FT2Ra: A Fine-Tuning-Inspired Approach to Retrieval-Augmented Code Completion

Qi Guo*
Tianjin University
Tianjin, China
bxguoqi@tju.edu.cn

Xiaohong Li
Tianjin University
Tianjin, China
xiaohongli@tju.edu.cn

Xiaofei Xie
Singapore Management University
Singapore
xfxie@smu.edu.sg

Shangqing Liu†
Nanyang Technological University
Singapore
liu.shangqing@ntu.edu.sg

Ze Tang
Nanjing University
Nanjing, China
zetang@smail.nju.edu.cn

Ruitao Feng
Singapore Management University
Singapore
rtfeng@smu.edu.sg

Junjie Wang
Tianjin University
Tianjin, China
junjie.wang@tju.edu.cn

Jidong Ge
Nanjing University
Nanjing, China
gjd@nju.edu.cn

Lei Bu
Nanjing University
Nanjing, China
bulei@nju.edu.cn

ABSTRACT

The rise of code pre-trained models has significantly enhanced various coding tasks, such as code completion, and tools like GitHub Copilot. However, the substantial size of these models, especially large models, poses a significant challenge when it comes to fine-tuning them for specific downstream tasks. As an alternative approach, retrieval-based methods have emerged as a promising solution, augmenting model predictions without the need for fine-tuning. Despite their potential, a significant challenge is that the designs of these methods often rely on heuristics, leaving critical questions about *what information should be stored or retrieved and how to interpolate such information for augmenting predictions.*

To tackle this challenge, we first perform a theoretical analysis of the fine-tuning process, highlighting the importance of $\Delta logits$ as a catalyst for improving model predictions. Building on this insight, we develop a novel retrieval-based method, *FT2Ra*, which aims to mimic genuine fine-tuning. While *FT2Ra* adopts a retrieval-based mechanism, it uniquely adopts a paradigm with a learning rate and multi-epoch retrievals, which is similar to fine-tuning.

We conducted a comprehensive evaluation of *FT2Ra* in both token-level and line-level code completions. Our findings demonstrate the remarkable effectiveness of *FT2Ra* when compared to state-of-the-art methods and its potential to genuine fine-tuning. In token-level completion, which represents a relatively easier task, *FT2Ra* achieves a 4.29% improvement in accuracy compared to the best baseline method on UniXcoder. In the more challenging line-level completion task, we observe a substantial ~2×+ increase

in Exact Match (EM) performance, indicating the significant advantages of our theoretical analysis. Notably, even when operating without actual fine-tuning, *FT2Ra* exhibits competitive performance compared to the models with real fine-tuning.

CCS CONCEPTS

• **Software and its engineering → Software development techniques**.

KEYWORDS

Code Completion, Retrieval-Augmented Language Models

1 INTRODUCTION

In the realm of software engineering, code pre-trained models (CPMs) specialized for code generation and completion are becoming increasingly prevalent. Recently, a series of code completion plugins such as GitHub Copilot [1], and Visual Studio IntelliCode [2] have significantly alleviated the burden on software developers and enhanced their development efficiency.

Code-centric pre-trained models are generally trained using vast amounts of source code data harvested from open repositories. In the inference phase, CPMs typically map the code prefixes to fixed-sized representations and use the representations to predict the next code token. However, despite the extensive training data, CPMs still struggle to capture rare or specialized patterns. On one hand, the rarity of certain patterns in the training data makes it difficult for the model to learn them adequately. On the other hand, the complex inter-dependencies between different data samples could include conflicting coding styles or logic that the model fails to reconcile.

---

*This work was done during the author's visit to Singapore Management University.
†Corresponding author

Furthermore, these CPMs may not excel in specific domains where task-oriented or project-specific knowledge is essential, such as project-specific API invocations. For example, the recent study [54] disclosed that these general-purpose pre-trained models are inferior in repository-level code completion where the interrelated dependencies among files within a repository are missed for these general models. Therefore, the post-training enhancement of these models becomes a crucial task.

To tackle the outlined challenges, a straightforward approach is to fine-tune the pre-trained models using specialized data, such as missing patterns or project-specific information. However, fine-tuning comes with its own set of limitations, particularly concerning the computational resources required and the quality of data necessary for effective adjustment. Fine-tuning the entire model necessitates storing and updating a colossal parameter set, an operation that becomes increasingly costly and often infeasible as model size escalates into billions of parameters. Furthermore, the success of this strategy hinges on the availability of high-quality, task-specific data. While parameter-efficient fine-tuning techniques have been proposed [18, 26, 35], they still demand considerable computational resources for fine-tuning.

Recent research [24, 47, 54] proposes an alternative route through the use of retrieval-augmented language models (RaLMs). These models supplement the capabilities of pre-trained models by incorporating retrieval mechanisms that source information (e.g., rare patterns) from an external database, thereby bypassing the need for additional fine-tuning. This method enables the model to explicitly store and retrieve rare patterns, as opposed to implicitly integrating them into the model's parameters [24]. This paradigm aligns well with human learning behavior, where sparse examples are leveraged to generalize effectively to new situations. Empirical results demonstrate that RaLMs can significantly enhance the performance of CPMs, particularly in the prediction of rare patterns.

For retrieval-augmented language models, two main challenges exist the identification of similar samples from an external database and the effective utilization of this retrieved information for making predictions. Typically, the former is addressed by retrieving nearest neighbors based on distance metrics in a pre-trained embedding space. For the latter, existing methods adopt different methods such as employing frequency analysis [51] and empirical probabilities [50] to integrate the retrieved information. For example, kNN-LM [24] retrieves the neighbors and computes a distribution over neighbors based on a softmax of their negative distances, which are used to augment the original predictions. The most recent work kNM-LM [47] retrieves the code tokens that the language model fails to predict and normalize into a distribution, which is merged with the predictions of the language model. While these heuristic approaches have yielded promising results, they largely depend on empirical settings, leaving theoretical gaps in terms of what information should be retrieved and how this information can be better exploited.

In this paper, to better understand the optimal use of retrieved knowledge, we first conduct a theoretical analysis of the fine-tuning process in CPMs. Our theoretical analysis and derivation reveal insights for designing a strategy that more closely approximates the effects of fine-tuning. While our theoretical derivation does incorporate certain approximations, the evaluation results still demonstrate

the effectiveness of the retrieval mechanism. Specifically, our analysis indicates that the logits discrepancy between the predicted and actual values associated with neighbors (i.e., $\Delta logits$) [1] serves as crucial information for augmenting CPM predictions. Based on the analysis, we develop a novel retrieval-augmentation technique, denoted as *FT2Ra*, for code completion tasks. CPMs can recalibrate and improve its predictions by adding the $\Delta logits$ to its logit output. Furthermore, akin to the iterative nature of the fine-tuning process, *FT2Ra* is designed to operate through an iterative retrieval cycle, progressively updating the external database to refine the quality of retrieved information, thereby continuously improving prediction accuracy.

To showcase the effectiveness of *FT2Ra*, we selected four state-of-the-art retrieval-based methods: kNN-LM [24], kNM-LM [47], ReACC [36], and BM25 [44]. Our evaluation encompassed both token-level and line-level code completion. The experimental results demonstrate that, guided by our theoretical analysis, *FT2Ra* significantly outperforms the baseline methods and achieves competitive performance similar to actual fine-tuned models. For instance, in the context of token-level completion, *FT2Ra* obtains an average accuracy of 74.22% (**4.29%+**) on UniXcoder, whereas UniXcoder and the top-performing baseline, kNM-LM, achieve accuracy of 54.07% and 69.93%, respectively. In the more challenging line-level completion task, *FT2Ra* achieves an average Exact Match (EM) score of 26.32 (~2×+) on UniXcoder. In contrast, UniXcoder and kNM-LM only manage scores of 1.63 and 13.93, respectively. We also observed that, in line-level completion using UniXcoder, *FT2Ra* achieves performance better than that of the fine-tuned UniXcoder model after 10 epochs, even when operating without fine-tuning. These results not only demonstrate the effectiveness of *FT2Ra* but also highlight its significant potential to achieve competitive results comparable to those of fine-tuned models. Furthermore, our additional evaluations reveal that the iterative retrieval mechanism designed within *FT2Ra* significantly contributes to its performance.

In summary, this paper makes the following contributions:

- **Theoretical Analysis:** We perform a theoretical analysis of the model fine-tuning process. This analysis provides valuable insights into *how to effectively exploit retrieval information in retrieval augmentation mechanisms.*
- **Methodology:** Building upon the insights derived from our theoretical analysis, we introduce a novel method called *FT2Ra*. This innovative approach emulates real fine-tuning through an iterative retrieval process, enhancing its effectiveness.
- **Comprehensive Evaluation:** We conduct an extensive evaluation to evaluate the effectiveness of *FT2Ra* in both token-level and line-level code completion tasks. The results highlight substantial improvements achieved by *FT2Ra*.
- **Open-Source Resources:** We have made the pertinent data, detailed experimental findings, and the tools publicly available [3].

---

[1]$\Delta logits$ represents the difference in logits before and after gradient descent, which can be expressed as $\Delta logits = logits' - logits$.

2 BACKGROUND AND PROBLEM

2.1 Retrieval-Augmented Language Models

Recently, a series of retrieval-augmented language models [14, 24, 47] have been proposed to augment language models with external knowledge [9, 17, 53]. Retrieval-augmented techniques can generally be divided into two types. The first type is at the input layer [14, 20, 42], where the retrieved information is text chunks. The second type is at the output layer [7, 24, 47], where the retrieved information is tokens. By combining the retrieved tokens with the tokens generated by the original model, the accuracy of the retrieval-augmented model's generation for each token can be improved. The first type of method can provide the model with more external knowledge, making it adept at handling tasks in the NLP field such as knowledge-based question answering [27, 45, 49]. The second type of method can refer to the retrieved information to correct the generated tokens, making it more suited for handling strictly structured generative tasks, such as code completion [7, 10, 11]. In this work, we mainly focus on the second category.

To better understand the mechanism, we take kNN-LM [24] as an example for a detailed explanation. Given a context sequence $c_t = (w_1, \ldots, w_{t-1})$, the language models (LMs) estimate $p_{LM}(w_t|c_t)$, i.e., the probability distribution over the next token w_t. kNN-LM is designed to augment a pre-trained language model with a set of nearest neighbours retrieved from an external text collection, which can be the training set D. Different from fine-tuning, retrieval augmentation does not need any retraining. In particular, RaLM includes two tasks, i.e., building a datastore and retrieval-augmented inference.

Datastore: The datastore is a retrieval set, which can be built with a forward pass by LM on the prepared text collection to store the context-target pairs as the subject of a query. We denote a function $f(\cdot)$ to map a context c to a fixed-length vector representation computed by a pre-trained LM. Given an example $(c_i, w_i) \in D$, we can pass c_i to a LM to get its vector representation, i.e., $k_i = f(c_i)$. The dataset D is a set of datasets such as the training data or other domain-specific data. In this way, we can obtain the key-value pair (k_i, v_i), where k_i is the context representation computed from LM and v_i is the target word w_i. Hence, the datastore (K, V) is a set of all context-target pairs built from D, which can be expressed as:

$$(K, V) = \{(f(c_i), w_i) \mid (c_i, w_i) \in D\} \tag{1}$$

Inference: The inference phase includes neighbour retrieval and the use of neighbour prediction information. Given a new input x, the model first computes its context representation i.e., $f(x)$. Using $f(x)$ as a query to retrieve the k-nearest neighbours $\mathcal{N}$ from the datastore (K, V) based on a defined distance function $dis(\cdot)$ such as Euclidean distance. Then it computes a distribution over these k neighbors using a softmax function. The probability for each vocabulary item is aggregated across all occurrences in the retrieved targets. Note that the items in the vocabulary set that do not appear in the retrieved targets have a probability of zero.

$$p_{kNN}(y|x) \propto \sum_{(k_i, v_i) \in \mathcal{N}} \mathbf{1}_{\{y=v_i\}} \exp(-dis(k_i, f(x))) \tag{2}$$

The final distribution is interpolated with the original LM distribution $p_{LM}(y|x)$ and $p_{kNN}(y|x)$ to obtain the joint distribution:

$$p(y|x) = (1 - \lambda)p_{LM}(y|x) + \lambda p_{kNN}(y|x) \tag{3}$$

where λ is a tuned hyper-parameter to control the weight of generation and retrieval.

2.2 Problem

From Equation 3, we can observe that the final distribution of kNN is the weighted sum of the original LM distribution i.e., $p_{LM}(y|x)$ and the retrieved nearest neighbor distribution i.e., $p_{kNN}(y|x)$. The key problem is how to interpolate the retrieved knowledge in the prediction, i.e., the design of $p_{kNN}(y|x)$. In kNN-LM, $p_{kNN}(y|x)$ is computed from Equation. 2 based on negative distances and the aggregated probability for each vocabulary item across all its occurrences in the retrieved targets. While the design is intuitive, it is still based on heuristics and lacks theoretical analysis and explanation. A key question to *identify what kinds of information should be retrieved and how best to leverage that information.*

3 APPROACH

In this section, we delve into a theoretical analysis aimed at identifying useful retrieval information, drawing inspiration from the fine-tuning process commonly employed for enhancing the performance of CPMs. Subsequently, we introduce our method, *FT2Ra*, which focuses on the effective interpolation of this retrieved information to improve the predictive accuracy of CPMs.

3.1 Inspiration From Fine-tuning

Fine-tuning serves as a practical technique for boosting the performance of pre-trained models, particularly when applied to domain-specific tasks or datasets that the original model may not adequately cover. Our goal is to distil insights from the mechanics of fine-tuning to inform the design of a retrieval-augmented method that approximates the performance improvements seen with fine-tuning, yet obviates the need for the fine-tuning process itself.

Let $\mathcal{M}$ represent a given language model capable of predicting the next token x_t based on its preceding context sequence $x = (x_1, x_2, \ldots, x_{t-1})$. We proceed with the following definitions:

- θ denotes the trained model parameters of $\mathcal{M}$.
- y is the ground-truth for x_t as a one-hot encoding, where the index corresponding to x_t is marked as 1 while other indices are 0. $y \in \mathbb{R}^v$ is a vector where v is the length of the vocabulary set.
- y' is the model prediction result for the next token, i.e., $y' = \mathcal{M}(x|\theta)$ and $y' \in \mathbb{R}^v$, which denotes the predicted probability of each token in the vocabulary set with the context. Typically, y' is the output for the probability layer of the model $\mathcal{M}$.
- $logits \in \mathbb{R}^v$ encapsulates the values in the logits layer, preceding the probability layer.
- $seqout \in \mathbb{R}^{dmodel}$ is the output of the decoder sequence output layer, preceding the logits layer, and $dmodel$ is the dimension of this layer.

Suppose the LM $\mathcal{M}$ undergoes fine-tuning through multiple epochs, following best practices. Without loss of generality, we assume that the loss for a given input x diminishes after each iteration of the fine-tuning (i.e., the gradient descent algorithm).

Qi Guo, Xiaohong Li, Xiaofei Xie, Shangqing Liu, Ze Tang, Ruitao Feng, Junjie Wang, Jidong Ge, and Lei Bu

Let θ and θ' denote the model's parameters before and after an epoch of fine-tuning, respectively, such that $\theta' = \theta + \Delta\theta$. The corresponding loss values before and after the fine-tuning are:

$$l = \mathcal{L}(\mathcal{M}(x|\theta), y), l' = \mathcal{L}(\mathcal{M}(x|\theta'), y)$$

where $\mathcal{L}$ is the loss function, and y is the ground truth for the given context sequence x. We define the change in the loss as $\Delta l = l' - l$. Given that the language model is differentiable, the change in loss Δl can be expressed as:

$$\Delta l = \mathcal{L}(\mathcal{M}(x|\theta + \Delta\theta), y) - \mathcal{L}(\mathcal{M}(x|\theta), y) \quad (4)$$

In gradient descent, the learning rate η_θ controls the magnitude of parameter updates:

$$\Delta\theta = -\eta_\theta \times \frac{\partial\mathcal{L}}{\partial\theta} \quad (5)$$

On the other hand, the loss can also be formulated in terms of logits, $l = \mathcal{L}(softmax(logits), y)$, where $logits$ is the model output on x and y is the ground truth. After one iteration of the gradient descent, the loss value l' can be described as $l' = \mathcal{L}(softmax(logits'), y)$, with $logits'$ denoting the output of the updated model. Let $\Delta logits = logits' - logits$, and we can derive:

$$\Delta l = \mathcal{L}(softmax(logits + \Delta logits), y) - \mathcal{L}(softmax(logits), y)$$
$$= (\Delta logits)^T \cdot \frac{\partial\mathcal{L}}{\partial logits}$$
$$(6)$$

Intuitively, if we can develop a method for approximating $\Delta logits$ without actually engaging in fine-tuning, then these approximated $\Delta logits$ could be directly interpolated into the predictions of the model. This mimics the effects of fine-tuning and may achieve comparable performance, depending on the accuracy of the $\Delta logits$ approximation.

We observe the final LM-head layer of the generative model, where $logits = lm\_head(seqout)$. We ignore the activation layer in LM-head and can approximately treat the LM-head as a linear layer, from which we can derive:

$$logits \approx W \cdot seqout + b \quad (7)$$

where W is a weight matrix with the dimension $v * dmodel$.

From equation 7, using the chain rule for differentiation [4], we get:

$$\frac{\partial l}{\partial W} = \frac{\partial l}{\partial logits} \cdot seqout^T \quad (8)$$

During the gradient descent process, since W is a part of θ, according to equation 5, it also follows the gradient descent rule:

$$\Delta W = -\eta_\theta \cdot \frac{\partial l}{\partial W} \quad (9)$$

When we fix the parameters preceding $seqout$ and only fine-tune the subsequent parameters of the model, then according to equation 7, 8 and 9, we make an approximation:

$$\Delta logits \approx \Delta W \cdot seqout$$
$$= -\eta_\theta \cdot \frac{\partial l}{\partial W} \cdot seqout$$
$$= -\eta_\theta \cdot \frac{\partial l}{\partial logits} \cdot seqout^T \cdot seqout \quad (10)$$
$$= -\eta_\theta \cdot ||seqout||_2^2 \cdot \frac{\partial l}{\partial logits}$$

We use $||seqout||_2$ to denote the L2 norm of $seqout$. Furthermore, we define $-\eta_{logits}$ as $-\eta_\theta \cdot ||seqout||_2^2$, and we can get:

$$\Delta logits \approx -\eta_{logits} \times \frac{\partial\mathcal{L}}{\partial logits} \quad (11)$$

Equation 11 offers a feasible methodology for calculating changes in logits, which can be employed to bolster the current model's performance on x reducing its loss. To obtain the value of $\frac{\partial\mathcal{L}}{\partial logits}$, we propose the retrieval-based method detailed in Section 3.2.1.

Our derivation implies new insights about what kind of information should be stored and retrieved (i.e., the $\Delta logits$) and how to leverage the information (i.e., add $\Delta logits$ to the predicted logits). In summary, it introduces the following benefits: 1) the retrieval mechanism is theoretically grounded, different from the existing mere heuristic approaches, 2) the retrieval mechanism tries to mimic the fine-tuning process, which has a high potential to achieve high performance and 3) the retrieved knowledge regarding $\Delta logits$ is more fine-grained compared to existing methods, and its integration into the prediction process is both straightforward and direct.

3.2 Algorithm

Building on the theoretical insights from fine-tuning, we introduce a novel retrieval-augmented method.

3.2.1 Approximation of $\frac{\partial\mathcal{L}}{\partial logits}$.
To approximate the value of $\frac{\partial\mathcal{L}}{\partial logits}$ shown in Equation 11, we employ the nearest k neighbors of the sample x for the estimation. The approximation is formulated as

$$\frac{\partial\mathcal{L}}{\partial logits} \approx \sum_i \lambda_i \times \frac{\partial\mathcal{L}_i}{\partial logits_i} \quad (12)$$

where $1 \leq i \leq k$ represents the ith neighbor and λ_i serves as a hyper-parameter to adjust the contribution of each neighbor to the approximation. Since $\frac{\partial\mathcal{L}_i}{\partial logits_i}$ is the partial derivative with respect to the logits layer, we have $\frac{\partial\mathcal{L}_i}{\partial logits_i} = y'_i - y_i$ for each neighbor. Incorporating this into Equation 12 yields:

$$\frac{\partial\mathcal{L}}{\partial logits} \approx \sum_i \lambda_i \times (y'_i - y_i) \quad (13)$$

Finally, we integrate Equation 13 into Equation 11 to derive:

$$\Delta logits \approx -\eta_{logits} \times \sum_i \lambda_i \times (y'_i - y_i)$$

$$logits' \approx logits - \eta_{logits} \times \sum_i \lambda_i \times (y'_i - y_i) \quad (14)$$

Given an input x, Equation 14 offers a mechanism to calculate new logits by leveraging both the original prediction and the contributions from the nearest neighbours.

Algorithm 1: *FT2Ra*

Input: Test sample: x, large model: $\mathcal{M}$, learning rate:
η_{logits}, datastore: (K, V),
the number of neighbours: N, the number of iterations: E
Output: The output: y'_x

1 $r \leftarrow f_{\mathcal{M}}(x)$
2 $(Y, L, D) \leftarrow retrieve(r, N, (K, V))$
3 **for** $e \in \{1, \dots, E\}$ **do**
4      $\Delta logits \leftarrow 0$
5      $logits_x \leftarrow cal\_logits(\mathcal{M}, x)$
6   **for** $(y_i, logits_i) \in (Y, L)$ **do**
7          $y'_i \leftarrow softmax(logits_i)$
8          $\lambda_i \leftarrow cal\_weight(d_i, D)$
9          $\Delta logits_i \leftarrow \lambda_i \times \eta_{logits}(y_i - y'_i)$
10          $\Delta logits \leftarrow \Delta logits + \Delta logits_i$
11      $logits_x \leftarrow logits_x + \Delta logits$
12   **for** $l_i \in L$ **do**
13          $l_i \leftarrow l_i + \Delta logits$
14 $y'_x = softmax(logits_x)$
15 **return** y'_x

3.2.2 Datastore Construction.

As with prior work in this area [24, 47], a retrieval set, referred to as datastore D, is essential for storing context-target pairs, often represented as key-value pairs. The nature of the knowledge encapsulated in this datastore depends on the specific retrieval mechanism employed, particularly the type of information used for the calculation.

In the datastore, each key is generated to facilitate distance calculation between the given input and the elements in the retrieval set. For a given training example $(c_i, w_i) \in D$, we map the context c_i to a fixed-length vector representation using a function $f(\cdot)$. In line with previous research [24], we utilize the last hidden states (i.e., the output of the final layer of the CPM as this mapping function $f(\cdot)$. Hence, the *key* for each entry is $k_i = f(c_i)$.

Considering Equation 14, the *value* associated with each key should include both the ground truth y_i (which corresponds to w_i in a one-hot encoded format) and the predicted probability distribution y'_i. Instead of storing the probability vector, we opt to store the corresponding logits vector $logits_i$. This is because: 1) The predicted probability y'_i can be easily recalculated from $logits_i$ whenever needed and 2) Storing $logits_i$ allows for their use in multiple retrieval iterations, as will be detailed in Section 3.2.3. Given these considerations, the datastore is formally defined as:

$$(K, V) = \{(f(c_i), (y_i, logits_i)) \mid (c_i, w_i) \in D\}$$

3.2.3 Iterative Nearest Neighbor Retrieving.

Algorithm 1 outlines the steps involved in the execution of *FT2Ra*, our retrieval-augmented language model. The inputs to *FT2Ra* include: an input context x, the original pre-trained model $\mathcal{M}$, the learning rate η_{logits}, the datastore (K, V), the number of neighbors to retrieve N, and the number of iterative retrieval cycles E. The output generated by *FT2Ra* is the updated prediction y'_x.

Initially, *FT2Ra* computes the representation vector r of the input x, using it to fetch the top-N nearest neighbors from the datastore

(lines 1–2). An iterative retrieval process then follows, which is a unique feature compared to existing methods. The iterative retrieval process is similar to the process of model fine-tuning conducted over a specified number of epochs, denoted as E (lines 3–13). At each iteration, the original model's prediction (retrieved from $logits_x$ in line 5) is adjusted based on the logits alteration computed from the retrieved neighbors (lines 6–10).

It's worth noting that neighbors may vary in their relevance to the input context. Accordingly, we introduce weights λ_i for each neighbor (line 8). These weights are calculated based on the inverse of the distance between the neighbors and the input:

$$\lambda_i = \frac{1/(d_i + 1)}{\sum_{d \in D}(1/(d + 1))} \tag{15}$$

Intuitively, a smaller distance between a neighbor and the input results in a higher weight, meaning that closer neighbors contribute more substantially to the updated prediction.

Finally, *FT2Ra* updates the logits using the calculated change in logits, which has been interpolated from retrieved samples (line 11). To facilitate further iterations of the retrieval process, the datastore is also updated (line 13). While an ideal update method would involve recalculating the entire datastore using Equation 14, we opt for a more computationally efficient strategy. Specifically, we maintain the same set of neighbors across all iterations and apply a constant $\Delta logits$ to the logits of these neighbors, balancing efficacy with computational efficiency.

Discussions. Differing from existing methods, *FT2Ra* provides two main advantages. First, it employs detailed retrieval information, $\Delta logits$, for a more precise evaluation of each retrieved neighbor's influence on the final prediction. Second, its iterative retrieval cycles can further improve performance. It's important to note that these multiple iterations are not actual fine-tuning, but rather a series of retrieval processes. These iterations are also optional and can be adjusted according to specific needs, like accuracy and inference efficiency. In our evaluation, we found that *FT2Ra* outperformed baseline models even with just one iteration (the conventional setting). With multiple iterations, however, *FT2Ra*'s performance can be further enhanced (see results in RQ4).

4 EXPERIMENTAL SETUP

The experimental design considers two completion scenarios: token-level and line-level completions, on models with or without fine-tuning. Specifically, we aim to answer the research questions:

- **RQ1**: How effective is *FT2Ra* in the two completion tasks?
- **RQ2**: To what extent can *FT2Ra* approximate the effect of actual fine-tuning?
- **RQ3**: How do different parameter settings, including the weighting strategies and the number of neighbours selected, affect *FT2Ra*'s performance?
- **RQ4**: How useful is the multi-round iteration strategy in *FT2Ra*?

4.1 Benchmarks

Completion Scenario. Based on the scale of completion, we consider two completion scenarios, i.e., token-level and line-level completions. For token-level completion, the model predicts the next

token, based on the given (correct) context. The metric for evaluation in token-level completion is *accuracy*, i.e., checking whether each completion is correct. For line-level completion, the model performs repeated execution of token-level completion until a line is completed, and retrieval occurs at every step of token prediction. Contrasting with token-level completions, predictions for each token depend on the prediction of the preceding token, which might be incorrect. In line with CodeXGLUE [37], the chosen evaluation metrics are *exact match* (EM) and *edit similarity* (ES).

Datasets. We have chosen two widely used benchmarks for our study: the dataset from kNM-LM [47] and the code completion benchmarks from CodeXGLUE [37]. Specifically, kNM-LM benchmark comprises 20 Java projects: 10 large-scale and 10 small-scale. In our experiments, we selected the ten larger projects. CodeXGLUE benchmarks contain code samples written in both Java and Python programming languages, i.e., JavaCorpus [6] and PY150 [43].

We follow the settings in [37, 47] for preparing and splitting the training and testing data. The training dataset can be used to fine-tune the pre-trained models. Note that the kNM-LM benchmarks do not provide a pre-defined test set tailored for line-level completion. To circumvent this, we follow the instructions in [6]. Specifically, we randomly extract 300 lines of code from the test data of each project to serve as targets for model completion. For evaluations regarding token-level predictions, we let the models predict each individual token in the test code samples.

Models. Following the state-of-the-art work [47], we selected two widely used code pre-trained models in our experiments: 1) **CodeGPT** [37]: It is a GPT-style code pre-trained model to support code completion. CodeGPT has the same model architecture and training objectives as GPT-2 [41], which consists of 12 layers of Transformer decoders. CodeGPT is pre-trained on Python and Java corpora from CodeSearchNet [19], which includes 1.1M Python code and 1.6M Java code. CodeGPT-adapted is pre-trained from GPT-2 and we use CodeGPT-small-java-adaptedGPT2 and CodeGPT-small-python-adaptedGPT2 to evaluate code completion in Java and Python datasets, respectively. 2) **UniXcoder** [13]: It is a cross-modal pre-trained model using mask attention matrices with prefix adapters to control the model behaviour. Furthermore, it leverages cross-modal contents such as AST and code comment to enhance the code representations. Specifically, it consists of 12 layers of Transformer with 768-dimensional hidden states. UniXcoder is pre-trained on the CodeSearchNet [19] dataset for six programming languages including Java and Python.

Although our method is general, in this paper, we did not select very large models like CodeGen, InCoder, and CodeLLama, as fine-tuning them demands substantial computing resources. This requirement arises particularly because 1) the dataset includes unique symbols (e.g., $< STR\_LIT >, < NUM\_LIT >, < CHAR\_LIT >$) that necessitate specialized fine-tuning and 2) our experimental setting in RQ2 requires fine-tuning. Consequently, we chose two large CPMs, as suggested in the recent study [47].

4.2 Baselines

Four state-of-the-art retrieval-based baselines are selected for comparisons, including kNN-LM, kNM-LM, BM25 and ReACC, where BM25 and ReACC are suitable to the line-level completions.

- **kNN-LM** [24]: It augments the prediction of a pre-trained language model by linearly interpolating its next word distribution with a k-nearest neighbours model. The nearest neighbours are computed based on the distance in the vector space with a single forward pass of a pre-trained model over the retrieved dataset. The final distribution is the weighted sum of the original model distribution and the nearest neighbour distribution.
- **kNM-LM** [47]: It utilizes the in-domain code to construct the retrieved datastore decouple from LM and then combines with LM by Bayesian Inference for code completion. Compared with kNN-LM, it is able to calculate the posterior probability and utilize it to merge the distributions of nearest neighbours and the original model, which avoids manual weight tuning between the model distribution and neighbour distribution.
- **BM25** [44]: It is a term-based retrieval approach, which considers each code fragment as a code token sequence and employs bag-of-words representations to compute the matching score based on the lexical similarity between the query and document. Hence, it is more suitable for the line-level completion. As BM25 is based on the term frequency, it is one kind of sparse retriever.
- **ReACC** [36]: It is a hybrid retriever framework by combining scores of sparse and dense retrievers. For sparse retrievers, it uses BM25 [44] for implementation. For dense retriever, it maps each code fragment to a dense vector based on the DPR model [23], which consists of two bidirectional transformer encoders to encode the query code and the retrieved code for the retrieval.

Configurations. Considering the hyper-parameters in Algorithm 1, in our experiments, we configured the number of neighbors (N) and the number of iterations (E) to 20 and 7, respectively. Additionally, we set the learning rates (η_{logits}) to specific values: 3 for JavaCorpus, 5 for PY150, and 5 for the kNM-LM dataset. It's worth noting that we thoroughly evaluated and discussed various settings of these hyperparameters in RQ2 and RQ3. For the other baseline methods, we selected the default configuration used in their papers. Notably, kNN-LM is not applied in code learning tasks, we followed the same configurations as described in [47].

5 EXPERIMENTAL RESULTS

5.1 RQ1: Effectiveness on pre-trained models

The main goal of the retrieval augmentation is to bolster the model's performance without the need for fine-tuning. Therefore, this experiment aims to evaluate the effectiveness of *FT2Ra* on pre-trained models without fine-tuning.

Token-level Completion. The results for token-level completion are shown in Table 1, including the results on the ten Java projects from kNM-LM and the two CodeXGLUE benchmarks. The *Original* column shows the results with the pre-trained models.

The overall results show that, compared with the original pre-trained models, all retrieval-augmented techniques have a higher accuracy, demonstrating the usefulness of retrieval-based augmentation. Furthermore, we can see that *FT2Ra* significantly outperforms the baselines across all datasets and models. For example, while the average accuracies of pre-trained models on CodeGPT and UniXcoder stand at 55.46% and 54.07%, respectively, *FT2Ra* increases the performance to 73.19% and 74.22%, outperforming all

Inputs:
...
form = SQLFORM.factory(Field('filename', requires=IS_IN_SET(files), label **?**

Ground Truth:	=T ("<STR_LIT>") , <EOL>
FT2Ra:	=T ("<STR_LIT>") , <EOL>
CodeGPT:	=T ("<STR_LIT>") , requires = IS_IN_SET (files) , label...
kNN-LM	=T ("<STR_LIT>") , require = IS_IN_SET (files) , require...
BM25&ReACC	=T ("<STR_LIT>") , requires = IS_IN_SET (files) , label...
kNM-LM:	, <EOL>

FT2Ra prediction on tokens (requires, <EOL>,...)				
Iter.	Logits	Δ logits	Updated logits	Pred.
1	(10.85,0.0,...)	(-0.001,2.88,...)	(10.85, 2.88,...)	requires
...	...	...	...	...
5	(10.84, 8.77,...)	(-0.001,1.94,...)	(10.84,10.71,...)	requires
6	(10.84,10.71,...)	(-0.001,1.40,...)	(10.84,12.11,...)	<EOL>
7	(10.84,12.11,...)	(-0.001,0.88,...)	(10.84,12.99,...)	<EOL>

Figure 1: Case study for CodeGPT line-level completion on PY150

Table 1: Results of token-level completion on pre-trained models (%).

Type	Dataset	CodeGPT				UniXcoder			
		Original	kNN-LM	kNM-LM	FT2Ra	Original	kNN-LM	kNM-LM	FT2Ra
kNM-LM	Rest.	46.99	54.99	70.71	**77.68**	42.59	50.86	71.26	**77.58**
	Amaze.	55.22	58.33	66.34	**71.00**	54.65	56.73	68.14	**71.79**
	Dropwizard	50.11	55.00	65.50	**71.14**	47.15	51.30	66.56	**70.12**
	Eureka	52.76	55.73	64.56	**70.15**	51.00	54.36	66.01	**69.35**
	Feign	48.71	54.47	70.63	**77.48**	45.01	50.36	70.87	**77.12**
	Galaxy	53.40	55.57	64.90	**69.24**	49.57	52.47	65.16	**69.25**
	Interview	64.70	66.46	69.29	**73.14**	62.91	64.80	71.54	**75.27**
	Logging.	60.06	65.10	79.10	**87.38**	56.95	61.85	79.69	**86.30**
	Requery	56.69	59.44	68.39	**75.75**	54.11	56.07	68.66	**74.91**
	Froyo.	59.53	62.56	67.64	**71.04**	58.79	61.31	69.38	**72.79**
	Avg.	54.82	58.77	68.71	**74.40**	52.27	56.01	69.73	**74.45**
CodeXGLUE	JavaCorpus	64.95	67.83	70.74	**72.07**	65.61	67.92	72.13	**74.73**
	PY150	52.41	55.35	60.94	**62.19**	60.44	64.62	69.81	**71.43**
	Total Avg.	55.46	59.24	68.23	**73.19**	54.07	57.72	69.93	**74.22**

baseline models. Moreover, in comparison with the best baseline kNM-LM, *FT2Ra* boosts an average increase of 4.96% for CodeGPT and 4.29% for UniXcoder. The results demonstrate the effectiveness of *FT2Ra* in token-level completion.

Line-level Completion. The results for line-level completion, evaluated on CodeGPT and UniXcoder, are presented in Table 2. The metrics of exact match and edit similarity are represented by the columns *EM* and *ES*, respectively. Similarly, we can find that all of the retrieval-based methods could still enhance the performance, but the improvement degree of baselines is generally limited. For example, on average, kNN-LM, kNM-KM, BM25 and ReACC achieve scores (7.12, 56.74), (11.68, 46.99), (5.30, 53.59) and (5.43, 53.64) on CodeGPT, respectively, while the pre-trained model achieves (4.19, 51.71). While the recent state-of-the-art kNM-LM can achieve higher EM scores than other baselines, its ES scores are lower. The low performance of baselines could be attributed to the difficulty of the line-level completion. Any incorrect token prediction (inaccurate context) could affect the prediction of the following tokens. It is obvious that *FT2Ra* significantly outperforms the baselines, manifesting its superiority in both the EM and ES metrics across all datasets and models. Considering the results on CodeGPT, *FT2Ra* increases the scores to (24.35, 67.90). While on UniXcoder, *FT2Ra* achieves higher improvement (26.32, 70.11) than the pre-trained model (1.63, 50.54) and the baselines. The results demonstrate the effectiveness of our proposed retrieval mechanism on line-level completion.

We have observed that the performance of various methods varies across different datasets and models. Considering the results on the dataset Froyo., we find that all methods consistently achieve higher EM scores on CodeGPT compared to UniXcoder. Interestingly, all baseline models, including pre-trained ones, exhibit poor performance on the PY150 dataset but demonstrate better results on the JavaCorpus dataset. Upon our in-depth analysis of CodeGPT, we discovered that the pre-trained model CodeGPT-small-py-adaptedGPT2 tends to underestimate the probability of end-of-line tokens (<EOL>). We randomly selected 30 test data instances from PY150, specifically targeting cases where *FT2Ra* succeeded while the original model failed. We discovered that 9 of these instances reached the maximum token prediction count (set at 100) when predicted by CodeGPT. In contrast, we randomly checked 100 instances in JavaCorpus predicted by CodeGPT-small-java-adaptedGPT2, and none of the predictions reached the token count limit. This discrepancy could be attributed to the natural line termination indicators present in Java code such as semicolons and braces, which allow the model to easily discern when to stop the prediction. However, in Python, the model must accurately predict the <EOL> symbol to recognize the end of a statement. Compared with others, *FT2Ra* exhibits significant enhancements on the PY150 dataset, with improvements of (18.48, 50.52) and (29.17, 59.00) when evaluated on CodeGPT and UniXcoder, respectively.

In Figure 1, we present an illustrative example that shows the advantage of *FT2Ra* when applied to PY150. Upon examining the results obtained by different methods, we observe that, except for kNM-LM, all models accurately predict the initial seven tokens.

Table 2: Results of line-level completion on pre-trained models (%).

		CodeGPT											
Type	Dataset	Original		kNN-LM		kNM-LM		BM25		ReACC		FT2Ra	
		EM	ES	EM	ES	EM	ES	EM	ES	EM	ES	EM	ES
	Rest.	1.00	49.36	1.00	53.68	9.63	47.05	3.99	53.48	3.99	53.05	**17.94**	**70.62**
	Amaze.	1.99	56.86	3.99	57.94	9.30	47.75	2.99	59.25	2.99	58.89	**22.92**	**66.54**
	Dropwizard	1.00	52.13	2.33	57.83	4.65	49.56	1.33	52.52	1.33	51.84	**22.59**	**69.34**
	Eureka	3.99	55.76	5.32	58.20	8.31	50.41	3.99	57.59	3.99	57.78	**20.93**	**68.00**
	Feign	1.33	47.72	3.32	52.77	10.63	47.19	3.99	53.50	3.99	53.21	**25.91**	**72.34**
kNM-LM	Galaxy	1.33	50.97	2.66	54.61	11.96	48.53	2.66	51.72	2.33	52.07	**22.26**	**64.14**
	Interview	8.97	61.63	13.95	62.80	19.60	57.28	9.30	61.41	9.97	62.91	**27.91**	**71.08**
	Logging.	2.66	59.04	5.98	63.13	15.28	58.59	6.31	63.42	6.64	63.64	**33.89**	**80.63**
	Requery	5.98	61.76	8.97	62.81	9.63	46.51	6.64	63.21	7.31	63.12	**28.24**	**73.11**
	Froyo.	8.31	63.96	11.30	65.85	16.28	55.58	7.97	63.90	8.31	63.93	**28.24**	**73.91**
	Avg.	3.65	55.92	5.88	58.96	11.53	50.85	4.92	58.00	5.08	58.04	**25.08**	**70.97**
CodeXGLUE	JavaCorpus	13.69	48.74	16.28	49.90	16.18	43.89	14.19	49.73	14.19	49.85	**22.88**	**54.54**
	PY150	0.00	12.54	10.39	41.40	8.69	11.56	0.20	13.35	0.20	13.36	**18.48**	**50.52**
	Total Avg.	4.19	51.71	7.12	56.74	11.68	46.99	5.30	53.59	5.43	53.64	**24.35**	**67.90**
		UniXcoder											
Type	Dataset	Original		kNN-LM		kNM-LM		BM25		ReACC		FT2Ra	
		EM	ES	EM	ES	EM	ES	EM	ES	EM	ES	EM	ES
	Rest.	0.66	50.12	1.66	52.07	11.30	54.05	1.99	63.46	1.99	64.66	**18.94**	**72.17**
	Amaze.	1.33	56.00	1.66	58.40	13.29	55.28	1.00	58.87	1.00	59.04	**23.26**	**66.92**
	Dropwizard	0.00	49.44	1.33	54.89	16.61	56.41	0.66	59.82	0.66	58.33	**20.27**	**69.94**
	Eureka	0.33	55.44	1.66	59.39	15.61	58.91	0.00	63.56	0.00	62.23	**22.26**	**69.98**
	Feign	0.00	50.06	1.00	52.46	8.97	52.99	4.65	67.68	4.65	67.39	**27.24**	**75.24**
kNM-LM	Galaxy	0.33	50.06	0.33	53.06	13.29	47.61	1.00	54.89	1.00	54.86	**21.26**	**64.56**
	Interview	4.65	59.89	7.31	61.88	20.93	62.78	2.66	58.28	3.32	57.65	**30.56**	**73.34**
	Logging.	0.00	55.11	4.98	60.78	17.28	59.45	3.32	72.28	3.99	72.45	**35.22**	**81.23**
	Requery	2.33	60.56	3.99	61.92	14.95	56.73	2.33	65.34	2.33	65.14	**30.56**	**73.79**
	Froyo.	1.33	63.60	1.99	65.14	21.93	62.75	2.33	63.94	2.66	64.09	**32.56**	**76.46**
	Avg.	1.10	55.03	2.59	58.00	15.42	56.70	1.99	62.81	2.16	62.58	**26.21**	**72.36**
CodeXGLUE	JavaCorpus	8.49	47.72	9.99	49.69	12.89	48.06	7.79	46.97	7.79	47.09	**24.58**	**58.67**
	PY150	0.10	8.43	0.00	8.60	0.10	12.74	0.00	7.46	0.00	7.43	**29.17**	**59.00**
	Total Avg.	1.63	50.54	2.99	53.19	13.93	52.31	2.31	56.88	2.45	56.70	**26.32**	**70.11**
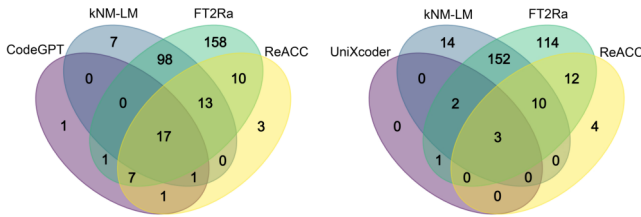
Figure 2: Venn diagram of the EM results on CodeGPT (left) and UniXcoder (right).

However, when reaching the eighth token, the baselines, including the original model, cannot predict the correct termination token <EOL>. Instead, they predict the token *requires*, which leads to an uninterrupted sequence of predictions until reaching the maximum token count. By checking CodeGPT's prediction on the eighth token, we discover that the token *requires* has the highest prediction probability (0.13), while the token <EOL> receives a prediction probability of 0, making it challenging for the baselines to correct the prediction. kNM-LM exhibits too much augmentation, resulting in incorrect predictions for even the first token. The table on the right provides detailed insights into how *FT2Ra* corrects the prediction. Despite the stubborn prediction of *requires*, *FT2Ra* leverages the calculation of Δ_{logits} across multiple iterations to steadily increase the logits of the token <EOL> while decreasing the logits of

Table 3: Results of average generation time per token (s).

		Input Retrieval			Output Retrieval		
	Original	BM25	ReACC		kNN-LM	kNM-LM	FT2Ra
CodeGPT	0.0163	0.0161	0.0164		0.0208	0.0193	0.0271
UniXcoder	0.0134	0.0143	0.0135		0.0163	0.0155	0.0214

requires. Ultimately, at the sixth iteration, *FT2Ra* successfully fixes the prediction.

In Figure 2, we present a Venn diagram depicting the completion lines that achieve an exact match with the ground truth. For the sake of clarity, we have excluded the results of BM25 and kNN-LM from the diagram since their outcomes closely resemble those of ReACC and kNM-LM. The findings clearly illustrate that *FT2Ra* outperforms other methods by generating a significantly larger number of unique code lines.

Performance. To evaluate *FT2Ra*'s performance, we measured the average time required to predict a token. We did not compare the line prediction time since the predictions of different methods can have different lengths. Specifically, we selected 1,000 line-completion tasks at random, using different methods to predict the line with a set length of 100 tokens. We then recorded the average token prediction time for comparison. All experiments were conducted on a single A5000 GPU card for consistency.

Table 3 presents the results. Note that the time used by *FT2Ra* is from its seven retrieval iterations. On the CodeGPT model, the
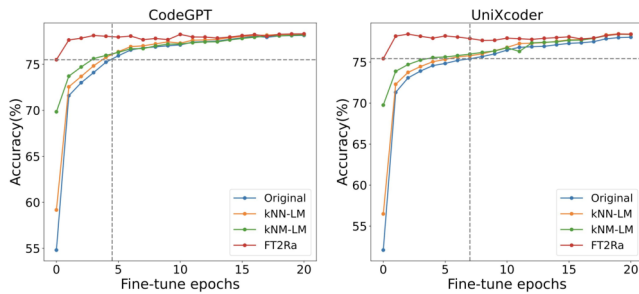
Figure 3: Results of token-level completion on fine-tuned models with different epochs.

Figure 4: Results of line-level completion.

average prediction times for the original model, BM25, ReACC, kNN-LM, kNM-LM, and *FT2Ra* are 0.0163s, 0.0161s, 0.0164s, 0.0208s, 0.0193s, and 0.0271s, respectively, and a similar trend is observed with UniXcoder. The results indicate that while input retrieval methods slightly impact prediction speed, output retrieval methods, which require more computation, tend to slow it down more noticeably. Compared to other output retrieval baselines, *FT2Ra*, which retrieves more detailed information and allows for multiple retrieval rounds, takes slightly longer. This represents a trade-off between effectiveness and efficiency, with *FT2Ra* sacrificing some speed for significant improvements in effectiveness.

Answers to RQ1: The results reveal *FT2Ra*'s dominant performance on pre-trained models over existing baselines in both token-level and line-level completions.

5.2 RQ2: Comparison with Fine-tuning

The key insight of *FT2Ra* lies an innovative approach that emulates the fine-tuning process with certain approximations (refer to Section 3.1). Hence, we compared the results of *FT2Ra* with genuine fine-tuning results on the kNM-LM dataset. We utilized the training data from all ten projects to fine-tune the pre-trained models and subsequently evaluated the methodologies on all test data of these projects. The pre-trained models were fine-tuned over a range of epochs. We compared the performance of various methods for both line-level and token-level completion, with the models fine-tuned across these different epochs.

Token-level Completion. For the token-level completion, we capped the maximum number of epochs at 20. This upper limit was chosen because it was observed that most methods tend to reach convergence within this period. Figure 3 presents the token-level completion performance of different methods on fine-tuned models over various epochs. Notably, the comparative results at each point are derived by the retrieval from the respective fine-tuned models at specific epochs (i.e., epochs 1, 2, . . . , 20). The data stores are also updated under different fine-tuned models. The results for epoch 0 are derived from the pre-trained model without fine-tuning.

Overall, we observe a progressive improvement in the performance of the original model with an increasing number of fine-tuning epochs (see blue lines). The results of the retrieval-based methods also exhibit an upward trend, showing the generalization capability across different fine-tuned models. However, as the model goes through multiple fine-tuning epochs, the improvements are
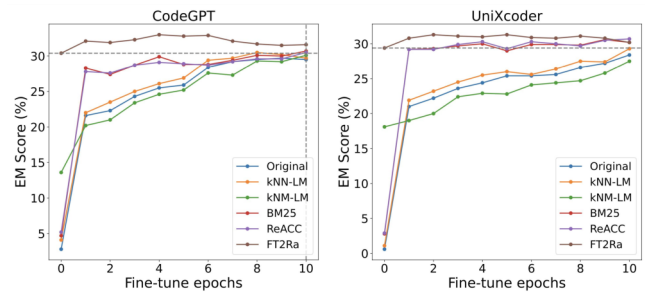
diminishing as the model nears its best performance after sufficient tuning. Comparing *FT2Ra* to the baselines, it is clear that *FT2Ra* consistently outperforms the baselines on fine-tuned models.

To assess how closely *FT2Ra*'s effect (simulating fine-tuning) aligns with real fine-tuning, we compare *FT2Ra*'s performance on the pre-trained model without any fine-tuning to that of the fine-tuned models. As indicated by the dotted line, *FT2Ra*, without fine-tuning the model, achieves similar performance to fine-tuned CodeGPT and UniXcoder models after approximately 4 and 7 epochs, respectively. In contrast, the best baseline, kNM-LM, only reaches a similar performance level with a model fine-tuned for about one epoch. These results underscore the value of our theoretical analysis from the fine-tuning process.

Line-level Completion. Figure 4 illustrates the results in terms of EM for line-level completion. Due to space constraints, the results for Edit Similarity can be accessed on our website [3]. We capped the maximum number of epochs at 10 due to the large computational overhead of line-based completion. When compared to the token-level completion results in Figure 3, it becomes evident that the impact of other baseline methods is notably diminished in line-level completion, primarily because this task is more difficult. We observe that BM25 and ReACC yield similar results, likely due to their adoption of similar methods. On the other hand, the performance of kNN-LM and kNM-LM is very close to that of the fine-tuned models, which indicates that they have limited improvement.

Conversely, *FT2Ra* continues to demonstrate clear advantages over other methods, due to its precise token prediction. Notably, when comparing the performance of *FT2Ra* at epoch 0 with that of other fine-tuned models, it becomes apparent that even without fine-tuning, FT2Ra can outperform the performance of fine-tuned models at 10 epochs.

Answers to RQ2: *FT2Ra* remains highly effective when applied to fine-tuned models. Furthermore, our results indicate that *FT2Ra* yields promising outcomes even without fine-tuning, achieving competitive or superior performance compared to fine-tuned models with multiple epochs.

5.3 RQ3: Impact of Weighting Strategy and the Number of Neighbors

To evaluate the effectiveness of our weighting strategy and to understand the impact of the number of neighbours, we collected four datasets, including the two Java projects that were randomly chosen from the kNM-LM dataset, and the two datasets from CodeXGLUE.

Qi Guo, Xiaohong Li, Xiaofei Xie, Shangqing Liu, Ze Tang, Ruitao Feng, Junjie Wang, Jidong Ge, and Lei Bu

Table 4: Results with different weighting strategies and different numbers of neighbors (%).

Dataset	Weighting Strategy				#Neighbors			
	Rec.	Uni.	Smax	Smax-T	5	10	20	50
Rest.	78.15	74.62	78.43	**78.46**	**78.69**	78.65	78.15	76.79
Eureka	**70.65**	68.80	67.12	66.84	69.38	70.14	**70.65**	69.79
JavaCorpus	**72.07**	71.97	67.98	68.15	70.58	71.40	72.07	**72.46**
PY150	**62.19**	61.17	56.85	56.94	60.93	61.69	**62.19**	61.93

The evaluation is performed on the token-level completion task, which serves as the foundation for line-level completion.

5.3.1 Effectiveness of the weighting strategy. Since various baseline methods employ different weighting strategies to determine the significance of the retrieved samples. For instance, kNN-LM utilizes the softmax (referred to as *Smax*), whereas kNM-LM employs softmax with temperate (denoted as *Smax-T*). Our method calculates weights based on distance, referred to as *Rec.* (see Equation 15). To provide a comparative evaluation, we incorporated these strategies into *FT2Ra* for the comparisons. Additionally, we introduced a baseline, i.e., uniform strategy (*Uni.*), which allocates equal weights to all samples. Detailed results can be found on the left of Table 4. Obviously, the weighting strategy *Rec.* outperforms other strategies when they are adopted to *FT2Ra*. An exception is the results on Rest., where *Smax* and *Smax-T* marginally exceed the performance of *FT2Ra*. Interestingly, the uniform strategy *Uni.* excels over the other two methods for the benchmark JavaCorpus and PY150, emphasizing the importance of designing a suitable weighting strategy.

5.3.2 Impact of the number of neighbours. We evaluated the performance of *FT2Ra* by setting the number of neighbors to 5, 10, 20, and 50. The findings, as presented in the right part of Table 4, suggest that *FT2Ra* exhibits relatively limited sensitivity to the number of neighbours chosen. There is not a single optimal parameter that is universally effective across all datasets. In general, selecting too few neighbours may not provide enough information to augment predictions. Conversely, selecting an excessive number might introduce negative effects, such as the interference of irrelevant neighbours.

> **Answers to RQ3**: Our weighting strategy is useful in enhancing the performance of *FT2Ra*. Moreover, *FT2Ra* generally exhibits limited sensitivity to changes in the number of neighbours.

5.4 RQ4: Usefulness of Multiple Iterations

To evaluate the effect of the multiple iteration strategy incorporated into *FT2Ra*, we configured *FT2Ra* with varying retrieval iterations (i.e., *E* in Algo. 1) ranging from 1 to 10. We also consider the impact of the parameter η_{logits}, which are configured with 4 values: 2.5, 5, 10, and 20. Evaluations were carried out using similar configurations as in RQ3, i.e., token-level completion on pre-trained models.

The results are presented in Figure 5. It is obvious that *FT2Ra*'s performance benefits from multiple retrieval rounds, which is a unique feature compared to existing retrieval-based baselines. By increasing the number of retrieval rounds, the performance of *FT2Ra* gradually gets better. From the results, we found that the performance of *FT2Ra* tends to stabilize after approximately 4 retrieval iteration cycles.
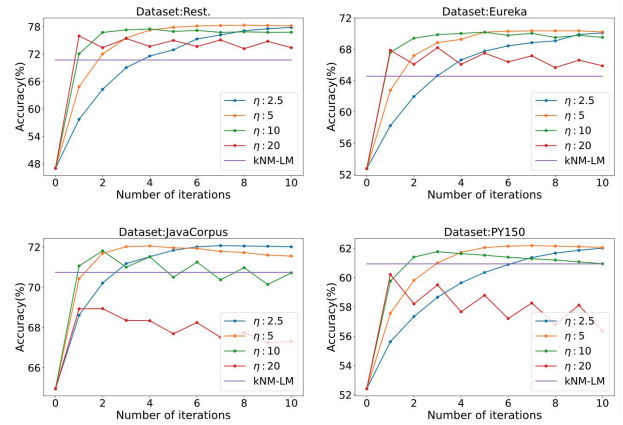
Figure 5: Results with different numbers of iterations.

With respect to different *learning rates* (i.e., η_{logits}), *FT2Ra*'s performance shows high sensitivity to this parameter. In general, larger values of η_{logits} enable *FT2Ra* to converge faster, whereas smaller ones necessitate multiple iterations. For instance, with η_{logits} set to 0.5, convergence tends to be achieved after 10 iterations. In contrast, a setting of 4 for η_{logits} reaches optimal performance after just one iteration. Yet, we also observed that excessively high learning rates could hamper *FT2Ra*'s performance. For instance, settings of η_{logits} at 2 and 4 typically yield results inferior to those achieved with 0.5 and 1. The configuration using a value of 4 for η_{logits} achieves the poorest performance. The *learning rate* in *FT2Ra* shows a similar effect to the learning rate of real training.

Even with just one iteration, *FT2Ra* surpasses the best-performing baseline, kNM-LM, as shown by the straight line in Figure 5. A higher learning rate, (e.g., η_{logits} = 10), is typically needed for faster convergence. Under this setting, *FT2Ra* outperforms kNM-LM in Rest., EureKa, and JavaCorpus. In PY 150, *FT2Ra* exceeds kNM-LM's performance after only 2 epochs. These results further highlight *FT2Ra*'s effectiveness, even with no or a few iterations.

While automatically selecting optimal parameters for learning rate and number of iterations can be challenging, there are some general guidelines that can aid in this process. Conducting initial trials on a small test set allows for the assessment of the model's performance. If the model exhibits rapid oscillation and a decline in performance, it suggests that the learning rate is too high. Conversely, if the model fails to converge after many iterations, it implies that the learning rate is too low. To adjust the number of iterations, early stopping techniques can be employed, ensuring that the tuning process is both computationally efficient and completed within a reasonable timeframe.

> **Answers to RQ4**: The multiple iteration strategy is useful in improving *FT2Ra*'s performance. In general, the more rounds, the better the results. Moreover, *FT2Ra* shows sensitivity to the learning rate parameter, η_{logits}. Smaller values tend to yield superior results, but they require a greater number of iterations for convergence.

6 THREATS TO VALIDITY

Potential biases from our choices of models and datasets represent a possible threat to our study. To mitigate it, we have followed the recent works [47] for guidance and selected two prominent datasets, i.e., the kNM-LM datasets and the CodeXGLUE benchmarks, and two widely-used pre-tained models, specifically CodeGPT and UniX-coder. We also plan to evaluate *FT2Ra* on the large models such as CodeLLama and CodeGen in future work. Furthermore, we were unable to establish a concrete theoretical framework to determine the weighting strategy. Instead, we empirically evaluated four common strategies in RQ3 and selected the most effective one. We acknowledge the significance of the weighting strategy and intend to investigate it further in future research. Another potential threat to our study is that the approximations inherent in retrieval-augmented methods may affect the precision of the results. This is particularly relevant when applying these methods to new models or datasets, where the impact of approximations might be more pronounced. In line with [47], there is a threat to the use of ReACC on Java programs. The original authors only made their retrieval models for Python available, leaving the Java version undisclosed. To circumnavigate this obstacle during our Java experiments, we utilized their Python version. In parallel, we incorporated the BM25 model, which has a similar performance to ReACC. For transparency, we have made our entire codebase, datasets, and experimental results public, thereby enabling independent verification.

7 RELATED WORK

Code Completion. Code completion is regarded as a vital aspect of enhancing software development efficiency in contemporary Integrated Development Environments (IDEs). Hindle [16] were pioneers in employing N-gram techniques to implement code completion using language models. Subsequently, deep neural networks [30] and pre-training techniques [12, 33, 34, 48] have been made great progress. While some of these efforts involve encoding code-specific structured information like Abstract Syntax Tree (AST) into inputs [25, 28], the prevailing trend in current research treats source code as sequences of code tokens, as exemplified by models like CodeGPT [37], and UniXcoder [13]. The advent of large language models like ChatGPT [39], CodeGen [38], StarCoder [29] has introduced new opportunities and challenges to code completion. large models entail a vast number of parameters, which significantly elevates the cost of fine-tuning. Therefore, research on retrieval-based enhancement is essential in this context.

Retrieval-augment Language Model. Retrieval-augmented techniques [15, 21, 31, 32, 45] are primarily categorized into two types: one being retrieval enhancement applied to inputs, also referred to as pre-task retrieval. This category encompasses techniques such as REACC [36], REDCODER [40] and DPR [23] as discussed in previous works. These retrieval techniques necessitate the preliminary segmentation of the data to be retrieved into fixed-length chunks, with each chunk typically containing several hundred tokens. They concatenated the most relevant information to the inputs for the enhancement. Some works go beyond simply retrieving from the original training set, they refine new information from the original training dataset. ASAP [5], in the task of code summarization, uses not just the conventional source code and summary as input but

also incorporates static analysis products such as the repository name, the fully qualified name of the target function, its signature and its data flow graph. RLPG [46] is proposed for single-line code auto-completion in an IDE. RLPG not only retrieves similar content as supplemental input but also utilizes repository-level code context such as Post Lines, Identifiers, Type Identifiers as additional input prompts. Joshi et al. [22] proposes a multi-lingual program repair method named RING. It retrieves relevant buggy-fix examples from an example bank, using the completed bug repair and repair methods as supplemental input prompts. On the other hand, some works also try to focus on retrieval enhancement for outputs such as kNN-LM [24, 52], kNM-LM [47], and RETRO [8]. This kind of retrieval paradigm involves the preliminary creation of a retrieval database, where information from this database is utilized to modify the output generated. For example, RETRO [8] integrates it within the Transformer, while kNN-LM [24, 52] employ probability interpolation at the final probability layer. Compared with these works, we develop a novel retrieval-based approach from theoretical analysis to mimic genuine fine-tuning for code completion.

8 CONCLUSION

In this paper, we introduce a novel retrieval augmentation method for code completion tasks. Guided by a theoretical analysis, we discerned the value of $\Delta logits$ as a pivotal retrieval metric. Building on this revelation, we designed *FT2Ra*, a method that is to simulate the fine-tuning process closely. Similarly, *FT2Ra* incorporates a learning rate and a multi-round iteration strategy, aiming to mirror the results of genuine fine-tuning. The experimental results demonstrated *FT2Ra*'s superiority against state-of-the-art methods and its competitive results with regards to fine-tuning.

9 DATA AVAILABILITY

Our source code and experimental data are available at [3].

REFERENCES
[1] 2022. GitHub Copilot. https://github.com/features/copilot.
[2] 2022. intellicode. https://visualstudio.microsoft.com/services/intellicode.
[3] 2023. ft2ra website. https://sites.google.com/view/ft2ra/home.
[4] 2023. Stanford University CS229: Machine Learning. https://cs229.stanford.edu/. Accessed: 2023-12-10.

[5] Toufique Ahmed, Kunal Suresh Pai, Premkumar Devanbu, and Earl T Barr. 2024. Automatic semantic augmentation of language model prompts (for code summarization). In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 1004–1004.

[6] Miltiadis Allamanis and Charles Sutton. 2013. Mining Source Code Repositories at Massive Scale using Language Modeling. In *2013 10th Working Conference on Mining Software Repositories (MSR)*. IEEE, 207–216.

[7] Uri Alon, Frank Xu, Junxian He, Sudipta Sengupta, Dan Roth, and Graham Neubig. 2022. Neuro-symbolic language modeling with automaton-augmented retrieval. In *International Conference on Machine Learning*. PMLR, 468–485.

[8] Sebastian Borgeaud, Arthur Mensch, Jordan Hoffmann, Trevor Cai, Eliza Rutherford, Katie Millican, George Bm Van Den Driessche, Jean-Baptiste Lespiau, Bogdan Damoc, Aidan Clark, et al. 2022. Improving language models by retrieving from trillions of tokens. In *International conference on machine learning*. PMLR, 2206–2240.

[9] Xiang Chen, Lei Li, Ningyu Zhang, Xiaozhuan Liang, Shumin Deng, Chuanqi Tan, Fei Huang, Luo Si, and Huajun Chen. 2022. Decoupling knowledge from memorization: Retrieval-augmented prompt learning. *Advances in Neural Information Processing Systems* 35 (2022), 23908–23922.

[10] Michiel De Jong, Yury Zemlyanskiy, Nicholas FitzGerald, Fei Sha, and William Cohen. 2021. Mention memory: incorporating textual knowledge into transformers through entity mention attention. *arXiv preprint arXiv:2110.06176* (2021).

[11] Andrew Drozdov, Shufan Wang, Razieh Rahimi, Andrew McCallum, Hamed Zamani, and Mohit Iyyer. 2022. You can't pick your neighbors, or can you? When and how to rely on retrieval in the *k* NN-LM. *arXiv preprint arXiv:2210.15859* (2022).

[12] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).

[13] 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* (2022).

[14] Kelvin Guu, Kenton Lee, Zora Tung, Panupong Pasupat, and Mingwei Chang. 2020. Retrieval augmented language model pre-training. In *International conference on machine learning*. PMLR, 3929–3938.

[15] Junxian He, Graham Neubig, and Taylor Berg-Kirkpatrick. 2021. Efficient nearest neighbor language models. *arXiv preprint arXiv:2109.04212* (2021).

[16] Abram Hindle, Earl T Barr, Mark Gabel, Zhendong Su, and Premkumar Devanbu. 2016. On the naturalness of software. *Commun. ACM* 59, 5 (2016), 122–131.

[17] Sebastian Hofstätter, Jiecao Chen, Karthik Raman, and Hamed Zamani. 2023. Fidlight: Efficient and effective retrieval-augmented text generation. In *Proceedings of the 46th International ACM SIGIR Conference on Research and Development in Information Retrieval*. 1437–1447.

[18] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685* (2021).

[19] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436* (2019).

[20] Gautier Izacard, Patrick Lewis, Maria Lomeli, Lucas Hosseini, Fabio Petroni, Timo Schick, Jane Dwivedi-Yu, Armand Joulin, Sebastian Riedel, and Edouard Grave. 2022. Few-shot learning with retrieval augmented language models. *arXiv preprint arXiv:2208.03299* (2022).

[21] Zhengbao Jiang, Frank F Xu, Luyu Gao, Zhiqing Sun, Qian Liu, Jane Dwivedi-Yu, Yiming Yang, Jamie Callan, and Graham Neubig. 2023. Active retrieval augmented generation. *arXiv preprint arXiv:2305.06983* (2023).

[22] Harshit Joshi, José Cambronero Sanchez, Sumit Gulwani, Vu Le, Gust Verbruggen, and Ivan Radiček. 2023. Repair is nearly generation: Multilingual program repair with llms. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 37. 5131–5140.

[23] Vladimir Karpukhin, Barlas Oğuz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. 2020. Dense passage retrieval for open-domain question answering. *arXiv preprint arXiv:2004.04906* (2020).

[24] Urvashi Khandelwal, Omer Levy, Dan Jurafsky, Luke Zettlemoyer, and Mike Lewis. 2020. Generalization through Memorization: Nearest Neighbor Language Models. In *International Conference on Learning Representations*. https://openreview.net/forum?id=HklBjCEKvH

[25] Seohyun Kim, Jinman Zhao, Yuchi Tian, and Satish Chandra. 2021. Code prediction by feeding trees to transformers. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 150–162.

[26] Brian Lester, Rami Al-Rfou, and Noah Constant. 2021. The power of scale for parameter-efficient prompt tuning. *arXiv preprint arXiv:2104.08691* (2021).

[27] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems* 33 (2020), 9459–9474.

[28] Jian Li, Yue Wang, Michael R Lyu, and Irwin King. 2017. Code completion with neural attention and pointer networks. *arXiv preprint arXiv:1711.09573* (2017).

[29] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023. StarCoder: may the source be with you! *arXiv preprint arXiv:2305.06161* (2023).

[30] Chang Liu, Xin Wang, Richard Shin, Joseph E Gonzalez, and Dawn Song. 2016. Neural code completion. (2016).

[31] Shangqing Liu, Yu Chen, Xiaofei Xie, Jingkai Siow, and Yang Liu. 2020. Retrieval-augmented generation for code summarization via hybrid gnn. *arXiv preprint arXiv:2006.05405* (2020).

[32] Shangqing Liu, Cuiyun Gao, Sen Chen, Lun Yiu Nie, and Yang Liu. 2020. Atom: Commit message generation based on abstract syntax tree and hybrid ranking. *IEEE Transactions on Software Engineering* 48, 5 (2020), 1800–1817.

[33] Shangqing Liu, Yanzhou Li, Xiaofei Xie, and Yang Liu. 2022. Commitbart: A large pre-trained model for github commits. *arXiv preprint arXiv:2208.08100* (2022).

[34] Shangqing Liu, Bozhi Wu, Xiaofei Xie, Guozhu Meng, and Yang Liu. 2023. Contrabert: Enhancing code pre-trained models via contrastive learning. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2476–2487.

[35] Xiao Liu, Yanan Zheng, Zhengxiao Du, Ming Ding, Yujie Qian, Zhilin Yang, and Jie Tang. 2023. GPT understands, too. *AI Open* (2023).

[36] Shuai Lu, Nan Duan, Hojae Han, Daya Guo, Seung-won Hwang, and Alexey Svyatkovskiy. 2022. Reacc: A retrieval-augmented code completion framework. *arXiv preprint arXiv:2203.07722* (2022).

[37] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664* (2021).

[38] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474* (2022).

[39] OpenAI. 2023. ChatGPTblog. https://openai.com/blog/chatgpt.

[40] Md Rizwan Parvez, Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Retrieval augmented code generation and summarization. *arXiv preprint arXiv:2108.11601* (2021).

[41] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.

[42] Ori Ram, Yoav Levine, Itay Dalmedigos, Dor Muhlgay, Amnon Shashua, Kevin Leyton-Brown, and Yoav Shoham. 2023. In-context retrieval-augmented language models. *arXiv preprint arXiv:2302.00083* (2023).

[43] Veselin Raychev, Pavol Bielik, and Martin Vechev. 2016. Probabilistic Model for Code with Decision Trees. *ACM SIGPLAN Notices* (2016), 731–747.

[44] Stephen Robertson, Hugo Zaragoza, et al. 2009. The probabilistic relevance framework: BM25 and beyond. *Foundations and Trends® in Information Retrieval* 3, 4 (2009), 333–389.

[45] Weijia Shi, Sewon Min, Michihiro Yasunaga, Minjoon Seo, Rich James, Mike Lewis, Luke Zettlemoyer, and Wen-tau Yih. 2023. Replug: Retrieval-augmented black-box language models. *arXiv preprint arXiv:2301.12652* (2023).

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

[47] Ze Tang, Jidong Ge, Shangqing Liu, Tingwei Zhu, Tongtong Xu, Liguo Huang, and Bin Luo. 2023. Domain Adaptive Code Completion via Language Models and Decoupled Domain Databases. *arXiv preprint arXiv:2308.09313* (2023).

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

[49] Zhiruo Wang, Jun Araki, Zhengbao Jiang, Md Rizwan Parvez, and Graham Neubig. 2023. Learning to Filter Context for Retrieval-Augmented Generation. *arXiv preprint arXiv:2311.08377* (2023).

[50] Wikipedia. 2023. Empirical Probability. https://en.wikipedia.org/wiki/Empirical_probability.

[51] Wikipedia. 2023. Frequency (statistics). https://en.wikipedia.org/wiki/Frequency_(statistics).

[52] Frank F Xu, Uri Alon, and Graham Neubig. 2023. Why do Nearest Neighbor Language Models Work? *arXiv preprint arXiv:2301.02828* (2023).

[53] Kaiyu Yang, Aidan M Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan Prenger, and Anima Anandkumar. 2023. Leandojo: Theorem proving with retrieval-augmented language models. *arXiv preprint arXiv:2306.15626* (2023).

[54] Fengji Zhang, Bei Chen, Yue Zhang, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023. Repocoder: Repository-level code completion through iterative retrieval and generation. *arXiv preprint arXiv:2303.12570* (2023).