I Know What You Are Searching for: Code Snippet Recommendation from Stack Overflow Posts

ZHIPENG GAO, Shanghai Institute for Advanced Study of Zhejiang University, China
XIN XIA, Huawei, China
DAVID LO, Singapore Management University, Singapore
JOHN GRUNDY, Monash University, Australia
XINDONG ZHANG, Alibaba Group, China
ZHENCHANG XING, CSIRO's Data61 & Australian National University, Australia

Stack Overflow has been heavily used by software developers to seek programming-related information. More and more developers use Community Question and Answer forums, such as Stack Overflow, to search for code examples of how to accomplish a certain coding task. This is often considered to be more efficient than working from source documentation, tutorials, or full worked examples. However, due to the complexity of these online Question and Answer forums and the very large volume of information they contain, developers can be overwhelmed by the sheer volume of available information. This makes it hard to find and/or even be aware of the most relevant code examples to meet their needs. To alleviate this issue, in this work, we present a query-driven code recommendation tool, named Que2Code, that identifies the best code snippets for a user query from Stack Overflow posts. Our approach has two main stages: (i) semantically equivalent question retrieval and (ii) best code snippet recommendation. During the first stage, for a given query question formulated by a developer, we first generate paraphrase questions for the input query as a way of query boosting and then retrieve the relevant Stack Overflow posted questions based on these generated questions. In the second stage, we collect all of the code snippets within questions retrieved in the first stage and develop a novel scheme to rank code snippet candidates from Stack Overflow posts via pairwise comparisons. To evaluate the performance of our proposed model, we conduct a large-scale experiment to evaluate the effectiveness of the semantically equivalent question retrieval task and best code snippet recommendation task separately on Python and Java datasets in Stack Overflow. We also perform a human study to measure how real-world developers perceive the results generated by our model. Both the automatic and human evaluation results demonstrate the promising performance of our model, and we have released our code and data to assist other researchers.

CCS Concepts: • **Software and its engineering → Software evolution**; **Maintaining software;**

Additional Key Words and Phrases: Code Search, Stack Overflow, paraphrase mining, Duplicate questions

This research was partially supported by ARC Laureate Fellowship FL190100035 and National Research Foundation, Singapore, under its Industry Alignment Fund – Pre-positioning (IAF-PP) Funding Initiative.

Authors' addresses: Z. Gao, Building #5, Zhangjiang Guochuang Center Phase III, No. 799 Dangui Road, Pudong New District, Shanghai 310058, China; email: zhipeng.gao@zju.edu.cn; X. Xia (corresponding author), No. 360 Jiangshu Road, BinJiang District, Hangzhou 310056, Zhejiang Province, China; email: xin.xia@acm.org; D. Lo, School of Information Systems, Singapore Management University, 80 Stamford Road, Singapore 178902, Singapore; email: davidlo@smu.edu.sg; J. Grundy, Building 6, 29 Ancora Imparo Way, Clayton Campus, Monash University VIC 3800, Australia; email: john.grundy@monash.edu; X. Zhang, 969 West Wen Yi Road, Yu Hang District, Hangzhou 311121, Zhejiang Province, China; email: zxd139923@alibaba-inc.com; Z. Xing, Building #108, 108 North Road, Australian National University, Canberra ACT 2600, Australia; email: Zhenchang.Xing@anu.edu.au.

1049-331X/2023/04-ART80 $15.00
https://doi.org/10.1145/3550150

ACM Reference format:
Zhipeng Gao, Xin Xia, David Lo, John Grundy, Xindong Zhang, and Zhenchang Xing. 2023. I Know What You Are Searching for: Code Snippet Recommendation from Stack Overflow Posts. *ACM Trans. Softw. Eng. Methodol.* 32, 3, Article 80 (April 2023), 42 pages.
https://doi.org/10.1145/3550150

1 INTRODUCTION

To deliver high-quality software more effectively and efficiently, many developers frequently use community **Question and Answer (Q&A)** sites, such as Stack Overflow, for solutions to their programming problems and tasks [69]. Code search is one such task that plays an important role in software development. Software developers spend about 19% of their development time searching for relevant code snippets on the web [8]. To help in programming problem solving, searching for useful code snippets from Stack Overflow has become a common part of developers' daily work [69]. Typically, when developers encounter a technical problem, they formulate their problem as a query and use a search engine (e.g., Google or Stack Overflow's own search tool) to obtain a list of possible relevant posts that may contain useful solutions to their problem. After that, developers have to read answers with various levels of quality included in the returned posts to identify possible solutions.

While search engines such as Google are widely used for searching for required information in Stack Overflow, there are a significant number of user queries that can not be satisfied [18]. One primary reason for failed Stack Overflow searches are poorly constructed queries and/or low-quality queries [17, 48, 68]. Unsuccessful searches using low-quality queries are difficult and ineffective due to the following challenges:

- *Query Mismatch.* Low-quality queries suffer from the *query mismatch problem*. The query mismatch problem refers to different user expressions of the same problem. Because different developers often describe their problems in their own way, the queries formulated by different developers may be semantically related but expressed in a variety of different ways. According to our empirical study, around 80% of the lexical terms are expressed differently regarding a duplicate question pair. Considering a general search engine heavily relies on whether similar posts exist and how similar the user search queries are, **the query mismatch problem greatly hinders the performance of the standard information retrieval tools for searching relevant questions**. Therefore, there is a need by developers to find semantically relevant questions for a wide variety of user query expressions.
- *Information Overload.* Low-quality queries can lead to an *information overload problem* [68, 69]. The information overload problem refers to the huge amount of search results returned by a general search engine, where the expected result can be easily buried. Searching Stack Overflow using low-quality queries often brings in a large number of irrelevant results, and developers can easily get lost in the massive amount of contents and thus fail to locate their desired result. Xu et al. [69] conducted a survey with 72 developers and, according to their interview and survey-based study, there is too much noisy and redundant information online, and developers often wasted their time on reading irrelevant posts, which was really time-consuming. Similarly, Xia et al. [68] surveyed 235 developers from 21 countries, and they found it is hard for developers to get the desired solutions for the search tasks due to too many noisy results, and the best solution for a problem is often ranked low in the results of search engines. Therefore, there is a need by developers to **receive the most suitable code solutions for their current programming tasks high up in their search results list**.

We focus on Stack Overflow in this work due to the large number of posts in Stack Overflow that provide variable user descriptions about different types of problems. Moreover, a tremendous number of reusable code snippets archived in Stack Overflow enable information seekers to directly get solutions from the repositories. In this study, we aim not to replace Google searches for Stack Overflow solutions, but to help developers to search for the best code fragments in Stack Overflow that more closely match diverse developer user queries.

We formulate this task as a *query-driven code recommendation* task for a given input question to Stack Overflow. Given an input question, instead of naively choosing the answers from relevant questions, we present a novel model and tool, named Que2Code, to achieve this goal of searching for the best code snippet to answer a user query. We use a two-stage model: In the first stage, we use a query rewriter to tackle the *query mismatch* challenge. The idea is to use rewritten version of a query question to cover different forms of semantically equivalent expressions. In the second stage, we use a code selector to tackle the *information overload* challenge. We extract all the code snippets from the collected answers to construct a candidate pool and then train a Pairwise Learning to Rank neural network by automatically establishing positive and negative training samples. We then select the best code snippet from the code snippet candidates via pairwise comparisons. We conduct extensive experiments to evaluate our Que2Code model. To evaluate the first stage of semantically equivalent question retrieval, we collect duplicate question pairs of Python and Java from Stack Overflow and verify the effectiveness of our approach for identifying the semantically equivalent question for a given user query question. To evaluate the second stage of best code snippet recommendation, we collect more than 218K **QC (question-code snippet)** pairs for Python and more than 270K QC pairs for Java. We then evaluate the effectiveness of our approach for choosing the right code snippet in the code snippet candidate pool. Our experimental results show that our proposed Que2Code model outperforms several baselines in both stages.

This article makes the following three main contributions:

- All previous studies of question routing in CQA systems [12, 15, 64, 70, 73, 78] work on finding similar questions. However, it is hard to measure the relevance between different questions automatically and experts are often asked to manually rate the relevance score [69, 70]. In our study, we propose a new task of semantically equivalent question retrieval. By utilizing duplicate question pairs archived in Stack Overflow, we present a novel model and an evaluation method to automatically evaluate the semantically equivalent questions without a labor-intensive labeling process.
- All current studies that have investigated code snippet searching [10, 21, 53, 74] rely on calculating a matching score between a query and a code snippet. We argue that code snippet recommendation is more about predicting relative orders rather than precise relevance scores. Hence, we propose a novel pairwise learning to rank model to recommend code snippet from Stack Overflow posts, and we first use the BERT model for searching for code snippets in Stack Overflow.
- Our experimental results show that our Que2Code is more effective for code snippet recommendation than several state-of-the-art baselines and performs better than Google search engine in identifying the duplicate questions for the low-quality user queries. We have released the source code of our Que2Code and our dataset[1] to help other researchers replicate and extend our study.

The rest of the article is organized as follows: Section 2 presents a motivating example and user scenario of our approach. Section 3 presents details of our approach for semantically equivalent

[1] https://github.com/beyondacm/Que2Code.

	Example 1	Example 2	Example 3
Duplicate Question	**Difference in divison of -a//b and a//b in python 2.7 [duplicate] (41448836)**	**Is there a way to change the attribute of an object assigned to the attribute of another object? [duplicate] (53962630)**	**Set a maximum value for cells in a csv file [duplicate] (56862047)**
Master Question	**Floor division with negative number (37283786)**	**Python Class dynamic attribute access (34831505)**	**Set maximum value (upper bound) in pandas DataFrame (40836208)**
Google search Q1	Different results in Python v 3.5.2 and v 2.7.12, but the v 2.7.12 is correct?	Change attribute of another object B within object A in Python	Set a maximum value for cells in a csv file
Google search Q2	What is the reason for having '//' in Python?	Change OBJECT ATTRIBUTE that it is ATTRIBUTE of an other OBJECT in PYTHON	how to extract the min value and max value from csv file using python
Google search Q3	using python 3.5 code in python 2.7 gives different answer	In python, how to update attribute based on attribute of another object passed as an argument?	Finding Max value in a column of csv file PYTHON
Google search Q4	Difference between modulus (%) and floor division(//) in NumPy?	Is there a "clean" way to replace all attribute values of an object in Java with values from another object of the same type?	Highlight the maximum value in the dataframe and save as csv
Google search Q5	Why is floating-point division in Python faster with smaller numbers?	How to access attribute of object from another object's method, which is one of attributes in Python?	How to print out row of CSV file with max value in one column Python
Google search Q6	Why does the division get rounded to an integer?	Is there a way to assign a reference to an object attribute in Python?	CSV find max in column and append new data
Google search Q7	Why does Python 2 use '/' only as integer division?	Property for changing an object-attribute in an object	How to find min and max value for each column in the entire csv file
Google search Q8	Slight deviation between datetime.datetime calculations in Python 2 and 3	Python: Modifying an object by assigning it to another object	Trying to get the largest number in a column of a .csv file
Google search Q9	Why does integer division yield a float instead of another integer?	How do you change the value of one attribute by changing the value of another? (dependent attributes)	CSV field with less than max cell limit of excel is getting truncated
Google search Q10	What does the percentage sign mean in Python	OOP/JAVA: Does assinging an object to another changes its attributes(class variables) too?	Maximum number of rows of CSV data in excel sheet
Google search Q11	...	...	**Set maximum value (upper bound) in pandas DataFrame**

Fig. 1. Motivating examples.

question retrieval and best code snippet recommendation. Section 4 presents the automatic experimental results of our approach with respect to the two stages separately. Section 5 presents the results of our approach on human evaluation. Section 7 discusses the strengths and of our approach and threads to validity. Section 8 presents key related work associated with our study. Section 9 concludes the article.

2 MOTIVATION

In this section, we first show motivating examples from Stack Overflow of the sorts of problems mentioned above (i.e., *query mismatch* and *information overload*), we then present the user scenarios of employing our approach, which can help developers to address these problems. Figure 1 shows three motivating examples of user query questions in Stack Overflow and their corresponding top results in appearance order returned by the Google search engine (the screenshots of the Google search results are saved in our replication package). From the figure, we can observe the aforementioned two challenges with respect to the low-quality questions:

(1) The *query mismatch* challenge for low-quality questions. Consider Example 1—the objective of the user question (i.e., duplicate question) is related to floor mechanism in Python. However, due to lack of knowledge or terminology of the problem, the developer was unable to summarize the key point, therefore, he/she expressed this problem in his/her own way, i.e., "*difference in division of -a//b and a//b in python2.7*." This duplicate question shares only one lexical unit (i.e., division) with the target master question (i.e., "*Floor division with negative number*"). Another example is shown in Example 2: The user query question (i.e., "*Is there a way to change the attribute of an object assigned to the attribute of another object*") and its duplicate question (i.e., "*Python Class dynamic attribute access*") also varies considerably. Such a query mismatch problem often results in low-quality questions due to missing key information or important technical details. We manually checked the top-50 posts returned by Google search engine and found that the target post (i.e., master question) can not be

successfully retrieved based on taking user query questions (i.e., duplicate question) as input, which verifies the Google search engine lacks the capacity to retrieve duplicate questions due to query mismatch challenge. Therefore, it is necessary to have an approach to fill the gap between diverse user expressions.

(2) The *information overload* challenge for low-quality questions. The information overload problem can be detrimental to the developers. Searching with low-quality user queries often brings in irrelevant and useless results [68]; If the search engine fails to return the desired result (as shown in Examples 1 and 2), then the developer has to spend significant time and effort browsing useless posts. Even if the search engine successfully retrieves the target post, if the target post is not ranked higher up among other searching results, then the potential solution to the programming task can be easily buried in the overwhelming amount of information. For example, as shown in Example 3, the Google search engine returns the target post at the 11th position. There are 10 irrelevant posts in front of the target post, and each post often includes multiple answers with many not useful code examples. Even just reading all these answers can cost enormous amount of time for developers, not to mention digesting the associated undesirable code solutions. For such cases, the developers may lose interest in browsing the large number of the irrelevant posts before finding the correct code solution; it is thus beneficial to have an approach to rank the desirable code snippet to a higher position among the searching results.

We illustrate the usage scenario of our proposed tool, Que2Code, as follows:

Without Our Tool: Consider Bob is a developer. Daily, Bob encounters a technical problem and wants a code snippet to help solve it. He tries his best to write a query to summarize his problem and searches related questions on Stack Overflow. However, due to lack of the knowledge and terminology about the problem, the query formulated by Bob does not match any post with potential answers. Furthermore, Bob has to painstakingly browse a lot of low-quality and/or irrelevant posts to identify any possible solutions. Therefore, Bob loses interest and is unsatisfied with the overwhelming number of seemingly irrelevant posts. As a result, he posts a duplicate question on Stack Overflow.

With Our Tool: Now consider Bob adopts our Que2Code. When Bob types in his query question, our *QueryRewriter* first generates a list of paraphrase questions for his problem, which increases the likelihood of retrieving semantically equivalent questions in Stack Overflow. Following that, instead of naively returning a massive amount of similar questions, our *CodeSelector* sorts the returned code snippet candidates and recommends the most relevant ones that may contain possible solutions. With the help of our tool, Bob can quickly get answers for his problem without spending much time on reviewing and digesting the low-quality and/or irrelevant information. This time, Bob quickly identifies a useful code snippet from an existing Stack Overflow post for his problem by using our tool.

3 APPROACH

We present a novel query-driven code recommendation system. Que2Code consists of two stages: *Semantically Equivalent Question Retrieval* and *Best Code Snippet Selection*. Our approach takes in a technical question as a query from a developer and recommends a sorted list of code snippets.

3.1 Overview of Approach

Figure 2 demonstrates the workflow of Que2Code. Our model contains two stages: (i) semantically equivalent question retrieval and (ii) best code snippet recommendation. It has two subcomponents, i.e., *QueryRewriter* and *CodeSelector*. The first can qualitatively retrieve semantically

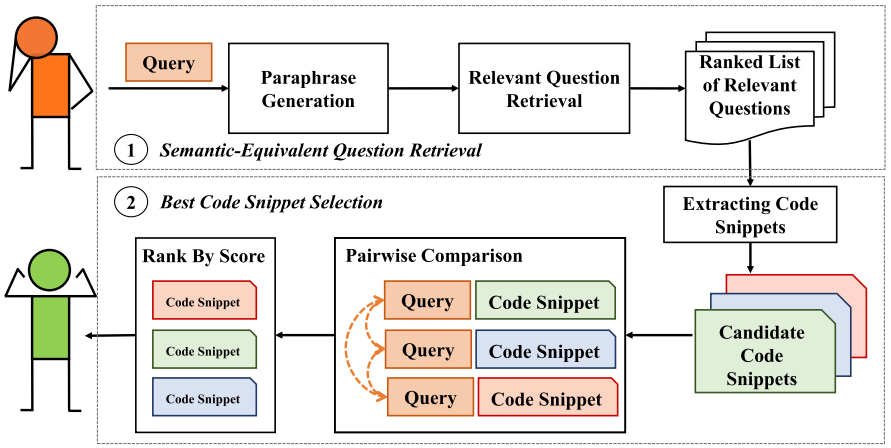

Fig. 2. Workflow of Que2Code.

equivalent questions, and the second can quantitatively rank the most relevant code snippets to the top of the recommendation candidates.

In the first stage, our *QueryRewriter* component tackles the *query mismatch* challenge. To bridge the gap between different expressions of semantically equivalent questions, we introduce the idea of *query rewriting*. The idea is to use a rewritten version of a query question to cover a variety of different forms of semantically equivalent expressions. In particular, we first collect the duplicate question pairs from Stack Overflow, because duplicate questions can be considered as semantically equivalent questions of various user descriptions. We then frame this problem as a sequence-to-sequence learning problem, which directly maps a technical question to its corresponding duplicate question. We train a text-to-text transformer, named *QueryRewriter*, by using the collected duplicate question pairs. After the training process, for any given query question, *QueryRewriter* outputs semantically equivalent paraphrased questions of the input query. Following that, the query question and its generated paraphrased questions are encoded by *QueryRewriter* to measure their relevance with other question titles.

In the second stage, our *CodeSelector* component tackles the *information overload* challenge. To do this, we first collect all the answers of the semantic relevant questions retrieved in the first stage. We then extract all the code snippets from the collected answer posts to construct a candidate code snippets pool. For the given query question, we pair it with each of the code snippet candidates. We then fit them into the trained *CodeSelector* to estimate their matching scores and judge the preference orders. *CodeSelector* can then select the best code snippet from the code snippet candidates via pairwise comparison. Our approach is fully data-driven and does not rely on hand-crafted rules.

3.2 Semantically Equivalent Question Retrieval

In this stage, given a technical problem formulated as a query, we propose a *QueryRewriter* to generate paraphrase questions and retrieve the semantically equivalent questions in Stack Overflow. Figure 3 demonstrates the workflow of *QueryRewriter*. *QueryRewriter* has three steps: paraphrase generation, question embedding, and questions retrieval.

3.2.1 Paraphrase Generation. To obtain the features of semantically equivalent questions for a given user query, we utilize the historical archives of duplicate questions in Stack Overflow, which

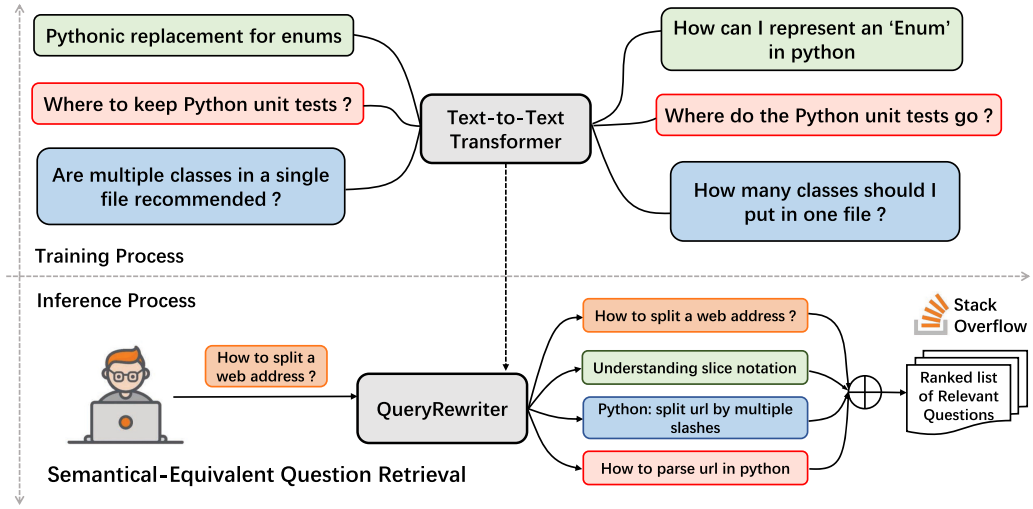

Fig. 3. QueryRewriter workflow.

are manually marked by users and moderators. These duplicate question pairs can be viewed as questions of same intent but written in different ways. In this step, we first automatically generate paraphrase questions for the query question to represent different forms of user expressions. The underlying idea is that by adding these paraphrase questions, we are more likely to find the relevant question that matches the intent expressed in the user query.

In this step, we model the task of paraphrase question generation as a sequence-to-sequence learning problem, where the question title is viewed as a sequence of tokens and its corresponding duplicate question as another sequence of tokens. We adopt the Seq2Seq Transformer architecture [63], which includes an Encoder Transformer and a Decoder Transformer. Both the Encoder and the Decoder Transformers have multiple layers, and each layer contains a multi-head attentive sub-layer followed by a fully connected sub-layer with residual connections [24] and layer normalization [3].

More formally, given a question $\mathbf{X}$ as a sequence of tokens $(x_1, x_2, \ldots, x_M)$ of length M and its duplicate question $\mathbf{Y}$ as a sequence of tokens $(y_1, y_2, \ldots, y_N)$ of length N. The encoder takes the question $\mathbf{X}$ as input and transforms it to its contextual representations. The decoder learns to generate the corresponding duplicate question $\mathbf{Y}$ one token at a time based on the contextual representations and all preceding tokens that have been generated so far. Mathematically, the paraphrase generation task is defined as finding $\overline{y}$, such that:

$$\overline{y} = \underset{\mathbf{Y}}{argmax}\, P_\theta(\mathbf{Y}|\mathbf{X}), \tag{1}$$

where $P_\theta(\mathbf{Y}|\mathbf{X})$ is defined as:

$$P_\theta(\mathbf{Y}|\mathbf{X}) = \prod_{i=1}^{L} P_\theta(y_i|y_1, \ldots, y_{i-1}; x_1, \ldots, x_M). \tag{2}$$

$P_\theta(\mathbf{Y}|\mathbf{X})$ can be seen as the conditional log-likelihood of the predicted duplicate question $\mathbf{Y}$ given the input question $\mathbf{X}$. This model can be trained by minimizing the negative log-likelihood of the training question duplicate question pairs. Once the model is trained, we do inference using beam search [29]. Beam Search returns a list of most likely output sequences (i.e., paraphrase questions). It searches question tokens produced at each step one-by-one. At each timestep, it selects b tokens

with the least cost, where b is the beam wise. It then prunes off the remaining branches and continues selecting the possible tokens that follow on until it meets the end-of-sequence symbol. We repeat the process and generate the top-N most likely paraphrase questions for our study.

A working example is demonstrated in Figure 3, when we input the user query *"how to split a web address?"* described in the motivation example into our trained model, the *QueryRewriter* automatically generates paraphrase questions of different expressions, such as *"understanding slice notation," "python: split url by multiple slashes," "how to parse url in python."* Since the user query question (i.e., "*how to split a web address*") and the target question (i.e., "*slicing url with python*") do not share any lexical units, incorporating the aforementioned paraphrase questions can successfully bridge this gap and solve the *query mismatch* problem.

3.2.2 Question Embedding. After the text-to-text transformer is trained, we can generate multiple paraphrase questions for a newly posted user query. We use these paraphrase questions to boost the user query for the downstream task of question retrieval. By incorporating the paraphrase questions, we can alleviate the *query mismatch* problem by covering the different forms of duplicate question expressions. Thus, we have a better chance to retrieve semantically equivalent questions in Stack Overflow that are intended to solve the same problem.

For a given user query question q_u, we first construct $\mathbf{Q_u} = \{q_u, pq_k\}(1 \leqslant k \leqslant N)$, where q_u is the user query question and $pq_k(1 \leqslant k \leqslant N)$ are the top-N generated paraphrase questions. To capture the overall features and semantics of the user query q_u, we embed each question q_i in $\mathbf{Q_u}$ (including the user query question and the paraphrase questions) to a fixed dimensional vector e_{q_i} via the Encoder Transformer. We then use the average embeddings of all possible questions as the final representation for the user query question. More formally, the question embedding step is defined as follows:

$$e_{q_i} = Encoder(q_i), q_i \in \mathbf{Q_u}, \tag{3}$$

$$e_{q_u} = \frac{1}{N} \sum e_{q_i}, q_i \in \mathbf{Q_u}. \tag{4}$$

3.2.3 Question Retrieval. Given a newly posted user query question q_u and a question title q in the Stack Overflow repository, we use Euclidean distance to estimate the semantic distance, because Euclidean distance has shown to be effective for different software engineering tasks [16, 27], especially when the candidates are similar to each other. The definition of distance as well as the relevance between two questions are as follows:

$$Distance(q_u, q) = \frac{Euclidean(e_{q_u}, e_q)}{|e_{q_u}| + |e_q|}, \tag{5}$$

$$Relevance(q_u, q) = 1 - Distance(q_u, q). \tag{6}$$

For a given user query, we can easily compute a relevance score between the user query and any candidate question in our database. Following that, all the relevance scores are sorted and the top-K ranked questions are returned as the most semantically relevant questions for the query. Considering the number of paraphrase questions N is the key parameter added to our approach, we further investigated the optimal parameter settings of N for the *QueryRewriter* by performing a parameter tuning analysis. We vary N from 0 to 20 with step size 1 to select the optimal parameter, and we find that: (i) the performance of our model rapidly increases as N is increased from 0 to 3, (ii) achieves its best performance when N reaches around 5, (iii) the overall performance of our model reaches a plateau after achieving the best performance. We thus recommend setting the number of generated paraphrase questions to 5, which is close to the optimal settings of N in our approach.

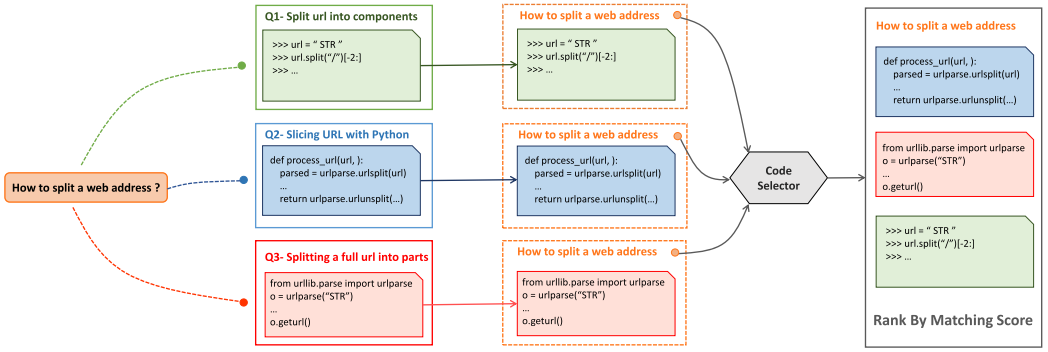

Fig. 4. CodeSelector workflow.

3.3 Best Code Snippet Recommendation

Theoretically, after retrieving the semantically equivalent questions for a given user query, we can naively choose the code snippet from the top ranked questions and recommend the solution to the developers. However, we argue this is not sufficient regarding the following reasons: (i) The technical queries submitted by developers are complicated or sometimes inaccurate [17], there is no guarantee that the top ranked solution is correct and can satisfy the needs of developers. Therefore, we need to collect as many as possible relevant potential code snippet candidates. (ii) Although the search engine can return a list of relevant questions to their problems, the large number of relevant posts and the sheer amount of information in them makes it difficult for developers to find the most needed answer [69]. Therefore, how to pick the best solution from the massive amounts of information is a non-trivial task.

To address this time-consuming task of online code searching, we propose *CodeSelector* to help developers effectively select the most relevant and suitable code snippet for a specific query question. Particularly, the *CodeSelector* reranks all the code snippets by comparing the matching score among different QC (query-code) pairs. Snippets ranked in the top of the final result indicate that these code snippets are more likely and suitable for the programming task. Figure 4 demonstrates the workflow of our *CodeSelector*. For a given query question, a list of relevant questions is retrieved from stage one. After that, all the code snippets associated with these questions are collected, and each code snippet is paired with the given query to make a QC pair. Then, the *CodeSelector* reranks all the code snippet candidates by conducting pairwise comparison among QC pairs. To build the *CodeSelector*, two steps are performed: Preference Pairs Construction and Pairwise Comparison.

3.3.1 Preference Pairs Construction. In this step, we use the available crowdsourced data on Stack Overflow for preference pairs construction. We propose three heuristic rules that can automatically establish the preference pairs and construct the training sets. Our approach is fully data driven and it does not need manual effort. Our three heuristic rules are as follows:

- *For a given question, its best code snippet is preferable to a non-relevant code snippet.* We define the *best code snippet* for a question as the code snippet associated with an accepted answer to the question or the one associated with the highest-vote answer if there are multiple answers to the question. A non-relevant code snippet is randomly selected from the repository. This rule suggests that the quality of the best code snippet is better than the non-relevant ones.
- *For a given question, its non-best code snippet is preferable to a non-relevant code snippet.* We define the *non-best code snippet* as other code snippets apart from the best one within the

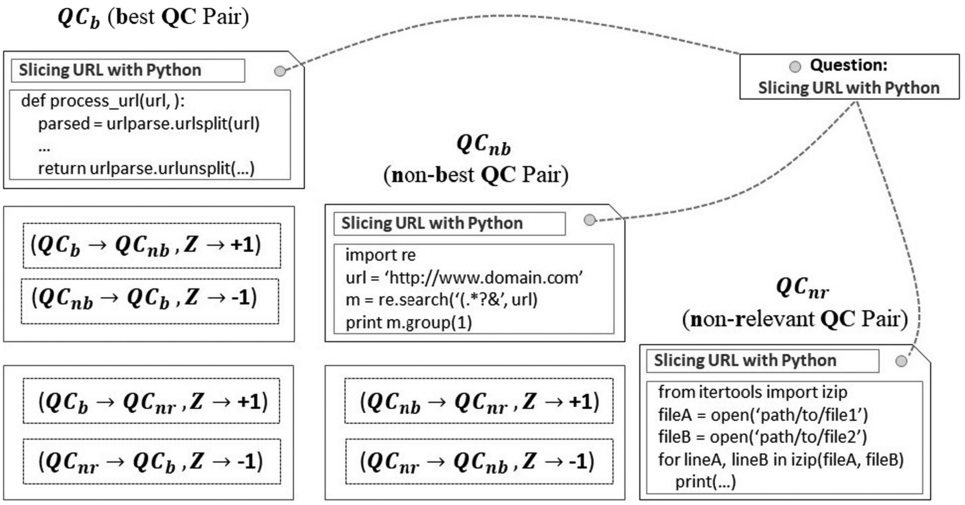

Fig. 5. Preference-pairs construction.

same question thread. This rule suggests that a question prefers the code snippets associated with answers to itself to those of others.

- *For a given question, its best code snippet is preferable to its non-best code snippet.* Even though an individual user vote may not be very reliable, the aggregation of a great number of user votes can provide a very powerful indicator of relevance preference. We argue that the *best code snippet* from an answer post for a question is better than the *non-best ones* in different answer posts for the same question, in most cases.

According to the above heuristic rules, for each given question, three **query-code (QC)** pairs can be generated: We pair it with its best code snippet as the QC_b (**b**est **QC** pair), we pair it with its non-best code snippet as the QC_{nb} (**n**on-**b**est **QC** pair), we pair it with a non-relevant code snippet as the QC_{nr} (**n**on-**r**elevant **QC** pair). We then automatically construct the training samples $\langle QC_1, QC_2, Z\rangle$ for this study. In particular, a training sample contains three parts: two QC pairs (i.e., QC_1 and QC_2) and a label (i.e., Z), the label is automatically determined by the preference relationship between the two QC pairs. Figure 5 demonstrates our data labeling process. Given the query question "*Slicing URL with Python*," we first make three QC pairs, i.e., QC_b, QC_{nb}, and QC_{nr}, as mentioned above. Then, a Pairwise comparison between any two QC pairs can establish a label Z for this training sample. It is worth emphasizing that the comparison order of the two QC pairs matters. For example, a comparison between QC_b and QC_{nb} will be labelled as positive, while a comparison between QC_{nb} and QC_b will be labelled as negative.

There are several advantages of employing this data labeling process: (i) due to the professionality of technical queries, only experts with domain knowledge are qualified to judge the usefulness of a code snippet to a query question. Therefore, manually labeling the relevance scores for all code snippets is very time-consuming and requires a substantial effort [18, 28]. Our heuristic rules can automate the labeling process without any human efforts. (ii) By using these heuristic rules, we gather more training samples, which can provide enough data points for training a deep learning based model.

3.3.2 Pairwise Comparison. After collecting large amounts of labeled training data via preference pairs construction, we develop a learning-to-rank model to sort all the code snippets for a

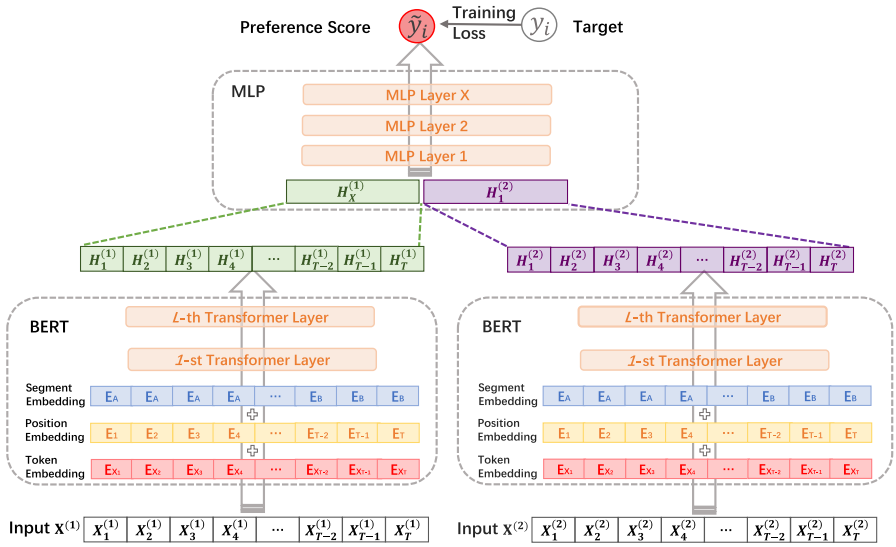

Fig. 6. CodeSelector workflow.

given query question. We design a novel semi-supervised network for ranking **query-code (QC)** pairs. Figure 6 demonstrates the architecture of our proposed model. The input to *CodeSelector* are two QC pairs. *CodeSelector* then learns to judge the preference relationship between two QC pairs based on the positive and negative training samples. In other words, we not only consider the program semantics between a query and a code snippet, but also investigate the relevance preference among different QC pairs.

- **BERT Embedding Layer.** We use the modeling power of BERT [14], which is one of the most popular pre-trained models trained using Transformers [63]. BERT consists of 12-layer transformers, each of the transformers being composed of a self-attention sub-layer with multiple attention heads. Since BERT has been proven to be effective for capturing semantics and context information of sentences in other work, we use BERT as the feature extractor for our task. The input to the BERT embedding layer are two parallel QC pairs. Given each QC pair as a sequence of tokens $\mathbf{x} = \{x_1, \ldots, x_T\}$ of length T, BERT takes the tokens as input and calculates the contextualized representations $\mathbf{H}^l = \{h_1^l, \ldots, h_T^l\} \in \mathbb{R}^{T \times D}$ as output, where l denotes the lth transformer layer and D denotes the dimension of the representation vector. The underlying sub-components work in parallel, mapping each QC pair to its distributional vectors $\mathbf{h}^{(1)}$ and $\mathbf{h}^{(2)}$, respectively, which are then used to perform the predictions for the downstream task.
- **Multi-Layer Perceptron.** After obtaining the BERT representations, we add a **Multi-Layer Perceptron (MLP)** on top of BERT embedding layer to calculate the preference score between the input two QC pairs. Since *CodeSelector* adopts BERT to model two QC pairs, respectively, it is intuitive to combine the features of two pathways by concatenating them. This design has been widely adopted in other deep learning work [18, 25]. To further capture the preference between the latent features of $\mathbf{h}^{(1)}$ and $\mathbf{h}^{(2)}$, we add a standard MLP on the concatenated vector. In this sense, we can endow the model a large level of flexibility and non-linearity to learn the interactions between the two QC pairs. The contextualized representations (i.e., $\mathbf{h}^{(1)}$ and $\mathbf{h}^{(2)}$) are fed to the MLP layer to predict the final preference $\mathbf{y} = \{0, 1\}$. More precisely, the MLP is defined as follows:

$$
\begin{aligned}
\mathbf{z}_1 &= \phi_1(\mathbf{h}^{(1)}, \mathbf{h}^{(2)}) = \begin{bmatrix} \mathbf{h}^{(1)} \\ \mathbf{h}^{(2)} \end{bmatrix} \\
\mathbf{z}_2 &= \phi_2(\mathbf{z}_1) = \mathbf{a}_2(\mathbf{W}_2^T \mathbf{z}_1 + \mathbf{b}_2) \\
&\dots \\
\mathbf{z}_L &= \phi_L(\mathbf{z}_{L-1}) = \mathbf{a}_L\left(\mathbf{W}_L^T \mathbf{z}_{L-1} + \mathbf{b}_L\right) \\
P(\mathbf{y} = j|\mathbf{x}^{(1)}, \mathbf{x}^{(2)}) &= \sigma(\mathbf{z}_L).
\end{aligned} \tag{7}
$$

$\mathbf{W}_x$, $\mathbf{b}_x$, and $\mathbf{a}_x$ denote the weight matrix, bias vector, and activation function for the x-layer's perceptron, respectively. σ is the sigmoid function $\sigma(x) = 1/(1 + e^{-x})$, which will output the final preference score between 0 and 1. For the preference score, we want this score to be high if the first QC pair is preferable to the second one (i.e., $\mathbf{x}^{(1)} \succ \mathbf{x}^{(2)}$) and to be low if the second QC pair is preferable to the first one (i.e., $\mathbf{x}^{(2)} \succ \mathbf{x}^{(1)}$).

3.4 Experimental Settings

We implemented our system in Python using the Pytorch framework. For the *QueryWriter*, we trained the text-to-text transformer on the duplicate question pairs, we follow the parameter settings from Reference [47], which has achieved state-of-the-art results on many benchmarks covering summarization, question answering, text classification, and more. For the *CodeSelector*, we use the pre-trained BERT model released by Reference [14] as our feature extractor. We use the ReLu as the activation function and employ three hidden layers for MLP. The size of the first hidden layer in MLP is equal to the size of the joint vector obtained after concatenating two QC vectors from the BERT model. We fix the parameters of the BERT model and fine-tune the MLP parameters for our task, and the *CodeSelector* is learned by optimizing the log loss of Equation (7).

4 AUTOMATIC EVALUATION

To recommend the code snippets for developers in Stack Overflow, our Que2Code is divided into two stages: semantically equivalent question retrieval and best code snippet recommendation. We wanted to evaluate the performance of the proposed *QueryRewriter* to address the *query mismatch* problem in the first stage and *CodeSelector* to address the *information overload* problem in the second stage. We want to answer the following key research questions:

- RQ-1: How effective is our *QueryRewriter* for semantically equivalent question retrieval?
- RQ-2: How effective is our *QueryRewriter* compared with Google search engine?
- RQ-3: How effective is our *QueryRewriter* for capturing the domain-pecific contextual information?
- RQ-4: How effective is the paraphrase generation added to our *QueryRewriter*?
- RQ-5: How effective is our *CodeSelector* for best code snippet selection?
- RQ-6: How effective is the BERT and preference pairs added to our *CodeSelector*?
- RQ-7: How robust is our *CodeSelector* with different parameter settings?

4.1 RQ-1: Effectiveness of QueryRewriter

We want to identify the best code snippet from a list of semantically equivalent questions for a given query question. If the retrieved questions are not relevant to the query question, then it is unlikely that our tool is able to find the suitable code snippets to solve the target problem. In this study, we consider the duplicate questions in Stack Overflow as semantically equivalent question pairs. After training with the duplicate question pairs archived in Stack Overflow, *QueryRewriter*

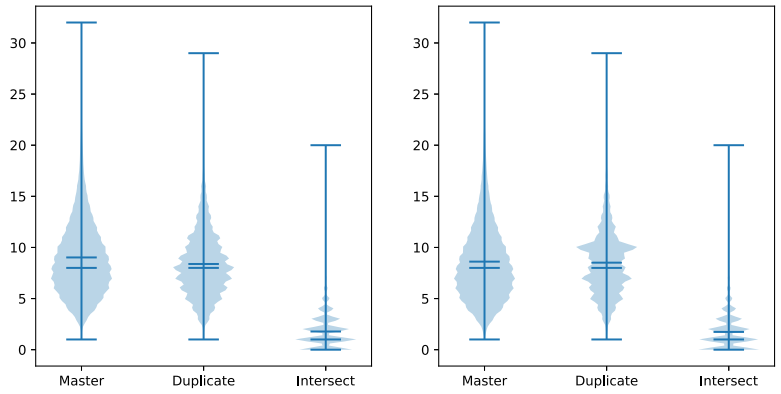

Fig. 7. Violin plots of question distribution for Python (left) and Java (right) datasets.

is able to generate paraphrase questions for a given user query question. By jointly embedding the user query question and the generated paraphrase questions, *QueryRewriter* retrieves the most relevant questions in the repository. We want to investigate the effectiveness of our *QueryRewriter* for retrieving semantically equivalent questions in Stack Overflow.

4.1.1 Data Preparation. We first downloaded the official data dump of Stack Overflow from the StackExchange[2] website. The raw data dump contains timestamped information about the *Posts*, *Comments*, *Users*, *Tags*, *Postlinks*, and so on. We extracted the duplicate question pairs as follows: We first parsed the *PostLinks* and *Posts* database files. Because duplicate questions are marked with a special marker in *PostLinks* database, we can easily identify the `PostId` of the source post and the target post if they are duplicate question pairs. After that, we extracted the question title of the source post and the target post by checking the `PostId` in *Posts* database. We regard the question title of the source post as a duplicate question and the question title of the target post as a master question. We then paired each master question q_m and duplicate question q_d as $\langle q_m, q_d \rangle$ pair. After that, these collected $\langle q_m, q_d \rangle$ pairs are fed into the text-to-text Transformer to train our *QueryRewriter*. In this study, we only focused on Python and Java programming languages for our experiment. As a result, we obtained more than 47K $\langle q_m, q_d \rangle$ pairs for Python and more than 56K pairs for the Java dataset. We further investigated the *query mismatch* problem between the master questions and its corresponding duplicate questions. Specifically, we counted the number of tokens of the master question q_m and its duplicate question q_d, and then we counted the common lexical tokens between the master question and the duplicate question. The violin plot for Python and Java is demonstrated in Figure 7, and we can see that the overlap tokens between the master question and duplicate question are small. For example, for the Python dataset, the average number of tokens of the master question and duplicate question are 9.0 and 8.4, respectively, while the average number of overlap tokens is 1.8. Even though the master question and its duplicate question are semantic-equivalent, they only share a few tokens in common. This also justifies our assumption that the *query mismatch* problem frequently occurs when developers submit their search queries. Therefore, it is necessary to propose a model to address the *query mismatch* problem. Table 1 shows the statistics of our collected datasets of Python and Java duplicate questions. We randomly sampled 2,000 $\langle q_m, q_d \rangle$ pairs for validation and 2,000 $\langle q_m, q_d \rangle$ pairs for testing and kept the rest for

[2]https://archive.org/download/stackexchange.

Table 1. Duplicate Questions Statistics

Python	# Duplicate Questions	47,170	# Avg. Tokens (Duplicate)	8.4
	# Master Questions	20,430	# Avg. Tokens (Master)	9.0
	# $\langle q_m, q_d \rangle$ Pairs (Train)	43,170	# Avg. Tokens (Intersect)	1.8
	# $\langle q_m, q_d \rangle$ Pairs (Val)	2,000	# $\langle q_m, q_d \rangle$ Pairs (Test)	2,000
Java	# Duplicate Questions	56,938	# Avg. Tokens (Duplicate)	8.5
	# Master Questions	23,889	# Avg. Tokens (Master)	8.6
	# $\langle q_m, q_d \rangle$ Pairs (Train)	52,938	# Avg. Tokens (Intersect)	1.7
	# $\langle q_m, q_d \rangle$ Pairs (Val)	2,000	# $\langle q_m, q_d \rangle$ Pairs (Test)	2,000

training. We clarify that different duplicate questions can point to the same master question and this is the reason why the number of duplicate questions is more than the master questions.

4.1.2 Experimental Setup. To determine the effectiveness of our *QueryRewriter* for retrieving semantic-equivalent questions, we designed the following experiment: We first build an index with Lucene using all the question titles in testing set, and the generated index is later used to retrieve the most relevant questions against a given query. In such a way, for each duplicate question q_d in the testing set, we first obtained a set of top-K similar questions $Q = \{q_i\}(1 \leqslant i \leqslant k)$ via the Apache Lucene. We then added its corresponding master question q_m to Q, where now $Q = \{q_m, q_i\}(1 \leqslant i \leqslant k, q_m \neq q_i)$. For a given testing question q_d, we can ensure its corresponding master question q_m is in evaluation candidate pool Q. Since Q includes the semantic-equivalent question q_m for q_d. One way to evaluate our approach is to look at how often the master question q_m can be retrieved successfully among other members of Q. Thus, we adopted the metrics *P@K* and *DCG@K* [37], which are widely used in previous studies [28, 69], to measure the ranking performance of the approach in our study. The evaluation metrics are defined as follows:

- *P@K* is the precision of the master question in top-K candidate questions. Given a question, if one of the top-K ranked questions includes the master question, then we consider the recommendation to be successful and set $success(q_m \in topK)$ to 1, otherwise, we consider the recommendation to be unsuccessful and set $success(q_m \in topK)$ to 0. The *P@K* metric is defined as follows:

$$P@K = \frac{1}{N} \sum_{i=1}^{N} [success(q_m \in topK)]. \tag{8}$$

- *DCG@K* is another popular top-K accuracy metric that measures a recommender system performance based on the graded relevance of the recommended items and their positions in the candidate set. Different from *P@K*, the intuition of *DCG@K* is that highly ranked items are more important than low-ranked items. According to this metric, a recommender system gets a higher reward for ranking the correct answer at a higher position. The $success(q_m \in topK)$ is same with the previous definition, while the $rank_{q_m}$ is the ranking position of the master question q_m. The *DCG@K* is defined as follows:

$$DCG@K = \frac{1}{N} \sum_{i=1}^{N} \frac{[success(q_m \in topK)]}{\log_2(1 + rank_{q_m})}. \tag{9}$$

4.1.3 Experimental Baselines. To demonstrate the effectiveness of our proposed method for relevant question retrieval task, we compared it to the following baselines:

- **IR** stands for the information retrieval baseline. For a given testing question q_i, it retrieves the question that is closest to q_i from the testing set. We use the traditional TF-IDF metric

Table 2. Effectiveness Evaluation (Python)

Model	P@1	P@2	P@3	P@4	DCG@2	DCG@3	DCG@4	DCG@5
IR	32.1 ± 2.4%	44.6 ± 2.6%	54.0 ± 3.6%	64.8 ± 4.8%	40.0 ± 2.1%	44.7 ± 2.5%	49.3 ± 2.9%	63.0 ± 1.3%
Word2Vec	23.9 ± 2.8%	40.6 ± 1.9%	56.1 ± 1.9%	72.0 ± 2.0%	34.4 ± 1.9%	42.2 ± 1.8%	49.0 ± 1.5%	59.9 ± 1.2%
FastText	28.4 ± 3.6%	42.1 ± 2.9%	53.9 ± 3.1%	66.7 ± 2.3%	37.0 ± 2.9%	42.9 ± 3.0%	48.4 ± 2.6%	61.3 ± 1.7%
Sent2Vec	26.0 ± 2.6%	45.9 ± 2.5%	61.9 ± 3.2%	78.6 ± 3.1%	38.6 ± 2.1%	46.5 ± 2.0%	53.7 ± 2.2%	62.0 ± 1.2%
AnswerBot	33.9 ± 2.7%	54.5 ± 4.2%	68.9 ± 3.2%	81.4 ± 2.6%	46.9 ± 3.4%	54.1 ± 2.9%	59.5 ± 2.4%	66.7 ± 1.7%
CROKAGE	36.7 ± 2.6%	51.5 ± 3.1%	64.4 ± 1.9%	78.4 ± 2.5%	46.0 ± 2.7%	52.4 ± 2.0%	58.5 ± 1.6%	66.8 ± 1.3%
Ours	**45.8 ± 3.2%**	**60.5 ± 2.5%**	**72.3 ± 2.7%**	**85.1 ± 2.6%**	**55.0 ± 2.6%**	**61.0 ± 2.5%**	**66.5 ± 2.1%**	**72.2 ± 1.6%**

in our experiment, which is often used to calculate the relevance between a document and a user query in software engineering tasks, such as question retrieval [12, 68] and code search [53].

- **Word2Vec** is a model that embeds words in a high-dimensional vector space using a shallow neural network [38]. This word-embedding technique has provided a strong baseline for information retrieval tasks. Yang et al. [71] used average word embeddings of words in a document as the vector representation for a document. The average word embeddings can be used to calculate the relevance between two question titles in our task.
- **FastText** is another word-embedding model proposed by Reference [7]. Similar to the Word2Vec model, each word is represented as a high-dimensional vector in such a way that similar words have similar vector representations. Same with the Word2Vec baseline, the average FastText word embeddings are used to estimate the similarity between different question titles.
- **Sent2Vec** method, which is also known as para2vec or sentence embedding [32]. This method modifies the Word2Vec algorithm to generate semantic embeddings of longer pieces of text (e.g., sentences or paragraphs) via unsupervised learning. The generated sentence embeddings have been applied in textual similarity tasks [32]. With the help of publicly accessible tool [51], we train the Sent2Vec model using the duplicate question corpus and obtain the sentence embeddings for each question title.
- **AnswerBot**. Xu et al. [69] proposed a three-stage framework called AnswerBot to generate an answer summary for a non-factoid technical question (i.e., non-factoid questions are defined as open-ended questions that require complex answers, such as descriptions, opinions, or explanations, and technical questions are often non-factoid questions [23, 59, 69]). In their first stage, they combined the word embeddings with the traditional traditional IDF metrics for retrieving relevant questions. Their method has been proven to be effective in the task of relevant question retrieval compared with a set of baselines.
- **CROKAGE.** More recently, Da et al. [13] proposed a model named CROKAGE to recommend relevant solutions from Stack Overflow for a searching query. They aimed to address the lexical gap problem between the query and the solutions via using a multi-factor relevance mechanism. To be more specific, they calculated the final relevance score by combining four types of scores (*lexical score*, *semantic score*, *API method score*, *API class score*). We adapt their approach for our task of retrieving semantically equivalent questions. Particularly, we only retain the *lexical scores* and *semantic scores* for this research question, since question titles usually do not contain API classes.

4.1.4 Experimental Results. The experimental results of our *QueryRewriter* compared to the above baselines for Python and Java are summarized in Tables 2 and 3, respectively. We do not report *P*@5 and *DCG*@1 in our tables; since *P*@5 is always equal to 1 and *DCG*@1 is always equal to *P*@1, both can be easily inferred from the tables. The best-performing system for each column is highlighted in boldface. From the table, several points stand out:

Table 3. Effectiveness Evaluation (Java)

Model	P@1	P@2	P@3	P@4	DCG@2	DCG@3	DCG@4	DCG@5
IR	31.1 ± 2.0%	43.1 ± 2.5%	53.7 ± 2.4%	62.8 ± 2.5%	38.6 ± 2.1%	43.9 ± 2.1%	47.9 ± 1.8%	62.3 ± 1.2%
Word2Vec	26.4 ± 3.2%	37.7 ± 4.3%	51.7 ± 4.1%	66.3 ± 3.3%	33.6 ± 3.7%	40.5 ± 3.6%	46.8 ± 3.0%	59.9 ± 2.0%
FastText	29.8 ± 1.9%	42.7 ± 2.6%	53.0 ± 5.2%	65.8 ± 3.8%	38.0 ± 2.2%	43.1 ± 3.5%	48.6 ± 2.7%	61.9 ± 1.4%
Sent2Vec	24.8 ± 2.2%	45.8 ± 3.4%	62.0 ± 2.8%	80.8 ± 2.6%	38.0 ± 2.9%	46.1 ± 2.5%	54.2 ± 1.8%	61.7 ± 1.4%
AnswerBot	33.3 ± 3.0%	48.1 ± 2.4%	61.1 ± 2.5%	75.0 ± 2.3%	42.6 ± 2.4%	49.1 ± 2.5%	55.1 ± 2.3%	64.8 ± 1.6%
CROKAGE	38.9 ± 4.3%	53.4 ± 5.5%	65.3 ± 4.8%	78.0 ± 2.8%	48.0 ± 5.0%	54.0 ± 4.4%	59.4 ± 3.3%	68.0 ± 2.6%
Ours	**58.6 ± 4.3%**	**73.9 ± 1.4%**	**84.3 ± 2.0%**	**92.9 ± 1.8%**	**68.2 ± 2.3%**	**73.4 ± 2.0%**	**77.1 ± 1.4%**	**79.9 ± 1.7%**

(1) It is a little surprising that the **word-embedding-based approaches (e.g., Word2Vec, FastText, and Sent2Vec) achieve the worst performance** regarding $P@1$. This indicates that retrieving semantically equivalent questions from a set of similar questions is a non-trivial task. Word2Vec and Sent2Vec map each question to a fixed-length vector, so the vectors of similar questions are also close together in vector space. However, due to the reason that all candidate questions in Q are similar to each other, it is thus very hard for Sent2Vec and Word2Vec approaches to distinguish the duplicate questions from a list of similar questions. This is the reason why their performance is weak and ineffective for retrieving semantically equivalent questions.

(2) Regarding the $P@1$ score, the traditional **IR method performs better** than the word-embedding-based methods (i.e., Word2Vec and Sent2Vec). For the IR-based approach, it relies heavily on how similar the testing question and its duplicate question are. Considering that many duplicate questions may share same lexical units with the testing questions, these duplicate questions can be easily retrieved by the IR-based approach. However, there is still a large number of duplicate questions that are semantically equivalent with only a few common words or without at all (e.g., as shown in Figure 1); the IR-based approach is unable to retrieve these questions correctly solely based on the similarity between words or tokens. This may also explain its surprisingly low score as K increases.

(3) AnswerBot and CROKAGE perform better than other baselines excluding our proposed model. This is because both AnswerBot and CROKAGE combine the lexical-based model (i.e., IR) and semantic-based model (e.g., Word2Vec, FastText, and Sent2Vec) for modeling the question titles, which also signals that solely based on the lexical features or semantic features is not sufficient for our task. It is also notable that the performance of CROKAGE is better than the AnswerBot approach. We attribute this to the different word-embedding techniques they employed. AnswerBot combines IDF metrics with Word2Vec model, while CROKAGE combines IDF metrics with FastText model. This signals that the FastText model has its advantage as compared to Word2Vec model for modeling the duplicate question titles.

(4) It is clear that **our model outperforms all the other methods by a large margin** in terms of $P@K$ and $DCG@K$ scores of different depth. We attribute this to the following reasons: First, *QueryRewriter* generates multiple paraphrase questions for a given testing question. Adding these paraphrase questions can reduce the lexical gap between the testing questions and its corresponding duplicate questions and alleviate the *query mismatch* problem for different developers. Second, all of the baseline methods including our approach can be viewed as variants of embedding algorithm(s), which can map the questions into vectors of a high-dimensional space and then calculate the relevance score between vectors. Hence, the key of retrieving semantically equivalent questions relies on how good the embeddings are for capturing the semantics of different duplicate questions. Our approach has its advantage as compared to other baselines, because our *QueryRewriter* trains the historical duplicate question pairs in Stack Overflow by using a text-to-text transformer. As a result, the embeddings

Table 4. QueryRewriter vs. Google (Python)

Dataset	**Overall Dataset**		**Successful Dataset**		**Failed DataSet**	
	Count	2,000	Count	393	Count	1,607
	Avg. Score	3.01	Avg. Score	6.99	Avg. Score	2.05
Approach	Google	Ours	Google	Ours	Google	Ours
P@1	20.0%	**35.6%**	59.3%	**47.0%**	10.4%	**32.9%**
P@2	36.4%	**54.2%**	78.4%	**67.9%**	26.1%	**50.9%**
P@3	44.6%	**65.4%**	89.3%	**79.9%**	33.6%	**61.9%**
P@4	47.1%	**76.2%**	97.7%	**92.6%**	34.7%	**72.2%**
P@5	75.4%	**90.9%**	100.0%	**99.0%**	69.4%	**90.0%**
DCG@1	20.0%	**35.6%**	59.3%	**47.0%**	10.4%	**32.9%**
DCG@2	30.3%	**47.4%**	71.3%	**60.2%**	20.3%	**44.2%**
DCG@3	34.4%	**53.0%**	76.8%	**66.2%**	24.0%	**49.7%**
DCG@4	35.5%	**57.6%**	80.4%	**71.7%**	24.5%	**54.2%**
DCG@5	46.5%	**63.3%**	81.3%	**74.2%**	37.9%	**60.7%**

generated by our approach are more suitable for identifying the semantically equivalent questions. The superior performance of our approach also verifies the embeddings generated by our approach convey a lot of valuable information.

Answer to RQ-1: How effective is our approach for retrieving semantically equivalent questions? We conclude that our approach is highly effective for semantically equivalent question retrieval in Stack Overflow.

4.2 RQ-2: Google Search Results Analysis

Google search engine has been widely used by developers to search online resources and improve their daily productivity. Typically, when developer encounters a technical problem, he or she usually use a web search engine (e.g., Google) to obtain a list of relevant posts in Stack Overflow. Therefore, the Google search engine can be considered as a baseline for searching duplicate questions intuitively. In this research question, for a given question, we would like to investigate whether our *QueryRewriter* can rank its duplicate question higher up among other question candidates compared with Google search.

4.2.1 Experimental Setup. Regarding the Google search results analysis, for each duplicate question pair $\langle q_d, q_m \rangle$ in the testing set, we first use the question title of q_d as a searching query and feed to the Google search engine. We then crawl the first page returned by the Google search engine and extract all the Stack Overflow related posts as the Google search results; the Google search results on our testing set are also saved in our replication package. Similar to RQ-1, for each testing question q_d, we obtain a set of relevant questions $\mathcal{G} = \{q_i\}(1 \leqslant i \leqslant k)$, where k is the number of relevant questions returned by the Google search engine. We remove q_d from $\mathcal{G}$ if $\mathcal{G}$ contains q_d, and add q_m to $\mathcal{G}$ if q_m is not within $\mathcal{G}$. In the light of this, for the given testing question q_d, we can ensure its master question q_m is in the evaluation candidate pool $\mathcal{G}$. Hereafter, we pair the given testing question with each of the candidate questions in $\mathcal{G}$ and utilize our model to generate a ranking list of the candidate questions by estimating their relevance scores. We evaluate our approach and Google search engine with $\mathcal{G}$ in terms $P@K$ and $DCG@K$.

4.2.2 Experimental Results. The experimental results of *QueryRewriter* and Google search engine for Python and Java are presented in Tables 4 and 5, respectively. To have a deeper

Table 5. QueryRewriter vs. Google (Java)

Dataset	Overall Dataset		Successful Dataset		Failed Dataset	
	Count	2,000	Count	410	Count	1,590
	Avg. Score	2.52	Avg. Score	6.93	Avg. Score	1.38
Approach	Google	Ours	Google	Ours	Google	Ours
P@1	21.5%	**39.0%**	62.7%	**48.5%**	10.9%	**35.5%**
P@2	39.6%	**59.8%**	80.7%	**71.9%**	29.0%	**56.7%**
P@3	48.8%	**70.5%**	92.1%	**82.4%**	37.7%	**67.5%**
P@4	51.2%	**79.6%**	98.0%	**94.1%**	39.1%	**75.8%**
P@5	76.0%	**91.6%**	99.5%	**98.6%**	70.0%	**89.8%**
DCG@1	21.5%	**39.0%**	62.7%	**48.5%**	10.9%	**36.5%**
DCG@2	33.0%	**52.1%**	74.0%	**63.3%**	22.4%	**49.2%**
DCG@3	37.6%	**57.5%**	79.8%	**68.6%**	26.7%	**54.7%**
DCG@4	38.6%	**61.4%**	82.3%	**73.6%**	27.3%	**58.2%**
DCG@5	48.2%	**66.0%**	82.9%	**75.3%**	39.2%	**63.7%**

understanding about the performance of the Google search engine and our approach, we further split our testing dataset into *successful dataset* and *failed dataset* based on whether the duplicate question can be successfully retrieved by the Google search engine. To be more specific, for a given testing question, if Google search engine successfully finds its duplicate question, then we add this testing question to the *successful dataset*, otherwise it will be added to the *failed dataset*. From the tables, we can observe the following points:

(1) The Google search achieves a poor performance for retrieving the duplicate questions on the testing set. Regarding the 2,000 testing questions, the Google search engine only finds 393 duplicate questions successfully for the Python and 410 duplicate questions for Java dataset, which means that around 80% of the duplicate questions can not be retrieved effectively by Google search engine. Since we manually added the master question to the *failed dataset* for evaluation, the *P@K* and *DCG@K* are not equal to zero for Google search engine.
(2) The quality of the questions within the *successful dataset* is much higher than the *failed dataset*. The successfully retrieved questions are grouped together into the *successful dataset*, while the other questions are grouped together into the *failed dataset*. We then compute the average question scores with respect to these two datasets, which has been widely used for measuring a post quality in Stack Overflow [2, 4, 52, 72]. For the questions within the *successful dataset*, the average score is 6.99 for Python and 6.93 for Java, much higher than the questions of the *failed dataset* (e.g., 2.05 for Python and 1.38 for Java). The large number of low-quality posts lead to the comparatively suboptimal performance of Google search engine for retrieving duplicate questions.
(3) Our approach performs better than the Google search engine on *failed dataset*, while the Google search engine outperforms our approach on *successful dataset*. The superior performance of the Google search engine on the *successful dataset* indicates its capability of handling the high-quality posts. Considering these high-quality posts often obtain more attentions (e.g., user views and clicks), it is not surprising that the Google search engine can easily record and index them for recommendation. However, the Google search engine fails to return the desired results on the *failed dataset*, which suggests the Google search engine lacks the ability to deal with low-quality posts.
(4) Our approach outperforms the Google search engine by a large margin on the overall testing dataset. This is because our approach stably and substantially performs better than the

Google search baseline on the *failed dataset*, which accounts for the majority part of the testing questions. When dealing with the low-quality posts in the *failed dataset*, our approach can successfully rank the desired results (i.e., duplicate question) higher up among other question candidates. The advantages of our approach for this scenario are due to the following reasons: First, without relying on the feedback from online users, our approach is trained with the historical duplicate question pairs, and the various quality of duplicate questions can increase the ability and generality for handling the low-quality posts. Second, our approach can map the semantic-equivalent questions into the vector space that are close to each other, and the lexical gap between duplicate question pairs can be filled by using the question embeddings.

Answer to RQ-2: How effective is our approach compared with Google search engine? We conclude that our approach performs better than Google search engine when dealing with low-quality posts in Stack Overflow.

4.3 RQ-3: Context Analysis

As the technical Q&A sites (i.e., Stack Overflow) are used by developers and professional experts, the questions in these Q&A communities are, more often than not, very professional with specific domain context. For example, these questions often include software-specific entities (e.g., software libraries/frameworks, software-specific concepts). To investigate whether the domain-specific context could influence the performance of our approach, or in other words, whether our model can learn the domain-specific context features from the training corpus, we perform a context analysis for this study.

4.3.1 Experimental Setup. For the context analysis, we create a training set without domain-specific context and use the same testing set for evaluation. To do this, we check each token in the training corpus if the token is a normal English word (using the NLTK package), and we only keep the English words and remove the other non-proper English words. As a result, the software-specific terms within the master questions and duplicate questions are deleted. After that, we retrain our *QueryRewriter* and all the baselines on the non-domain-specific context training corpus and perform the same evaluation as in RQ-1.

4.3.2 Experimental Results. The experimental results of our *QueryRewriter* and other baselines for Python and Java dataset are presented in Tables 6 and 7, respectively. From the tables, we can deduce the following key findings:

(1) The performance of all models decreases on the training set without domain-specific context. This suggests that the domain-specific context has a major influence on the overall performance. We further counted the unique tokens of the training corpus with and without the domain-specific context: For Python dataset, only 4,756 out of 19,208 tokens are remained; for Java dataset, only 4,874 out of 22,344 tokens are remained. The large proportion of domain-specific context in Stack Overflow may also explain the performance drop in all baseline methods.
(2) It is notable that after adding the domain-specific context, our model achieves the biggest performance rise among different models. This reveals that our model is effective in learning the domain-specific features and knowledge. Moreover, the performance of our proposed model still outperforms the other baseline approaches even under the training corpus without domain-specific context, which justifies the robustness of our model.

Table 6. Context Analysis of *P*@1 (Python)

Approach	Without Context	With Context	△ Improve
IR	24.9 ± 2.2%	32.1 ± 1.2%	28.9%
Word2Vec	22.2 ± 2.9%	23.9 ± 2.8%	7.7%
FastText	24.4 ± 1.5%	28.4 ± 3.6%	16.4%
Sent2Vec	24.2 ± 2.1%	26.0 ± 2.6%	7.4%
AnswerBot	29.4 ± 3.8%	33.9 ± 2.7%	15.3%
CROKAGE	26.9 ± 2.3%	36.7 ± 2.6%	36.4%
Ours	31.1 ± 2.4%	45.8 ± 3.2%	47.3%

Table 7. Context Analysis of *P*@1 (Java)

Approach	Without Context	With Context	△ Improve
IR	26.9 ± 2.1%	31.1 ± 2.0%	15.6%
Word2Vec	24.5 ± 2.4%	26.4 ± 3.2%	7.6%
FastText	25.5 ± 2.6%	29.8 ± 1.9%	16.7%
Sent2Vec	23.4 ± 2.5%	24.8 ± 2.2%	6.0%
AnswerBot	26.8 ± 3.3%	33.3 ± 3.0%	24.3%
CROKAGE	28.2 ± 2.2%	38.9 ± 4.3%	37.9%
Ours	37.7 ± 4.2%	58.6 ± 4.3%	55.4%

Answer to RQ-3: How effective is our approach for capturing contextual information? We conclude that the domain-specific context can influence the model's performance, and our approach is highly effective for learning domain-specific context information.

4.4 RQ-4: Ablation Analysis

Ablation analysis is a common method to estimate the contribution of a component to the overall system [17]. It studies the performance of a system by removing certain components. Our *QueryRewriter* learns to encode the duplicate question pairs from Stack Overflow, so two semantically equivalent questions are close in terms of vector representation. When we perform question retrieval tasks, a main novelty of our approach is adding paraphrase questions generated by *QueryRewriter*. In this research question, we perform an ablation analysis to investigate if the novel aspect that we introduce helps. To be more specific, we investigate the effectiveness of the added paraphrase question to our model.

4.4.1 Experimental Setup. For the ablation analysis, we compare our approach with one of its incomplete variants, named **Drop-PQ**. Different from our proposed model, **Drop-PQ** removes all the generated paraphrase questions added to our model and only keeps the embedding of the original testing question. By going through the same steps as our approach in Section 3.2, we can evaluate **Drop-PQ** model for the semantic-equivalent question retrieval task.

4.4.2 Experimental Results. The comparison results between **Drop-PQ** and our approach are displayed in Table 8. We observe the following points from the table:

(1) By comparing the results of our approach and **Drop-PQ**, it is clear that **incorporating the paraphrase questions improves the overall performance**. When adding the paraphrase questions to our model, the *P*@1 score is improved by 24.1% in Python and 18.3% in Java

Table 8. Ablation Analysis

Measure	Python		Java	
	Drop-PQ	**Ours**	**Drop-PQ**	**Ours**
P@1	36.9%	**45.8%**	49.5%	**58.6%**
P@2	50.0%	**60.5%**	65.3%	**73.9%**
P@3	62.1%	**72.3%**	76.9%	**84.3%**
P@4	74.5%	**85.1%**	87.5%	**92.9%**
DCG@2	45.1%	**55.0%**	59.4%	**68.2%**
DCG@3	51.2%	**61.0%**	65.3%	**73.4%**
DCG@4	56.5%	**66.5%**	69.8%	**77.1%**
DCG@5	66.4%	**72.2%**	74.7%	**79.9%**

dataset. We attribute this to the ability of paraphrase questions to reduce the lexical gap between the semantically equivalent questions.

(2) By comparing the results of **Drop-PQ** and our previous baselines in RQ-1, we can see that **even by dropping the paraphrase questions, Drop-PQ still achieves better or comparable results than other baselines**. This is because, even removing the paraphrase questions, the question embeddings are generated from the same Encoder of our text-to-text transformer. This further verifies the importance and necessity of the embeddings of our approach.

Answer to *RQ-4:* How effective is the paraphrase generation component added to our *QueryRewriter*? We conclude that adding paraphrase questions significantly improves the overall performance of our model.

4.5 RQ-5: Effectiveness of CodeSelector

When trying to solve daily coding problems, developers often formulate their problems as a question and/or a few keywords to some search engines. The search engine returns a list of potential posts that may contain useful answers. Due to the complexity of the online CQA forums and the large volume of information generated from it, software developers may encounter the *information overload* problem wherein the massive amounts of information makes it hard to be aware of the most relevant resources to meet the information needs of the developers. To alleviate this *information overload* problem, we propose a *CodeSelector* to rank the code snippets candidates via pairwise comparisons. To evaluate our approach, we conducted a large-scale automatic evaluation experiment to evaluate the effectiveness of our approach to identify the best code solution for a technical problem.

4.5.1 Data Preparation. To train the *CodeSelector*, we first constructed positive and negative training samples in terms of preference pairs. For each Stack Overflow post, we extracted the code snippets (using ⟨*code*⟩ tags) within the post's question body and corresponding post question title. To avoid being context-specific, numbers and strings within a code snippet are replaced with special tokens "NUMBER" and "STRING," respectively. We first adopted the NLTK [6] toolkit to tokenize the code snippets, and we removed the code snippets that are too long (more than 512 tokens) or too short (less than 5 tokens). This is because for a given code snippet, it is unable to capture the code semantics if it is too short. We set the 512 as the maximum number of code snippet tokens, since the maximum input sequence length of BERT [14] is restricted to 512 tokens, and this setting is sufficient for most cases of code snippets in Stack Overflow [17]. For question titles, we only preserved the "how" related questions in Stack Overflow. The resulting

Table 9. QC Dataset Statistics

Python	# $\langle q, cs \rangle$ Pairs	218,717
	# best code snippet	112,447
	# non-best code snippet	106,270
	# non-relevant code snippet	112,447
	# Positive Samples	141,074
	# Negative Samples	141,224
Java	# $\langle q, cs \rangle$ Pairs	272,120
	# best code snippet	134,993
	# non-best code snippet	137,127
	# non-relevant code snippet	134,993
	# Positive Samples	177,340
	# Negative Samples	176,942

$\langle question, code\ snippet \rangle$ pairs are added to our corpus. Towards this end, we collected 218K QC pairs for Python dataset and 272K QC pairs for Java dataset.

For each question in the corpus, we make the code snippet associated with the accepted answer or the highest-vote answer as the *best code snippet*, the code snippet associated with the non-accepted answers as the *non-best code snippet*, and randomly select the code snippet from other questions as the *non-relevant code snippet*. According to our heuristic rules described above in Section 3.3, we constructed our dataset with balanced positive and negative preference pairs. We randomly sampled 5,000 samples for validation and 5,000 pairs for testing, respectively, and kept the rest for training. The details of the statistics of our collected dataset are summarized in Table 9.

4.5.2 Experimental Setup. To evaluate our *CodeSelector* performance for identifying the best code snippet for a technical question, for each question q in the testing set, we employ the same KNN strategy in RQ-1 to search its top-K similar questions over the whole dataset. Undoubtedly, the testing question itself can be found. We then constructed a code snippet candidates pool C by gathering all the code snippets associated with the returned questions. In the light of this, we can ensure the ground truth code snippet (*best code snippet*) is in the code snippet candidates pool C. Following that, we pair the given question q with each of the code snippets in C to make a QC pair. Hereafter, by doing a pairwise comparison between each two QC pairs, we can generate a ranking list of preference scores for each code snippet. All the code snippet candidates can be ranked by their preference scores. For this code selection task, we also employ the same automatic evaluation metrics *P@K* and *DCG@K* used in RQ-1. *P@K* and *DCG@K* stands for the proportion of the selected code snippets in the top-K that are the ground truth.

4.5.3 Experimental Baselines. To demonstrate the effectiveness of our proposed approach, we compare it with several competitive baseline approaches. We adapt these approaches slightly for our specific task, i.e., selecting the best code snippet from a pool of code snippet candidates. We briefly introduce these approaches and our evaluation experimental settings below. For each method below, the involved parameters are carefully tuned, and the best performance of each approach is used to report the final results.

- **Traditional Classifiers.** Considering that our *CodeSelector* ranks the code snippet candidates by doing classification between QC pairs, it is hence natural to compare our approach with traditional classifiers. Recently, Calefato et al. [9] proposed an approach for best answer prediction problem by formulating it as a binary-classification task. The binary-classification methods output a score referring to a probability of relevance. They assessed 26 traditional

classifiers for predicting the best answer in Stack Overflow. We choose the two most effective traditional classifiers, xgbTree and RandomForest, to apply to our code snippet recommendation task. In our experiments, we treated the pair of $\langle question, best\ code\ snippet \rangle$ as positive sample and the pair of $\langle question, non - best\ code\ snippet \rangle$ as negative sample, and we then train the traditional classifiers with these training samples. Thereafter, we rank the code snippet candidates by the relevance scores generated by the above trained classifiers.

- **Answer-ranking Methods.** The code snippet selection need of our task is similar to the answer-ranking problem in CQA forums. Hence, our task is transformed to find an optimal ranking order of the code snippet candidates according to the their relevance to the given query question. Two answer-ranking methods, i.e., AnswerBot [69] and DeepAns [18], are chosen as baselines. Regarding the AnswerBot baseline, their user study showed a promising performance for selecting salient answers in the second stage of their approach. Regarding the DeepAns baseline, they calculated a matching score via a deep neural network between each answer and the question title; the experimental results show that DeepAns is effective for selecting the most relevant answer compared with several state-of-the-art benchmarks. For both of these answer-ranking methods, an overall score is computed to estimate the relevance between each answer and the question title. For our task, we can replace the answer with the code snippet and thus adapt their answer-ranking methods to our task of ranking code snippets among a set of code snippet candidates.
- **DL-based Code Search Methods.** Another thread of similar research that is relevant to our work is code search. A plethora of approaches have been investigated for searching code snippet in software repositories, and recent DL (deep learning)-based approaches have achieved promising results for this task. The DL-based code search methods advocate the idea of mapping and matching data in a high-dimensional vector space. Three state-of-the-art DL-based code search methods, i.e., NCS [53], DeepCS [21], and CROKAGE [13], are chosen for our study. NCS is an unsupervised technique for neural code search proposed by Facebook [53]. They combine the word embeddings and TF-IDF weighting derived from a code corpus. In our experiment, we used all the collected QC pairs as the code corpus for training the word embeddings and TF-IDF weightings. DeepCS is a supervised technique that jointly embeds code and natural language description into a high-dimensional vector space proposed by Gu et al. [21]. The author constructed the $\langle C, D+, D- \rangle$ triples to train their model for minimizing the ranking loss, where C is the code snippet, $D+$ and $D-$ are the correct and incorrect description, respectively. In our experiment, for each code snippet C, we treat the associated question title as the correct description $D+$ and treat a randomly selected question title as the incorrect description $D-$. CROKAGE [13] is a tool to deliver a comprehensible solution for a programming task. It chooses the top quality answers related to the task. To mitigate the lexical gap problem, CROKAGE calculates the scores for candidate QA pairs using four similarity factors (e.g., *lexical score*, *semantic score*, *API method score,* and *API class score*). Specifically, they use TF-IDF and fastText embedding to calculate the *lexical scores* and *semantic scores*, respectively. They also reward the QA pairs that contain the top API methods and relevant API classes with *API method scores* and *API class scores*. Their final outputs contain both code examples and code explanations. Because our study mainly focuses on the code snippet recommendation task, we only retain the code examples part and remove the code explanation part.

4.5.4 Experimental Results. The experimental results of our proposed model and the above baselines over our Python and Java datasets are summarized in Tables 10 and 11, respectively. From the table, we can observe the following points:

Table 10. Effectiveness Evaluation of CodeSelector (Python)

Model	P@1	P@2	P@3	P@4	DCG@2	DCG@3	DCG@4	DCG@5
RandomForest	26.6 ± 1.6%	49.6 ± 2.6%	69.7 ± 2.0%	86.4 ± 1.4%	41.1 ± 2.1%	51.1 ± 1.8%	58.3 ± 1.5%	63.5 ± 1.0%
xgbTree	24.3 ± 1.5%	47.3 ± 2.4%	69.1 ± 2.5%	85.8 ± 1.8%	38.8 ± 1.9%	49.7 ± 1.8%	56.9 ± 1.4%	62.4 ± 0.9%
AnswerBot	31.0 ± 1.5%	51.1 ± 2.3%	70.4 ± 2.1%	87.4 ± 1.5%	43.7 ± 1.8%	53.3 ± 1.4%	60.6 ± 1.1%	65.5 ± 0.8%
DeepAns	29.6 ± 2.2%	52.3 ± 1.9%	71.2 ± 1.2%	88.5 ± 1.2%	43.9 ± 1.9%	53.4 ± 1.5%	60.8 ± 1.3%	65.3 ± 1.1%
NCS	29.8 ± 1.6%	52.7 ± 2.8%	71.3 ± 2.0%	88.3 ± 1.9%	44.2 ± 2.2%	53.5 ± 1.6%	60.9 ± 1.6%	65.3 ± 1.0%
DeepCS	29.5 ± 2.2%	51.3 ± 2.3%	70.3 ± 1.5%	86.7 ± 1.4%	43.3 ± 2.2%	52.8 ± 1.7%	59.8 ± 1.4%	64.9 ± 1.2%
CROKAGE	33.3 ± 2.3%	55.0 ± 2.0%	71.9 ± 1.3%	86.5 ± 1.1%	47.0 ± 1.9%	55.4 ± 1.5%	61.7 ± 1.2%	66.9 ± 1.1%
Ours	**42.6 ± 2.5%**	**64.6 ± 1.1%**	**80.0 ± 1.0%**	**92.3 ± 0.9%**	**56.5 ± 1.5%**	**64.2 ± 1.2%**	**69.5 ± 1.0%**	**72.5 ± 1.0%**

Table 11. Effectiveness Evaluation of CodeSelector (Java)

Model	P@1	P@2	P@3	P@4	DCG@2	DCG@3	DCG@4	DCG@5
RandomForest	24.8 ± 2.9%	47.5 ± 2.3%	67.8 ± 2.1%	85.0 ± 1.4%	39.1 ± 2.2%	49.2 ± 1.8%	57.0 ± 1.7%	62.5 ± 1.3%
xgbTree	25.8 ± 1.7%	47.5 ± 2.6%	68.1 ± 2.4%	85.0 ± 1.5%	39.5 ± 2.1%	49.8 ± 2.0%	57.1 ± 1.5%	62.8 ± 1.1%
AnswerBot	31.4 ± 2.1%	52.1 ± 1.3%	70.9 ± 1.1%	86.9 ± 1.7%	44.5 ± 1.4%	53.9 ± 1.0%	60.7 ± 0.8%	65.8 ± 0.8%
DeepAns	29.1 ± 2.3%	52.5 ± 2.3%	71.3 ± 2.3%	86.7 ± 1.4%	43.9 ± 2.2%	53.2 ± 1.9%	59.9 ± 1.6%	65.1 ± 1.2%
NCS	30.4 ± 1.8%	52.1 ± 1.7%	71.4 ± 1.5%	86.9 ± 1.4%	44.1 ± 1.6%	53.7 ± 1.3%	60.4 ± 1.2%	65.5 ± 0.9%
DeepCS	28.5 ± 3.9%	48.2 ± 3.4%	65.8 ± 4.6%	81.6 ± 3.5%	40.9 ± 3.3%	49.7 ± 3.6%	56.5 ± 2.9%	63.6 ± 2.0%
CROKAGE	33.6 ± 1.8%	54.6 ± 1.8%	72.3 ± 1.9%	86.6 ± 1.1%	46.8 ± 1.6%	55.7 ± 1.4%	61.8 ± 1.1%	67.0 ± 0.8%
Ours	**42.4 ± 1.9%**	**66.1 ± 1.8%**	**81.6 ± 1.5%**	**92.6 ± 1.6%**	**57.3 ± 1.6%**	**65.1 ± 1.5%**	**69.8 ± 1.5%**	**72.7 ± 0.9%**

(1) The performance of **traditional classifiers are comparatively suboptimal**. This indicates that traditional classifiers are unable to capture the semantics between the code snippets and the questions.

(2) The **answer-ranking methods and the DL-based code search methods achieve similar results**. The underlying idea of these two kinds of methods is similar, namely, the application of applying the embedding technique to map the raw data (including the questions and code snippets) into a high-dimensional space and then estimate the match score between them. This may explain the reason why their performances are comparable with each other. CROKAGE has its advantage as compared to other benchmarks. This is caused by several reasons: First, it combines the lexical features and semantic features (using lexical score and semantic score), which is why it is superior to the traditional classifiers. Second, in addition to lexical features and semantic features, it incorporates the API related features; this results in its superiority to the other answer-ranking methods and DL-based code search methods.

(3) **Our proposed model is substantially better than all of the baseline methods.** We attribute this to the following reasons: First, all of the baseline approaches (including traditional classifiers, answer-ranking methods, as well as the DL-based code search methods) can be viewed as pointwise approaches. Pointwise approaches transform the task of ranking into classification or regression on single QC pairs. They are thus unable to consider the relative order or preference between different code snippets. Nevertheless, ranking is more about predicting relative orders rather than precise relevance scores. In light of this, we propose a pairwise approach to judge the preference relationship between any two given QC pairs rather than the absolute relevance value of a single QC pair. Compared with the pointwise approaches, our model not only considers the relevance between a query and a code snippet, but also investigates the relevance preference of two QC pairs. This is why it is superior to other pointwise baselines. Second, in addition to constructing the preference pairs, our model incorporates the BERT model to embed the QC pairs. The attention mechanism behind BERT makes it possible to express sophisticated functions beyond the simple weighted average, which results in its superiority to the DL-based code search methods. This also

Table 12. Component-wise Evaluation (Python)

Measure	Drop-pairwise	Drop-Bert	Ours
P@1	36.4 ± 1.8%	33.9 ± 1.2%	**42.6 ± 2.5%**
P@2	59.7 ± 2.1%	57.5 ± 1.1%	**64.6 ± 1.1%**
P@3	78.1 ± 1.7%	76.7 ± 1.9%	**80.0 ± 1.0%**
P@4	91.5 ± 1.0%	90.5 ± 1.2%	**92.3 ± 0.9%**
DCG@2	51.1 ± 1.8%	48.8 ± 1.0%	**56.5 ± 1.5%**
DCG@3	60.3 ± 1.5%	58.4 ± 1.2%	**64.2 ± 1.2%**
DCG@4	66.1 ± 1.2%	64.4 ± 0.9%	**69.5 ± 1.0%**
DCG@5	69.4 ± 1.0%	68.0 ± 0.6%	**72.5 ± 1.0%**

signals that the embeddings produced by BERT convey much valuable information, which can better capture the semantics between the user query question and the code snippets.

(4) By comparing the evaluation results of the different datasets (i.e., Python and Java), we can see that our proposed model behaves consistently across different programming languages. This also indicates the generalization ability of our approach.

Answer to *RQ-5: How effective is our CodeSelector for best code snippet selection?* We conclude that our *CodeSelector* is effective for selecting the correct code snippet for a given technical question.

4.6 RQ-6: Component-wise Evaluation

Compared with other methods, the key advantages of our *CodeSelector* are its two sub-components: incorporating the BERT model and employing the pairwise comparisons. To verify the effectiveness of both aforementioned components, we conduct a component-wise evaluation to evaluate their performance one-by-one.

4.6.1 Experimental Setup. For our component-wise evaluation, we compare our model with two of its incomplete versions:

- **Drop-pairwise** removes the pairwise comparison component from our *CodeSelector*. In this experiment, for a given question, we drop the process of constructing the positive and negative preference pairs. To do this, we reconstruct the best QC pairs as positive samples and make the non-best and non-relevant QC pairs as negative samples. The **Drop-pairwise** model is then trained as a binary classification model same as ours.
- **Drop-BERT** removes the BERT component from our *CodeSelector*. In this experiment, we keep the preference pairs construction process but drop the BERT embedding process. To do this, we replace the BERT embedding layers (described in Section 3.3) with the traditional Word2Vec embedding layers. The **Drop-BERT** can then be trained with the preference QC pairs in just the same way.

4.6.2 Experimental Results. The evaluation results of **Drop-pairwise** and **Drop-BERT** are displayed in Tables 12 and 13, respectively. It can be seen that:

(1) **Dropping either component does hurt the performance of *CodeSelector*.** This justifies the importance and effectiveness of both components.

(2) **Drop-BERT achieves the worst performance.** This indicates that a good embedding technique has a major influence on the overall performance. For example, when adding BERT component for embedding the QC pairs, the *P*@1 score is improved by 20.4% and 18.2% on

Table 13. Component-wise Evaluation (Java)

Measure	Drop-pairwise	Drop-Bert	Ours
P@1	36.9 ± 2.2%	34.6 ± 1.7%	**42.4 ± 1.9%**
P@2	60.9 ± 2.6%	59.5 ± 2.6%	**66.1 ± 1.8%**
P@3	78.4 ± 1.8%	78.6 ± 1.8%	**81.6 ± 1.5%**
P@4	91.4 ± 1.2%	91.1 ± 1.0%	**92.6 ± 1.6%**
DCG@2	52.0 ± 2.3%	50.3 ± 2.1%	**57.3 ± 1.6%**
DCG@3	60.8 ± 1.8%	59.8 ± 1.6%	**65.1 ± 1.5%**
DCG@4	66.4 ± 1.6%	65.3 ± 1.2%	**69.8 ± 1.5%**
DCG@5	69.7 ± 1.2%	68.7 ± 0.9%	**72.7 ± 0.9%**

Python and Java datasets, respectively. We attribute this to the advantage of BERT for capturing the intent of the query question as well as the program code. This is why our model outperforms other deep learning-based code searching methods.

(3) By comparing the results of **Drop-pairwise** and **Ours**, we can measure the performance improvement achieved due to the employment of pairwise comparison component. It is clear that by **removing the pairwise comparison component, there is a significant drop with respect to different metrics**. This shows that the pairwise comparison component has a significant contribution to the overall performance of our model. This is the reason why our model outperforms other pointwise baselines.

Answer to *RQ-6: How effective is the BERT component and preference pairs added to our CodeSelector?* We conclude that both the BERT component and the preference pairs are effective and helpful to enhance the performance of our *CodeSelector*.

4.7 RQ-7: Robustness Analysis

Considering the complexity and diversity of the CQA sites, there is little chance to find the past solved questions that exactly match the user query questions. We thus have to enlarge K—the number of retrieved questions—to improve the recall of the relevant questions as well as the potential code snippets within these questions. However, a larger K may often bring more noise into the code snippet candidates pool. This increases the complexity and difficulty for recommending the potential code snippet. We conduct a robustness analysis to investigate our model's performance with respect to different number of retrieved questions.

4.7.1 Experimental Setup. To verify the robustness of our proposed model, we set different thresholds for the number of returned questions. In particular, we increase the number of returned questions k from 5 to 10 (k = 5 corresponds to our model described in RQ-4) and then evaluate the performance of our *CodeSelector* with respect to different parameter settings of K.

4.7.2 Experimental Results. The average P@1-5 over Python and Java datasets are shown in Figure 8. By jointly analyzing these two figures, we can have the following observation:

(1) The **overall performance trend of our model goes down as k increases**. This justifies our previous concerns that more noise is introduced when enlarging k, which incurs bigger challenges for our task.

(2) The **performance of our model on the Java dataset decreases faster than that on Python dataset**. The reason for this phenomenon may be that the Java code snippets are more complex and contain more noise compared with Python code snippets even under the same k settings.

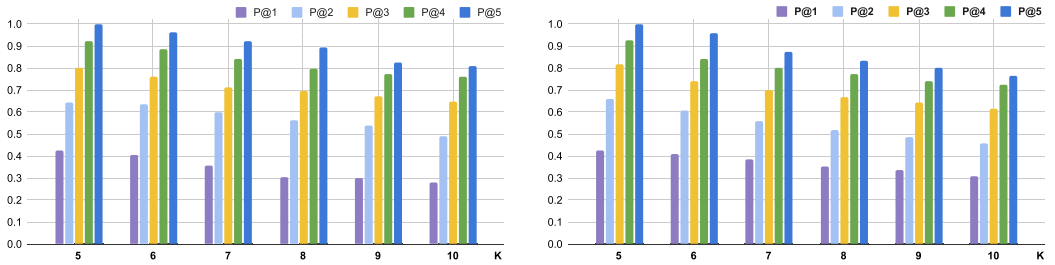

Fig. 8. Robustness analysis on Python (left) and Java (right).

(3) The **performance drop of our model with increasing k is not very large.** For example, when we set k to 10, the performance of our model on $P@1$ is still better than or comparable with the best performance on several baselines. This further shows the robustness of our model.

Answer to *RQ-7: How robust is our CodeSelector with different parameter settings?* We conclude that our approach is robust to noises.

5 HUMAN EVALUATION

The goal of our tool is to recommend the best code snippets that most closely match a developer's intent from past solved questions in Stack Overflow. We perform a user study to measure how developers actually perceive the results produced by our approach.

5.1 Human Evaluation Preliminary

5.1.1 Participants Selection. Since we are recommending code snippets for newly posted queries, we thus sampled 50 unanswered Python posts from Stack overflow for our human evaluation. We recruited 10 participants to join our human evaluation, which is larger than or comparable to the size of the user study participants in previous studies [18, 34, 69]. Our user study includes 1 postdoctoral fellow and 4 Ph.D. students majoring in Computer Science and 5 software developers from industry. All of these selected participants have working experience with Python development. The years of their working experience on Python range from 3 to 10 years. In practice, our tool aims to help different levels of practitioners, from novice to senior developers. The diversity of their background (i.e., from academic and industry) and their working years can improve the generality of our user study. Our human evaluation includes two user studies: the user study on question relevance and the user study on code usefulness. All the selected evaluators are asked to participate in the above two user studies.

5.1.2 Human Evaluation Baselines. We use the following baselines for our human evaluation:

- **Google Search Engine.** Considering developers usually search for technical help using the Google search engine, we employ the Google search engine as one of our baselines. For the Google search engine, we use the question title of the post as the search query; we then add the "site:stackoverflow.com" to the end of the search query so it only searches on Stack Overflow. We treat the first ranked question returned by Google search engine as the most relevant question, and we treat the code fragment within its best answer (the accept answer or the highest vote answer) as the most relevant code snippet.
- **Stack Overflow Search Engine.** Similar to Google search engine, Stack Overflow also provides a service to search relevant posts. For the Stack Overflow search engine engine, we

refer to the first ranked related question suggested by the Stack Overflow as the most relevant question and the code snippet within its best answer as the recommended solution.

- **CROKAGE. CROKAGE** outperforms the other baselines in both stage one (semantically equivalent question retrieval) and stage two (best code snippet recommendation). Therefore, we also employ the **CROKAGE** to find the relevant questions and code snippets for a given query in our human evaluation.
- **1stRanked.** Given a user query, our approach first retrieves the semantically equivalent questions, then it reranks all the code snippets associated with these questions. Different from our approach, the **1stRanked** method drops the second stage of the code reranking process and simply uses the code snippet from the first ranked question as the recommended solution.
- **Que2Code.** For our approach, we perform an end-to-end evaluation using our Que2Code tool. In particular, we use the *QueryRewriter* to retrieve the semantic relevant questions in Stack Overflow and then use the *CodeSelector* to select the best code snippet for the unanswered question. After *CodeSelector* reranks all the code snippets, we treat the first ranked code snippet and its associated question as the recommended solution.

5.2 User Study on Question Relevance

5.2.1 Experimental Setup. Since we are recommending code snippets from semantically equivalent questions, if the retrieved questions are not relevant to the user query question, then it is unlikely that we can select the appropriate code snippet from the answer candidates pool. Therefore, we first conduct a user study to measure how humans perceive the question retrieval results. To do this, we consider the *question relevance* modality for this user study. The *question relevance* metric measures how relevant is the retrieved question to the user query question. To be more specific, we asked participants to do a web questionnaire. For each unanswered question, the evaluator is displayed with this user query question along with five retrieved questions from the above baselines. For each retrieved question, the participant is asked to give a score between 1 and 4 to measure the relevance between the user query question and the retrieved question. We define the scoring guidelines as follows:

- Score 1: The two questions have no relevance.
- Score 2: The two questions share some common words but are semantically irrelevant.
- Score 3: The two questions are semantically similar.
- Score 4: The two questions are semantically equivalent (they are identical in meaning).

We provide the scoring guidelines in the beginning of each questionnaire to guide participants. Figure 9 shows one example in our survey. It is worth mentioning that the order of the five retrieved questions is randomly decided, so the participants do not know which question is generated by our approach.

5.2.2 Experimental Results. We obtained 500 groups of scores from the above user study. Each group contains five scores for five retrieved questions, respectively. We regard a score of 1 as *low* quality, a score of 2 as *medium*− quality, a score of 3 as *medium*+ quality, a score of 4 as *high* quality. The score distribution and the mean score of *question relevance* across baselines are presented in Table 14. We also evaluate whether the differences between our approach and the baselines are statistically significant by performing Wilcoxon signed-rank test [67]. From the table, we can see that:

(1) **The Stack Overflow search engine achieved the worst performance among all the baselines.** The very large proportion of low-quality questions reflects that the Stack Overflow search engine is ineffective for searching relevant questions for newly posted questions.

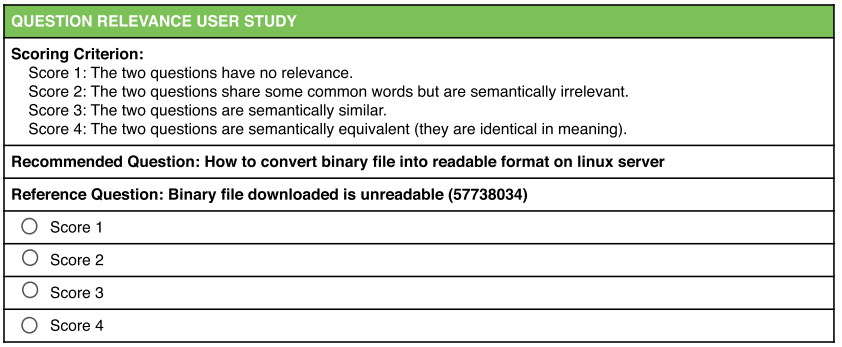

Fig. 9. Example of question relevance user study.

Table 14. User Study on Question Relevance

Measure	Low	Medium-	Medium+	High	Mean Score	P-value
Google	3.20%	31.60%	36.80%	28.40%	2.91	>0.5
Stack Overflow	58.40%	28.80%	6.80%	6.00%	1.60	<0.01
CROKAGE	12.80%	40.00%	36.40%	10.80%	2.45	<0.01
1stRanked	8.00%	33.20%	45.60%	13.20%	2.64	<0.01
Ours	2.80%	27.20%	47.20%	22.80%	2.90	–

We manually checked the 50 questions produced by Stack Overflow search engine; it recommended the same common question 15 times, which explains its weak performance in question retrieval tasks.

(2) **Our approach outperforms the CROKAGE baseline regarding the *question relevance* metric.** The results of human evaluation are consistent with large-scale automatic evaluation results, which further justifies the effectiveness of our approach for retrieving relevant questions.

(3) **The 1stRanked method has its advantages as compared to other baselines (i.e., Stack Overflow and CROKAGE).** This is reasonable, because both the 1stRanked method and our approach employs *QueryRewriter* for embedding relevant questions. The *QueryRewriter* incorporates historical duplicate question pairs from Stack Overflow, such that two semantically equivalent questions are close in terms of vector representations.

(4) **Google search engine performs better than our approach regarding the *high* quality questions.** Considering Google's capability, i.e., the larger searching database and accumulated user searching histories, it is not surprising that the Google search engine can identify the high-quality relevant questions for the user query. **However, our approach still achieves comparable mean score regarding the *question relevance* metric, and the difference between our approach and Google search engine is also not statistically significant.** This is because the proportion of relevant questions (including the *medium+* and *high* quality questions) outnumber those of the Google search engine. This reflects that, for a given unseen question, our approach is more likely to retrieve relevant questions in general.

5.3 User Study on Code Usefulness

5.3.1 Experimental Setup. Our final goal is searching for useful code snippets to help developers solve unanswered questions. We also conduct a user study to measure the *code usefulness* of our

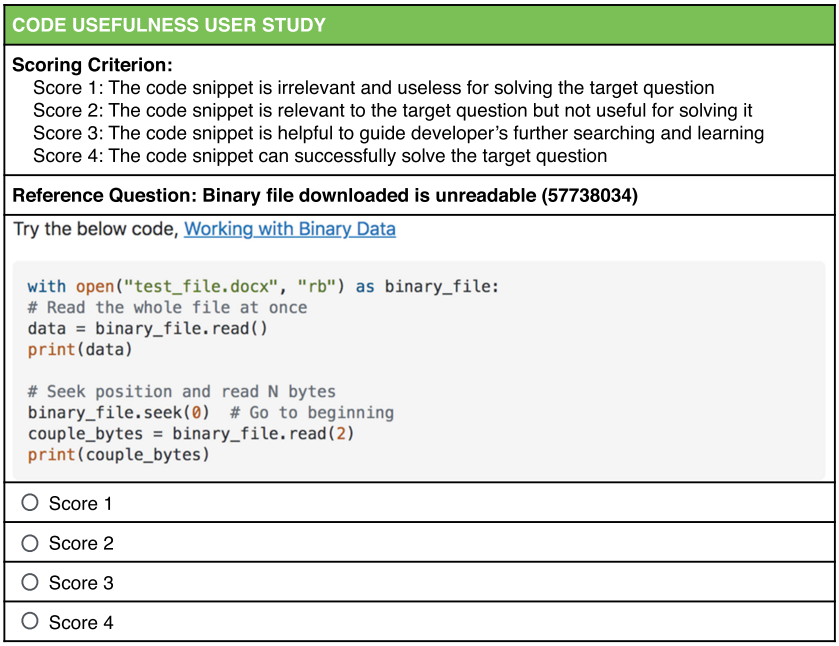

Fig. 10. Example of code usefulness user study.

recommended code snippets. The *code usefulness* metric refers to how useful the recommended code snippet is for solving the user query questions. Similar to our previous user study, for each unanswered question, we provide five code snippets candidates recommended by five baselines, respectively. After that, each evaluator was asked to rate five code snippets from 1 to 4 according to its *code usefulness* with respect to the following scoring guidelines:

- Score 1: The code snippet is irrelevant and useless for solving the target question.
- Score 2: The code snippet is relevant to the target question, but not useful for solving it.
- Score 3: The code snippet can guide developer's further searching and learning, which is useful to solve the target question.
- Score 4: The code snippet can successfully solve the target question.

We provide the scoring guidelines in the beginning of the questionnaire to guide the evaluators. Figure 10 demonstrates one example of our survey. The evaluators were blinded to which code snippet is generated by our approach.

5.3.2 Experimental Results. Same as for the user study on *question relevance*, after obtaining the evaluator's feedback, we regard a score of 1 as *low* quality, a score of 2 as *medium*− quality, a score of 3 as *medium*+ quality, and a score of 4 as *high* quality. The score distribution and the mean score of *code usefulness* across different baselines are presented in Table 15. The Wilcoxon signed-rank test [67] is also performed between our approach and each baseline method; the results are displayed in the last column of Table 15. From the table, we can see that:

(1) **Our model significantly outperforms all the baselines (including the Google search engine) regarding the *code usefulness* metric.** This suggests that the code snippets recommended by our approach are considered to be more useful to the given query question compared with baselines. The reason may be due to the two stages of our approach,

Table 15. User Study on Code Usefulness

Measure	Low	Medium-	Medium+	High	Mean Score	P-value
Google	17.60%	23.60%	32.40%	26.40%	2.68	<0.01
Stack Overflow	73.60%	18.00%	6.40%	2.00%	1.37	<0.01
CROKAGE	30.80%	26.00%	28.40%	14.80%	2.27	<0.01
1stRanked	28.40%	31.60%	23.20%	16.80%	2.28	<0.01
Ours	12.00%	22.00%	34.00%	32.00%	2.86	–

i.e., *semantically equivalent question retrieval* and *best code snippet recommendation*. The first stage focuses on retrieving as many as possible relevant questions to construct the candidate set. The more relevant the retrieved question is to the query, the more likely the code snippet associated with the question is helpful to solve the problem. The second stage tries to rank the useful code snippet to the top of the recommendation result.

(2) **Compared with the 1stRanked method, our approach has its advantages.** Even though the 1stRanked method retrieves the same relevant question candidates as ours, it naively picks the code snippet from the most relevant question. However, considering the complexity of the technical queries, it is very hard to find identical questions to the given query. Therefore, it is necessary to consider the correlation between the code snippet and the user query. Different from the 1stRanked method that is solely based on *question relevance*, our *CodeSelector* reranks all the code snippet candidates by performing comparisons pairwisely. The *CodeSelector* not only considers the program semantics between the code snippet and the query question, but also investigates the relevance preference between different QC pairs. Thus, a useful code snippet can be ranked higher up among other candidates. The superior performance of our approach regarding the mean score of *code usefulness* further supports the ability of our *CodeSelector* model for recommending useful code snippet.

(3) **By comparing the mean score of *code usefulness* and *question relevance*, there is a significant drop overall in every baseline method.** This indicates that, compared with relevant question retrieval tasks, identifying useful code snippets is more challenging. Technical questions in Stack Overflow are rather complicated and specific; even though two technical questions are semantically relevant, the code snippet is not applicable or reusable for the programming task. **We also observe that the performance drop of our approach is much smaller than other baseline models.** We attribute this to the effectiveness of *CodeSelector* for putting relevant snippet on higher positions than the irrelevant ones; this also verifies the importance and necessity of our *CodeSelector* in the second stage.

5.4 Qualitative Analysis

5.4.1 Experimental Setup. In this work, we aim to alleviate the *query mismatch* and *information overload* problems by using the *QueryRewriter* and *CodeSelector*, respectively. To vividly demonstrate the workflow details of our model (i.e., from generating paraphrase questions, semantically equivalent question retrieval, and best code snippet selection), we further conduct a case study to manually investigate some questions used in the previous user study. Figure 11 shows five examples from the previous user studies to demonstrate the detailed results. For each unanswered question, we present the intermediary results of the generated paraphrase questions, the top-five ranked retrieved question, as well as the recommended code snippets generated by our approach. The words that do not appear in the original query question, but include both in the generated paraphrase questions and the target retrieved questions, are highlighted in yellow color. We also highlight the question in boldface, which provides the recommended code snippet.

Query Question	Generated Paraphrase Question	Retrieved Questions	Recommend Code Snippet
Ex1. (12702617) automatic deletion of directory in linux (centos)	how to quickly remove directory in python	(27700114) How to delete a python directory effectively?	import shutil shutil.rmtree("path_to_dir")
	how to delete files in python with subdirectory	**(43756284) How to remove a directory including all its files in python?**	
	how to rename a directory with ipython script	(51080451) How do I delete a specified number of files in a directory in Python?	
	setting a directory for deletion in python	(36156426) How rename files in a directory using python?	
	how to delete a directory in python2.7 on centos?	(31223312) how to rename file in a directory using python	
Ex2. (12745563) use of plot3d in python	plotting 3d vector using matplotlib	**(47806784) how to plot a 3d graph in python using matplotlib?**	import matplotlib.pyplot as plt from sklearn.datasets import make_s_curve from mpl_toolkits.mplot3d import Axes3D # make and plot 3d X, y = make_s_curve(n_samples=1000) ax = plt.axes(projection='3d') ax.scatter3D(X[:, 0], X[:, 1], X[:, 2], c=y) ax.view_init(10, -60) plt.show()
	how to plot 3d on a computer	(4802157) How to make 3D plots in Python?	
	use the 3d symbol when plotting with matplotlib	(49561740) How to make a 3D data surface plot using matplotlib in python	
	use of 3d plots in python	(25286811) How to plot a 3D density map in python with matplotlib	
	pyplot 3d plots, variables	(36811960) How do I plot a 2D array graph in Python using matplotlib	
Ex3. (13090635) Diagnose python memory usage	python memory management	**(552744) How do I profile memory usage in Python?**	# heaps is quite simple to use from guppy import hpy h = hpy() print(h.heap()) # This gives you some output like this: … Index Count % Size % Cumulative % Kind 0 25773 53 1612820 49 1612820 49 str 1 11699 24 483960 15 2096780 64 tuple …
	using a python module to monitor memory usage	(43737948) How do I determine the memory usage of a python type?	
	memory leak on python object	(21701434) How to manage memory error in python?	
	python memory error	(11596371) How Does Python Memory Management Work?	
	how can i easily determine memory usage of python packages?	(55392166) How to detect memory leak in python code?	
Ex4. (3201964) merging 2 columns in a csv file through python	how to merge dataframe columns?	**(38049548) How to merge two columns into one in csv format (Python Pandas)?**	A, B = [], [] with open(your_file) as f: for line in f: if …: A.append(line.split(your_seperator)) else: B.append(line.split(your_seperator)) A = pd.DataFrame(A, columns = list_of_columns) B = pd.DataFrame(B, columns = list_of_columns_2) .drop(columns_to_drop, 1) df = pd.concat([A, B]).reset_index(drop = True)
	merging dataframe rows of different lengths	(31384177) How to merge two pandas DataFrames in Python?	
	using pandas to merge two column in a file	(37894654) How to merge dataframes using pandas python?	
	merge text data in csv file using python	(37697195) how to merge two data frames based on particular column in pandas python?	
	pandas merging 101	(50708959) How to merge two columns from a dataframe	
Ex5. (5939210) is python faster than php?	speed of python vs .php	(5497540) How to call a Python Script from PHP?	$activateScript = $_GET['activeScript']; exec("python / path / to / file.py $activateScript");
	python vs php: which is faster ?	**(36859666) How run python with html or php?**	
	python over php	(1686192) How fast is Python?	
	how php compares to python's model of program	(1060436) How do I include a PHP script in Python?	
	are python two times faster than php	(16288021) How to combine php & python code in Python	

Fig. 11. Qualitative analysis.

5.4.2 Experimental Results. From the cases demonstrated in Figure 11, we can see that:

(1) **The paraphrase questions generated by our *QueryRewriter* are meaningful for the given user query question.** Note that the *QueryRewriter* automatically transforms the user query question into different forms of semantically equivalent user expressions. For example, in the third case, the original user query question is about "*Diagnose python memory usage*," and our approach generates multiple paraphrase questions such as "*python memory management*," "*using a python module to monitor memory usage*," "*memory leak on python object*." These generated paraphrase questions can be viewed as meaningful outputs to capture the developer's intent and used as a way of question boosting for the original user query question.

(2) **It is clear that adding the paraphrase questions can reduce the lexical gap between different user expressions**, which increases the likelihood of retrieving the semantic relevant questions in Stack Overflow. As shown in the second case, the developer formulates his/her problem as a user query "*use of plot3d in python*"; the generated paraphrase question "*plotting 3d vector using matplotlib*" can add missing information for the user query question and better link to the target semantically equivalent question in Stack Overflow (i.e., "*how to plot a 3d graph in python using matplotlib*"), which verifies the ability of our QueryRewriter to alleviate the *query mismatch* problem.

(3) **A large number of code snippets recommended by our system are relevant and useful with respect to the user query question.** Some code snippets can well satisfy the developer's programming tasks directly. For example, as shown in the first case of Figure 11, the developer's query question "*automatic deletion of directory in linux (centos)*" can be successfully solved by the code snippets within a relevant question "*How to remove a directory including all its files in python?*" This verifies the effectiveness and possibility of our approach for recommending code solutions from the existing historical answers.
(4) **We also notice that the recommended code snippets are not always selected from the first ranked question candidate.** Instead of naively choosing the first ranked code snippet like the Google search engine and Stack Overflow search engine, our *CodeSelector* selects the best code snippet among a set of code snippet candidates via pairwise comparisons, which justifies the ability of our *CodeSelector* to alleviate the *information overload* problem.
(5) However, **the recommended code snippets from our system are not always useful**. For example, in the second case, the developer would like to inquire about "*use of plot3d in python*"; our recommended code snippet is about "plot 3d graph using matplotlib," which can be viewed as helpful by looking at the intent of the developer. In the fourth sample, even though the recommended code snippet can not be applied directly, it can be easily adapted to the user query question with minor modification.
(6) Also, **the recommended code snippets from our system are not always relevant**. For example, in the last sample, even though the generated paraphrase questions are meaningful and relevant to the user query question, the final recommended code snippet is still irrelevant to the problem described in the query question. This is because some user queries posted by developers are often complex and sophisticated; there may not always exist semantically equivalent questions in our code database. It is thus very hard to search the query-specific code snippet to solve the corresponding problem.

Overall, our approach is more effective for retrieving relevant questions and searching useful code snippets compared with other baselines under human evaluation.

6 PRACTICAL USAGE

The experiment was conducted on an Nvidia GeForce GTX 2080 GPU with 12 GB memory. The time cost of our approach is mostly for the training process, which takes approximately 20 to 24 hours for training Python and Java datasets, respectively. However, after finishing the training process, each question title and code snippet in our database can be converted into vector representations, which is highly efficient for later computing and searching processes. For example, the searching process on 5,000 examples takes five to eight minutes, while searching a single code snippet only costs 60 to 80 ms.

Considering that searching code snippets in Stack Overflow with our approach is efficient, we have implemented Que2Code as a prototype web-based tool, which can facilitate developers in using our approach and inspire follow-up research. Figure 12 shows the web interface of Que2Code. Developers can type or paste their query question into our web application, after that Que2Code goes through the question retrieval and code snippet reranking process and recommends the top ranked questions and code snippets to the developers. We below describe the details of the input and output of our tool.

- **Input:** The input to the Que2Code is a user query question, which is a sequence of tokens. The input box in Figure 12 shows an example of the user query question, i.e., "*how to find the most frequent item in array*." After inputting the user query question, the developer can click the "Search" button to submit their query.

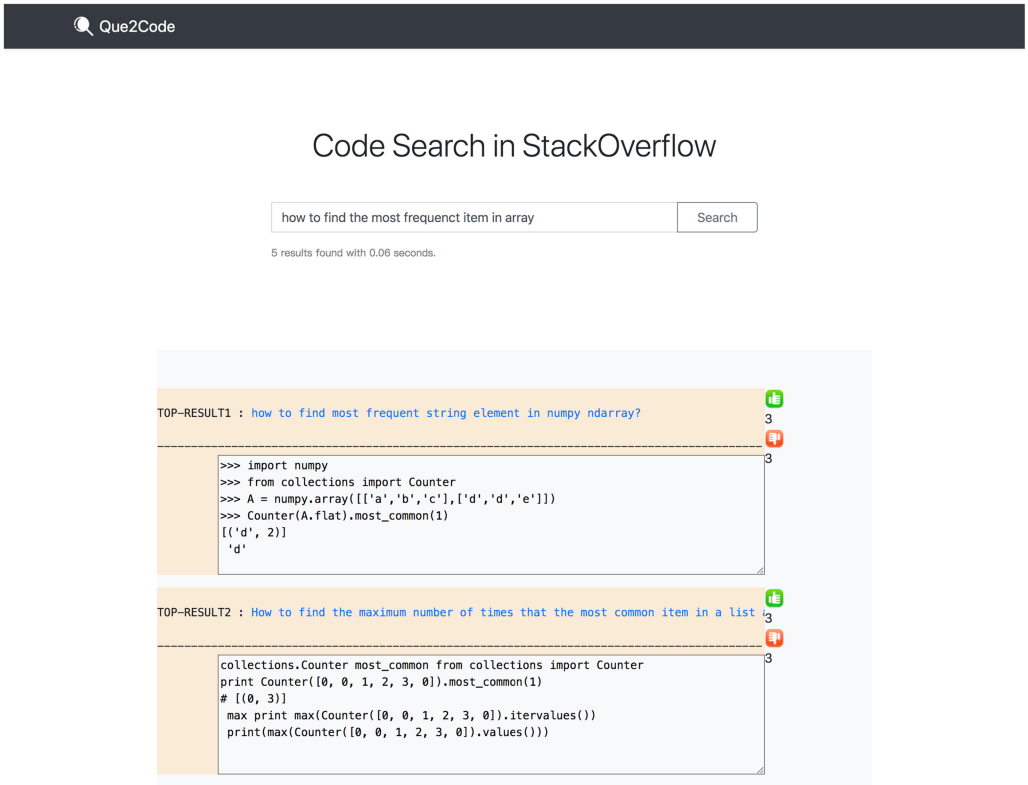

Fig. 12. Prototype of Que2Code.

- **Output:** The output of the Que2Code is two-fold: relevant questions and code snippets. After the developer submits his/her query to the server, the Que2Code searches through the codebase and returns top-five relevant code snippets with their associated question titles. The link to these question posts on Stack Overflow is also provided for user references. For example, the relevant question post "*how to find the most frequent string element in numpy array*" and its code snippet are retrieved from our database to guide developers for solving their problems. Developers can use our tool to quickly locate the potential solutions to their query programming tasks and have a better understanding of their problems.

The goal of our tool is helping developers effectively search code snippets from Stack Overflow to their programming tasks and saving their time to do so. After browsing these code snippets, developers still need to manually modify these code snippets for further refactoring and testing.

Overall, our approach is efficient enough for practical use, and we have implemented a web service tool, Que2Code, to apply our approach for practical use.

7 DISCUSSION

We discuss the strengths and weaknesses of our approach as well as the threats to validity for our experiments.

7.1 Strengths and Limitations of Our Approach

7.1.1 Strengths of Our Approach. To address the *query mismatch* and *information overload* problem in the Stack Overflow community, we proposed a novel query-driven code snippet recommendation model. Its key strengths are summarised below.

(1) Paraphrase Question Generation. A key advantage of our model is training a text-to-text transformer, named *QueryRewriter*, for generating paraphrase questions as a way of question boosting. This greatly improves the likelihood of our approach of retrieving semantically equivalent questions for a user query question. By training on the duplicate question pairs in Stack Overflow, for a user query question, *QueryRewriter* is able to generate different user descriptions for the same problem, which is helpful for our semantically equivalent question retrieval tasks. The experimental results in Section 4.4 verify the effectiveness of adding paraphrased questions to our model.
(2) Pairwise Learning to Rank. To recommend the most relevant code snippets in Stack Overflow, we propose a novel pairwise learning to rank model in our work. Guided by our three heuristic rules, we can automatically construct the preference QC pairs and transform the code snippet recommendation task to a binary classification task. Rather than calculating a precise relevance score for a single QC pair, we estimate the preference relationship between two QC pairs. Ranking code snippets by pairwise comparison is more suitable for the code recommendation task in our study.
(3) BERT Model for Embeddings. BERT is designed to pre-train deep bidirectional representations from unlabeled text. It is conceptually simple and empirically powerful, which obtains new state-of-the-art results on 11 natural language processing tasks, such as question answering, language inference, and so on. In our research, we investigated the BERT model for embedding query and code snippet pairs, and the ablation analysis in Section 4.6 demonstrates that it greatly enhanced the performance of our proposed model.

7.1.2 Limitations of Our Approach. We have identified the following limitations of our approach:

(1) Model Limitations. We have collected our dataset from official data dump from Stack Overflow. Our work is up to the release date of the official data dump that we used, and we cannot automatically update our model with the newly added data after the release date. However, this limitation can be supported by adding extra engineering work (e.g., automatically updating the model every month by checking the newest version of Stack Overflow data dump); we will try to expand our approach to handle the newly added data in our future work. In terms of practical usage, the experimental results show that our model outperforms Google search for low-quality questions, while our work does not provide mechanisms to estimate the question quality currently. However, this limitation can be alleviated by incorporating a question quality prediction model, which has been widely studied in prior works [5, 45, 62]. For example, a developer can first leverage the quality prediction model to estimate the question quality, then our tool can be used to search suitable code snippets from Stack Overflow for low-quality questions.
(2) Query Limitations. Our model aims to identify the best code solutions for a user query from Stack Overflow posts. However, there are different types of user queries that are relevant to different search tasks (e.g., explanations for unknown terminologies, explanations for exceptions/error messages), which may not always be searching for code snippets. The Que2Code model currently recommends code snippet for a user query if it is asking about code solutions; we will try to focus on other types of questions in our future work.

7.2 Threats to Validity

We have identified the following threats to validity:

7.2.1 Internal Validity. Internal validity relates to potential errors in our model implementation and experimental settings. To reduce errors in automatic evaluation, we have carefully tuned the parameters of the baseline approaches and used them in their highest-performing settings for comparison, but there may still exist errors that we did not notice. Considering such cases, we have released the code and data of our research to facilitate other researchers to repeat our work and verify their ideas. Regarding the distance metric, we choose Euclidean distance to estimate the semantic distance between different questions; we did not consider other distance metrics (e.g., cosine distance) in this preliminary study. We believe our model will generalize to other distance metrics due to the valuable information of our embeddings. We will explore the effectiveness of different distance metrics in future studies.

7.2.2 External Validity. Threats to external validity are concerned with the generalizability of our dataset. Our dataset is collected from the official Stack Overflow data dump. We focus on two popular programming languages, i.e., Python and Java, for our experiment. However, there are still many other programming languages that are not considered in our study. We believe that our model will generalize to other programming languages due to the effectiveness and robustness of our approach. We will try to extend our approach to other programming languages to benefit more developers in future studies.

7.2.3 Model Validity. The model validity relates to model structure that could affect the learning performance of our approach. In the first stage, we choose an encoder-decoder architecture for our *QueryRewriter*. Such an encoder-decoder architecture targets the sequence-to-sequence learning problem, which requires a large amount of manually labeled duplicate question pairs. However, our model may not generalize to other technical Q&A sites if the training set is limited. In the second stage, we choose the basic BERT model as our embedding layer due to its promising results on a wide range of NLP tasks [14, 19]. Recent research has proposed new models, such as GPT [46], RoBERTa [33], DistilBERT [54], ALBERT [31], that can achieve better performance than BERT and/or similar performance with much less parameters. However, our results do not shed light on the effectiveness of employing other deep learning models with respect to different structures and new advanced features. We will try to use other deep learning models for our tasks in future work and compare them to the results that we report in this article.

8 RELATED WORK

8.1 Code Search in Software Engineering

The goal of code search is to find code fragments from a large code repository that most closely match a developer's intent. Many code search methods have been proposed in the literature [10, 21, 35, 36, 40, 44, 53, 74]. Existing code search methods can be classified into two mainstreams: Information Retrieval-based methods and Deep Learning-based methods.

Ye et al. [74] proposed a model to fill the gap between natural-language queries and code snippets by projecting them into the same high-dimensional vector space. Sachdev et al. [53] proposed a neural code search method that combined the of token-level embeddings with the traditional information retrieval techniques TF-IDF. They found that the basic word-embedding techniques can achieve good performance on code search task. Gu et al. [21] proposed a supervised technique, named DeepCS, for code searching using deep neural networks. They used multiple sequence-to-sequence-based networks to capture the features of the natural language queries and the code snippets.

The above code search methods are designed to measure the relevance degree between an individual QC (natural language query-code snippet) pair. However, in our study, rather than modeling a single QC pair to predict the precise relevance score, we model the preference relationship between two QC pairs. In other words, our model not only considers the relevance between a query and a code fragment, but also investigates the preference relationship between different QC pairs.

8.2 Duplicated Questions in Stack Overflow

The quality of the user-generated content is a key factor to attract users to visit the CQA sites, such as Stack Overflow. Prior work suggests that a quality decay problem occurs in these CQA community due to the growth in the number of duplicate questions [60]. This makes finding answers to a question harder and may dilute quality of answers. To maintain the quality of posts in Stack overflow, many studies have investigated the duplicate questions in Stack Overflow [1, 39, 56, 65, 75, 76].

Zhang et al. [76] proposed an approach, named DupPredictor, to predict whether a question is duplicate question in Stack Overflow. They considered multiple factors, such as similarity scores of topics, titles, descriptions, and tags for each question pair and calculated an overall similarity score by combining these features. Followed by their research, Ahasanuzzaman et al. [1] first manually investigated why duplicate questions are asked by users in Stack Overflow, then proposed Dupe, which extracted features from question corpus and then built a binary classifier to judge if a question pair is duplicated or not. More recently, Wang et al. [65] presented a deep-learning based approach to detect duplicate questions in Stack Overflow, which can capture the document-level and word-level semantic information, respectively.

In our work, instead of considering the negative aspect of the duplicated questions in Stack Overflow, we consider duplicated question as semantically equivalent questions pairs. We then train a query rewriting model for retrieving relevant questions in Stack Overflow.

8.3 Query Reformulation in Software Engineering

The effectiveness of code search heavily relies on the quality of the search query. If a query performs poorly, then searching useful code snippets becomes increasingly difficult. Therefore, it is necessary to reformulate and/or improve the user query when the query is poorly expressed. This need has motivated researchers to investigate the query reformulation (or expansion) approaches for software engineering tasks [11, 22, 26, 28, 35, 41, 49, 50, 55, 58].

Shepherd et al. [55] presented an approach, V-DO, that automatically extracts verbs and objects from source code comments for misspelled query terms. Haiduc et al. [22] developed a query reformulation strategy by performing machine learning on a set of historical queries and relevant results. Following that, Hill et al. [26] proposed a query expansion tool, named *Conquer*, which combines the V-DO and contextual searching technique to suggest alternative query words. Lu et al. [35] implemented an approach to expand a query by using synonyms with the help of WordNet. Nie et al. [41] proposed a model, named *QECK*, to identify the software-specific expansion words from the high-quality pseudo feedback on Stack Overflow and generate expansion queries. After that, Rahman et al. [50] proposed a query reformulation approach that suggests a list of relevant API classes for code search. Most recently, Cao et al. [11] proposed an automated deep-learning based query reformulation approach by using the query logs in Stack Overflow.

Different from the existing query expansion approaches, in this study, we first investigate the possibility of using duplicate question pairs from Stack Overflow for query rewriting. Our *QueryRewriter* can capture features between semantically equivalent questions and address the query mismatch problem.

8.4 Question Answering in CQA Sites

Finding similar questions and/or appropriate answers from historical archives has been applied in CQA sites. Great effort has been dedicated to various tasks, such as question retrieval [12, 15, 64, 70, 73, 78], answer selection [18, 42, 57, 69], tagging [20, 66, 77], and expert identification [30, 43, 61].

Conventional techniques for retrieving answers primarily focus on complementary features of the CQA sites. Calefato et al. [9] transform the answer selection task to a binary classification problem; they empirically evaluated 26 answer prediction model in Stack Overflow. Xu et al. [69] proposed a novel framework for generating relevant, useful, and diverse answer summary for technical questions in Stack Overflow. Rather than directly ranking community answers, Tian et al. [61] predicted the best expert for a technical question in Stack Overflow by assuming that good respondents will give better answers. More recently, Gao et al. [17] proposed a model for generating good question titles for developers by mining the code snippets in Stack Overflow.

Different from the aforementioned studies, we aim to search the best code fragment from the historical data in CQA database. We frame this task as a query-driven code recommendation task, and we propose a two-stage framework to address the semantic-equivalent question retrieval and best code recommendation task, respectively.

9 CONCLUSION AND FUTURE WORK

We have presented a fully data-driven approach, named Que2Code, for recommending the best code snippet in Stack Overflow for a user query question. We formulate this task as a query-driven code recommendation problem. Our proposed Que2Code model contains two components: *QueryRewriter* and *CodeSelector*. In particular, we proposed a *QueryRewriter* for retrieving semantically equivalent questions (as the first stage) and a *CodeSelector* for selecting the best code fragment in Stack Overflow (as the second stage). We have conducted extensive experiments to evaluate our approach on Stack Overflow dataset. Compared with several existing baselines, experimental results have comparatively demonstrated the effectiveness and superiority of our proposed model in both evaluation and human evaluation.

ACKNOWLEDGMENTS

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore. The authors would like to thank the reviewers for the insightful and constructive feedback.

REFERENCES

[1] Muhammad Ahasanuzzaman, Muhammad Asaduzzaman, Chanchal K. Roy, and Kevin A. Schneider. 2016. Mining duplicate questions of stack overflow. In *Proceedings of the IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*. IEEE, 402–412.

[2] Syed Ahmed and Mehdi Bagherzadeh. 2018. What do concurrency developers ask about? A large-scale study using stack overflow. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. 1–10.

[3] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. 2016. Layer normalization. *arXiv preprint arXiv:1607.06450* (2016).

[4] Kartik Bajaj, Karthik Pattabiraman, and Ali Mesbah. 2014. Mining questions asked by web developers. In *Proceedings of the 11th Working Conference on Mining Software Repositories*. 112–121.

[5] Antoaneta Baltadzhieva and Grzegorz Chrupała. 2015. Predicting the quality of questions on stackoverflow. In *Proceedings of the International Conference Recent Advances in Natural Language Processing*. 32–40.

[6] Steven Bird and Edward Loper. 2004. NLTK: The natural language toolkit. In *Proceedings of the ACL Conference on Interactive Poster and Demonstration Sessions*. Association for Computational Linguistics.

[7] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. 2017. Enriching word vectors with subword information. *Trans. Assoc. Computat. Ling.* 5 (2017), 135–146.

[8] Joel Brandt, Philip J. Guo, Joel Lewenstein, Mira Dontcheva, and Scott R. Klemmer. 2009. 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.

[9] Fabio Calefato, Filippo Lanubile, and Nicole Novielli. 2019. An empirical assessment of best-answer prediction models in technical Q&A sites. *Empir. Softw. Eng.* 24, 2 (2019), 854–901.

[10] Jose Cambronero, Hongyu Li, Seohyun Kim, Koushik Sen, and Satish Chandra. 2019. When deep learning met code search. In *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 964–974.

[11] Kaibo Cao, Chunyang Chen, Sebastian Baltes, Christoph Treude, and Xiang Chen. 2021. Automated query reformulation for efficient search based on query logs from stack overflow. In *Proceedings of the IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1273–1285.

[12] Xin Cao, Gao Cong, Bin Cui, and Christian S. Jensen. 2010. A generalized framework of exploring category information for question retrieval in community question answer archives. In *Proceedings of the 19th International Conference on World Wide Web*. 201–210.

[13] Rodrigo Fernandes Gomes da Silva, Chanchal K. Roy, Mohammad Masudur Rahman, Kevin A. Schneider, Klérisson Paixão, Carlos Eduardo de Carvalho Dantas, and Marcelo de Almeida Maia. 2020. CROKAGE: Effective solution recommendation for programming tasks by leveraging crowd knowledge. *Empir. Softw. Eng.* 25, 6 (2020), 4707–4758.

[14] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[15] Debasis Ganguly and Gareth J. F. Jones. 2015. Partially labeled supervised topic models for RetrievingSimilar questions in CQA forums. In *Proceedings of the International Conference on The Theory of Information Retrieval*. 161–170.

[16] Zhipeng Gao, Lingxiao Jiang, Xin Xia, David Lo, and John Grundy. 2020. Checking smart contracts with structural code embedding. *IEEE Trans. Softw. Eng.* 47, 12 (2020), 2874–2891. DOI: 10.1109/TSE.2020.2971482

[17] Zhipeng Gao, Xin Xia, John Grundy, David Lo, and Yuan-Fang Li. 2020. Generating question titles for stack overflow from mined code snippets. *ACM Trans. Softw. Eng. Methodol.* 29, 4 (2020), 1–37.

[18] Zhipeng Gao, Xin Xia, David Lo, and John Grundy. 2020. Technical Q8A site answer recommendation via question boosting. *ACM Trans. Softw. Eng. Methodol.* 30, 1 (2020), 1–34.

[19] Zhipeng Gao, Xin Xia, David Lo, John Grundy, and Thomas Zimmermann. 2021. Automating the removal of obsolete TODO comments. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 218–229.

[20] José R. Cedeno González, Juan J. Flores Romero, Mario Graff Guerrero, and Felix Calderón. 2015. Multi-class multi-tag classifier system for stackoverflow questions. In *Proceedings of the IEEE International Autumn Meeting on Power, Electronics and Computing (ROPEC)*. IEEE, 1–6.

[21] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. In *Proceedings of the IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 933–944.

[22] Sonia Haiduc, Gabriele Bavota, Andrian Marcus, Rocco Oliveto, Andrea De Lucia, and Tim Menzies. 2013. Automatic query reformulations for text retrieval in software engineering. In *Proceedings of the 35th International Conference on Software Engineering (ICSE)*. IEEE, 842–851.

[23] Helia Hashemi, Mohammad Aliannejadi, Hamed Zamani, and W. Bruce Croft. 2020. ANTIQUE: A non-factoid question answering benchmark. In *Proceedings of the European Conference on Information Retrieval*. Springer, 166–173.

[24] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 770–778.

[25] Xiangnan He, Lizi Liao, Hanwang Zhang, Liqiang Nie, Xia Hu, and Tat-Seng Chua. 2017. Neural collaborative filtering. In *Proceedings of the 26th International Conference on World Wide Web*. 173–182.

[26] Emily Hill, Manuel Roldan-Vega, Jerry Alan Fails, and Greg Mallet. 2014. NL-based query refinement and contextualized code search results: A user study. In *Proceedings of the IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*. IEEE, 34–43.

[27] Rubing Huang, Chenhui Cui, Weifeng Sun, and Dave Towey. 2020. Poster: Is Euclidean distance the best distance measurement for adaptive random testing? In *Proceedings of the IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 406–409.

[28] He Jiang, Liming Nie, Zeyi Sun, Zhilei Ren, Weiqiang Kong, Tao Zhang, and Xiapu Luo. 2016. ROSF: Leveraging information retrieval and supervised learning for recommending code snippets. *IEEE Trans. Serv. Comput.* 12, 1 (2016), 34–46.

[29] Philipp Koehn. 2004. Pharaoh: A beam search decoder for phrase-based statistical machine translation models. In *Proceedings of the Conference of the Association for Machine Translation in the Americas*. Springer, 115–124.

[30] Varun Kumar and Niranjan Pedanekar. 2016. Mining shapes of expertise in online social Q&A communities. In *Proceedings of the 19th ACM Conference on Computer Supported Cooperative Work and Social Computing Companion*. 317–320.

[31] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. 2019. Albert: A lite BERT for self-supervised learning of language representations. *arXiv preprint arXiv:1909.11942* (2019).

[32] Quoc Le and Tomas Mikolov. 2014. Distributed representations of sentences and documents. In *Proceedings of the International Conference on Machine Learning*. 1188–1196.

[33] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. RoBERTa: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692* (2019).

[34] Zhongxin Liu, Xin Xia, Ahmed E. Hassan, David Lo, Zhenchang Xing, and Xinyu Wang. 2018. Neural-machine-translation-based commit message generation: How far are we? In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 373–384.

[35] Meili Lu, Xiaobing Sun, Shaowei Wang, David Lo, and Yucong Duan. 2015. Query expansion via WordNet for effective code search. In *Proceedings of the IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 545–549.

[36] Fei Lv, Hongyu Zhang, Jian-Guang Lou, Shaowei Wang, Dongmei Zhang, and Jianjun Zhao. 2015. CodeHow: Effective code search based on API understanding and extended Boolean model (E). In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 260–270.

[37] Christopher D. Manning, P. Raghavan, and Hinrich Schütze. 2008. *Introduction to Information Retrieval*. Vol. 39, Cambridge University Press, Cambridge, 234–265.

[38] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S. Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. In *Advances in Neural Information Processing Systems*. 3111–3119.

[39] Yuji Mizobuchi and Kuniharu Takayama. 2017. Two improvements to detect duplicates in stack overflow. In *Proceedings of the IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 563–564.

[40] Laura Moreno, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Andrian Marcus. 2015. How can I use this method? In *Proceedings of the IEEE/ACM 37th IEEE International Conference on Software Engineering*. IEEE, 880–890.

[41] Liming Nie, He Jiang, Zhilei Ren, Zeyi Sun, and Xiaochen Li. 2016. Query expansion based on crowd knowledge for code search. *IEEE Trans. Serv. Comput.* 9, 5 (2016), 771–783.

[42] Liqiang Nie, Xiaochi Wei, Dongxiang Zhang, Xiang Wang, Zhipeng Gao, and Yi Yang. 2017. Data-driven answer selection in community QA systems. *IEEE Trans. Knowl. Data Eng.* 29, 6 (2017), 1186–1198.

[43] Aditya Pal, F. Maxwell Harper, and Joseph A. Konstan. 2012. Exploring question selection bias to identify experts and potential experts in community question answering. *ACM Trans. Inf. Syst.* 30, 2 (2012), 1–28.

[44] Luca Ponzanelli, Gabriele Bavota, Andrea Mocci, Massimiliano Di Penta, Rocco Oliveto, Barbara Russo, Sonia Haiduc, and Michele Lanza. 2016. CodeTube: Extracting relevant fragments from software development video tutorials. In *Proceedings of the IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 645–648.

[45] Luca Ponzanelli, Andrea Mocci, Alberto Bacchelli, Michele Lanza, and David Fullerton. 2014. Improving low quality stack overflow post detection. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution*. IEEE, 541–544.

[46] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. 2018. Improving language understanding by generative pre-training, 1–12. Available: https://www.cs.ubc.ca/~amuham01/LING530/papers/radford2018improving.pdf.

[47] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2019. Exploring the limits of transfer learning with a unified text-to-text transformer. *arXiv preprint arXiv:1910.10683* (2019).

[48] Md Masudur Rahman, Jed Barson, Sydney Paul, Joshua Kayani, Federico Andrés Lois, Sebastián Fernandez Quezada, Christopher Parnin, Kathryn T. Stolee, and Baishakhi Ray. 2018. Evaluating how developers use general-purpose web-search for code retrieval. In *Proceedings of the 15th International Conference on Mining Software Repositories*. 465–475.

[49] Mohammad Masudur Rahman and Chanchal Roy. 2018. Effective reformulation of query for code search using crowdsourced knowledge and extra-large data analytics. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 473–484.

[50] Mohammad M. Rahman, Chanchal K. Roy, and David Lo. 2019. Automatic query reformulation for code search using crowdsourced knowledge. *Empir. Softw. Eng.* 24, 4 (2019), 1869–1924.

[51] Radim Řehůřek and Petr Sojka. 2010. Software framework for topic modelling with large corpora. In *Proceedings of the LREC Workshop on New Challenges for NLP Frameworks*. ELRA, 45–50. Retrieved from http://is.muni.cz/publication/884893/en.

[52] Christoffer Rosen and Emad Shihab. 2016. What are mobile developers asking about? A large scale study using stack overflow. *Empir. Softw. Eng.* 21, 3 (2016), 1192–1223.
[53] Saksham Sachdev, Hongyu Li, Sifei Luan, Seohyun Kim, Koushik Sen, and Satish Chandra. 2018. Retrieval on source code: A neural code search. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. 31–41.
[54] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. 2019. DistilBERT, a distilled version of BERT: Smaller, faster, cheaper and lighter. *arXiv preprint arXiv:1910.01108* (2019).
[55] David Shepherd, Zachary P. Fry, Emily Hill, Lori Pollock, and K. Vijay-Shanker. 2007. Using natural language program analysis to locate and understand action-oriented concerns. In *Proceedings of the 6th International Conference on Aspect-oriented Software Development*. 212–224.
[56] Rodrigo F. G. Silva, Klérisson Paixão, and Marcelo de Almeida Maia. 2018. Duplicate question detection in stack overflow: A reproducibility study. In *Proceedings of the IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 572–581.
[57] Priyanka Singh and Elena Simperl. 2016. Using semantics to search answers for unanswered questions in Q&A forums. In *Proceedings of the 25th International Conference Companion on World Wide Web*. 699–706.
[58] Raphael Sirres, Tegawendé F. Bissyandé, Dongsun Kim, David Lo, Jacques Klein, Kisub Kim, and Yves Le Traon. 2018. Augmenting and structuring user queries to support efficient free-form code search. *Empir. Softw. Eng.* 23, 5 (2018), 2622–2654.
[59] Hongya Song, Zhaochun Ren, Shangsong Liang, Piji Li, Jun Ma, and Maarten de Rijke. 2017. Summarizing answers in non-factoid community question-answering. In *Proceedings of the 10th ACM International Conference on Web Search and Data Mining*. 405–414.
[60] Ivan Srba and Maria Bielikova. 2016. Why is stack overflow failing? Preserving sustainability in community question answering. *IEEE Softw.* 33, 4 (2016), 80–89.
[61] Yuan Tian, Pavneet Singh Kochhar, Ee-Peng Lim, Feida Zhu, and David Lo. 2013. Predicting best answerers for new questions: An approach leveraging topic modeling and collaborative voting. In *Proceedings of the International Conference on Social Informatics*. Springer, 55–68.
[62] László Tóth, Balázs Nagy, Dávid Janthó, László Vidács, and Tibor Gyimóthy. 2019. Towards an accurate prediction of the question quality on stack overflow using a deep-learning-based NLP approach. In *Proceedings of the International Conference on Software Technologies*. 631–639.
[63] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems*. 5998–6008.
[64] Kai Wang, Zhaoyan Ming, and Tat-Seng Chua. 2009. A syntactic tree matching approach to finding similar questions in community-based QA services. In *Proceedings of the 32nd International ACM SIGIR Conference on Research and Development in Information Retrieval*. 187–194.
[65] Liting Wang, Li Zhang, and Jing Jiang. 2020. Duplicate question detection with deep learning in stack overflow. *IEEE Access* 8 (2020), 25964–25975.
[66] Xin-Yu Wang, Xin Xia, and David Lo. 2015. TagCombine: Recommending tags to contents in software information sites. *J. Comput. Sci. Technol.* 30, 5 (2015), 1017–1035.
[67] Frank Wilcoxon. 1992. Individual comparisons by ranking methods. In *Breakthroughs in Statistics*. Springer, 196–202.
[68] Xin Xia, Lingfeng Bao, David Lo, Pavneet Singh Kochhar, Ahmed E. Hassan, and Zhenchang Xing. 2017. What do developers search for on the web? *Empir. Softw. Eng.* 22, 6 (2017), 3149–3185.
[69] Bowen Xu, Zhenchang Xing, Xin Xia, and David Lo. 2017. AnswerBot: Automated generation of answer summary to developers' technical questions. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 706–716.
[70] Bowen Xu, Zhenchang Xing, Xin Xia, David Lo, and Shanping Li. 2018. Domain-specific cross-language relevant question retrieval. *Empir. Softw. Eng.* 23, 2 (2018), 1084–1122.
[71] Xinli Yang, David Lo, Xin Xia, Lingfeng Bao, and Jianling Sun. 2016. Combining word embedding with information retrieval to recommend similar bug reports. In *Proceedings of the IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 127–137.
[72] Xin-Li Yang, David Lo, Xin Xia, Zhi-Yuan Wan, and Jian-Ling Sun. 2016. What security questions do developers ask? A large-scale study of stack overflow posts. *J. Comput. Sci. Technol.* 31, 5 (2016), 910–924.
[73] Ting Ye, Bing Xie, Yanzhen Zou, and Xiuzhao Chen. 2014. Interrogative-guided re-ranking for question-oriented software text retrieval. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*. 115–120.

[74] Xin Ye, Hui Shen, Xiao Ma, Razvan Bunescu, and Chang Liu. 2016. From word embeddings to document similarities for improved information retrieval in software engineering. In *Proceedings of the 38th International Conference on Software Engineering*. 404–415.

[75] Wei Emma Zhang, Quan Z. Sheng, Jey Han Lau, and Ermyas Abebe. 2017. Detecting duplicate posts in programming QA communities via latent semantics and association rules. In *Proceedings of the 26th International Conference on World Wide Web*. 1221–1229.

[76] Yun Zhang, David Lo, Xin Xia, and Jian-Ling Sun. 2015. Multi-factor duplicate question detection in Stack Overflow. *J. Comput. Sci. Technol.* 30, 5 (2015), 981–997.

[77] P. Zhou, J. Liu, X. Liu, Z. Yang, and John C. Grundy. 2019. Is deep learning better than traditional approaches in tag recommendation for software information sites? *Inf. Softw. Technol.* 109 (2019), 1–13.

[78] Yanzhen Zou, Ting Ye, Yangyang Lu, John Mylopoulos, and Lu Zhang. 2015. Learning to rank for question-oriented software text retrieval (T). In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1–11.

Received 9 November 2020; revised 4 June 2022; accepted 15 July 2022