Fusing Code Searchers

Shangwen Wang, Mingyang Geng, Bo Lin, Zhensu Sun, Ming Wen, Yepang Liu, Li Li, Tegawendé F. Bissyandé and Xiaoguang Mao

Abstract—Code search, which consists in retrieving relevant code snippets from a codebase based on a given query, provides developers with useful references during software development. Over the years, techniques alternatively adopting different mechanisms to compute the relevance score between a query and a code snippet have been proposed to advance the state of the art in this domain, including those relying on information retrieval, supervised learning, and pre-training. Despite that, the usefulness of existing techniques is still compromised since they cannot effectively handle all the diversified queries and code in practice. To tackle this challenge, we present `Dancer`, a data fusion based code searcher. Our intuition (also the basic hypothesis of this study) is that existing techniques may complement each other because of the intrinsic differences in their working mechanisms. We have validated this hypothesis via an exploratory study. Based on that, we propose to fuse the results generated by different code search techniques so that the advantage of each standalone technique can be fully leveraged. Specifically, we treat each technique as a retrieval system and leverage well-known data fusion approaches to aggregate the results from different systems. We evaluate six existing code search techniques on two large-scale datasets, and exploit eight classic data fusion approaches to incorporate their results. Our experiments show that the best fusion approach is able to outperform the standalone techniques by 35% - 550% and 65% - 825% in terms of MRR (mean reciprocal rank) on the two datasets, respectively.

Index Terms—Code Search, Information Retrieval, Data Fusion.

1 Introduction

Software programming is often redundant: a given functionality usually already has a number of implemented code available in codebases in the wild [1]. Developers therefore tend to refer to or directly reuse existing code snippets during development to increase their productivity [2], [3]. As reported by prior studies, overall developers can spend about 20% of their time searching for relevant code examples [4], [5]. Code search is an active research field in software engineering that aims at facilitating the code reuse activities of practitioners [6], [7], [8], [9], [10], [11], [12], [13]. Typically, code search techniques consider a natural language query by a developer describing the intended functionality, and employ retrieval mechanisms to identify code snippets that are relevant to that functionality within a large-scale codebase collected from real-world repositories.

Existing code search techniques can generally be classified into three categories according to the mechanisms that they develop for computing the relevance score between the query and a given code snippet [14], [15], [16]. In the first category, techniques rely on traditional *information retrieval*, where they focus on the proportion of overlapping tokens between the query and the code snippet [17], [18], [19], [20]. In the second category, *supervised learning* is leveraged where models are trained on a labeled dataset to capture the semantic relationship between the query and the code [21], [22], [23], [24]. In the third category, researchers explore *pre-training*, which utilizes self-supervised techniques to pre-train models on an unlabeled dataset that is usually much larger than those used by the second category, expecting that the huge-amount data can help capture the connection between the query and code [25], [26], [27], [28]. Beyond this categorisation, it is noteworthy that techniques from the same category may also have different focuses. For instance, while DeepCS [21] and DeGraphCS [29] both implement supervised learning, they differ in the information that they leverage for modeling code semantics: the former focuses on the token sequences of programs while the latter focuses on the data flow and control flow.

Over the years, the performance of state-of-the-art code search techniques has been improved continuously by constructing more complex models [22], [30] or leveraging more advanced deep learning techniques [31]. Nevertheless, due to the diversity of the queries and code in practice, standalone techniques inevitably work effectively on some queries but fall short on others, which compromises their usefulness [14], [15]. To address this limitation, our intuition, which is also the basic hypothesis of this study, is that **existing techniques may complement each other** due to the intrinsic differences among their working mechanisms. That is to say, queries which are not effectively handled by a given code search technique may be successfully handled using another technique. We first propose to validate this hypothesis by performing an exploratory study on the complementarity of code search techniques. This study is

- *Shangwen Wang, Mingyang Geng, Bo Lin, and Xiaoguang Mao are with the National University of Defense Technology, Changsha, Hunan, China. E-mails: wangshangwen13@nudt.edu.cn, gengmingyang13@nudt.edu.cn, linbo19@nudt.edu.cn, and xgmao@nudt.edu.cn*
- *Zhensu Sun is with ShanghaiTech University, Shanghai, China. E-mail: sunzhs@shanghaitech.edu.cn*
- *Ming Wen is with Huazhong University of Science and Technology, Wuhan, China. E-mail: mwenaa@hust.edu.cn*
- *Yepang Liu is with Southern University of Science and Technology, Shenzhen, China. E-mail: liuyp1@sustech.edu.cn*
- *Li Li is with Monash University, Australia. E-mail: lilicoding@ieee.org*
- *Tegawendé F. Bissyandé is with University of Luxembourg, Luxembourg. E-mail: tegawende.bissyande@uni.lu*
- *Ming Wen and Yepang Liu are the corresponding authors.*

conducted from two perspectives: the complementarities among techniques from different categories (which we refer to as *inter-category complementarities*) and the complementarities among techniques in the same category (which we refer to as *intra-category complementarities*). Specifically, we select three example techniques across different categories for the inter-category experiment and three techniques from the supervised learning based category (i.e., the most widely-studied category so far [14]) for the intra-category experiment. We evaluate these techniques on a dataset containing more than 22K queries and then investigate the distributions of the queries whose corresponding code snippets (referred to as the oracle code) are ranked at top positions. Results show that for the three inter-category techniques, there are totally 16,227 queries whose oracle code can be ranked at the first position by at least one of them, 66% of which (i.e., 10,720) can only be effectively handled by a unique technique. Similarly, this percentage is 44% for the three intra-category techniques. Such a result indicates the complementarity of existing code search techniques and thus validates our hypothesis.

Motivated by the above observation, in this paper, we propose to boost the effectiveness of code search by leveraging the complementarity of existing techniques. We were inspired by noting the work of group decision support systems [32]. Specifically, such systems take input from multiple decision-makers simultaneously, bring together the produced ideas, and finally arrive at a best decision. Therefore, we postulate that effectiveness enhancement could be achieved for code search if the results from multiple techniques can be fused. Based on that, we propose a DAta fusioN based Code searchER, `Dancer`, which uses data fusion approaches to combine search results from different code search techniques. Data fusion is widely studied in the information retrieval domain [33], [34], [35], and is considered as an effective way to combine multiple sources of information into a single one [36], [37], [38]. Specifically, we treat each code search technique as a retrieval system and use both rank-based and score-based data fusion methods to combine the results from each system.

To evaluate the effectiveness of our fusion approach, we perform extensive experiments on two large-scale datasets from CodeSearchNet [39], i.e., the CSN-Python and CSN-Java. We use six state-of-the-art code searchers as standalone techniques, among which one is based on information retrieval, three are based on supervised learning, and the remaining two are based on pre-training techniques. We first evaluate each of the standalone techniques and then the fusion results generated by eight different fusion approaches. Results show that both the rank-based and the score-based fusion approaches can significantly boost the code search effectiveness compared with the standalone techniques. Specifically, the score-based fusion approach, $\texttt{Dancer}_{CombSUM}$, achieves the highest Mean Reciprocal Rank (MRR) value on the CSN-Python dataset, which is 0.922. Such a value exceeds that of each standalone technique by 35% - 550%. On the CSN-Java dataset, the rank-based fusion approach, $\texttt{Dancer}_{Borda\,count}$, has the most effective performance with its MRR being 0.878. Such a value outperforms standalone techniques by 65% - 825%. We also estimate the average response time of `Dancer` for a given query and we consider the result (i.e., around 0.2 second) as affordable. Further in-depth investigation shows that fusing the results of three code search techniques can already gain significant effectiveness enhancement compared with standalone techniques, which demonstrates the practicality of our approach in a computation resource restricted situation.

TABLE 1: The data fusion approaches we use and their categories.

Category	Approaches
Rank based	`Borda count` [36], `Reciprocal rank` [40], `Condorcet criterion` [41]
Score based	`CombMIN` [33], `CombMAX` [33], `CombSUM` [33], `CombANZ` [33], `CombMNZ` [33]

To summarize, our study makes the following contributions:

- ❶ We perform an exploratory study, which demonstrates the substantial complementarities among existing code search techniques. This observation can open a novel direction for code search researches.
- ❷ We devise a data fusion based approach for code search, named `Dancer`, which builds on the complementarities of existing code search techniques. `Dancer` exploits off-the-shelf data fusion approaches to integrate results from multiple code searchers.
- ❸ We extensively evaluate the effectiveness of `Dancer` and achieve promising results. Specifically, in terms of MRR, the best fusion approach outperforms the standalone techniques by 35% - 550% and 65% - 825% respectively on two large-scale datasets.

2 Background

2.1 Data Fusion

Data fusion, i.e., combining multiple ranked answer lists into a single one, is a widely-studied problem in the domain of Information Retrieval (IR). Typically, for an information retrieval system (where given a query, relevant documents are expected to be retrieved from a database of candidate documents), the query representation and the document representation are two critical components. The key idea of data fusion is thus the document and/or query representations of each IR system may differ from the others and enhanced retrieval results may be provided if multiple sources of information could be combined [33], [34], [35]. Such a strategy has achieved impressive achievements in the web search. For instance, results from multiple independent search systems are combined into a single ranking in meta-search [36], and results in different languages can also be combined [37].

Data fusion approaches can be broadly grouped into two types, i.e., the rank-based ones and the score-based ones [42], [43]. In our study, we explore the combination of different code search techniques using both types of data fusion approaches, and Table 1 lists the eight approaches we utilize in this study. We next briefly introduce each of them. The rank-based ones rely on the rank position of a retrieved document. The well-known approaches include the **Borda count** [36] where each document receives a score determined by how many other documents are ranked lower than it, and the score is summed across all input lists;

TABLE 2: The code search techniques we investigate and their categories.

Category	Techniques
Information retrieval based	`BM25` [44]
Supervised learning based	`Self-attention` [39], `DeGraphCS` [29], `Multi-modal` [22]
Pre-training based	`CodeBERT` [25], `GraphCodeBERT` [26]

the **Reciprocal rank** [40] which sorts the documents according to the reciprocal of their rankings; and the **Condorcet criterion** [41] where the rank of a document is based on the pairwise comparison with all other candidates.

The score-based ones use the relevance score of a retrieved document, which represents the likelihood of the document being relevant to the query. Fox and Shaw [33] proposed several basic fusion approaches including: **CombMIN**, a document's final relevance score is the minimum of its individual relevance scores, aiming at minimizing the probability that a non-relevant document would be highly ranked; **CombMAX**, a document's final relevance score is the maximum of its individual relevance scores, aiming at avoiding the cases where relevant documents are poorly ranked; **CombSUM**, a document's final relevance score is the summation of its individual relevance scores, aiming at comprehensively considering the document's similarity scores instead of simply attempting to select a single similarity value from a set of result lists; **CombANZ**, a document's final relevance score is the value of CombSUM divided by the number of non-zero relevance scores, aiming at punishing the documents that occur in the list but with low scores; and **CombMNZ**, a document's final relevance score is the value of CombSUM multiplied by the number of non-zero relevance scores, aiming at promoting a document that occurs in multiple lists.

2.2 Code Search

Many code search approaches have been proposed during the years to advance this common activity in software development practices [2], [9], [11]. Given a natural language (NL) query from the developer, code search approaches search for the code snippets from a large-scale code corpus and return the relevant ones to serve as references for developers' implementations. Based on the methods to calculate the relevance score between the query and the code, existing approaches can be broadly grouped into three categories [14], [15]. Table 2 summarizes the code search techniques which are selected as our study subjects and their corresponding categories. In the following content, we briefly introduce each category and the representative techniques that are selected as our study subjects. Readers can refer to the survey [14] for more details about code search techniques.

The first category represents the traditional IR-based techniques, which mainly rely on the keyword mapping between the query and the code. Such techniques focus on the textual overlapping relations to calculate the relevance scores. For instance, Lu et al. [17] augmented the query with the synonyms obtained from WordNet and then performed keyword matching. Lv et al. [45] propose to combine textual similarity and API matching into an extended Boolean model. In our study, we choose to use a state-of-the-art IR approach, the **BM25** algorithm [44], to calculate the relevance score for each code snippet. It is a TF-IDF-like function and it ranks a set of code snippets based on the query tokens appearing in each code snippet, regardless of their proximity within the code.

A fundamental problem faced by the first category is its inability to capture the correlation between the semantics of the query and the code. To overcome this limitation, researchers later propose to use supervised learning techniques to embed the query and code into a shared vector space, where the relevance is calculated as the similarity between the query vector and the code vector. Therefore, such techniques differ with each other in how they represent the semantics of the code. For instance, DeepCS [21] considers code snippets as token sequences and embeds the code using a recurrent neural network (RNN). Wan et al. [22] proposed MMAN which uses an attention model to aggregate different types of code information including the token sequence, the AST structure, and the graph-based semantic information. In our study, we select three state-of-the-art techniques as our study subjects, which are **Self-attention** [39], **DeGraphCS** [29], and **Multi-modal** [22]. These techniques have been used as baselines for a number of follow-up studies [5], [23], [25], [46]. The *Self-attention* model also treats code as token sequences but it uses a Transformer to embed the sequence, which is expected to outperform the RNN on dealing with sequential data and thus can build a more qualified baseline than DeepCS. *DeGraphCS* utilizes the variable-based flow graph to depict data and control flows in the program. It is expected to outperform the original graph based component of MMAN which merely considers control flow information. We implement our own *Multi-modal* approach by using the above two approaches to replace the original components in MMAN which target the token sequence and program graph information and keeping the original component that captures AST structure information, i.e., the Tree-LSTM [47]. Information from each component is also fused through an attention mechanism and with more advanced components, our *Multi-modal* is expected to build a more state-of-the-art baseline than the original MMAN. Note that for all of these three approaches, the query is embedded by a Transformer-based encoder [48].

The second category of techniques, however, require large-scale labeled data for training, which is hard to obtain in practice. With the blooming of pre-training techniques in the natural language processing (NLP) filed, researchers also propose to design pre-trained models for programming tasks. The core idea of such pre-training techniques is to utilize the huge-amount readily-available code-comment pairs for capturing the semantic connection between natural language (NL) and programming language (PL). To this end, different pre-trained models with diverse pre-training objectives have been proposed. For instance, **CodeBERT** [25], the first NL-PL pre-training technique, uses the Masked Language Modeling task in which the model is trained to predict the randomly masked tokens, and the Replaced Token Detection task in which the model is trained to determine if a token is replaced. To address the limitation of *CodeBERT* that the program structure is ignored, **GraphCodeBERT** [26] designs specialized tasks in the pre-training phase to make the model structure-aware, such as the edge

prediction, in which the model learns to predict which program tokens contain data flow relations. In our study, we select these two techniques as our study subjects since the previous study has shown that they are the most effective ones in program understanding tasks including code search among all the pre-training techniques [49].

Typically, code search techniques rank candidate code snippets in a descending order based on their relevance scores with the query. That is to say, with executing a code search technique, both the rank and the relevance score of each candidate code snippet can be obtained. Hence, both rank-based and score-based fusion approaches can be applied to aggregate the results from different code search techniques and in our study we explore both directions.

3 Hypothesis Validation

As we have introduced, existing code search techniques can be mainly classified into three categories according to their working mechanisms. Additionally, even techniques from the same category may have different focuses. For instance, *Self-attention* and *DeGraphCS* differ in how to embed the semantics of code, while *CodeBERT* and *GraphCodeBERT* have different pre-training objectives. Given that, our hypothesis in this study is that *different code search techniques complement each other, i.e., they may work effectively on different queries*. To validate the existence of such a complementarity, we conduct an exploratory experiment to investigate if the existing code search techniques indeed work effectively on different queries. We define a metric **uniqueness degree** here to quantitatively assess such a complementarity, which is calculated as the number of queries effectively handled by a unique technique divided by the number of queries effectively handled by at least one technique. The uniqueness degree values range from 0 to 1. Intuitively, higher uniqueness degrees mean that more queries can be effectively addressed only by specific techniques, and thus there exists a higher degree of complementarity among the techniques.

Our experiment focuses on two perspectives: the inter-category complemantarity (i.e., the complemantarity of techniques from different categories) and intra-category complementarity (i.e., the complemantarity of techniques from the same category). To achieve the first target, we select the most effective technique from each category to serve as representatives. The effectiveness comparison is based on the previous studies [22], [49], [50] and finally *BM25*, *Multi-modal*, and *GraphCodeBERT* are selected. For the second target, we explicitly focus on the supervised learning category since techniques in this category are the most popularly studied [14]. Specifically, three state-of-the-art techniques in this category (i.e., *Self-attention*, *DeGraphCS*, and *Multi-modal*), which are also our study subjects, are used. We then train and evaluate them on the CodeSearchNet-Python dataset [39], which will be detailed in Section 5. Specifically, we investigate the distributions of the queries whose corresponding code snippet can be ranked at top-k positions by each code search technique. The results are shown in Figure 1 where k = 1, 5, and 10.

From the figure, we note that each technique can work effectively on a certain number of unique queries from the inter-category perspective. For instance, *GraphCodeBERT* can uniquely rank the oracle at the first position for 7,420 queries, and the number for *Multi-modal* and *BM25* are 2,981 and 319, respectively. Totally, there are 16,227 queries whose corresponding code snippets can be ranked at the first position by at least one of the code search techniques. Among them, 10,720 queries are effectively handled by a standalone technique uniquely (only one specific technique can rank its oracle at top-1), leading to a uniqueness degree of 66%. While the uniqueness degree decreases with respect to the top-10 results, different techniques still perform effectively on certain queries exclusively. Specifically, *GraphCodeBERT* and *Multi-modal* uniquely rank the oracle at the top-10 position for a large amount of queries (5,352 and 1,758, respectively). Totally, the three techniques uniquely rank the oracle at top-10 for 7,201 queries, still occupying 34% of the queries whose oracle code can be ranked at top-10 by at least one technique (7,201/21,176). We also observe similar phenomenon from the intra-category perspective. Take the top-1 results as an example: the oracle code of 11,298 queries can be ranked at top-1 by at least one technique, among which 4,970 are effectively handled by a unique technique, leading to a uniqueness degree of 44%.

Our results indicate the complementarity of existing code search techniques: different techniques (whether they are from the same category or not) effectively handle different queries. The validation of the existence of such a complementarity motivates our study, where data fusion approaches are leveraged to combine results from different code search techniques into a single result list for effectiveness enhancement.

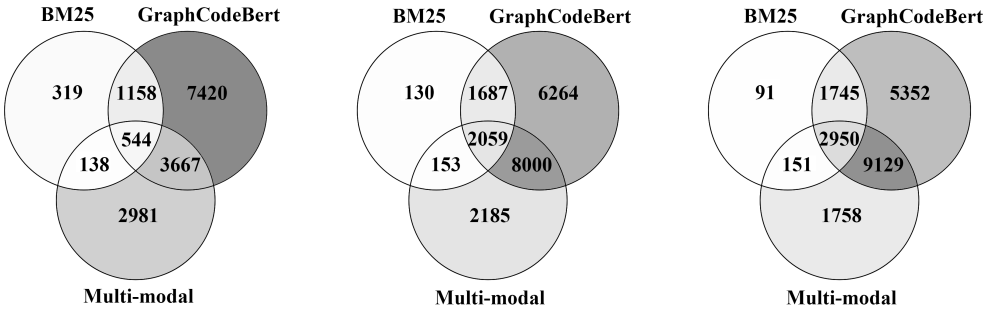
We also note that in this figure, different areas have different shades. The darker the area, the more queries are located in it. We find that the inter- and intra-category results demonstrate different patterns: for the former, the area of *GraphCodeBERT* is much darker than those of the other techniques; while for the latter, the areas of the three techniques are in similar shades. This is because techniques from different categories have diverse effectiveness while such difference is not that significant for techniques from the same category (details will be analyzed in Section 6).

4 Methodology

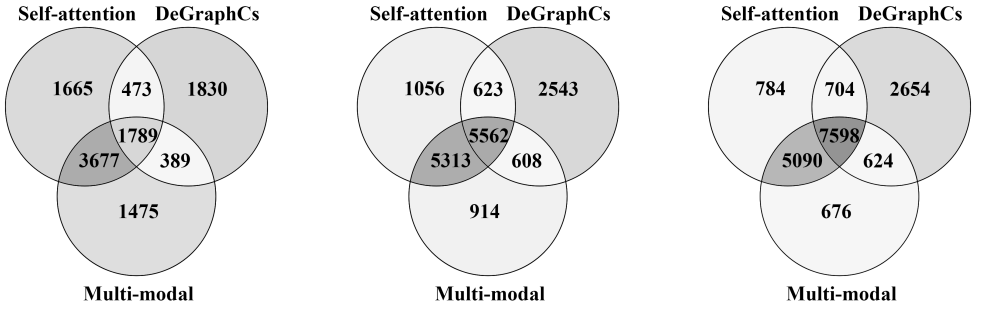
In this section, we present our fusion based approach that incorporates retrieval results from different code searchers.

4.1 Overview

The overall workflow of our fusion approach is illustrated in Figure 2. Given a natural language query and a codebase, our approach first applies off-the-shelf code search techniques to generate their standalone search results. Then, for the candidate code snippets, both their ranks in the result list and their relevance scores produced by each code search technique are recorded. Finally, according to the recorded data in the last step, the rank/score-based data fusion approach is applied to re-rank the candidate code snippets and the output of this step is considered as the final result.



(a) Top-1 results of inter-category **(b)** Top-5 results of inter-category **(c)** Top-10 results of inter-category



(d) Top-1 results of intra-category **(e)** Top-5 results of intra-category **(f)** Top-10 results of intra-category

Fig. 1: The distributions of the queries whose oracles can be ranked at top positions by different code search techniques.

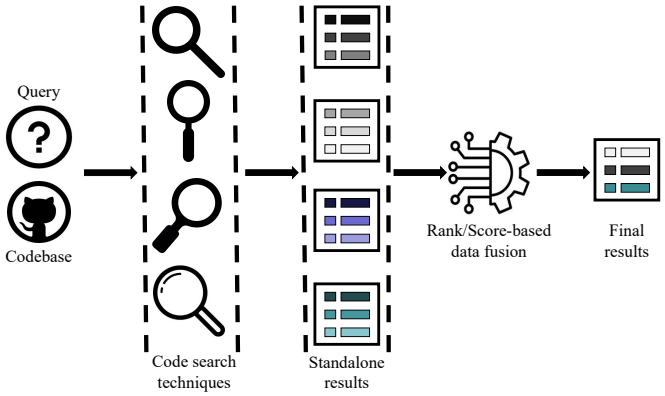


Fig. 2: The workflow of our fusion approach.

4.2 Executions of Standalone Code Search Techniques

Given a query and a codebase to search for the relevant code snippets, we execute the six code search techniques listed in Table 2 respectively. The mechanisms of our study subjects have been introduced in Section 2.2. We briefly review how each technique calculate the relevance score here.

For the IR based *BM25* algorithm, the relevance score of each candidate code snippet is calculated based on the number of query tokens appearing in it. For the supervised learning based models, the code snippet and query are embedded as vectors separately and the similarity of the vectors is considered as the relevance score. For the pre-training techniques, during fine-tuning, the query and source code are concatenated as the sequence input (e.g., $\{[CLS], W, [SEP], C\}$ for *CodeBERT* and $\{[CLS], W, [SEP], C, [SEP], V\}$ for *GraphCodeBERT*), where W denotes the query segment, C denotes the source code segment, V denotes the variables in the source code, $[CLS]$ is a special token in front of these segments, and $[SEP]$ is a special token to split two kinds of data. The representation of the $[CLS]$ token is considered as the aggregated sequence representation and it will be connected with a *softmax* layer to output the relevance score.

4.3 Data Fusion

In this work, we leverage eight well-known data fusion approaches listed in Table 1. Each of them can serve for our `Dancer`. Recall that we have briefly introduced their working mechanisms in Section 2. We give formal definitions to them in the following paragraphs.

4.3.1 Rank-based data fusion

Given a set of code snippets to be ranked C and a set of rankings R returned by different code search techniques, where each $r \in R$ is a permutation on $1 \ldots |C|$, **Borda count** works as follows:

$$BCscore(c \in C) = \sum_{r \in R} (|C| - r(c))$$

where $r(c)$ denotes the rank of the code snippet c in the permutation of r, and thus $|C| - r(c)$ means how many code snippets are ranked lower than c in this permutation. Then, each $c \in C$ is re-ranked according to its $BCscore$ value in a descending order. Suppose two code snippets a and b, the behind intuition of this approach is that a should be ranked higher than b in the final result if the number of code snippets that are ranked lower than a is more than that of b, according to all the rankings in R.

Reciprocal rank works as follows:

$$RRscore(c \in C) = \sum_{r \in R} \frac{1}{r(c) + k}$$

where $r(c)$ denotes the same meaning as above and the constant k, which is used to mitigate the impact of high rankings to non-relevant code, is empirically fixed to 60 according to the existing study [40]. Then, each c is also re-ranked in a descending order according to the $RRscore$ value. The rationale of this approach is utilizing the aggregated reciprocal value to provide a holistic view for each code snippet's relevance.

Condorcet criterion works as follows:

$$CCscore(c \in C) = \sum_{c' \in C, c \neq c'} MV(c, c', R)$$

where $MV(c, c', R)$ denotes the majority voting result of the code snippet pair (c and c') on the ranking set R. It returns 1 if c is ranked higher than c' on more than half of the rankings and 0 otherwise. After that, each c is re-ranked in a descending order according to the $CCscore$ value. The behind intuition of this approach is that the code snippet which achieves the highest number of wins in such pairwise comparisons should be ranked at the top position.

4.3.2 Score-based data fusion

As we have introduced before, different code search techniques use different ways to calculate relevance scores. Consequently, the relevance scores from different techniques may have different scales. One essential step before calculating the overall result is normalizing the scores so that scores produced by different techniques can be compared with one another. In this study, we use the *Linear Normalization* strategy to normalize the scores. Such a strategy is to re-scale the scores into a fixed scale (i.e., [0, 1] in our study) and thus help alleviate the scale bias. It has been demonstrated to be effective by a number of previous studies [38], [51], [52].

Given a set of code snippets to be ranked C and a set of rankings R returned by different code search techniques, where each $r \in R$ assigns a relevance score $RS(c, r)$ to each code snippet $c \in C$. The relevance score of a retrieved code snippet is first normalized by the maximum range of the scores:

$$NRS(c, r) = \frac{RS(c, r) - Min_r}{Max_r - Min_r}$$

where Max_r and Min_r denote the maximum and minimum relevance scores in the list returned by the ranking r. After that, the number of non-zero relevance score of the code snippet c (denoted as $NNZ(c)$) can be calculated.

Then, for each code snippet c, its final relevance score (FRS) with respect to the **CombMIN**, **CombMAX**, **CombSUM**, **CombANZ**, and **CombMNZ** is calculated as:

$$FRS_{min}(c) = MIN(NRS(c, r_1), \ldots, NRS(c, r_{|R|}))$$
$$FRS_{max}(c) = MAX(NRS(c, r_1), \ldots, NRS(c, r_{|R|}))$$
$$FRS_{sum}(c) = \sum_{r \in R} NRS(c, r)$$
$$FRS_{anz}(c) = \frac{FRS_{sum}}{NNZ(c)}$$
$$FRS_{mnz}(c) = FRS_{sum} \times NNZ(c)$$

where MIN (MAX) is the function to return the minimum (maximum) value from a list of data. Finally, all the code snippets are re-ranked based on their final relevance scores in a descending order.

TABLE 3: The statistics of our evaluation datasets.

Dataset	Training	Validation	Test
CSN-Python	412,178	23,107	22,176
CSN-Java	394,471	15,328	26,909

5 Experiment Settings

5.1 Dataset

We choose to use the popular CodeSearchNet (CSN) dataset [39] as the benchmark for our experiments. CSN is a large-scale dataset for semantic code search and has been widely used upon released [25], [46], [49]. It extracts millions of code functions and their corresponding comments from GitHub for totally six programming languages (PLs) including Go, Java, JavaScript, PHP, Python, and Ruby. In our study, to increase the generalizability of our results, we use two mostly-used datasets (i.e., the CSN-Python and CSN-Java).

The CSN dataset is split into three parts: the training/-validation/test sets. The detailed statistics of our evaluation dataset are listed in Table 3. In our experiments, the code comment is used as query and its corresponding code is expected to be retrieved. The training and validation sets are used to train the three supervised learning based techniques (i.e., *Self-attention*, *DeGraphCS*, and *Multi-modal*) as well as fine-tune the two pre-trained models (i.e., *CodeBERT* and *GraphCodeBERT*), while the test set is used to evaluate the effectiveness of each technique. The *BM25* algorithm does not require a training phase so that we directly apply it on the test set. Note that for each query, we search for code snippets from the whole test set, which mimics the real-world scenario where a huge amount of information is provided by the web and an effective code search technique is supposed to accurately identify the needed code snippets.

5.2 Research Questions

In our study, we seek to answer the following research questions (RQs):

RQ1: How effective are existing code search techniques on the large-scale dataset? In this first RQ, we aim to investigate the effectiveness of each standalone technique. The necessity of such an evaluation is twofolds. First, since some of our study subjects have not been evaluated on our selected datasets (e.g., the *BM25* and *DeGraphCS*), the literature lacks a comprehensive comparison among existing techniques on the large-scale dataset. Such a comparison can understand the advantages and disadvantages of existing studies, and thus facilitate future code search researches. Second, the experiment results also pave the way for this study since we need to compare with each standalone technique to evaluate the effectiveness of our fusion approach.

RQ2: Can data fusion help boost the effectiveness of code search? In this RQ, we aim to investigate can `Dancer` that combines search results from all our study subjects outperform each standalone technique. We investigate both the rank-based and the score-based data fusion approaches.

RQ3: To what extent do different code search techniques contribute to the final fusion results? The hypothesis of

our study is that different code search techniques work effectively on different queries and thus all the study subjects could contribute to the final fusion results to different degrees. Although we have illustrated the complementarity of existing code search techniques through our exploratory study, we in this RQ seek to quantitatively assess the contribution of each involved technique.

RQ4: What is the response time of `Dancer` for a given query? To deploy the fusion approach to real-world scenarios, it is important to understand the response time for a given query. In this RQ, we investigate such a question through analyzing the time cost of `Dancer` on the test sets.

5.3 Evaluation Metrics

We adopt two widely-used metrics to measure the effectiveness of code search techniques, which are the Mean Reciprocal Rank (MRR) and the SuccessRate@k [21], [22], [45].

MRR is the average of the reciprocal ranks for the results of a set of queries Q. MRR provides a holistic perspective for evaluating the overall effectiveness and it is calculated as:

$$MRR = \frac{1}{|Q|}\sum_{q \in Q}\frac{1}{Rank_q}$$

where $|Q|$ is the size of the query set, and $Rank_q$ is the rank of the code snippet which corresponds to the query q. Generally, the higher the MRR value is, the better the code search effectiveness is.

SuccessRate@k measures the percentage of queries for which the corresponding code snippet could exist in the top k ranked results. It is an important metric because it can measure how many returned results could be manually checked by the developers before finding the relevant one. This metric is calculated as:

$$SuccessRate@k = \frac{1}{|Q|}\sum_{q \in Q}\delta(Rank_q \leq k)$$

where $|Q|$ and $Rank_q$ denote the same meanings as above, $\delta(\cdot)$ is a function which returns 1 if the input is true and returns 0 otherwise. Generally, the higher the SuccessRate value is, the better the code search effectiveness is. The values of k are set to 1, 5, and 10 respectively, following a number of previous studies [1], [21], [22].

5.4 Implementation Details

All our experiments are performed on a server which possesses 8 interconnected NVIDIA Tesla V100 with 32GB memory. Our implementation of the *BM25* algorithm is reused from the open-accessed GitHub repository. [1] As for the other five deep learning models, their implementations are all open sourced. We therefore reuse all the source code and hyper-parameters to avoid replication bias. Note that *DeGraphCS* uses the Intermediate Representation of programs to extract the graph information. Their approach was originally implemented for C language. To apply it on Java and Python, we exploit JLang [2], and the `Dis` module [3] to acquire the Intermediate Representations for Java and Python languages respectively. After that, the original source code can process the obtained Intermediate Representations and the graph information can be constructed. Note that in our study we do not reuse any previously-reported effectiveness of our study subjects. Instead, we re-train and re-evaluate all of them on our own server to avoid replication bias [49]. The data fusion approaches are implemented by ourselves.

1. https://github.com/SpringMagnolia/Bm25Vectorizer
2. https://polyglot-compiler.github.io/JLang/
3. https://pypi.org/project/dis/

6 Experiment Results

6.1 RQ1: The Effectiveness of Standalone Techniques

The effectiveness of standalone code search techniques is shown in the first part of Table 4. We note that generally, pre-training based techniques achieve the best performance among all the techniques. For instance, the MRRs of *GraphCodeBERT* and *CodeBERT* on the CSN-Python dataset are 0.682 and 0.669 respectively, exceeding those of the rest techniques to a large extent (e.g., *GraphCodeBERT* outperforms *Self-attention*, the most effective one from the rest techniques whose MRR is 0.444, by around 55%). In contrast, the IR-based technique, *BM25*, achieves the lowest performance with its MRR being 0.140 and 0.095 on the two datasets. This result reflects the rationale of recently-proposed code search techniques to focus more on the semantic information rather than the textual information. Such significant effectiveness differences among diverse categories also explain our observation in Figure 1 that the shade of *GraphCodeBERT* is much darker than those of the others. We also note that supervised learning based techniques can outperform pre-training based ones under certain settings. Specifically, on the CSN-Java dataset, the SuccessRate@10 of *DeGraphCS* is the highest among the six techniques, even slightly exceeding that of *GraphCodeBERT* (0.742 vs. 0.721).

Several techniques such as *DeGraphCS* and *CodeBERT* have different performances on the two datasets (e.g., the MRRs of *DeGraphCS* are 0.307 and 0.489 on the Python and Java datasets respectively). This may be caused by the programming language differences: previous studies that evaluate on multiple PLs also observe such a phenomenon [25], [26]. Besides, we use the official released hyper-parameters in our experiments. As observed by Zeng et al. [49], the optimal hyper-parameters for different datasets may be different. We also note that combining information from multiple modals (i.e., the *Multi-modal* technique) does not necessarily provide better performance compared with its individual components that utilize more advanced techniques than the original ones. For instance, *Self-attention* achieves an MRR of 0.444 on the CSN-Python dataset, which is slightly higher than that of *Multi-modal*, i.e., 0.434. Recall that the original *Multi-modal* technique is evaluated on 1k queries for C language. Our results illustrate the threats to its external validity: different PLs and large-scale query sets may incur effectiveness difference.

Pre-training based techniques are generally the most effective ones while the IR-based has the lowest effectiveness. Supervised learning based techniques could achieve the optimum performance under certain settings.

TABLE 4: The effectiveness of standalone code search techniques and fusion approaches on our evaluation datasets.

Category	Technique	CSN-Python				CSN-Java			
		MRR	**SR@1**	**SR@5**	**SR@10**	**MRR**	**SR@1**	**SR@5**	**SR@10**
Standalone	BM25	0.140	0.097	0.182	0.223	0.095	0.068	0.122	0.150
	Self-attention	0.444	0.343	0.566	0.639	0.436	0.312	0.579	0.683
	DeGraphCS	0.307	0.202	0.421	0.522	0.489	0.361	0.642	**0.742**
	Multi-modal	0.434	0.331	0.559	0.631	0.486	0.359	0.639	0.737
	CodeBERT	0.669	0.563	0.802	0.857	0.522	0.420	0.646	0.707
	GraphCodeBERT	**0.682**	**0.577**	**0.812**	**0.865**	**0.534**	**0.432**	**0.659**	0.721
Rank-based Fusion	$\texttt{Dancer}_{Borda\,count}$	**0.913**	**0.881**	**0.955**	**0.969**	**0.878**	**0.844**	**0.919**	**0.939**
	$\texttt{Dancer}_{Reciprocal\,rank}$	0.822	0.755	0.894	0.947	0.743	0.656	0.835	0.905
	$\texttt{Dancer}_{Condorcet\,criterion}$	0.856	0.794	0.936	0.966	0.751	0.670	0.850	0.911
Score-based Fusion	$\texttt{Dancer}_{CombMIN}$	0.135	0.075	0.198	0.256	0.148	0.064	0.233	0.314
	$\texttt{Dancer}_{CombMAX}$	0.518	0.287	0.855	0.917	0.418	0.210	0.742	0.826
	$\texttt{Dancer}_{CombSUM}$	**0.922**	**0.894**	**0.956**	**0.968**	**0.872**	**0.839**	**0.913**	**0.936**
	$\texttt{Dancer}_{CombANZ}$	0.802	0.727	0.893	0.930	0.805	0.744	0.882	0.906
	$\texttt{Dancer}_{CombMNZ}$	0.917	0.889	0.952	0.967	0.850	0.819	0.888	0.909

6.2 RQ2: The Effectiveness of Fusion Approach

The effectiveness of the eight fusion approaches is listed in the second and third parts of Table 4. We note that generally, both rank-based fusion approach and score-based fusion approach can provide promising fusion results. For instance, the MRRs of $\texttt{Dancer}_{Reciprocal\,rank}$ and $\texttt{Dancer}_{CombANZ}$ always reach around 0.8 on the CSN-Python and CSN-Java datasets. The optimal performances of the rank-based and score-based approaches, as highlighted in the table, can be even better. Specifically, $\texttt{Dancer}_{CombSUM}$ achieves the best performance on the CSN-Python dataset with its MRR reaching 0.922. It outperforms the most effective standalone technique, *GraphCodeBERT*, by around 35% (0.922 vs. 0.682), and the least effective technique, *BM25*, by around 550% (0.922 vs. 0.140). The SuccessRate@1 of $\texttt{Dancer}_{CombSUM}$ is nearly 0.9, indicating that for around 90% of the queries, it can rank the oracle code at the first position. Furthermore, the SuccessRate@10 of $\texttt{Dancer}_{CombSUM}$ is nearly 0.97, which means that for almost all the queries, their corresponding code can be ranked at top-10 positions. As for the CSN-Java dataset, $\texttt{Dancer}_{Borda\,count}$ achieves the best performance with its MRR being 0.878, which is slightly higher than that of $\texttt{Dancer}_{CombSUM}$ (0.872). Such an MRR exceeds those of *GraphCodeBERT* (0.534) and *BM25* (0.095) by around 65% and 825% respectively. The SuccessRate@1 and SuccessRate@10 of $\texttt{Dancer}_{Borda\,count}$ reach 0.87 and 0.94 respectively, which are also extremely high values.

We also note that two fusion approaches (i.e., $\texttt{Dancer}_{CombMIN}$ and $\texttt{Dancer}_{CombMAX}$) do not obtain promising results. The effectiveness of such fusion approaches is even not as good as standalone techniques. For instance, the MRR of $\texttt{Dancer}_{CombMIN}$ on the CSN-Python dataset is 0.135, lower than that of *BM25* (0.140). Such results may indicate that these fusion approaches are not suitable for combining different code searchers. We investigate the queries whose oracle can be ranked at top-k positions by at least one standalone technique but are unsuccessfully handled by $\texttt{Dancer}_{CombMIN}$. The numbers of such queries when k=1/5/10 are 16,851/16,996/16,111 (18,834/18,791/17,426) respectively on the CSN-Python (CSN-Java) dataset. Similarly, the numbers for $\texttt{Dancer}_{CombMAX}$ are 12,149/2,432/1,463 (14,911/5,097/3,670) respectively. Such results reveal that both approaches could be easily affected by scores inaccurately assigned to non-relevant code snippets. For instance, if a non-relevant code snippet obtains a high score from a code search technique, it is likely to be ranked at top positions when using $\texttt{Dancer}_{CombMAX}$. In contrast, all the other six approaches fuse the results from a holistic perspective and thus can alleviate the effects of such inaccurate values. For instance, $\texttt{Dancer}_{CombSUM}$ adds the scores returned by all the systems for a code snippet and thus the effects of inaccurate values will be mostly eliminated.

TABLE 5: Effectiveness of variants of $\texttt{Dancer}_{CombSUM}$ on our evaluation datasets.

Fused techniques	CSN-Python		CSN-Java	
	MRR	**↓ (%)**	**MRR**	**↓ (%)**
-BM25	0.919	0.3	0.865	0.8
-Self-attention	0.902	2.2	0.831	4.7
-DeGraphCS	0.875	5.1	0.823	5.6
-Multi-modal	0.904	2.0	0.823	5.6
-CodeBERT	0.827	10.3	0.794	8.9
-GraphCodeBERT	0.826	10.4	0.792	9.2
$\texttt{Dancer}_{CombSUM}$ (6 techniques)	0.922		0.872	

TABLE 6: Effectiveness of variants of $\texttt{Dancer}_{Borda\,count}$ on our evaluation datasets.

Fused techniques	CSN-Python		CSN-Java	
	MRR	**↓ (%)**	**MRR**	**↓ (%)**
-BM25	0.911	0.2	0.876	0.2
-Self-attention	0.895	2.0	0.870	0.9
-DeGraphCS	0.895	2.0	0.862	1.8
-Multi-modal	0.897	1.8	0.862	1.8
-CodeBERT	0.812	11.1	0.812	7.5
-GraphCodeBERT	0.811	11.2	0.802	8.7
$\texttt{Dancer}_{Borda\,count}$ (6 techniques)	0.913		0.878	

Data fusion approaches can help boost the code search effectiveness significantly. In terms of MRR, the most effective fusion approach can bring 35% - 550% and 65% - 825% effectiveness enhancement compared with standalone techniques on the two datasets respectively.

6.3 RQ3: Contributions of Each Standalone Technique

To dissect the contribution of each standalone technique to the final fusion results, we remove one technique at each time and evaluate the effectiveness achieved by the remaining five techniques. We focus on the $\texttt{Dancer}_{CombSUM}$ and

$\texttt{Dancer}_{Borda\,count}$ since according to our results in the last RQ, they are the most effective fusion approaches.

In this RQ, we investigate the degradation of the MRR metric, since it reflects the overall effectiveness. Results are shown in Table 5 and Table 6. We note each standalone technique contributes to the final results, which also demonstrates the rationale of our hypothesis. Regarding to the quantitative analysis, we find a consistent trend across different datasets for both fusion approaches: the pre-training based techniques contribute the most to the final results while the IR based one contributes the least. Specifically, on the CSN-Python dataset, the MRR of fusion results will drop to 0.826/0.811 without *GraphCodeBERT*, a decrease of 10.4%/11.2% for the $\texttt{Dancer}_{CombSUM}$/$\texttt{Dancer}_{Borda\,count}$ approach. In contrast, under the same setting, the fusion effectiveness will only experience a decrease of 0.2% - 0.3% if *BM25* is excluded, an order of magnitude lower than the decrease experienced without *GraphCodeBERT*. We also note the contribution of each of the supervised learning based technique is considerable, but is not the most significant. For instance, on the CSN-Java dataset, the MRR of the results of $\texttt{Dancer}_{CombSUM}$ will decrease by 5.6% if *DeGraphCS* or *Multi-modal* is excluded. Such a trend correlates to the effectiveness of each standalone technique: the most effective technique contributes the most to the fusion results while the least effective one contributes the least.

Generally, each involved technique contributes to the fusion performance. The contribution from the most effective standalone technique, GraphCodeBERT, is an order of magnitude higher than that from BM25, which contributes the least.

6.4 RQ4: The Time Cost of Dancer

We first introduce how we calculate the time cost of each standalone technique. Usually, each deep learning model has a parameter named *batch size*. For a model with the batch size of BS, it means that during testing, the number of BS queries are handled simultaneously. We suppose that in practice, the "first come first served" principle is adopted, which means each query will be handled individually. Therefore, to approximate the time cost under real-world scenarios, we set the batch size to 1 for each deep learning model and record their time cost on the test sets. Suppose a model spends T seconds on the test set, and the total number of queries is denoted as Q, the average time cost of the model for a given query (t) is calculated as the following: $t = \frac{T}{Q}$. Since we may not have enough computation resource in practice, we also assume that each standalone technique is executed sequentially. As a result, the overall response time of Dancer for a given query is estimated by summing up the time costs of each standalone technique and the fusion algorithm.

Table 7 illustrates the time costs of each standalone technique. We first note that all the involved techniques are generally efficient, i.e., they generally spend less than 0.01s dealing with a query. For instance, *GraphCodeBERT*, the least efficient one among the learning based techniques, spends less than 0.05 searching for candidate code snippets for a given query on the CSN-Python dataset. On average, the IR-based technique *BM25* has the highest time cost for a given query on the two datasets, i.e., 0.081s and 0.133s, respectively. This can be explained by the fact that other techniques are executed on GPU, which is more powerful for numeric calculation than CPU. We also observe that on average, a technique spends more time dealing with queries from the CSN-Java dataset compared with those from the CSN-Python dataset. Through in-depth analysis, we find that it is because queries from the CSN-Java dataset are generally more complex. Specifically, the average token numbers of the queries from the CSN-Java and CSN-Python datsets are 21.2 and 15.3 respectively.

By summing up the time costs of each standalone technique, we can obtain the overall time cost for the six standalone techniques to generate their search results for a given query, which is 0.151s on the CSN-Python dataset and 0.210s on the CSN-Java dataset. We have also investigated the time cost of using the data fusion approaches to generate the final ranking results. The results show that all the eight fusion approaches take less than 5 milliseconds to combine the results, which is negligible compared with that spent on generating the results of standalone techniques. To summarize, in practice, the response time for a given query would be around 0.2s, which is affordable. Table 7 also illustrates the time costs of training each model. We note that training is much more time-consuming than test. For instance, on the CSN-Python dataset, it takes 30,888/30 seconds to train/test *Self-attention*, the former being around 5000X larger than the latter. Totally, it takes 610,506 seconds ($\approx$ 7 days) to train these models while only 3,340 seconds (less than 1 hour) to obtain the test results, on the CSN-Python dataset. This also indicates that once trained, these models can be comparatively efficiently queried. As for training, *Multi-modal* is the most time-consuming technique, which is within our expectation since it needs to combine three modals and the gradient descending could be comparatively slow.

In practice, the response time of Dancer for a given query would be around 0.2s, which is an affordable time cost.

7 Discussion

7.1 Case Analysis

To deeply understand the success of our fusion approach, we investigate the queries whose corresponding code snippets cannot be ranked at top-k positions by any standalone technique but are successfully handled by our $\texttt{Dancer}_{CombSUM}$ fusion approach. Totally, the numbers of such queries when k=1/5/10 are 2,422/354/108 (3,454/567/188) respectively on the CSN-Python (CSN-Java) dataset. Table 8 gives a concrete example from the CSN-Python dataset. The first part of the table demonstrates the query and the code. The intention of this query is to generate an integer array whose range is detailed, and the corresponding code fulfills this functionality by calling the `arange` API from the *Numpy* package. The second part of the table illustrates the ids of the code snippets that are ranked at top-10 by each standalone technique and the fusion approach. The id of the oracle code is 21841.

From the results, we note that none of the existing technique can successfully rank the oracle code at top-1 position:

TABLE 7: The time cost of Dancer on two test sets.

Technique	CSN-Python (Query: 22176)			CSN-Java (Query: 26909)		
	Training (s)	Test (s)	Time/query (s)	Training (s)	Test (s)	Time/query (s)
BM25	-	1,786	0.081	-	3,573	0.133
Self-attention	30,888	30	0.001	29,686	44	0.002
DeGraphCS	164,800	98	0.004	157,722	132	0.005
Multi-modal	317,296	186	0.008	303,666	264	0.010
CodeBERT	8,640	160	0.007	8,304	179	0.007
GraphCodeBERT	88,882	1,080	0.049	85,069	1,458	0.054
Total	610,506	3,340	0.151	584,447	5,650	0.210

TABLE 8: A test query and its corresponding code which is successfully ranked at top-1 by our fusion approach but failed by each standalone technique. The numbers refer to the ids of different candidate code snippets.

Query:
Generate integers from start to (and including!) stop.

Code:

```
def iseq(start=0, stop=None, inc=1):
    if stop is None:
        stop = start
        start = 0
        inc = 1
    return arange(start, stop+inc, inc)
```

BM25	14345	**21841**	11997	7549	7584	7583	12026	1052	12028	12695
Self-attention	19239	459	**21841**	4719	16685	460	15407	18888	875	9510
DeGraphCS	7108	**21841**	6979	4520	17155	13642	5284	15160	20641	19645
Multi-modal	11997	**21841**	14345	21818	7548	12695	3117	7549	16399	19139
CodeBERT	9265	7108	**21841**	2935	367	5047	13642	17155	697	19645
GraphCodeBERT	9265	2935	**21841**	7108	367	12414	19645	17155	5047	21772
$\texttt{Dancer}_{CombSUM}$	**21841**	7108	17155	19645	13642	697	11997	16891	9265	5284

they are disturbed by other candidate code snippets in the codebase to varying degrees. For instance, the *BM25* technique ranks the 14345_{th} code snippet higher than the oracle. By in-depth analysis, we find that this code snippet shares high token similarities with the oracle code: they both contain two overlapped tokens with the query, i.e., *start* and *stop*. Moreover, the token *start* appears for six times in that code snippet, which exceeds the number of the oracle (i.e., four) and thus leads to a higher term frequency value. As we have introduced in Section 2, the *BM25* technique is a TF-IDF-like algorithm and hence that code snippet obtains a relevance score higher than that of the oracle, leading to the inaccuracy ranking results. We also observe that different techniques are disturbed by different candidate code snippets. For instance, unlike *BM25* which prioritizes a code snippet with high textual similarity to the oracle, *CodeBERT* and *GraphCodeBERT* both rank another code snippet (i.e., the 9265_{th}) at the top position. This code snippet has high structure similarity to the oracle since it also contains an `if-else` structure. However, this code snippet is not within the top-10 results from any other standalone technique. Fortunately, the oracle is always ranked at top-3 by all the standalone techniques. Therefore, after combining the results from all the techniques, the oracle is successfully ranked at top-1.

This example shows that different standalone techniques have unique weaknesses on searching for the correct code, and fusing the results from different techniques have potential to alleviate the inaccuracy of each standalone technique. This is the reason for the success of our fusion approach.

7.2 Comparison with Another Fusion Approach

Besides the data fusion approach, another widely-used approach in the IR domain for fusion is learning to rank [53], which typically relies on training a machine learning model for a ranking task. To further compare the effectiveness of these two approaches on fusing code search techniques, we perform another experiment where we use a well-known learning to rank model, i.e., rankSVM [54], to combine the six studied techniques. Specifically, each code snippet c is associated with a vector $Scores(c) =< NRS(c, r_1), \ldots, NRS(c, r_6) >$, where $NRS(c, r_i)$ denotes the normalized relevance score of the retrieved code snippet by the i_{th} technique. After that, this vector is projected to a high-dimension vector space by a kernel function. Finally, a linear function is used to output the final score for the code snippet c based on the projected vector, and it is learned during the training process. The training goal is to rank the oracle code snippet at the top position (i.e., with the highest output score).

Following the previous study [55], we perform a 10-fold cross validation on the evaluation set and sum up the result in each fold as the final result on the test set. The comparison between data fusion and learning to rank is shown in Table 9. From the results, we observe that the learning to rank model has slightly lower performance compared with the two most effective data fusion approaches. For instance, on the CSN-Python dataset, the MRR of the learning to rank model is 0.909, lower than that of $\texttt{Dancer}_{CombSUM}$, which is 0.922. Similarly, on the CSN-Java dataset, the MRR of the learning to rank model is 0.863, still lower than that of $\texttt{Dancer}_{Borda\,count}$, which is 0.878. Such results indicate that (1) combining different code search techniques is promising, since both fusion approaches outperform standalone techniques significantly; and (2) the data fusion approach outperforms the learning to rank approach on combining different code search techniques, which demonstrates the rationale of our study.

Table 10 illustrates a concrete example. The oracle code snippet first checks the output (i.e., `message`) of a function and then writes the `message` into an argument. *BM25* successfully ranks the oracle code at top-1 because there are a number of overlapped tokens between the query and the code such as *message* and *key*. Relying on the score returned by *BM25*, $\texttt{Dancer}_{CombSUM}$ also ranks the oracle code at top-1. In contrast, rankSVM ranks the 4777_{th} candidate code snippet at the first position, which is ranked at top positions by *CodeBERT* (the 5_{th}) and *GraphCodeBERT* (the 2_{nd}). This case indicates that the weights assigned by the learning to rank model tend to favor the scores from more effective standalone techniques (e.g., *GraphCodeBERT*), but such incline will not always be appropriate and may lead to the inaccuracy of the ranking results.

TABLE 9: The comparison between the data fusion and learning to rank approaches on our evaluation datasets.

Technique	CSN-Python				CSN-Java			
	MRR	SR@1	SR@5	SR@10	MRR	SR@1	SR@5	SR@10
$\texttt{Dancer}_{Borda\ count}$	0.913	0.881	0.955	0.969	0.878	0.844	0.919	0.939
$\texttt{Dancer}_{CombSUM}$	0.922	0.894	0.956	0.968	0.872	0.839	0.913	0.936
Learning to rank	0.909	0.883	0.941	0.953	0.863	0.827	0.905	0.932

TABLE 10: A test query and its corresponding code which is successfully ranked at top-1 by our fusion approach but failed by the learning to rank model.

Query:
Check for message and write the message with key "message".

Code:

```
def poke(self, context):
    message = self.pubsub.get_message()
    if message and message['type'] == 'message':
        context['ti'].xcom_push(key='message', value=message)
        self.pubsub.unsubscribe(self.channels)
        return True
    return False
```

BM25	**307**	17340	10204	6996	18048	2337	10203	11931	11932	11933
Self-attention	9066	21245	12866	17778	18401	5455	9696	4538	9226	3991
DeGraphCS	13373	21290	2196	15773	17857	11972	7380	13575	18284	7477
Multi-modal	13020	10141	18874	13675	17923	21674	1187	6337	20759	15430
CodeBERT	22009	**307**	14369	4539	4777	14376	14044	8423	20259	14609
GraphCodeBERT	20454	4777	21266	20103	307	22015	15733	22009	20259	9571
$\texttt{Dancer}_{CombSUM}$	**307**	4777	22009	20454	14369	8423	14376	20259	14044	15733
rankSVM	4777	**307**	20454	22009	8423	14369	14376	20259	14044	15733

TABLE 11: Effectiveness achieved by $\texttt{Dancer}_{CombSUM}$ when fusing different numbers of code search techniques.

Step	Involved techniques	MRR (CSN-Python)	MRR (CSN-Java)
1	GraphCodeBERT	0.682	0.534
2	+ CodeBERT	0.699	0.546
3	+ Self-attention	0.874	0.765
4	+ Multi-modal	0.900	0.858
5	+ DeGraphCS	0.919	0.865
6	+ BM25	0.922	0.872

In each step, the listed technique is combined with those used in the last step for fusion.

7.3 The Trade-Off between the Effectiveness and the Computation Resource

In our experiments, we show that by fusing totally six different code search techniques, the effectiveness can be enhanced to a large extent compared with each standalone technique. The majority of the involved techniques are learning based, which require adequate training before they can be used. Therefore, one may not have the enough computation resource if he/she would like to apply our approach in practice. To address this limitation, we investigate the question that can we gain significant effectiveness enhancement against the standalone techniques but use as few techniques as possible at the same time. To this end, we design a strategy where each time we select the most effective one from techniques that are not fused yet and combine it with those already fused. We then calculate the effectiveness achieved at each time. This experiment differs from RQ3: in RQ3, we always fuse five techniques to dissect the contribution of the excluded one; while here we fuse different numbers of techniques and investigate the achieved effectiveness.

We use $\texttt{Dancer}_{CombSUM}$ as the fusion approach to perform such an experiment and the results are shown in Table 11. We find that when involving the first three techniques, the effectiveness of the fusion approach already exceeds that of the standalone techniques to a large extent. Specifically, on the CSN-Python dataset, if we combine *GraphCodeBERT*, *CodeBERT*, and *Self-attention* via the CombSUM approach, the obtained MRR will be 0.874, exceeding that of *GraphCodeBERT* by around 30%. Also, such a value already reaches 95% of the upper-bound effectiveness of combining totally six techniques (0.874/0.922). Similarly, on the CSN-Java dataset, such a strategy can achieve an MRR of 0.765, exceeding that of *GraphCodeBERT* by around 45%. We also observe that if we combine four or more techniques, the obtained effectiveness enhancement is significantly less than that obtained when three techniques are used.

Such results indicate that in practice, one could only select to use three of the best-performing techniques if the computation resource is restricted. Then, the obtained effectiveness is still significantly better than that of the standalone techniques.

7.4 The Effect of Score Normalization

In our approach, we apply a score normalization step before applying score-based data fusion approaches, aiming at alleviating the scale bias of different code search techniques. To investigate the effect of the score normalization step, we disable this step and evaluate the effectiveness of the resultant $\texttt{Dancer}_{CombSUM}$ approach, which is the most effective score-based approach in our experiment.

Results are shown in Table 12. We find that after removing this step, the effectiveness of the $\texttt{Dancer}_{CombSUM}$ approach experiences a systematic decrease on all the metrics. For instance, the MRR decreases from 0.922 to 0.888 on the CSN-Python dataset and from 0.872 to 0.790 on the CSN-Java dataset, a degree of 3.7% and 9.4% on the two datasets respectively. Similar trends are observed when it comes to the SuccessRate metrics. Specifically, the three SuccessRate metrics will be degraded by 3.4% - 4.0% on the CSN-Python dataset and by 9.5% - 10.4% on the CSN-Java dataset. Such results show that our score normalization step contributes to the final performance of the score-based fusion approaches significantly, and thus our design decision is reasonable.

7.5 Implications

Evaluating Code Search Techniques. Traditionally, code search techniques are often used and evaluated individually. Our work shows that it is easy to combine different code search techniques. We recommend that users should try to combine multiple techniques, if the computation resource is allowed. This further implies that, for tool makers proposing a new code search technique in the future, they should not only evaluate the performance of the technique in isolation but also understand how the technique contributes when combining with existing techniques.

Efficiency. In the existing evaluation of code search techniques, efficiency often receives less attention than effectiveness. Our study reveals that state-of-the-art code search

TABLE 12: The effectiveness of $\texttt{Dancer}_{CombSUM}$ with and without score normalization.

Variant	CSN-Python								CSN-Java							
	MRR	↓ (%)	SR@1	↓ (%)	SR@5	↓ (%)	SR@10	↓ (%)	MRR	↓ (%)	SR@1	↓ (%)	SR@5	↓ (%)	SR@10	↓ (%)
$\texttt{Dancer}_{CombSUM}$ w/o normalization	0.888	3.7	0.858	4.0	0.923	3.5	0.935	3.4	0.790	9.4	0.759	9.5	0.820	10.2	0.839	10.4
$\texttt{Dancer}_{CombSUM}$ w/ normalization	0.922		0.894		0.956		0.968		0.872		0.839		0.913		0.936	

techniques are generally efficient, e.g., on average, our study subjects usually spend less than 0.1 second dealing with a single query. This suggests that efficiency should be taken into consideration when evaluating newly-proposed code search techniques: a newly-proposed approach should be efficient to make the time cost of combining it with existing techniques in a reasonable limit.

Methods for Combining Approaches. Our study uses a classical approach from the information retrieval domain, i.e., data fusion, to combine different code search techniques. Our results show that such a simple combination can outperform standalone techniques significantly. However, our combination approach may have limitation. Specifically, we treat each technique as a black box and only combine their results (i.e., the scores and ranks of code snippets). This strategy misses the opportunity to utilize the intermediate results produced during the calculation. Therefore, more novel ways to combine different code search techniques are left to be explored in the future.

Technique Categorization. Our study demonstrates for the first time the complementarity of existing code search techniques from the perspectives of both the inter- and intra-categories. This observation suggests that future technical improvement within any of the three categories should be appreciated since it holds potential to boost the fusion effectiveness.

7.6 Threats to Validity

External Validity. All our results and findings may be restricted to the CodeSearchNet dataset. However, this dataset is the largest and most popular dataset in the code search domain [1], [30], [56]. As we have shown in Table 3, the test set of this dataset usually contains tens of thousands of queries. Such a large-scale ensures the diversity of the test set and we consider performing experiments on other benchmarks as our future work. Also, our study focuses on the Java and Python programming languages since they are widely-studied languages in the code search domain [1], [5], [7], [10] and there exist well-maintained program analysis tools for these languages. We recall that some of our investigated approaches need to parse the code snippets into tree structures, obtain the intermediate representations, or build graph structures (e.g., DegraphCS and Multi-modal). Therefore, to evaluate datasets for other languages, one should consider whether appropriate tools are readily available for reproduction. Nevertheless, we will explore such evaluations in future research.

Internal Validity. We totally include six code search techniques in this study. To avoid replication bias, we reuse the source code released by the authors of each technique. The data fusion approaches are implemented by ourselves. To ensure there is no implementation errors, the authors carefully checked the code. Furthermore, all the code and results of this study are open accessed at our online repository.

8 Related Work

8.1 Combinations via Data Fusion

As an effective approach in the IR domain, data fusion has been utilized by researchers to address software engineering (SE) tasks. Fusion Localizer [38] tries to combine fault localization results from different techniques using five classic data fusion approaches such as the CombSUM and Borda count. Revelle et al. [57] applied data fusion to feature location, which is to identify the source code that implements specific functionality in software. They fused program textual information, dynamic execution features, and dependency relations. Rahman et al. [58] provided actionable insights for avoiding non-reproducible bugs by combining the analyses from multi-modal studies. Motivated by these studies, we propose to apply data fusion approaches to improve code search effectiveness. To our best knowledge, we are the first to explore this direction.

8.2 Combinations via Learning to Rank

As another widely-used technique in the IR domain, learning to rank has also been utilized by researchers to address various SE tasks. Xuan and Monperrus [59] explored to combine different spectra-based fault localization (SBFL) formulae with a learning to rank model. Later, Zou et al. [55] showed that all the fault localization formulae, including the spectra-based ones and those built upon other types of information such as the stack trace and program slicing, can be combined through a learning to rank model. Ye et al. [60] utilized a learning to rank model to recommend relevant files for bug reports in which features encoding domain knowledge such as the API documentation and code change history are fused.

8.3 Other Specially-Designed Combinations

Beyond leveraging off-the-shelf IR algorithms, researchers also proposed specially-designed strategies to fuse information from differetn sources. Benton et al. [61] utilized the patches generated by different automated program repair techniques to help refine the fault localization results through *specially-designed heuristics*. Fang et al. [62] fused the syntax and semantic features of code for detecting functional code clones through a joint code representation model. Wang et al. [63] proposed that the effectiveness for assessing patch correctness can be boosted through combining the results from different techniques, using the *majority voting* strategy. Lin et al. [64] combined heuristics with a learning-based model by predicting which technique to use for better comment update results. These studies also motivate us to explore the combination of different code search techniques.

9 Conclusion

We study code search techniques from the literature and show, through an explorative investigation, that various

approaches proposed so far are complementary. We then propose `Dancer`, a fusion-based approach for enhancing code search effectiveness. Our approach fuses the search results obtained by existing code search techniques in a manner that enables to leverage their complementarity in order to achieve substantially higher performance. Our experiments investigated six state-of-the-art code search techniques and explored eight classic data fusion approaches. Applied on two large-scale datasets, the off-the-shelf data fusion approaches significantly outperform each state-of-the-art technique. Future studies in code search domain could devote to explore more specially-designed fusion approaches for further effectiveness enhancement, which is a promising direction. **Availability:** For open science, we release our artifact at: **https://doi.org/10.5281/zenodo.7016108**, including the source code and results of both standalone techniques and our fusion approaches.

Acknowledgments

References

[1] W. Sun, C. Fang, Y. Chen, G. Tao, T. Han, and Q. Zhang, "Code search based on context-aware code translation," in *Proceedings of the 44th International Conference on Software Engineering*, 2022.

[2] J. Brandt, M. Dontcheva, M. Weskamp, and S. R. Klemmer, "Example-centric programming: integrating web search into the development environment," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2010, pp. 513–522.

[3] M. Gharehyazie, B. Ray, and V. Filkov, "Some from here, some from there: Cross-project code reuse in github," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 2017, pp. 291–301.

[4] J. Brandt, P. J. Guo, J. Lewenstein, M. Dontcheva, and S. R. Klemmer, "Two studies of opportunistic programming: interleaving web foraging, learning, and writing code," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2009, pp. 1589–1598.

[5] J. Shuai, L. Xu, C. Liu, M. Yan, X. Xia, and Y. Lei, "Improving code search with co-attentive representation learning," in *Proceedings of the 28th International Conference on Program Comprehension*, 2020, pp. 196–207.

[6] K. Kim, D. Kim, T. F. Bissyandé, E. Choi, L. Li, J. Klein, and Y. Le Traon, "FaCoY: a code-to-code search engine," in *Proceedings of the 40th International Conference on Software Engineering*. ACM, 2018, pp. 946–957.

[7] J. Cambronero, H. Li, S. Kim, K. Sen, and S. Chandra, "When deep learning met code search," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 964–974.

[8] C. McMillan, N. Hariri, D. Poshyvanyk, J. Cleland-Huang, and B. Mobasher, "Recommending source code for use in rapid software prototypes," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 848–858.

[9] C. Sadowski, K. T. Stolee, and S. Elbaum, "How developers search for code: a case study," in *Proceedings of the 2015 10th joint meeting on foundations of software engineering*, 2015, pp. 191–201.

[10] L. Xu, H. Yang, C. Liu, J. Shuai, M. Yan, Y. Lei, and Z. Xu, "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)*. IEEE, 2021, pp. 342–353.

[11] X. Xia, L. Bao, D. Lo, P. S. Kochhar, A. E. Hassan, and Z. Xing, "What do developers search for on the web?" *Empirical Software Engineering*, vol. 22, no. 6, pp. 3149–3185, 2017.

[12] C. Ling, Z. Lin, Y. Zou, and B. Xie, "Adaptive deep code search," in *Proceedings of the 28th International Conference on Program Comprehension*, 2020, pp. 48–59.

[13] Z. Yao, J. R. Peddamail, and H. Sun, "Coacor: Code annotation for code retrieval with reinforcement learning," in *The world wide web conference*, 2019, pp. 2203–2214.

[14] L. Di Grazia and M. Pradel, "Code search: A survey of techniques for finding code," *arXiv preprint arXiv:2204.02765*, 2022.

[15] C. Liu, X. Xia, D. Lo, C. Gao, X. Yang, and J. Grundy, "Opportunities and challenges in code search tools," *ACM Computing Surveys (CSUR)*, vol. 54, no. 9, pp. 1–40, 2021.

[16] S. Yan, H. Yu, Y. Chen, B. Shen, and L. Jiang, "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)*. IEEE, 2020, pp. 344–354.

[17] M. Lu, X. Sun, S. Wang, D. Lo, and Y. Duan, "Query expansion via wordnet for effective code search," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 2015, pp. 545–549.

[18] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu, "Portfolio: finding relevant functions and their usage," in *Proceedings of the 33rd International Conference on Software Engineering*, 2011, pp. 111–120.

[19] O. A. Lemos, A. C. de Paula, F. C. Zanichelli, and C. V. Lopes, "Thesaurus-based automatic query expansion for interface-driven code search," in *Proceedings of the 11th working conference on mining software repositories*, 2014, pp. 212–221.

[20] F. Zhang, H. Niu, I. Keivanloo, and Y. Zou, "Expanding queries for code search using semantically related api class-names," *IEEE Transactions on Software Engineering*, vol. 44, no. 11, pp. 1070–1082, 2017.

[21] X. Gu, H. Zhang, and S. Kim, "Deep code search," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 933–944.

[22] Y. Wan, J. Shu, Y. Sui, G. Xu, Z. Zhao, J. Wu, and P. Yu, "Multi-modal attention network learning for semantic source code retrieval," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 13–25.

[23] Q. Zhu, Z. Sun, X. Liang, Y. Xiong, and L. Zhang, "Ocor: an overlapping-aware code retriever," in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2020, pp. 883–894.

[24] W. Ye, R. Xie, J. Zhang, T. Hu, X. Wang, and S. Zhang, "Leveraging code generation to improve code retrieval and summarization via dual learning," in *Proceedings of The Web Conference 2020*, 2020, pp. 2309–2319.

[25] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*, 2020, pp. 1536–1547.

[26] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu *et al.*, "Graphcodebert: Pre-training code representations with data flow," in *ICLR*, 2021.

[27] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, 2021, pp. 8696–8708.

[28] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang *et al.*, "Codexglue: A machine learning benchmark dataset for code understanding and generation," in *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*, 2021.

[29] C. Zeng, Y. Yu, S. Li, X. Xia, Z. Wang, M. Geng, L. Bai, W. Dong, and X. Liao, "degraphcs: Embedding variable-based flow graph for neural code search," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2022.

[30] J. Gu, Z. Chen, and M. Monperrus, "Multimodal representation for neural code search," in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2021, pp. 483–494.

[31] N. D. Bui, Y. Yu, and L. Jiang, "Self-supervised contrastive learning for code retrieval and summarization via semantic-preserving transformations," in *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2021, pp. 511–521.

[32] G. DeSanctis and R. B. Gallupe, "A foundation for the study of group decision support systems," *Management science*, vol. 33, no. 5, pp. 589–609, 1987.

[33] E. A. Fox and J. A. Shaw, "Combination of multiple searches," *NIST special publication SP*, vol. 243, 1994.

[34] C. C. Vogt and G. W. Cottrell, "Fusion via a linear combination of scores," *Information retrieval*, vol. 1, no. 3, pp. 151–173, 1999.

[35] W. B. Croft, "Combining approaches to information retrieval," in *Advances in information retrieval*. Springer, 2002, pp. 1–36.

[36] J. A. Aslam and M. Montague, "Models for metasearch," in *Proceedings of the 24th annual international ACM SIGIR conference on Research and development in information retrieval*, 2001, pp. 276–284.

[37] F. C. Gey, N. Kando, and C. Peters, "Cross-language information retrieval: the way ahead," *Information processing & management*, vol. 41, no. 3, pp. 415–431, 2005.

[38] D. Lo and X. Xia, "Fusion fault localizers," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, 2014, pp. 127–138.

[39] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "Codesearchnet challenge: Evaluating the state of semantic code search," *arXiv preprint arXiv:1909.09436*, 2019.

[40] G. V. Cormack, C. L. Clarke, and S. Buettcher, "Reciprocal rank fusion outperforms condorcet and individual rank learning methods," in *Proceedings of the 32nd international ACM SIGIR conference on Research and development in information retrieval*, 2009, pp. 758–759.

[41] C. Dwork, R. Kumar, M. Naor, and D. Sivakumar, "Rank aggregation methods for the web," in *Proceedings of the 10th international conference on World Wide Web*, 2001, pp. 613–622.

[42] D. Frank Hsu and I. Taksa, "Comparing rank and score combination methods for data fusion in information retrieval," *Information retrieval*, vol. 8, no. 3, pp. 449–480, 2005.

[43] P. Bailey, A. Moffat, F. Scholer, and P. Thomas, "Retrieval consistency in the presence of query variations," in *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2017, pp. 395–404.

[44] C. D. Manning, *Introduction to information retrieval*. Syngress Publishing,, 2008.

[45] F. Lv, H. Zhang, J.-g. Lou, S. Wang, D. Zhang, and J. Zhao, "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)*. IEEE, 2015, pp. 260–270.

[46] Z. Sun, L. Li, Y. Liu, X. Du, and L. Li, "On the importance of building high-quality training datasets for neural code search," in *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. ACM, 2022.

[47] K. S. Tai, R. Socher, and C. D. Manning, "Improved semantic representations from tree-structured long short-term memory networks," in *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, 2015, pp. 1556–1566.

[48] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[49] Z. Zeng, H. Tan, H. Zhang, J. Li, Y. Zhang, and L. Zhang, "An extensive study on pre-trained models for program understanding and generation," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2022.

[50] S. Robertson, H. Zaragoza *et al.*, "The probabilistic relevance framework: Bm25 and beyond," *Foundations and Trends® in Information Retrieval*, vol. 3, no. 4, pp. 333–389, 2009.

[51] M. Wu, D. Hawking, A. Turpin, and F. Scholer, "Using anchor text for homepage and topic distillation search tasks," *Journal of the American Society for Information Science and Technology*, vol. 63, no. 6, pp. 1235–1255, 2012.

[52] M. Montague and J. A. Aslam, "Relevance score normalization for metasearch," in *Proceedings of the tenth international conference on Information and knowledge management*, 2001, pp. 427–433.

[53] T.-Y. Liu *et al.*, "Learning to rank for information retrieval," *Foundations and Trends® in Information Retrieval*, vol. 3, no. 3, pp. 225–331, 2009.

[54] T.-M. Kuo, C.-P. Lee, and C.-J. Lin, "Large-scale kernel ranksvm," in *Proceedings of the 2014 SIAM international conference on data mining*. SIAM, 2014, pp. 812–820.

[55] D. Zou, J. Liang, Y. Xiong, M. D. Ernst, and L. Zhang, "An empirical study of fault localization families and their combinations," *IEEE Transactions on Software Engineering*, 2019.

[56] X. Ling, L. Wu, S. Wang, G. Pan, T. Ma, F. Xu, A. X. Liu, C. Wu, and S. Ji, "Deep graph matching and searching for semantic code retrieval," *ACM Transactions on Knowledge Discovery from Data (TKDD)*, vol. 15, no. 5, pp. 1–21, 2021.

[57] M. Revelle, B. Dit, and D. Poshyvanyk, "Using data fusion and web mining to support feature location in software," in *2010 IEEE 18th International Conference on Program Comprehension*. IEEE, 2010, pp. 14–23.

[58] M. M. Rahman, F. Khomh, and M. Castelluccio, "Why are some bugs non-reproducible?:–an empirical investigation using data fusion–," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2020, pp. 605–616.

[59] J. Xuan and M. Monperrus, "Learning to combine multiple ranking metrics for fault localization," in *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 2014, pp. 191–200.

[60] X. Ye, R. Bunescu, and C. Liu, "Learning to rank relevant files for bug reports using domain knowledge," in *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*, 2014, pp. 689–699.

[61] S. Benton, X. Li, Y. Lou, and L. Zhang, "On the effectiveness of unified debugging: An extensive study on 16 program repair systems," in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2020, pp. 907–918.

[62] C. Fang, Z. Liu, Y. Shi, J. Huang, and Q. Shi, "Functional code clone detection with syntax and semantics fusion learning," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020, pp. 516–527.

[63] S. Wang, M. Wen, B. Lin, H. Wu, Y. Qin, D. Zou, X. Mao, and H. Jin, "Automated patch correctness assessment: How far are we?" in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2020.

[64] B. Lin, S. Wang, Z. Liu, X. Xia, and X. Mao, "Predictive comment updating with heuristics and ast-path-based neural learning: A two-phase approach," *IEEE Transactions on Software Engineering*, pp. 1–20, 2022.