



OPEN ExCS: accelerating code search with code expansion

Siwei Huang^{1,2}, Bo Cai^{1,2}✉, Yaxiang Yu^{1,2} & Jian Luo^{1,2}

Efficiently searching and reusing code from expansive codebases is pivotal for enhancing developers' productivity. In recent times, the emergence of deep learning-driven neural ranking models, characterized by their vast dimensions and intricate interaction mechanisms, has been noteworthy. Yet, these models, in real-world scenarios, pose computational challenges due to their high dimensionality. Moreover, models rooted in interaction necessitate querying every piece of code within a voluminous corpus. While these methodologies offer superior accuracy, their online retrieval process is considerably more time-consuming compared to traditional Information Retrieval (IR) techniques. Addressing this, we introduce "ExCS", an innovative code search tool designed to expedite the code search process without compromising on accuracy. ExCS innovatively employs code expansion in its offline phase, leveraging predictions on potential queries for specific codes, thereby enriching the code's semantic depth. During online retrieval, ExCS prioritizes IR-based methods to pinpoint a concise set of persuasive candidates. Our evaluations, conducted on the Java dataset from CodeSearchNet, reveal that ExCS achieves a remarkable 90% reduction in retrieval duration while maintaining an impressive 99% retrieval accuracy.

Code search is a fundamental task in software development, facilitating troubleshooting, code reuse, and understanding. Estimates suggest developers allocate around 15% of their time to search-related tasks. This statistic accentuates the necessity for proficient code search tools, which could substantially boost development productivity and trim costs¹.

Navigating through vast codebases to locate desired code snippets is a challenging endeavor, primarily due to the semantic disparity between natural language queries and code^{2–4}. A variety of strategies have been probed to surmount this challenge, encompassing information retrieval (IR)-based methods and deep learning-based techniques. While the former boasts speed, they stumble when it comes to bridging the semantic chasm. On the flip side, deep learning methods shine in unraveling intricate query-code relationships, albeit at the cost of computational efficiency owing to the large dimensions of the models and complex mechanisms employed^{5–13}.

Historically, the bulk of research efforts have been channeled towards ameliorating retrieval accuracy, leaving a void in the domain of retrieval efficiency, particularly on extensive code bases. The crux of deep learning approaches for code search revolves around direct ranking of the entire corpus of source code snippets during the search process, which inevitably spirals into high computational expenses. Strategies utilizing separate representation vectors for code and description necessitate the computation of similarity between the target query vector and all code representation vectors in the corpus for each retrieval, setting a high bar for the dimension of representation vectors to achieve commendable retrieval accuracy. For instance, models like DeepCS invariably fix the final representation vector dimension at 512, meaning that the similarity computation between a code and query vector demands 512 multiplications and 512 additions with double data type variables for each pair. The total calculation for a single linear scan of a corpus with 1 million code snippets is staggering, encompassing around 1 billion multiplications and additions. The situation is further exacerbated by approaches utilizing binary classification, which, although absolve the need for pre-storage of representation vectors, necessitate real-time inference of target token sequences alongside all description token sequences for every retrieval. This real-time requirement considerably bloats the computational cost, given the hefty number of parameters in contemporary deep learning models. Moreover, models employing interaction mechanisms like TabCS and SANCS are compelled to interact with all code documents for each query, resulting in an astronomical computational cost. Nonetheless, IR methods can supply a small-scale set of candidate codes to neural ranking models, accelerating the retrieval process significantly. This expedited approach, however, has its accuracy tethered to the performance of the IR methods, thus illuminating the imperative for augmenting these methods.

¹School of Cyber Science and Engineering, Wuhan University, Wuhan 430072, China. ²Key Laboratory of Aerospace Information Security and Trusted Computing, Ministry of Education, Wuhan 430072, China. ✉email: caib@whu.edu.cn

Against this backdrop, we put forth CodeEx, an innovative sequence-to-sequence model crafted to alleviate the “vocabulary mismatch” dilemma. Envisioned as a question-answering mechanism, CodeEx predicts plausible queries for a given code and spawns corresponding expansions, thereby enriching the semantic association between code and queries. This nuanced tactic not only facilitates a more potent bridging of the semantic gap but also elevates the accuracy of IR methods, enabling them to furnish more reliable candidate codes to the succeeding neural ranking models, thereby forging a synergistic framework for advanced code search.

When dealing with large codebases, such as those on GitHub, it is only practical to use IR methods for code search¹⁴. To address this, we introduce ExCS, a groundbreaking code search tool that combines CodeEx with IR to enhance the efficiency of existing deep learning methods. ExCS scales on the initial search by leveraging IR methods. ExCS then applies deep learning models exclusively for re-ranking the shortlisted candidates, merging the efficiency of IR with the precision of neural models. This approach makes the search process both practical and scalable, even in extensive codebases. In its offline stage, ExCS utilizes CodeEx to generate code expansions from the code corpus, indexing the keywords from these expansions. During the online phase, ExCS leverages IR to extract the code candidates, which are then re-ranked using neural ranking models. This strategy keeps the time cost of deep learning-based code search within a manageable range while maintaining accuracy. Figure 1 clearly demonstrates the positive impact of ExCS on various code search methods. By comparing standard models with their ExCS-enhanced versions, the graph illustrates the balance between Mean Reciprocal Rank (MRR) and search speed, highlighting how ExCS improves both efficiency and accuracy in code search. Our contributions are encapsulated as follows:

- We unveil a unique deep neural network, CodeEx, aimed at enriching code with pertinent natural language (NL) sequences by anticipating likely queries.
- We engineer ExCS, a tool that capitalizes on CodeEx and IR to refine the efficiency of prevailing deep learning-based methods.
- We undertake a comprehensive experimental evaluation on public benchmarks, demonstrating that ExCS substantially uplifts retrieval efficiency while maintaining comparable accuracy to baseline models, thus marking a significant stride towards resolving the efficiency-accuracy trade-off in code search.

Background and motivation

Code search

Code search aims to remedy the semantic disconnection between queries in Natural Language (NL) and codes in the programming language. The so-called semantic gap originates from the “vocabulary mismatch” in discrete thermal representation and far distance in continuous vector representation. Various code search methods have been proposed^{6,7,9,15–23} to address this issue. Early methods^{16,22,23} employed Information Retrieval (IR) techniques to return results aligned with the query intent. They tackled the issue by enhancing the query with related keywords. For example, CodeHow, a state-of-the-art IR-based model, utilizes related APIs to expand user queries and improve query results. Although deep learning’s assistance is absent in these approaches, they compensate with the maturity and variety of methodologies and techniques from the field of IR²⁴. Nonetheless, as we discussed, IR alone is insufficient in bridging the semantic gap between code and NL¹.

With the surge of deep learning, numerous methods have integrated it into code search. Such models frequently utilize the common technique of joint embedding^{25–27}, which collectively embeds diverse data types into a unified vector space, causing semantically similar concepts (across different modalities) to cluster in nearby regions of the vector space. According to²⁸, code search models are of two types, based on their architecture—siamese²⁹ and interaction-based. In the siamese type, codes and queries are processed independently. Gu et al.⁷ were the pioneers in introducing deep learning to code search, proposing DeepCS which embeds code and

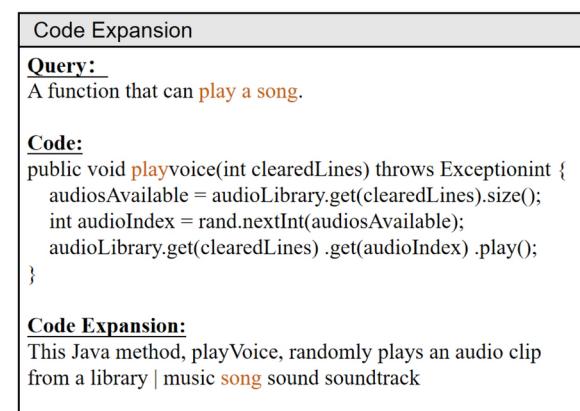
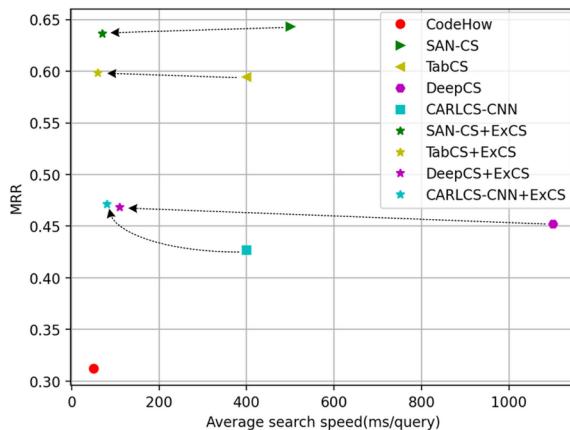


Fig. 1. The left figure depicts an Accuracy/Efficiency Map juxtaposing various code search models against their ExCS-enhanced versions. The diagram underlines the equilibrium between MRR and search speed, spotlighting the potential of assimilating ExCS to optimize both efficiency and accuracy in code search. And the right figure shows an example of code expansion.

query into vectors through LSTM. Other models under this type also leverage Language Models (LMs) due to their simplicity. On the other hand, interaction-based models, such as CARLCS-CNN⁹ and TabCS¹⁸, generate an interaction matrix to evaluate the relevance between the representations of query and code. For instance, CARLCS-CNN uses convolutional neural networks and LSTM to learn code and query embedding. With self-attention³⁰ gaining widespread usage in the NLP field, TAB-CS has incorporated a self-attention mechanism to embed code and query. Similarly, SAN-CS¹⁹ applies a global-attention mechanism to map the relationship between code and query. Notably, all the state-of-the-art models use interaction-based methods, effectively bridging the semantic gap between codes and queries.

Recent developments have introduced Large Language Models (LLMs) like StarCoder³¹, which use extensive pre-training and fine-tuning to help with code search and code generation tasks. Another notable LLM is Code Llama³², which helps with coding tasks by understanding natural language queries and providing related code snippets. The large number of parameters and extensive training data in these LLMs help them understand the subtle meanings between code and natural language, which is key to addressing the vocabulary mismatch problem. However, these models work in a generative way³³. So, when users use these LLMs for code search, it's more like they are doing code generation. A more practical way of code search, similar to what's done in AutoGPT³⁴, is to use search tools like Google or the vector library in Langchain³⁵ to find related content on web pages, which is then given to the LLMs for further processing. This way, the performance of the search tools also affects how well code search works with LLMs.

Efficiency in code search

In the spirit of detail, we shall analyze the time-consuming-operations in code search. The search methods previously discussed fall into two categories - those based on IR and those based on deep learning, which can further be classified by whether they incorporate an interaction mechanism. In IR-based code search, the principal time-consuming operations include computing the similarity score between query q and code document set $D = [d_1, \dots, d_n]$, and sorting to return the top-k results. BM25 is the commonly used approach to calculate scores. Following the similarity calculation:

$$Score(q, d) = \sum_{t \in Q} \log\left[\frac{N}{df_t}\right] \cdot S(q_i, d) \cdot \frac{(k_1 + 1)tf_{tq}}{k_1 + tf_{tq}} \quad (1)$$

$$S(q_i, d) = \frac{(k_2 + 1)tf_{td}}{k_2[(1 - b) + b \times (L_d/L_{ave})] + tf_{td}}, \quad (2)$$

where d is a document in D , N is the number of documents in D , df_t is the number of documents that contain term t , k_1 and k_2 are fixed parameters, tf_{xy} denotes the number of occurrences of x in y and L_d is the length of the corresponding document d . L_{ave} is the average length of all documents. The time complexity of BM25 is:

$$O(BM25) = O(|q| \cdot |D| \cdot |T|), \quad (3)$$

where $|\cdot|$ is the mod function, T is the set of entities in the vocabulary of code document set D . It is necessary to sort the similarity scores to obtain the top-k list. Because all code search methods require this step, we use the universal quicksort in reordering, with a time complexity of $O(n \log n)$.

For siamese model, codes and queries are processed independently, so the time-consuming operations for searching encompass embedding the query q , computing the cosine similarity between the query embedding q and the stored code embedding set $C = [c_1, \dots, c_n]$ that had been previously calculated and stored during the offline phase, and finally sorting to return the top-k result list. Embedding queries depend on the embedding models. For example, the time complexity of widely-used LSTM and self-attention are calculated as follows:

$$O(LSTM) = O(|q| \cdot |q|^2) \quad (4)$$

$$O(Self - attention) = O(|q|^2 \cdot |q|) \quad (5)$$

For the query q is usually much shorter than embedding size $|q|$, LSTM takes longer embedding time than self-attention. Then, the formula for calculating cosine similarity and the search time complexity is as follows:

$$CosSimilarity(q, c) = \frac{q \cdot c}{\|q\| \cdot \|c\|} \quad (6)$$

$$O(CosSimilarity) = O(|q| \cdot |d|) \quad (7)$$

$$O(Search_{siamese}) = O(|C| \cdot |q| \cdot |d|) \quad (8)$$

Interaction-based models, on the other hand, necessitate three steps: jointly embedding codes and queries, calculating similarity, and sorting. However, as code representations must be calculated through joint embedding at search time, the embedding step cannot be deferred, unlike in siamese models. The time complexity of jointly embedding depends on the embedding models like (4) and (5). The similarity calculation is the same as (6). The search time complexity is as follows:

$$O(\text{search}_{\text{interaction}}) = O(|C| \cdot |q| \cdot |d| \cdot \text{Emb}), \quad (9)$$

where Emb is the dimension of the jointly embedding time complexity.

Motivation

The previous sections demonstrate that evaluating every piece of code against a query can significantly enhance the effectiveness of code search. However, this approach is impractical when dealing with large codebases, such as those found on GitHub. Our objective is to accelerate this process by reducing the amount of code that needs to be examined. By doing so, we can leverage powerful yet computationally intensive neural networks to re-rank a smaller set of code snippets initially identified by a term-based search engine.

To achieve this, we focus on improving how code is represented before it is indexed—using language models, for example, to better capture the semantic meaning of the code. This allows us to maintain the efficiency of the initial term-based search while enhancing its ability to retrieve relevant code snippets. The retrieved results can then be re-ranked by a more advanced neural ranking model, such as SANCS, to ensure accuracy.

This two-step method—first rapidly identifying a smaller set of potential matches and then re-ranking them for precision—aims to strike a balance between speed and accuracy, which is crucial for practical code search tools. This approach is particularly beneficial for large codebases, where scanning all the code for each query would be prohibitively time-consuming. By implementing this method, we aim to accelerate the code search process while maintaining or even improving the accuracy of the results, making code search more practical and effective for real-world software development scenarios.

Methods

In this section, we introduce the proposed code expansion model, CodeEx, and the related code search tool, ExCS. The following subsections present the method details. Specifically, Section III-B describes the details of CodeEx, and Section III-C describes the two stages of the proposed ExCS.

CodeEx

Inspired by document expansion, we propose a fine-tuned code expansion model named CodeEx (Code Expansion). Figure 2 illustrates the overall structure of CodeEx. CodeEx expands the code using two approaches: overall expansion and keyword expansion. Formally, for a given code c , the process is denoted as follows:

$$E = RD([E_o : E_k]), \quad (10)$$

where E is the output expansion of CodeEx, E_o is the expansion generated by overall expansion, and E_k is the expansion generated by keyword expansion. $RD()$ is the function that removes repeated terms in the sequence to shorten the expansion length.

Recently many state-of-the-art natural language processing LMs have been adapted to code while their computation costs are impractical on online code retrieval. However, CodeEx can perform term representation with LMs in the offline stage to save the time costs of the online stage. Since these two approaches are sequence-to-sequence tasks, we take them as downstream tasks of an encoder-decoder LM on code, CodeT5. CodeT5

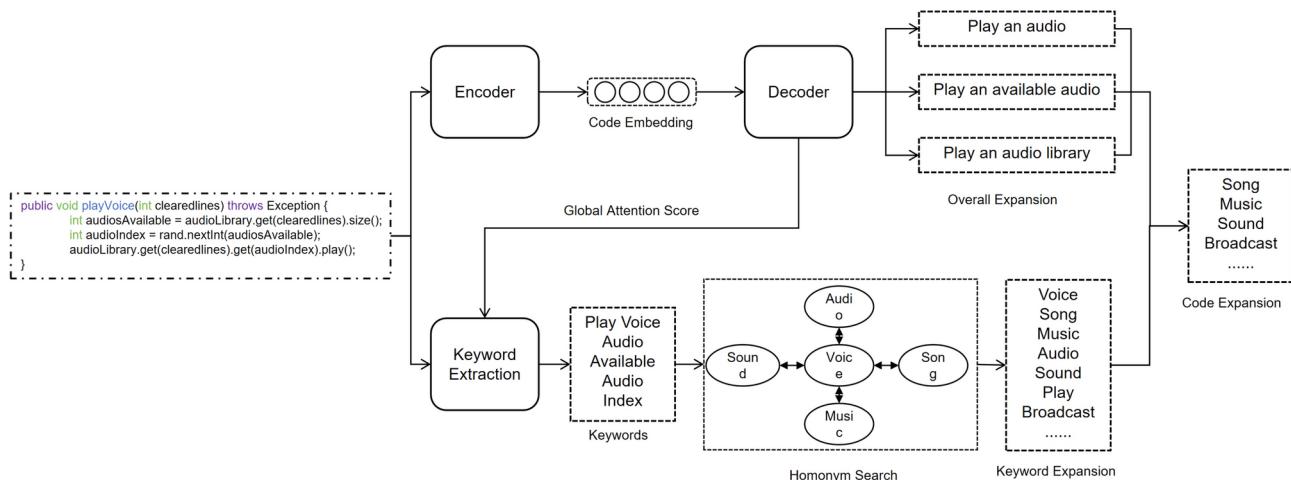


Fig. 2. The overall structure of CodeEx is illustrated using a Java code script as an example. In the upper part, the overall expansion process involves passing the code through an encoder-decoder structure to obtain a semantic expansion for the term 'play voice.' Utilizing global attention during this expansion, CodeEx extracts keywords from the code snippet and performs a homonym search to further expand these keywords. Finally, the two expansions are combined to produce the complete code expansion.

leverages a state-of-the-art LM, T5, to pre-train an identifier-aware code understanding network. Compared to encoder-only models, like CodeBert, and decoder-only models, like PolyCoder, CodeT5 performs better generation tasks thanks to its encoder-decoder structure. The following subsections describe the detailed design of these modules.

- (1) Overall expansion Overall expansion is a cross-modal sequence-to-sequence task aimed at obtaining the NL (Natural Language) sequences that match the input code semantics. Overall expansion takes the entire code tokens as input and outputs sequential tokens.

For fine-tuning, we adopt CodeT5 for the overall expansion task. As depicted in Fig. 3, the input to CodeT5 is a sequence of tokens. We first utilize tree-sitter to extract the annotations as labels and tokenize the code snippets following the implementation in CodeT5. Then, we input each code with a delimiter token as $input = ([CLS], c_1, \dots, c_n)$, where c_n represents the tokens in the code. We also adopt the beam search algorithm³⁶, to keep the top-B best hypotheses at each time step t , where B is the beam size. With beam search, we obtain the top-k possible generations. We concatenate the generations and remove the duplicated tokens as output, denoted as follows:

$$[s_1 : \dots : s_k] = \text{Sample}(\text{CodeT5}(c), k) \\ E_o = [s_1 : \dots : s_n], \quad (11)$$

where Sample is the function to return the top-k possible generation samples, and s_k represents the k th possible sample.

- (2) Keyword expansion Keyword expansion aims to supplement the overall expansion from the perspective of code tokens, which means keyword expansion is a code-to-code task. In contrast, overall expansion is a code-to-NL (Natural Language) task. As discussed in Section ii-C, method name, variable name, and API are artificially defined, although they are code features, making them more like NL. These features are essential because they are usually mentioned in the query. Compared to overall expansion, which generally summarizes these features in one word, keyword expansion tends to retain the variety of code features. The keyword refers to the tokens in the code that are semantically related to the query, such as the method

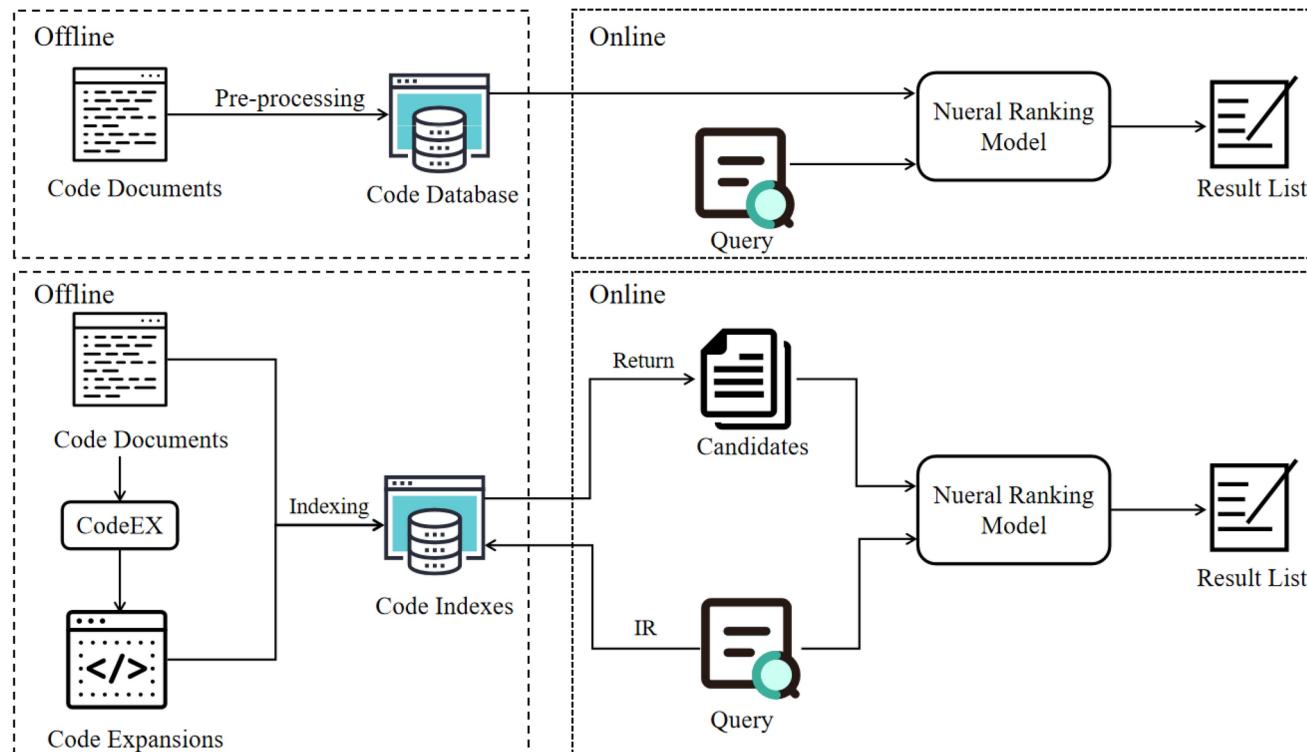


Fig. 3. This picture depicts two methods of code search. The top part illustrates the common practice where, in the offline phase, code is pre-processed and stored in a database. During the online phase, the database and query are inputted into a neural ranking model for searching. On the other hand, the bottom part illustrates the ExCS method where, in the offline phase, code is processed through CodeEX to generate Code Expansion. Then index them and store the indexes in an index database. During the online phase, the query is first processed through IR methods in the index database to retrieve candidates. Subsequently, the query and candidates are inputted into the ranking model for searching.

name, API name, and variable name. The keyword expansion comprises two steps: keyword extraction and homonym search.

Keyword extraction

Initially, we extract all method names, API names, and variable names from the code tokens. Then, after the decoding of the overall expansion, we calculate the relation score between the above tokens $T = [t_1, \dots, t_n]$ and the first sample s_1 using a global-attention mechanism as follows:

$$\begin{aligned} Q_t &= t_n \cdot W_Q, \\ K_s &= s_1 \cdot W_K, \\ R &= \sum \left(\text{softmax} \left(\frac{Q_t \cdot K_s^T}{\sqrt{d}} \right) \right), \end{aligned} \quad (12)$$

where t_n and s_1 are the representation vectors learned from the overall expansion of t_n and s_1 , W_Q and W_K are the weighted matrices, Q_t and K_s are the feature matrices of t_n and s_1 , $\text{softmax}()$ is the softmax function, and $\sum()$ is a function that summarizes the items. R reflects how much the token is related to the generation. We select the top-k tokens above a threshold value as the keywords.

Homonym search

Subsequently, we identify the homonyms of the keywords. For the terms in the code vocabulary, we represent them in vectors using the encoder model. According to word2vec³⁷, the vectors represent the semantics of each term, and the distance between two vectors can reflect their semantic similarity. For the keywords, we find the top-k closest terms, also known as the homonyms, to expand the code.

For the loss function, we aim for the output to cover more terms in the query while maintaining grammatical correctness, thus we use a fused loss of cross-entropy and BLEU-N as the training objective. We adopt a multi-task approach. The loss function of a training sample is calculated as follows:

$$L_1 = \text{BLEU} - N(o, l) = BP \cdot \exp \left(\sum_{n=1}^N \omega_n \log p_n \right), \quad (13)$$

$$L_2 = \text{CrossEntropy}(o, l), \quad (14)$$

$$L = k_1 \cdot L_1 + k_2 \cdot L_2, \quad (15)$$

where p_n is the precision score of the n-gram matches between output o and label l , BP is the brevity penalty, and ω_n is the uniform weight $\frac{1}{N}$. $\text{CrossEntropy}()$ is the cross-entropy loss function. L_1 , L_2 , and L are the first, second, and final loss, and k_1 and k_2 are the hyperparameters.

In this section, we've provided a detailed methodology for our proposed CodeEx model and ExCS tool. The subsequent sections will delve into the implementation and evaluation of these methodologies.

ExCS

Figure 3 depicts the overall framework of ExCS which implements code search through offline and online stages. The offline stage utilizes CodeEx to generate code expansions for each code in the corpus respectively and indexes the expanded codes. The online stage comprises two steps: obtaining candidate codes via Information Retrieval (IR) methods, and re-ranking the candidates utilizing neural ranking models. Upon receiving a user query, ExCS initially employs the term-based IR method BM25 on the stored indexes to retrieve a ranked list of codes. Lastly, ExCS re-ranks the codes using neural ranking models.

- (1) Offline stage In the offline stage, ExCS stores the indexes of expanded codes and the code embedding for Siamese neural ranking models. Initially, ExCS applies CodeEx on the code stored in the corpus to obtain expanded codes \hat{C} . The expansions are placed as annotations at the beginning of the code snippets since annotations can enrich the code semantics without altering the code structure. Following this, we filter the expanded codes by removing stop words and operations in Java, like “if”, as these terms are not beneficial when the query is in Natural Language (NL), and as per (1), longer documents result in lower BM25 scores. A simple experiment showed that term-based IR methods perform better on the filtered codes. After filtering, we obtain the final code documents $\bar{C} = [\bar{c}_1, \dots, \bar{c}_n]$.

For the Siamese neural ranking models, like DeepCS, ExCS will embed the codes during the offline stage using the code encoder in the model. For instance, in DeepCS, three features are extracted from a code c : method name (a list of camel split tokens), API sequence (a list of API words in the method body), and tokens (a bag of words in the method body and annotation). DeepCS embeds the method name $M = [m_1, \dots, m_n]$ and API sequence $A = [a_1, \dots, a_n]$ with LSTM, and embeds the tokens $T = [t_1, \dots, t_n]$ with MLP. The vectors of the three features are then fused into one vector c as follows:

$$\begin{aligned}
 \mathbf{m} &= LSTM_m(\mathbf{M}), \\
 \mathbf{a} &= LSTM_a(\mathbf{A}), \\
 \mathbf{t} &= MLP_c(\mathbf{T}), \\
 \mathbf{c} &= \tanh(MLP_{\text{fuse}}([\mathbf{m}; \mathbf{a}; \mathbf{t}])),
 \end{aligned} \tag{16}$$

where \mathbf{m} , \mathbf{a} , \mathbf{t} are the embedded vectors of \mathbf{M} , \mathbf{A} , \mathbf{T} , $\tanh()$ is the activation function, and $[\mathbf{m}; \mathbf{a}; \mathbf{t}]$ represents the concatenation of the three vectors.

Following code embedding, we store the code documents as $d = \{“code”: \bar{c}, “codeembedding”: c\}$, where \bar{c} is the filtered code. For interaction-based models, like SANCS, we simply store the code document as $d_t = \{“code”: \bar{c}\}$. The code documents are then indexed and stored.

(2) Online stage When a developer submits a NL query q , the online stage follows two steps: IR-based filtering, and deep learning-based re-ranking.

IR-based filtering

Initially, ExCS performs a code search with BM25 on the codebase, selecting the top-k results as candidates and obtaining the relevant code documents $\mathbf{C}_k = [c_k^1, \dots, c_k^k]$.

Deep learning-based re-ranking

Upon obtaining \mathbf{C}_k , we regard it as a new, albeit smaller, codebase to execute neural ranking models on it.

For the Siamese neural ranking models, the code representation \mathbf{c} has already been stored in the code document. Hence ExCS needs to embed the query q into vector \mathbf{q} , denoted as follows:

$$\mathbf{q} = W_q \cdot q, \tag{17}$$

where W_q is the matrix of query encoder parameters of the neural ranking models. Subsequently, the re-ranking process calculates the cosine similarity between the query vector \mathbf{q} and the candidate code vectors $[c_k^1, \dots, c_k^k]$ respectively, and sorts the candidates to return the result list.

For interaction-based models, ExCS first constructs a data loader with the three code features: method name, API sequence, and tokens. Models like TabCS and SANCS initially embed the code features into vectors using self-attention, respectively, and fuse them into a code feature matrix \mathbf{C} prior to the interaction mechanism, denoted as follows:

$$\begin{aligned}
 \mathbf{m} &= \text{SelfAtt}_m(\mathbf{M}), \\
 \mathbf{a} &= \text{SelfAtt}_a(\mathbf{A}), \\
 \mathbf{t} &= \text{SelfAtt}_c(\mathbf{T}), \\
 \mathbf{C} &= [\mathbf{m}; \mathbf{a}; \mathbf{t}],
 \end{aligned} \tag{18}$$

Then, these models leverage interaction-based mechanisms to attain further representations of code and query, and calculate the similarity. For instance, TabCS employs a fuse matrix \mathbf{F} to fuse the representations as follows:

$$\begin{aligned}
 \mathbf{F} &= \tanh(\mathbf{C}^T \mathbf{U} \mathbf{q}), \\
 \tilde{\mathbf{c}} &= MLP_{\tilde{c}}(\text{maxpooling}(\mathbf{F})), \\
 \tilde{\mathbf{q}} &= MLP_{\tilde{q}}(\text{maxpooling}(\mathbf{F}^T)), \\
 \text{sim} &= \cos(\tilde{\mathbf{c}}, \tilde{\mathbf{q}}),
 \end{aligned} \tag{19}$$

where $\tilde{\mathbf{c}}$ and $\tilde{\mathbf{q}}$ are the new vectors of code and query after interaction.

Finally, the models recommend a code list wherein the k th code is the k th possibly corresponding code for the query. This structured approach facilitates a streamlined process from query input to code recommendation, ensuring that the most relevant code snippets are effectively retrieved and ranked according to their relevance to the given query.

Experiment

This section delineates the research questions(RQs) under investigation, the baseline models chosen for comparison, the evaluation metrics employed, and the experimental setup and results.

Research questions

The core focus of our experiments revolves around the following RQs:

- RQ1: Can our proposed method, ExCS, enhance model accuracy and efficiency?

RQ1 examines the performance enhancement of each baseline model post adaptation to ExCS. The baseline models are elucidated in the ensuing section.

- RQ2: What impact do different components of ExCS have on accuracy?

ExCS incorporates various elements such as the number of candidates pre-re-ranking, the length of code expansion, and the incorporation of new words and copied words in code expansion, which are pivotal in code search. We orchestrate comparative experiments adjusting these parameters to discern their influence on model accuracy and to ascertain the optimal settings.

- RQ3: How does ExCS augment efficiency?

RQ3 delves into the time expenditure across different code search phases. Given that ExCS impacts similar model types uniformly, we select the best-performing models from the Siamese network and interaction-based network categories, namely DeepCS and SANCS, for experimentation. We juxtapose the time cost pre and post ExCS utilization to gauge the efficiency enhancement rendered by ExCS.

Baselines

The baseline models, selected from state-of-the-art neural ranking models, are categorized based on their structure and deep learning methodologies.

BM25: BM25, a pivotal ranking function in Information Retrieval (IR), excels in ranking documents in large text collections. Utilizing the Anserini open-source IR toolkit, we harness BM25's prowess for document indexing in our experiment²⁸.

DeepCS: DeepCS employs a Siamese network and joint embedding to facilitate effective code retrieval, applying RNN to method names, API sequences, and MLP on code tokens to generate representative vectors for both code and queries.

CARLCS-CNN: CARLCS-CNN, an interaction-based code search model, employs LSTM and CNN for individual code and query embedding before leveraging a fusion matrix to derive interdependent representations, fostering nuanced code-query relationships.

TabCS: TabCS, with its two-stage Attention-based network, extracts code and query semantics individually, then employs a co-attention mechanism, akin to CARLCS-CNN, to discern semantic correlation between code and query, enhancing code search accuracy.

SANCS: SANCS leverages self-attention networks to construct a comprehensive code-description network and a method for code and description representation along with a similarity measure module, providing a nuanced representation of code and queries.

Evaluation metrics

To meticulously evaluate model accuracy, we adopt two widely acknowledged metrics, namely SuccessRate and Mean Reciprocal Rank (MRR), following the methodology proposed by Xu et al.¹⁸.

SuccessRate@k(SR@k), the proportion of queries in which the relevant code answer can be found in the top-k result list, is computed as follows:

$$\text{SuccessRate}@k = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \delta(FRank_{Q_i} \leq k), \quad (20)$$

where Q is the query set in the evaluation and $FRank_{Q_i}$ is the rank of the relevant answer for the i th query in Q . δ is an indicator function that returns 1 when the $FRank_{Q_i}$ is within the top k returning results. Otherwise, it returns 0.

MRR, the average of the reciprocal ranks of the first hit results¹² of a set of queries Q , is calculated as follows:

$$MRR = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \frac{1}{FRank_{Q_i}}, \quad (21)$$

For the model efficiency, we employ the average search time as a metric.

Average Search Time, the average time spent from entering a query to getting the result list, measured in milliseconds per query (ms/query), provides a tangible measure of efficiency.

Dataset

We conduct our Experiments on the java dataset provided by CodeSearchNet(CSN)³⁸. CSN is a collection of datasets and benchmarks for semantic code retrieval. The datasets generated and/or analysed during the current study are available in the `code_search_net` repository, https://huggingface.co/datasets/code_search_net. It extracts functions and their paired comments from GitHub repositories. It covers six programming languages, and we elect to use the training dataset for Java, which encompasses 394,471 data points. Following the implementation blueprint by Sun et al.³⁹, we filter the dataset to eliminate repeat and poor-quality data, resulting in a refined dataset of 192,031 data points for training. The evaluation and test data remain unaltered from CSN, comprising 4009 and 26909 data points, respectively.

Model configuration

We conduct fine-tuning on the codet5 base multi sum (220 m) variant of CodeT5 [13] from Huggingface to realize CodeEx. For the overall expansion, we set the maximum source and target sequence lengths to 128 and 16, respectively. The batch size is set to 16, and a peak learning rate of 2e-4 with linear attenuation is adopted. We aim to obtain the top-5 samples and employ BLEU-1 for the loss function as per equation (15). For the keyword expansion, the top 5 keywords are selected. The pre-training regimen is executed on four Nvidia Tesla V100 clusters, each with 16G memory. The entire training duration for fine-tuning spans three days.

Results

This section delves into the examination of the three RQs outlined in Sect "Research questions".

RQ1: Performance Evaluation of ExCS in Terms of Accuracy and Efficiency

Table 1 provides a comparative analysis concerning the accuracy and efficiency between the baseline code search models described in Section III and their corresponding adaptations with ExCS. The term "BM25+CodeEx" denotes the IR-based filtering component during the offline stage in ExCS. Symbols " \uparrow " and " \downarrow " denote an increase or decrease in the metric when ExCS is integrated with the baseline model, respectively, with the following percentages indicating the extent of change relative to the baseline models.

The model efficiency is notably enhanced, with the average search time being reduced to under 100 ms/query when the neural ranking models are adapted to ExCS. This results in an average acceleration of 87.0% in comparison to the baseline models. This efficiency boost is attributed to the neural ranking models operating on a narrowed down codebase of 1000 entries, which significantly reduces the retrieval time. The BM25+CodeEx component accounts for approximately 80% of the time spent processing the large-scale codebase to generate candidates.

In terms of model accuracy, the fundamental component of ExCS, BM25+CodeEx, exhibits a 4.77% decrement in SR@1 performance while showcasing a significant enhancement of 19.07%/37.53% in SR@5 and SR@10 performance compared to MRR. In contrast, other methods exhibit an average increment of 2.39%/20.80%/38.68% in SR@1, SR@5, and SR@10 compared to MRR. The decrement in SR@1 is reasoned to be due to the expansion terms not present in the query, which diminishes the similarity measure between the correct code and the query. As the expansion increases the code document length and for other code documents, the expansion adds related terms to the query. According to (1), the longer the document and the more frequent the term showing in other documents will decrease the BM25. So the possibility of the correct answer showing in the first place decreases while the SR in a range remains normal. Also, We find that BM25+CodeEx outperforms DeepCS by 4.7%/3.3%/1.7%/11.0% in terms of SR@1/5/10, and MRR. This result shows that even without neural re-ranking, the IR-based search method can also get good accuracy with CodeEx.

For the baseline neural ranking models, the results demonstrate that the ExCS can preserve the accuracy or even improve the accuracy of the baseline models. For DeepCS, ExCS with DeepCS outperforms the baseline model by 14.2%/15.2%/14.1%/15.0% in SR@1/5/10, and MRR. The increment in accuracy is mainly due to the semantic enrichment by the expansion. By adding NL expansion, although DeepCS is based on the siamese network and LSTM is not as powerful as self-attention, DeepCS can get the semantic relationship between query and codes. For CARLCS-CNN, ExCS with CARLCS-CNN outperforms the baseline model by 11.5%/13.3%/12.1%/10.3% in SR@1/5/10, and MRR. The increment decreases by an average of 25.2% compared to DeepCS. It is because the CARLCS-CNN has an interaction mechanism, which reduces the additional advantage of enriched codes. For TabCS, ExCS with TabCS outperforms the baseline model by 0.5%/0.5%/0.6%/0.7% in terms of SR@1/5/10, and MRR. The increment becomes relatively smaller than above. We can find that the better the code query relationship capture is the baseline model, the less the code expansion can help improve the accuracy. For SANCS, ExCS with DeepCS underperforms the baseline model by 1.1%/2.2%/1.2%/1.1% in SR@1/5/10, and MRR. The accuracy decreases with ExCS. It is due to the candidates can not cover all the correct codes. So the advantage of semantic enrichment is less than the disadvantage of not having all the correct codes in candidates. Since ExCS must first use IR methods for the initial search to be applicable to large-scale codebases, the more precise the retrieval model, the greater the accuracy loss during this step. However, more precise models often require larger parameter counts or additional interaction mechanisms, which means they take longer to execute. Therefore, ExCS offers a trade-off where some accuracy is sacrificed to make the model usable.

Model	SR@1	SR@5	SR@10	MRR	Search time
BM25+CodeEx	0.4056	0.5386	0.6220	0.4523	50 ms/query
DeepCS	0.4125	0.5214	0.6173	0.4072	1100 ms/query
ExCS with DeepCS	0.4712 (\uparrow 14.2%)	0.6061 (\uparrow 15.2%)	0.7037 (\uparrow 14.1%)	0.4685 (\uparrow 15.0%)	110 ms/query (\downarrow 90.0%)
CARLCS-CNN	0.4331	0.5241	0.6335	0.4271	400 ms/query
ExCS with CARLCS-CNN	0.4830 (\uparrow 11.5%)	0.5922 (\uparrow 13.3%)	0.7095 (\uparrow 12.1%)	0.4713 (\uparrow 10.3%)	80 ms/query (\downarrow 80.0%)
TabCS	0.6134	0.7145	0.8056	0.5946	400 ms/query
ExCS with TabCS	0.6168 (\uparrow 0.5%)	0.7179 (\uparrow 0.5%)	0.8103 (\uparrow 0.6%)	0.5986 (\uparrow 0.7%)	60 ms/query (\downarrow 85.0%)
SANCS	0.6628	0.7509	0.8369	0.6435	500 ms/query
ExCS with SANCS	0.6537 (\downarrow 1.1%)	0.7341 (\downarrow 2.2%)	0.8280 (\downarrow 1.2%)	0.6367 (\downarrow 1.1%)	70 ms/query (\downarrow 86.0%)

Table 1. Model effectiveness and model efficiency without and with ExCS.

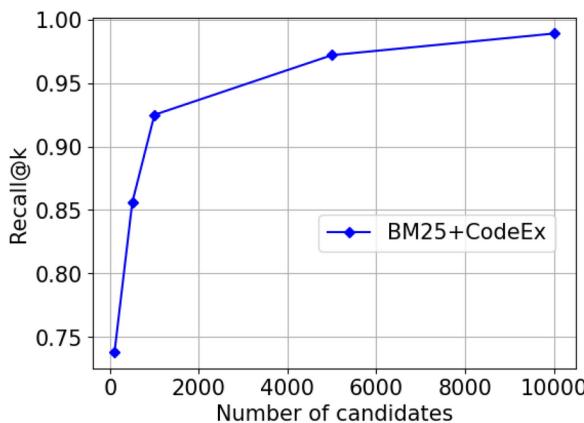
RQ2: Analyzing the Impact of Different Components of ExCS on Effectiveness

This subsection conducts a series of controlled experiments to dissect the influence of different components within ExCS—namely, the number of candidates pre-re-ranking, the extent of code expansion, and the distribution of new versus copied words in the code expansion. Given DeepCS's sensitivity to the alterations in the accuracy of BM25+CodeEx and the expansion, it was chosen as the baseline model for this segment of the analysis.

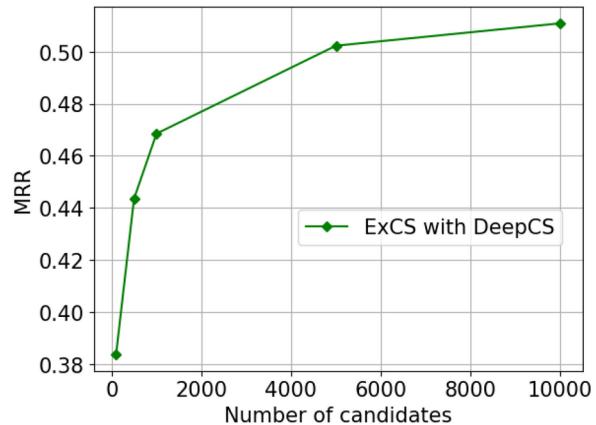
For the number of candidates, it affects the possibility the correct code be covered in the candidates and the time cost of the neural re-ranking. Figure 4a shows the Recall@k of the BM25+CodeEx, where k is the number of candidates and Recall@k represents the proportion of the number of correct documents in the top-k results to the number of all documents. We can find the Recall@k increases rapidly from 0.738 to 0.925 from 100 to 1000, and the increment goes slow from 0.925 to 0.989 from 1000 to 10,000. It is due to when k gets bigger, more correct codes with relatively low BM25 will be covered. However, when more correct codes are covered, the rest correct codes have a more negligible correlation to the query, in which case, we need to set way bigger k to obtain little correct codes. From Fig. 4b, we find the trend of MRR of ExCS with DeepCS is like the trend in Fig. 4a. It is because the accuracy of ExCS with DeepCS mainly relies on the Recall@k of the BM25+CodeEx while the semantic enrichment is the same. Figure 4c shows the average search time of the ExCS with DeepCS. We notice the time costs increase with the number of candidates increases. We argue that the time costs positively correlate to the scale of the codebase provided to DeepCS. From the above observation, considering accuracy and time cost, providing 1000 candidates is suitable for ExCS.

Then, we did a research on how the length of expansions affects the accuracy. We expanded the codes from 10 tokens to 100 tokens. The results are in Fig. 4d. From the we show some examples of expansions produced by the CodeEx. Also we notice that CodeEx refer to copy some tokens from the input codes(e.g., video, url, extract). This shows that CodeEx can effectively re-weight the terms and gets the key terms. Nevertheless, CodeEx also produces tokens not in the input document(e.g. voice, audio). This terms are characterized by synonyms and other related terms.

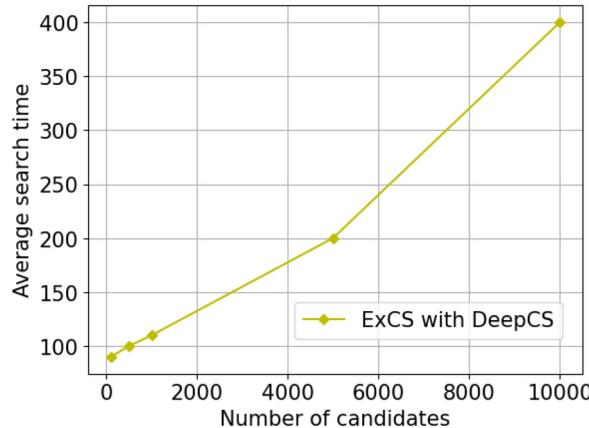
To quantify the above analysis, we measure the proportion of copied tokens and new tokens. In the test set, we find that 65% of the predictions are copied while 35% are new. We did ablation research and put results in



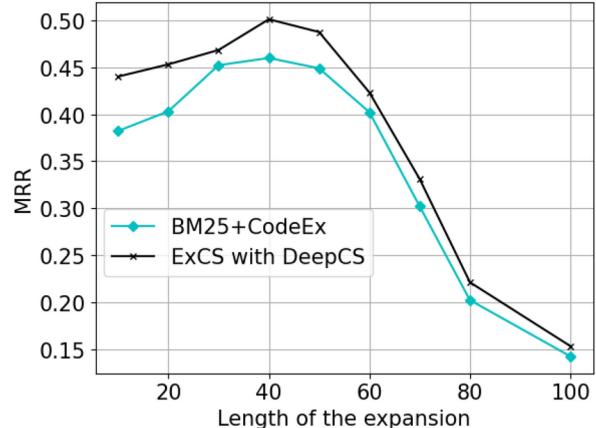
(a) The Recall@k comparison of BM25+CodeEx with various numbers of candidates.



(b) The MRR comparison of ExCS with DeepCS with various numbers of candidates.



(c) The average search time comparison of ExCS with DeepCS with various numbers of candidates.



(d) The MRR comparison of BM25+CodeEx and ExCS with DeepCS with various lengths of the expansion.

Fig. 4. Composite figure of various comparisons.

Dataset	Model effectiveness	
	BM25	ExCS with DeepCS
CSN	0.3193	0.4072
CSN+copied	0.3557	0.4221
CSN+new	0.4285	0.4572
CSN+all	0.4523	0.4685

Table 2. Model effectiveness and model efficiency without and with ExCS.

Model	Online retrieval		
	Embedding	Similarity	Array sorting
BM25+CodeEx	0 ms	40 ms	10 ms
DeepCS	1000 ms	100ms	10 ms
ExCS with DeepCS	50 ms	45 ms	11 ms
SANCS	350 ms	30 ms	10 ms
ExCS with SANCS	20 ms	42 ms	11 ms

Table 3. Model accuracy on different type of expansion.

Table 2. Expanding with copied terms outperforms the original documents MRR by 2.01% and new terms by 30.97%. We notice that the copied terms improve the performance less than the new terms. This is because the copied terms can only increase the term frequency of the tokens already in codes. In comparison, new terms can have more terms matching the query. Meanwhile, We achieve the highest MRR, 0.4527, with both types of terms, which shows that they are complementary.

RQ3: Efficiency Enhancement through ExCS

Table 3 provides a detailed breakdown of the time costs associated with embedding, vector similarity calculation, and array sorting processes across two state-of-the-art models, DeepCS and SANCS, as described in Sect “**Baselines**”, juxtaposed against BM25+CodeEx. This evaluation was conducted under a consistent experimental environment. We rounded the average experimental data, and during retrieval, there are other time-consuming procedures like loading documents so that the total time will be different from Table 1.

The vector similarity calculation time cost depends on the dimension of the models, in which DeepCS is 512 and SANCS is 128, so SANCS is faster than DeepCS. However, when adapted to ExCS, the time cost mainly depends on the BM25. The array sorting time is nearly the same. We can find that ExCS contributes most to the embedding step. For DeepCS, by storing the code embeddings in advance, ExCS significantly saves the embedding time, for the LSTM can only be calculated serially. For SANCS, self-attention can be calculated fast in parallel on GPUs, and the co-attention mechanism is relatively simple. So when SANCS only needs to perform neural ranking on a small-scale codebase, SANCS is very fast.

Threats to Validity

This section elucidates potential threats to the validity of our findings and delineates the steps taken to mitigate them where possible.

External validity: External validity concerns the generalizability of our findings to other settings or populations.

- **Generalizability:** The models were assessed utilizing a specific dataset under controlled settings. The evaluation was rigorous, yet the extension of these results to other datasets or different settings warrants further exploration. Our work lays a robust foundation for such future investigations, aiming to ensure the adaptability and applicability of the proposed ExCS model across diverse scenarios.
- **Dataset bias:** The dataset employed in our evaluation may embody inherent biases, which is a common concern in machine learning endeavors. We have endeavored to choose a representative dataset, yet the performance metrics might vary with different datasets. It's an inherent trade-off, yet it also sparks the avenue for evaluating the model's performance across a spectrum of datasets.

Internal validity: Internal validity pertains to the integrity of the experimental design and the causal relationships inferred.

- **Parameter tuning:** The models' performance is inherently tied to the choice of parameters. Our approach to parameter tuning was systematic and based on established practices, endeavoring to unveil the models' potential. Yet, there could be alternative tuning approaches or parameter sets that might yield different results, signifying an area for further exploration.
- **Implementation integrity:** Although meticulous testing was conducted to ensure the correctness of the implementation, the complex nature of software systems could harbor unnoticed issues that might affect the results. We have shared our code to promote transparency and facilitate validation by the community.

Construct validity: Construct validity is concerned with the adequacy of the measurement instruments.

- Measurement metrics: The metrics employed for evaluating model performance are standard in the field. However, they may not encapsulate all facets of model effectiveness and efficiency. Future work could consider employing or developing alternative metrics that might offer a more holistic view of the model's performance.
- Baseline comparisons: Our choice of baseline models was grounded on their relevance and common use in the field. Yet, the comparative analysis might exhibit different nuances with alternative baseline models or configurations. This aspect underscores the importance of continuous evaluation against emerging models and configurations.

Addressing these threats in a forthright manner not only underscores the robustness of our research process but also paves the way for future work. It invites the exploration of the proposed ExCS model across a broader spectrum of settings and against a wider array of baseline models. It also promotes the investigation of alternative parameter tuning strategies and performance metrics, all aimed at fostering a deeper and more comprehensive understanding of the model's strengths, areas of improvement, and its potential impact on the field.

Conclusion and future works

We introduce ExCS to speed up code search, which combines CodeEx-based Information Retrieval (IR) methods and neural ranking models. ExCS uses a two-step approach common in the IR field to cut down the time needed by neural ranking models. It first quickly finds a small set of candidate codes using IR methods and then re-ranks these candidates with neural ranking models. Moreover, ExCS is easy to adapt to any neural ranking models by using them as re-rankers, significantly reducing their retrieval time cost. Our thorough experiments on Java datasets from CodeSearchNet (CSN) show that compared to traditional code search models, ExCS greatly improves retrieval efficiency while maintaining nearly the same performance.

For future work, we plan to address some challenges. Firstly, our experiments are only done on method-level Java datasets. We plan to include datasets from other languages in CSN. Secondly, the performance of ExCS heavily relies on the pre-trained models and the chosen dataset. Most datasets use the annotation as a query, which isn't common in real-world situations. Therefore, we aim to collect manually marked data and realistic search data from sources like Google or Bing to test our method further in the future.

Data availability

The datasets generated and/or analysed during the current study are available in the code_search_net repository, https://huggingface.co/datasets/code_search_net.

Received: 29 May 2024; Accepted: 23 September 2024

Published online: 25 November 2024

References

1. Xia, X. *et al.* What do developers search for on the web?. *Empir. Softw. Eng.* **22**, 3149–3185 (2017).
2. de Rezende Martins, M. & Gerosa, M. A. Conra: A convolutional neural networks code retrieval approach. In *Proceedings of the 34th Brazilian Symposium on Software Engineering*, 526–531 (2020).
3. Yan, S., Yu, H., Chen, Y., Shen, B. & Jiang, L. Are the code snippets what we are searching for? a benchmark and an empirical study on code search with natural-language queries. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 344–354 (IEEE, 2020).
4. Brandt, J., Guo, P. J., Lewenstein, J., Dontcheva, M. & Klemmer, S. R. Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 1589–1598 (2009).
5. Nogueira, R., Yang, W., Lin, J. & Cho, K. Document expansion by query prediction. arXiv preprint [arXiv:1904.08375](https://arxiv.org/abs/1904.08375) (2019).
6. Feng, Z. *et al.* Codebert: A pre-trained model for programming and natural languages. arXiv preprint [arXiv:2002.08155](https://arxiv.org/abs/2002.08155) (2020).
7. Gu, X., Zhang, H. & Kim, S. Deep code search. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, 933–944 (IEEE, 2018).
8. Yu, Y., Si, X., Hu, C. & Zhang, J. A review of recurrent neural networks: Lstm cells and network architectures. *Neural Comput.* **31**, 1235–1270 (2019).
9. Shuai, J. *et al.* Improving code search with co-attentive representation learning. In *Proceedings of the 28th International Conference on Program Comprehension*, 196–207 (2020).
10. O'Shea, K. & Nash, R. An introduction to convolutional neural networks. arXiv preprint [arXiv:1511.08458](https://arxiv.org/abs/1511.08458) (2015).
11. Devlin, J., Chang, M.-W., Lee, K. & Toutanova, K. Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv preprint [arXiv:1810.04805](https://arxiv.org/abs/1810.04805) (2018).
12. Xu, F. F., Alon, U., Neubig, G. & Hellendoorn, V. J. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, 1–10 (2022).
13. Black, S., Gao, L., Wang, P., Leahy, C. & Biderman, S. Gpt-neo: Large scale autoregressive language modeling with mesh-tensorflow. In *If You Use this Software, Please Cite it Using these Metadata* **58** (2021).
14. GitHub. The technology behind github's new code search (2023). (Accessed 06 Feb 2023).
15. Wang, Y., Wang, W., Joty, S. & Hoi, S. C. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. arXiv preprint [arXiv:2109.00859](https://arxiv.org/abs/2109.00859) (2021).
16. Lv, F. *et al.* Codehow: Effective code search based on api understanding and extended boolean model (e). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 260–270 (IEEE, 2015).
17. Yao, Z., Peddamail, J. R. & Sun, H. Coacor: Code annotation for code retrieval with reinforcement learning. In *The World Wide Web Conference*, 2203–2214 (2019).
18. Xu, L. *et al.* Two-stage attention-based model for code search with textual and structural features. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 342–353 (IEEE, 2021).
19. Fang, S., Tan, Y.-S., Zhang, T. & Liu, Y. Self-attention networks for code search. *Inf. Softw. Technol.* **134**, 106542 (2021).
20. Gu, J., Chen, Z. & Monperrus, M. Multimodal representation for neural code search. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 483–494 (IEEE, 2021).

21. Wang, C. *et al.* Enriching query semantics for code search with reinforcement learning. *Neural Netw.* **145**, 22–32 (2022).
22. Grechanik, M. *et al.* A search engine for finding highly relevant applications. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, 475–484 (2010).
23. Bajracharya, S. *et al.* Sourcerer: a search engine for open source code supporting structure-based search. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, 681–682 (2006).
24. Marcos-Pablos, S. & García-Penalvo, F. J. Information retrieval methodology for aiding scientific database search. *Soft. Comput.* **24**, 5551–5560 (2020).
25. Neelakantan, A. *et al.* Text and code embeddings by contrastive pre-training. arXiv preprint [arXiv:2201.10005](https://arxiv.org/abs/2201.10005) (2022).
26. Toutanova, K. *et al.* Representing text for joint embedding of text and knowledge bases. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, 1499–1509 (2015).
27. Frome, A., Corrado, G., Shlens, J. *et al.* A deep visual-semantic embedding model. In *Proceedings of the Advances in Neural Information Processing Systems* 2121–2129.
28. Yang, P., Fang, H. & Lin, J. Anserini: Enabling the use of lucene for information retrieval research. In *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval*, 1253–1256 (2017).
29. Chicco, D. Siamese neural networks: An overview. *Artif. Neural Netw.* **73**, 73–94 (2021).
30. Vaswani, A. *et al.* Attention is all you need. *Adv. Neural Inf. Process. Syst.* **30** (2017).
31. Li, R. *et al.* Starcoder: may the source be with you! arXiv preprint [arXiv:2305.06161](https://arxiv.org/abs/2305.06161) (2023).
32. Roziere, B. *et al.* Code llama: Open foundation models for code. arXiv preprint [arXiv:2308.12950](https://arxiv.org/abs/2308.12950) (2023).
33. Jebara, T. *Machine Learning: Discriminative and Generative* Vol. 755 (Springer Science & Business Media, 2012).
34. Gravitas, S. Autogpt: An open source implementation (2023). (Accessed 28 Oct 2023).
35. langchain ai. Building applications with llms through composability (2023). (Accessed 28 Oct 2023).
36. Wiseman, S. & Rush, A. M. Sequence-to-sequence learning as beam-search optimization. arXiv preprint [arXiv:1606.02960](https://arxiv.org/abs/1606.02960) (2016).
37. Rong, X. word2vec parameter learning explained. arXiv preprint [arXiv:1411.2738](https://arxiv.org/abs/1411.2738) (2014).
38. Husain, H., Wu, H.-H., Gazit, T., Allamanis, M. & Brockschmidt, M. Codesearchnet challenge: Evaluating the state of semantic code search. arXiv preprint [arXiv:1909.09436](https://arxiv.org/abs/1909.09436) (2019).
39. Sun, Z., Li, L., Liu, Y., Du, X. & Li, L. On the importance of building high-quality training datasets for neural code search. In *Proceedings of the 44th International Conference on Software Engineering*, 1609–1620 (2022).

Author contributions

S.H. and Y.Y. wrote the main manuscript text. B.C. and J.L. help us with the experiment. All authors reviewed the manuscript.

Declarations

Competing interests

The authors declare no competing interests.

Additional information

Correspondence and requests for materials should be addressed to B.C.

Reprints and permissions information is available at www.nature.com/reprints.

Publisher's note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Open Access This article is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License, which permits any non-commercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if you modified the licensed material. You do not have permission under this licence to share adapted material derived from this article or parts of it. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by-nc-nd/4.0/>.

© The Author(s) 2024