{P}\}$$

where PATH denotes the type of BFS exploration path from kNN searched node $V_j \in A$ to any walking node V_i. For example, one might prioritize the exploration from a class node to its script node then to imported function to draw a possible invoke relationship for code completion (class-script-function path type), but one is unlikely to explore chain of method calls (method-method-method path type). More details are in Appendix A.3.

3.3 Re-ranking as Link Prediction

To manage the increased number of contexts from our Search-then-Expand strategy and reduce noise, we refine the context selection process for the decoder. Instead of using all retrieved contexts, we only consider the top-N_2 (where $N_2 < N_1$) most relevant ones. Initially, we tried ranking contexts by their embedding distances to the query, but this approach underperformed in our evaluation.

To improve relevance ranking, we treat it as a link prediction problem on a RSG. We embed the query as a node in the RSG and use a message passing network, along with a link prediction head, to score the connections between the query and other nodes in the graph. The final embeddings after going to message passing network f are used to calculate linking scores, which determine the relevance of each context to the query.

$$Z_i^{(L)} = f(Z_i^{(0)}, \mathcal{G} \oplus \mathcal{Q})$$

$Z_i^{(0)}$ is initial embedding taken from encoder for i^{th} node in graph, $Z_i^{(L)}$ is the last layer embedding after going to message passing network f. $\mathcal{G}_1 = \mathcal{G} \oplus \mathcal{Q}$ is concatenation of query $\mathcal{Q}$ node has query Q as the raw source code of the node and Z_Q as initial embedding to $\mathcal{G}$, by adding this into set of nodes V and add new relations to the query. For example, if the current code already have invoke relations with some function nodes in graph, we add these edges to the current node avoiding duplicate prediction.

$$s_i = W^T \text{concat}(Z_i^{(L)}, Z_Q^{(L)}) \;\; \forall i \in \{i | V_i \in A_{\text{exp}}\} \quad (1)$$

, where W is a trainable model parameter, s_i is the linking score between query node and all other nodes inside the RSG. In practice, we focus on nodes that are imported into the file being predicted, as suggested in RepoBench (Liu et al., 2023a). The training loss of node link prediction for each query is computed as

$$\mathcal{L} = -\frac{1}{N_1} \sum_{i=1}^{N_1} y_i \log \hat{y}_i \text{ where } \hat{y}_i = \frac{1}{1 + e^{-s_i}} \quad (2)$$

More details about incorporating query node into inference process can be found in Appendix A.6 In all experiments, we employ GraphSAGE (Hamilton et al., 2017) with L layers as the graph neural network (GNN) model to update the representation for each node based on the passage graph. The l-th layer of the GNN model updates the embedding of node i as follows:

$$Z_i^{(l)} = h\left(Z_i^{(l-1)}, \left\{Z_j^{(l-1)}\right\}_{(i,j)\in\mathcal{G}_1}\right) \quad (3)$$

where h is usually a non-linear learnable function which aggregates the embeddings of the node itself and its neighbor nodes. After re-ranking with linking scores, the final top-N_2 ($N_2 < N_1$) contexts are sent for decoding. Suppose their indices are $\{g_1, g_2, \cdots, g_{N_2}\}$, the decoding process for final prediction $\mathbf{C}$ is:

$$\mathbf{C} = \text{LLM}\left(Q, \left[\mathbf{P}_{g_1}; \mathbf{P}_{g_2}; \cdots; \mathbf{P}_{g_{N_2}}\right]\right) \quad (4)$$

where $\mathbf{P}_i$ is code representation of node $V_i \in V$

To train such a network to re-rank inside a repository, we solely optimize loss function defined in (1) on dataset $R = \left\{\left(\mathcal{G}^i, y^i_{\text{optimal}}, Q^i\right)\right\}$ with $\mathcal{G}^i$ is a RGS for a i^{th} repository and y^i_{optimal} is the optimal context node corresponding to query Q^i.

4 Empirical Evaluation

4.1 Tasks & Datasets

We choose RepoBench (Liu et al., 2023a)as the main dataset for our evaluation pipeline due to its large scale and comprehensive, making it an ideal candidate for assessing repository-level code completion from multiple perspectives. RepoBench consists of three distinct subsets, each designed to evaluate different aspects of repo-level code completion. Each subset contains up to 12000 samples for evaluation [2].

1. **RepoBench-R** focuses on evaluating the retrieval of relevant code snippets, crucial for accurate code prediction. It assesses the model's ability to sift through extensive repository data to identify useful snippets for code completion, termed as the **Context Retrieval** task.

2. **RepoBench-C** is designed to predict the next line of code using provided in-file and cross-file contexts, testing a model's ability to predict precise code completion from available contexts.

3. **RepoBench-P** combines the challenges of RepoBench-R and C, testing a model's pipeline from snippet retrieval to code prediction, reflecting real-world auto-completion (denoted as the **End-to-End Code Completion** task).

[2] Statistic of RepoBench can be found in Appendix Section.

Retrieval	Model	Easy				Hard					
		XF-F		XF-R		XF-F			XF-R		
		acc@1	acc@3	acc@1	acc@3	acc@1	acc@3	acc@5	acc@1	acc@3	acc@5
Random		15.68	47.01	15.61	46.87	6.44	19.28	32.09	6.42	19.36	32.19
Lexical	Jaccard	20.82	53.27	24.28	54.72	10.01	25.88	39.88	11.38	26.02	40.28
	Edit	17.91	50.61	20.25	51.73	7.68	21.62	36.14	8.13	22.08	37.18
Similarity-based	CodeBERT	16.47	48.23	17.87	48.35	6.56	19.97	33.34	7.03	19.73	32.47
	UniXcoder	25.94	59.69	29.40	61.88	17.70	39.02	53.54	20.05	41.02	54.92
	CodeT5+	18.29	53.31	19.61	53.05	9.51	25.24	37.81	13.21	28.15	38.76
	OpenAI	28.14	64.24	31.15	65.29	18.23	42.01	60.39	23.78	45.67	58.53
RepoHyper	UniXcoder	**32.56**	**68.91**	**33.79**	**69.67**	**25.81**	**49.51**	**61.19**	**27.12**	**51.12**	**63.23**
	CodeT5+	31.15	67.23	32.01	68.12	25.16	46.70	60.19	23.81	43.61	57.12

Table 1: Results of RepoHyper on RepoBench-R dataset. The studied encoder models include `codebert-base` for CodeBERT, `unixcoder-base` for UniXcoder, `codet5p` for using CodeT5+ 770M parameters. RepoHyper has UniXcoder and CodeT5+ 770M as the base encoders, respectively. Numbers are shown in percentage (%), with the best performance highlighted in bold. `text-embedding-large-3` for OpenAI.

Our work focuses on the creation of repository-level code graphs and retrieval strategies, primarily utilizes RepoBench-R and RepoBench-P. These subsets align with our goals to enhance context retrieval and code completion accuracy, directly showcasing our contributions.

Within these benchmarks, there are two settings (Liu et al., 2023b) to thoroughly assess performance: **Cross-File-First** (XF-F) challenges a model to predict the first occurrence of a cross-file line, requiring adept handling of long-range contexts; **Cross-File-Random** (XF-R) masks a random cross-file line where prior usage might offer clues for prediction. These settings enable a robust evaluation on these code completion scenarios.

4.2 Baselines & Metrics

4.2.1 Context Retrieval on RepoBench-R

We follow Liu et al. (2023a) to define 4 baselines: (1) **Random Retrieval**, where code snippets are chosen randomly, providing a basic comparison level. This process is repeated 100 times to average the results for consistency. (2) **Lexical Retrieval**, which uses simple text comparison techniques like Jaccard Similarity and Edit Distance to find relevant snippets based on the code tokens. (3) **Similarity-based Retrieval**, employing encoder models like CodeBERT, UnixCoder and OpenAI Text Embedding Models to generate code embeddings and Cosine Similarity to measure the semantic similarity between the cropped code and the candidate snippets. In RepoHyper setting, we employ our pipeline, which leverages different encoder models (UniXCoder (Guo et al., 2022), CodeT5+-770M (Wang et al., 2023)) to encode the query, construct semantic graph, node expansion, and link prediction to retrieve the contexts.

Evaluation Metrics We also follow (Liu et al., 2023a) use to Accuracy@k (acc@k) metric to evaluate performance on this task. For the easy subset of tasks, we assess performance using acc@1 and acc@3, while for the more challenging subset, we evaluate using acc@1, acc@3, and acc@5.

4.2.2 Code Completion on RepoBench-P

This end-to-end code completion task requires two components: Context Retrieval and Code Completion. Given our research's emphasis on developing novel context retrieval strategies, we examine how these strategies perform under various settings, using consistent code completion models for comparison. Specifically, we utilized GPT-3.5-Turbo and DeepSeek-Coder-33B (Guo et al., 2024) as our the LLM for code completion in our workflow.

We follow (Liu et al., 2023a) to define baselines according to different settings on the contexts: (1) **Gold-Only**, which uses only the 'gold snippet' for cross-file completions and leaves in-file completion contexts empty, testing a model's efficacy with minimal context. (2) **In-File-Only**, which uses maximum 30 lines up from prediction line in same file, indicates lower-bound performance without repo-level contexts. (3) **RepoBench**, which uses similarity-based search, UniXCoder, to retrieve top contexts, then prompts LLM for code completion.

We also assess snippet ranking strategies, **H2L** (High-to-Low) and **L2H** (Low-to-High), to see how the order of context relevance affects code completion performance. We also include **RepoCoder** (Liu et al., 2023b), which is a method on the repo-level code completion task as another baseline. This method is run iteratively with Jaccard retrieval method and maximum iterations of 4.

Evaluation Metrics We use Exact Match (EM) and CodeBLEU (Ren et al., 2020) to measure next-line completion accuracy as in (Liu et al., 2023a).

4.3 Implementation Details

To construct the Repo-level Semantic Graph (RSG) for a repository, we first parse functions, methods, and classes using *tree-sitter*[3], a tool for generating abstract syntax trees (AST). We then extract code entities from the AST and integrate them into a semantic graph[4]. In our methodology, we adopt *Pattern Search* for expansion with parameters set to a maximum depth $D = 4$, $M = 1000$, and $K = 3$. The number of prompt contexts for LLM completion, N_2, is dynamically selected to maximize token count within LLMs' context length limits. The Link Predictor is trained on training subset of RepoBench-R's gold context labels, with a text matching algorithm based on Jaccard distance used to align labels with RSG nodes. We use a homogeneous GraphSAGE network f with $L = 3$ GNN layers, optimized with Adam at a learning rate of 0.01 for 10 epochs, noting homogeneous networks performed adequately for RSG. Link Predictor is trained on 2 A100 GPUs in 6 hours.

5 Evaluation Results

5.1 Performance on Context Retrieval

The results presented in Table 1 demonstrate the enhanced performance of RepoHyper when compared to similarity-based approaches . By implementing graph-based semantic search strategies, RepoHyper significantly outperforms the baseline methods, including those utilizing CodeBERT and UniXCoder, as well as our own tests with CodeT5+. Specifically, it achieves high improvements, with UniXcoder and CodeT5+ showing relative increases of nearly 26% and 72%, respectively, across various subsets and task scenarios.

5.2 Performance on Code Completion

Table 2 shows that RepoHyper emerges as the most effective strategy, consistently achieving the highest scores across both EM (Exact Match) and CodeBLEU metrics in both XF-F and XF-R settings, with notable scores such as 52.76% EM and 61.49% CodeBLEU with gpt-3.5-turbo, and 49.49% EM and 58.48% CodeBLEU with DeepSeek-Coder. Compared to the baseline strategies like Gold-Only, RepoBench-L2H/H2L, and RepoCoder, RepoHyper's strategies (both L2H and H2L) demonstrate superior performance. The comparison between L2H (Low-to-High relevance) and H2L (High-to-Low relevance) within RepoHyper indicates that prioritizing snippets from low to high relevance (L2H) offers a significant advantage over the reverse, particularly highlighting the high performance of RepoHyper-L2H strategy.

	CR Strategy	**XF-F**		**XF-R**	
		EM	CodeBLEU	EM	CodeBLEU
Gpt-3.5-turbo	In-File-Only*	26.35	33.14	36.31	44.01
	Gold-Only*	30.59	38.37	40.65	49.12
	RepoBench-L2H	37.51	46.19	49.3	57.20
	RepoBench-H2L	39.89	48.01	51.21	59.44
	RepoCoder	48.73	57.62	59.55	67.42
	RepoHyper-L2H	**52.76**	**61.49**	**64.06**	**71.59**
	RepoHyper-H2L	48.8	57.21	59.81	66.85
DeepSeek-Coder	In-File-Only*	25.46	32.39	34.23	42.13
	Gold-Only*	27.29	35.15	37.10	46.08
	RepoBench-H2L	34.02	43.08	45.96	53.62
	RepoBench-L2H	36.55	44.53	47.71	55.81
	RepoCoder	45.47	54.32	56.24	64.23
	RepoHyper-L2H	**49.49**	**58.48**	**59.03**	**66.98**
	RepoHyper-H2L	45.17	53.71	56.56	63.41

Table 2: Comparison of various context retrieval strategies (CR Strategy) on the end-to-end code completion task on RepoBench-P for Python using GPT-3.5-turbo-16k and DeepSeek-Coder-33B.

6 Ablation Study

6.1 Hyper-parameters Sensitivity and Effects

We evaluated various graph-based semantic search strategies, including Exhausted, Pattern, and the baseline kNN search, each with different hyper-parameters. Table 3 reveals that while Exhausted Search achieves high hit rates, Pattern Search offers a more efficient solution, reaching up to 73% hit rates by exploring only 28% of the nodes, compared to 36.7% needed by Exhausted Search for similar outcomes. This efficiency highlights Pattern Search's ability to identify relevant sub-graphs more efficiently, enhancing precision with less computational resources. Moreover, both Exhausted and Pattern Search strategies significantly outperform the kNN baseline in hit rates, while maintain-

[3] https://github.com/tree-sitter/tree-sitter
[4] See Appendix A.1 for RSG construction details.

Search Algorithm	Parameter Combination	Hit/Coverage	
		Hits	Coverage
Exhausted Search	D=4, M=1000, K=3	80%	40%
	D=4, M=200, K=4	78%	44%
	D=4, M=10000, K=1	72%	40.4%
	D=2, M=10000, K=1	34%	10%
	D=2, M=10000, K=2	47%	16%
	D=2, M=10000, K=8	73%	36%
Pattern Search	D=4, M=1000, K=3	73%	28%
	D=4, M=200, K=4	70%	29%
	D=4, M=10000, K=1	65%	31%
	D=2, M=10000, K=1	62%	14%
	D=2, M=10000, K=2	51%	8%
	D=2, M=10000, K=8	68%	21%
kNN	D=M=1, K=0.35*$\|\mathcal{G}\|$	53%	35%

Table 3: Sensitivity Analysis. D : maximum depth; M : maximum number of nodes; K : in kNN algorithm. During expansion, the search algorithm will expand K nodes to Coverage % of the number nodes of the repolevel semantic graph, then extract explored node into a sub-graph. There's a Hits % probability that there's an optimal context in this sub-graph. In kNN baseline, we use K=35% of size of the original graph.

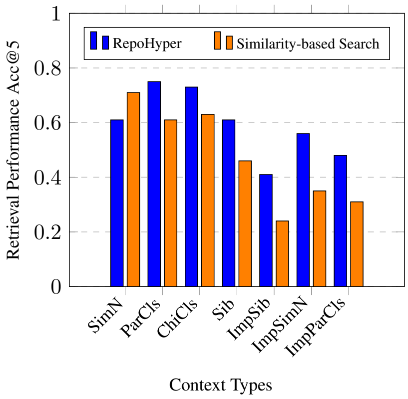

Figure 3: Retrieval performance comparison between RepoHyper and Similarity-based Semantic Search across different context types. We use kNN search within our RSG with UniXCoder encoder for encoding, this method is denoted as Similarity-based Semantic Search and RepoHyper with same encoder. Please see Appendix A.4 for more details on Context Types.

Models	Easy	Hard
(1) kNN	60.15	40.74
(2) w/ Exhausted Search	62.35	42.88
(3) w/ Pattern Search	64.23	44.50
(4) w/ E.S+Link Predictor	68.10	47.15
(5) w/ P.S+Link Predictor	69.12	47.83
(6) w/ P.S+Re-ranking	67.43	44.48

Table 4: Ablation study in the code retrieval task, we use Repobench-R testset with two *Easy* and *Hard* subsets. Acc@3 is used as the main metric. Exhausted Search is denoted as E.S, Pattern Search is denoted as P.S. For a combination with kNN and expansion strategy, we simply re-rank inside extracted sub-graph using cosine similarity between query and nodes in sub-graph.

ing smaller, more focused sub-graphs for quicker inference and reduced noise. These results emphasize the importance of choosing the right search strategy and hyperparameter tuning to balance search thoroughness and prediction efficiency.

6.2 Retrieved Nodes Analysis

RepoHyper aims to retrieve both semantically similar code contexts and program-semantic related contexts like *Sibling* (Sib) or *Import of Parent Class* (ImpParCls). In this study, we validated RepoHyper'retrieval capability across various context types identified in RepoBench-R's *Hard* subset, adhering to a classification scheme from prior research. We excluded rare context types due to their minimal representation in the dataset and focused our experiment on 100 instances for each context type under the XF-R setting, using the acc@5 metric for evaluation.

Results in Figure 3 show that our model excels in retrieving contexts tied to global program semantics, e.g., Class Hierarchy, underscoring its effectiveness. However, it underperforms in retrieving contexts with similar names, likely due to the limited number of anchors used during the kNN search phase. Adjusting the number of anchors might improve retrieval of similar snippets but could impact other context types' retrieval efficiency.

6.3 Ablation Study

Several meaningful observations can be drawn from Table 4: without Link Predictor, Pattern Search offers better accuracy than Exhausted Search. This is likely due to the fact that Exhausted Search has to include more nodes to obtain similar hits rate making the later re-ranking more difficult because of the noisy context nodes.

7 Conclusion

In this paper, we introduced RepoHyper, a novel framework aimed at enhancing repository-level code completion by addressing its complex challenges. RepoHyper advances this domain through three key components: the Repo-level Semantic Graph (RSG), Search-then-Expand Strategies, and a Link Predictor, collectively improving the accuracy and relevance of code suggestions. With extensive evaluation, RepoHyper show superiority in Repo-level Code Completion.

8 Limitations

This section outlines the limitations of our study, which we hope will serve as a catalyst for further research in this field:

Firstly, our Pattern Expansion strategy within the Repobench-R training set involves collecting the most frequent path types from kNN searched nodes to the nearest target nodes, which are then incorporated into the path type set $\mathcal{P}$ and used as filters for BFS. During this process, we manually select path types for exploration, which may not be optimal and could vary across different programming languages. A potential solution to this issue is the design of a learnable algorithm for navigating in RSG, as suggested by (Moon et al., 2019).

Secondly, our experiments on repository-level tasks were conducted using only the RepoBench dataset. Although, RepoBench is a well-designed benchmark with a sufficiently large sample size to statistically validate the effectiveness of our approach, the generalizability of our findings would be strengthened by performing detailed analyses on additional repository-level code completion benchmarks, such as those presented in (Ding et al., 2023; Liu et al., 2023b).

Lastly, our experiments rely on both public and proprietary large-scale CodeLLMs, which necessitate significant computational resources and contribute to carbon emissions, as highlighted by (Patterson et al., 2021). Moreover, the predictions generated by these models may not always align with user intentions, a concern that is further discussed in (Chen et al., 2021b). Addressing these issues is crucial for developing more environmentally sustainable and user-aligned CodeLLMs in the future.

References

Lakshya A Agrawal, Aditya Kanade, Navin Goyal, Shuvendu K Lahiri, and Sriram K Rajamani. 2023. Guiding language models of code with global context using monitors. *arXiv preprint arXiv:2306.10763*.

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

Ramakrishna Bairi, Atharv Sonwane, Aditya Kanade, Arun Iyer, Suresh Parthasarathy, Sriram Rajamani, B Ashok, Shashank Shet, et al. 2023. Codeplan: Repository-level coding using llms and planning. *arXiv preprint arXiv:2309.12499*.

Nghi DQ Bui and Lingxiao Jiang. 2018. Hierarchical learning of cross-language mappings through distributed vector representations for code. In *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results*, pages 33–36.

Nghi DQ Bui, Hung Le, Yue Wang, Junnan Li, Akhilesh Deepak Gotmare, and Steven CH Hoi. 2023. Codetf: One-stop transformer library for state-of-the-art code llm. *arXiv preprint arXiv:2306.00029*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri 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, Josh 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. 2021a. Evaluating large language models trained on code.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri 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, Josh 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. 2021b. Evaluating large language models trained on code.

Shouyuan Chen, Sherman Wong, Liangjian Chen, and Yuandong Tian. 2023. Extending context window of large language models via positional interpolation. *arXiv preprint arXiv:2306.15595*.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. 2023. Palm: Scaling language modeling with pathways. *Journal of Machine Learning Research*, 24(240):1–113.

Tri Dao. 2023. Flashattention-2: Faster attention with better parallelism and work partitioning. *arXiv preprint arXiv:2307.08691*.

Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems*, 35:16344–16359.

Yangruibo Ding, Zijian Wang, Wasi Uddin Ahmad, Hantian Ding, Ming Tan, Nihal Jain, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, et al. 2023. Crosscodeeval: A diverse and multilingual benchmark for cross-file code completion. *arXiv preprint arXiv:2310.11248*.

Yangruibo Ding, Zijian Wang, Wasi Uddin Ahmad, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. 2022. Cocomic: Code completion by jointly modeling in-file and cross-file context. *arXiv preprint arXiv:2212.10007*.

Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. Unixcoder: Unified cross-modal pre-training for code representation. *arXiv preprint arXiv:2203.03850*.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. 2024. Deepseek-coder: When the large language model meets programming – the rise of code intelligence.

William L. Hamilton, Rex Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. In *NIPS*.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, et al. 2021. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938*.

Vinoj Jayasundara, Nghi Duy Quoc Bui, Lingxiao Jiang, and David Lo. 2019. Treecaps: Tree-structured capsule networks for program source code processing. *arXiv preprint arXiv:1910.12306*.

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, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour 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!

Dianshu Liao, Shidong Pan, Qing Huang, Xiaoxue Ren, Zhenchang Xing, Huan Jin, and Qinying Li. 2023. Context-aware code generation framework for code repositories: Local, global, and third-party library awareness. *arXiv preprint arXiv:2312.05772*.

Tianyang Liu, Canwen Xu, and Julian McAuley. 2023a. Repobench: Benchmarking repository-level code auto-completion systems.

Yang Liu, Yuxuan Wang, Xinyu Zhang, Jian Li, and Xiaodong Liu. 2023b. Repocoder: Repository-level code completion through cross-file context retrieval. *arXiv preprint arXiv:2303.12570*.

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

Seungwhan Moon, Pararth Shah, Anuj Kumar, and Rajen Subba. 2019. OpenDialKG: Explainable conversational reasoning with attention-based walks over knowledge graphs. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 845–854, Florence, Italy. Association for Computational Linguistics.

Erik Nijkamp, Hiroaki Hayashi, Caiming Xiong, Silvio Savarese, and Yingbo Zhou. 2023a. Codegen2: Lessons for training llms on programming and natural languages. *arXiv preprint arXiv:2305.02309*.

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

David Patterson, Joseph Gonzalez, Quoc Le, Chen Liang, Lluis-Miquel Munguia, Daniel Rothchild, David So, Maud Texier, and Jeff Dean. 2021. Carbon emissions and large neural network training.

Ofir Press, Noah A Smith, and Mike Lewis. 2021. Train short, test long: Attention with linear biases enables input length extrapolation. *arXiv preprint arXiv:2108.12409*.

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297*.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Disha Shrivastava, Denis Kocetkov, Harm de Vries, Dzmitry Bahdanau, and Torsten Scholak. 2023a. Repofusion: Training code models to understand your repository.

Disha Shrivastava, Hugo Larochelle, and Daniel Tarlow. 2022. Repository-level prompt generation for large language models of code. In *ICML 2022 Workshop on Knowledge Retrieval and Language Models*.

Disha Shrivastava, Hugo Larochelle, and Daniel Tarlow. 2023b. Repository-level prompt generation for large language models of code. In *International Conference on Machine Learning*, pages 31693–31715. PMLR.

Yue Wang, Hung Le, Akhilesh Gotmare, Nghi Bui, Junnan Li, and Steven Hoi. 2023. CodeT5+: Open code large language models for code understanding and generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 1069–1088, Singapore. Association for Computational Linguistics.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021a. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*.

Yuxuan Wang, Xinyu Zhang, Jian Li, Yang Liu, and Xiaodong Liu. 2021b. Rlpg: A reinforcement learning based code completion system with graph-based context representation. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, page 1119–1129, New York, NY, USA. Association for Computing Machinery.

Kechi Zhang, Jia Li, Ge Li, Xianjie Shi, and Zhi Jin. 2024. Codeagent: Enhancing code generation with tool-integrated agent systems for real-world repo-level coding challenges. *arXiv preprint arXiv:2401.07339*.

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. 2023. Codegeex: A pre-trained model for code generation with multilingual evaluations on humaneval-x.

A Appendix

A.1 Building RSG

In this section, we present details on how to build nodes and relations of Repo-level Semantic Graph. Firstly, we use Tree-sitter [5] to parse functions, methods, classes out of Python files. After parsing these entities, we removed these entities from each file so the remaining codes in the file is not function or class. This ensures import statements and other statements (like main file) will be remained. In order to create *Import Relations* between script nodes and the imported nodes (functions, classes), we use built-in Abstract Syntax Tree (AST) module of Python to parse import statements, then identify which module in parsed entities is imported into the script node. For *Invoke Relations*, we use PyCG [6] to generate the call graph of the repository. Since, PyCG only works for Python3 repositories, and RepoBench contains a lot of Python2 repositories, we use Automated Python 2 to 3 code translation tool 2to3 [7] to translate these repositories into Python3, then apply PyCG to produce call graph. *Ownership* and *Encapsulate* relationships are straightforward to generate since the Tree-sitter allows us to identify which method belongs to which class exactly and we also parse functions, classes from each file then we also know which function, class belongs to which file exactly. For *Class Hierarchy Relations*, since Python is a language that needs to specify parent class in the implementation of the inherited class, we can use Tree-sitter to parse this parent class in declaration fields of the inherited class, then we can produce a class hierarchy edge between parent and child class.

A.2 Gold Snippet Definition

In the training set of RepoBench-R, for each sample, every snippet parsed from import statements is treated as a potential candidate for next-line prediction, for example, we have definitions of two following imported functions: add and minus as the snippets and the gold snippet is the optimal context for prediction, here which is "add" from src import add, minus nextline: add(1,2).

A.3 Pattern Search

In Repository Semantic Graph (RSG), there are three types of nodes: function (method), class, and script. The edges between these nodes represent five different relationships: import (1), invoke (2), ownership (3), encapsulate (4), and hierarchy (5). A path type is defined by the sequence of edge types along that path in the RSG. For example, let's con-

[5] https://tree-sitter.github.io/tree-sitter/
[6] https://github.com/vitsalis/PyCG
[7] https://docs.python.org/3/library/2to3.html

sider a node V_j identified by the k-nearest neighbor (kNN) search, and a target node V_{target} representing the "gold snippet" (e.g., a method called by V_j). There can be multiple paths from V_j to V_{target}. One such path could be: V_j (method) $\rightarrow$ (owned by) V_1 (class) $\rightarrow$ (encapsulated in) V_2 (script) $\rightarrow$ (import) V_{target} This path has a type of (ownership, encapsulate, import), represented by the edge types (3, 4, 1). However, since there are multiple paths from V_j to V_{target}, including all the nodes produced by these paths might introduce irrelevant contexts into the subgraph. For example, V_j can reach V_{target} through a more useful path like: V_j (method) $\rightarrow$ (owned by) $V_1 \rightarrow$ (owns) V_3 (method) $\rightarrow$ (call) V_{target} This path is more useful because V_3 (method) can provide hints on how to call V_{target}, whereas the previous path included V_2 (script), which is a longer context and less useful for understanding how to use V_{target}. In the training set of RepoBench-R, for each sample, we have a gold snippet and an in-file context. We first use kNN search to find the k most similar context nodes to the in-file context. Then, we perform an exhaustive search from these identified nodes to collect all possible paths leading to the gold snippet. For example, in sample 1, we might collect paths of types (3, 4, 1) and (1, 2, 3), while in sample 2, we might find paths of types (3, 4, 1) and (2, 4). The most frequent path type in the training set is (3, 4, 1), which appears twice in this example.

A.4 Context Types in Retrieved Node Analysis

We follow definitions of different contex types in (Shrivastava et al., 2022) :

- Parent Class (ParCls): code snippet taken from the parent class of the class that is having the target prediction line inside.
- Child Class (ChiCls): code snippet taken from the child class of the class that is having the target prediction line inside.
- Sibling (Sib): any code snippet in the files that are in the same directory as the current file (the file contains current target prediction line)
- Similar Name (SimN): take code snippet from files, functions, classes (objects) that have a similar name as the completing function, class or file. Similar names are determined by splitting the file name based on underscore or camel- case formatting and then matching parts of the filename. If one or more parts matches, two objects are considered to have similar names.
- Import Sibling (ImpSib): take code from the import objects (classes, functions) in the sibling files.
- Import Similar Name (ImpSimN): take code snippet from the import files used in the similar name files.
- Import Parent Class (ImpParCls): any code snippet from the import objects used in the parent class files. This implies the case when the child class is likely to re-use imported objects of its parent class.

A.5 RepoBench Benchmark Statistics

A.6 Rerank as Link Prediction

After extracting the subgraph using the k-nearest neighbor (kNN) search and expansion strategy, it is crucial to incorporate the query node, representing the in-file context, into the subgraph. To enable meaningful message passing between the query node and the extracted subgraph nodes, we need to establish edges connecting the query node to the subgraph. Fortuitously, in most cases, the query node (in-file context) already possesses certain connections to other nodes within the subgraph. For instance, the in-file context might invoke some functions that are represented as nodes in the subgraph for the task of next line prediction. By including the query node and its existing connections to the subgraph, the message passing network can effectively leverage the in-file context information during the link prediction process, consequently leading to more accurate and relevant code completion suggestions. This integration of the query node into the subgraph allows for a comprehensive understanding of the code context, facilitating the message passing network to make informed predictions. By considering both the subgraph extracted from the repository and the in-file context, the model can capture the relevant relationships and dependencies, resulting in code completion suggestions that are tailored to the specific code context at hand.

A.7 Samples

In this section, we present some samples in which the similarity-based retrieval method failed at and

Lang.	Task	Subset	XF-F	XF-R	IF	Mean Candidates	Mean Tokens
Python	RepoBench-R	Easy	12,000	6,000	-	6.7	-
		Hard	12,000	6,000	-	17.8	-
	RepoBench-C	2k	12,000	5,000	7,000	-	1,035
		8k	18,000	7,500	10,500	-	3,967
	RepoBench-P		10,867	4,652	6,399	24	44,028

Language	Task	XF-F	XF-R	IF
Python	Code Retrieval	175,199	86,180	-
	Code Completion	349,023	179,137	214,825

Table 5: *(Top)* Test data overview for RepoBench dataset for Python in 3 different tasks; *(Bottom)* Training data for RepoBench for Python

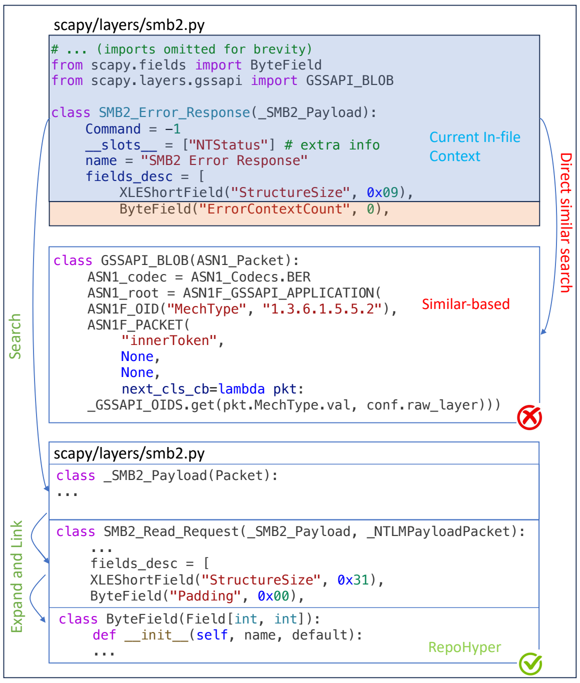

Figure 4: Sample ID 1430 in repository secdev/scapy

why, we use the UniXCoder as the main encoder. These samples are collected in RepoBench-R Cross-File First subset.