The Journal of Systems and Software 210 (2024) 111948

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Query-oriented two-stage attention-based model for code search ☆

Huanhuan Yang [a,b], Ling Xu [a,b,*], Chao Liu [a,b], Luwen Huangfu [c,d]

[a] Key Laboratory of Dependable Service Computing in Cyber Physical Society (Chongqing University), Ministry of Education, China
[b] School of Big Data and Software Engineering, Chongqing University, Chongqing, China
[c] Fowler College of Business, San Diego State University, CA, 92182, USA
[d] Center for Human Dynamics in the Mobile Age, San Diego State University, CA, 92182, USA

ARTICLE INFO

Keywords:
Code search
Attention mechanism
Query-oriented attention mechanism
Code structural feature

ABSTRACT

Applying code search models to search through a large-scale codebase can significantly contribute to developers finding and reusing existing code. Researchers have applied deep learning (DL) techniques to code search models, which first compute deeper semantics representation for query and candidate code snippets, and then rank code snippets. However, these models do not well deeply analyze the semantics gap (i.e., the difference and correlation between queries written in natural language and code in programming languages), or suitably apply the correlation to the code search task. Moreover, most DL-based models use complex networks, slowing down code search tasks.

To build the correlation of two languages, and apply the correlation well to code search task, we propose a query-oriented code search model named QobCS. QobCS leverage two attention-based stages, which are simple and quick, and the cooperation of the two stages bridges the semantic gap between code and query. Stage1 learns deeper semantics representation for code and query. Stage2 applies their deeper semantic correlation and query's intention to learn better code representation.

We evaluated QobCS on two datasets. On dataset1/dataset2 with 485k/542k code snippets, QobCS achieves the MRRs of 0.701/0.595, outperforming DL-based code search models DeepCS, CARLCS-CNN, UNIF, and our prior study TabCS. For efficiency, our model shows desirable performances on both datasets compared to DL-based models.

1. Introduction

Reusing existing code from open-source communities can improve the efficiency of software development. Researchers have used deep learning (DL) techniques to search the code that meets the intention of the developers from the existing large-scale code base. DL-based code search models generally independently obtained the embedding of query and code, whose cosine similarity is used to sort candidate codes. For example, Gu et al. (2018) proposed DeepCS, which uses the LSTM (long and short-term memory) model (Sundermeyer et al., 2012) for embedding. Cambronero et al. (2019) proposed UNIF, which uses an attention layer to embed code, showing good performance on efficiency.

To more accurately search for the corresponding code, researchers proposed leveraging joint networks to explore the correlation between the query written in natural language and code in the programming language. Here, we give an example to explain the semantic correlation. The natural word–"word" can be represented in the programming language by its abbreviation (i.e., "w"), specific class name (i.e., "string" and 'stringbuilder"), as well as the class name's abbreviation (i.e., "str", and "sb"). The key to bridging the semantic gap is to learn that these programming words mentioned above share the same semantics with "word", although they are far different from "word" in spelling.

Shuai et al. (2020) proposed CARLCS-CNN, which performs convolutional neural networks (CNN) followed by the co-attention mechanism. The co-attention mechanism is suitable for matching different words with the same semantics. However, the previous CNN integrates the semantics of many words together, which renders the following co-attention mechanism unable to build the semantic correlation between two single words. For example, we expect the co-attention mechanism to learn the semantics correlation between the natural word "word" and the programming word"str". However, the previous CNN modules feed the fused semantic representations of these words and their adjacent words to the following co-attention mechanism, instead of clear representations of "word" and "str". Therefore, compared with complex networks like CNN, the attention mechanism is more suitable because

☆ Editor: Nicole Novielli.

* Corresponding author at: Key Laboratory of Dependable Service Computing in Cyber Physical Society (Chongqing University), Ministry of Education, China.
E-mail addresses: yanghh@cqu.edu.cn (H. Yang), xuling@cqu.edu.cn (L. Xu), liu.chao@cqu.edu.cn (C. Liu), lhuangfu@sdsu.edu (L. Huangfu).

https://doi.org/10.1016/j.jss.2023.111948
Received 7 July 2022; Received in revised form 23 November 2023; Accepted 27 December 2023
Available online 3 January 2024

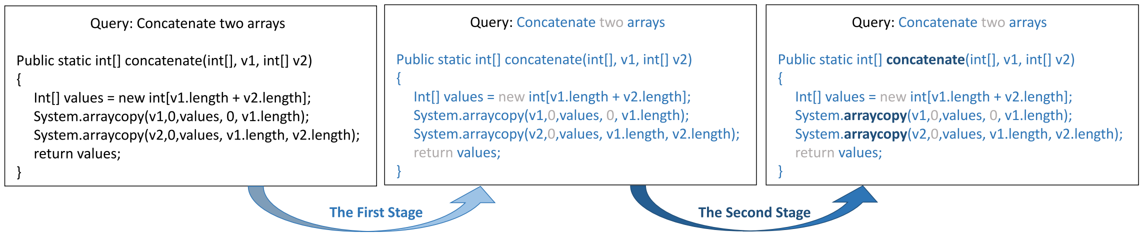


Fig. 1. The attentions computed by two stages of QobCS.

it only assigns weight to a single word, and does not mix the semantics of multiple words.

Moreover, although the co-attention mechanism shows good performance in building the correlation between natural language and programming language, it is not completely suitable to the code search task. Co-attention mechanism could filter out unrelated information between code and query, but the unrelated information may be the important semantics in the query's intention. Missing important intentions in the query can be fatal for code search, because code search depends largely on the semantics of the query.

Therefore, we proposed an idea that a code search model should first extract the query's semantics, and then compute the code representation according to two aspects: (1) the semantics of the query; (2) the correlation between natural language and programming language. In this way, the code representation could contain the information related to the query, and the query representation contains complete intentions. Finally, the similarity measure of the query representation and code representation is suitable and reasonable for the code search task.

In this paper, we propose QobCS — a query-oriented code search model. It is implemented by two simple and quick attention-based stages; The first stage applies attention mechanisms to independently learn deeper semantics representation for code and query; The following stage applies query-oriented attention mechanism, which builds the correlation of two languages (i.e., the natural language and the programming language) and leverages the query's semantics to compute code representation.

Firstly, we extract three textual features (i.e., the method name, API, and tokens) and a structural feature (i.e., AST sequence) to represent the code. Independent attention mechanisms in the first stage pay greater attention to words that represent the code semantics and query's intention, and less attention to meaningless words. As illustrated in Fig. 1, words that frequently appear in either programming language or natural language (e.g., "two", "new", and "return") are assigned with fewer weights, and meaningful words (e.g., "concatenate", "arrays", "values", "system", and "length") higher weights. Moreover, the second stage leverages independent query-oriented attention mechanisms, which find out and assign greater weights to code words related to the query's intention. Therefore, only the core point for achieving the query's intention is preserved in the final code representation. In Fig. 1, the code words ("concatenate" and "arraycopy") are highly related to the query's intention ("concatenate two arrays"), and thus receive greater weights. Unrelated words ("system", "values", and "length") receive small weights although they are meaningful.

This paper extends our preliminary study, which appeared as a conference paper in SANER 2021 (Xu et al., 2021). In particular, we extend our preliminary work in the following ways:

1. We propose QobCS — an extended version of TabCS proposed in our preliminary work (Xu et al., 2021). There are three major differences between QobCS and TabCS: (1) In TabCS, the second stage applies a co-attention mechanism. In QobCS, we apply query-oriented attention mechanisms. (2) In TabCS, in the second stage, we directly fuse four code feature matrices into a matrix, and feed it to a co-attention mechanism. In QobCS, the second stage fuses four features after applying four independent query-oriented attention mechanisms to them. (3) In TabCS, for a query, the final query representation changes with the different candidate code. In QobCS, the final query representation never changes for the same query. Our experiments show that QobCS outperforms the TabCS.
2. We further conduct experiments and discuss how the query-oriented attention mechanism performs on real queries and pre-trained models.
3. We conduct experiments to explore how the margin of loss function influences QobCS's performance.

We evaluated QobCS on two datasets. On dataset1/dataset2 with 485k/542k code snippets, QobCS achieves the MRRs of 0.701/0.595, outperforming four code search models DeepCS (Gu et al., 2018), CARLCS-CNN (Shuai et al., 2020), UNIF (Cambronero et al., 2019), and our prior study TabCS by 109.25%/93.81%, 44.54%/45.48%, 35.85%/42.00%, and 24.73%/15.98%, respectively. For efficiency, on both datasets, our model shows better performances than compared DL-based models.

In summary, the contributions of this study are as follows:

- Proposing a query-oriented two-stage attention-based model QobCS, which leverages attention mechanisms followed by query-oriented attention mechanisms on textual and structural features of code to achieve code search task.
- Evaluating the performance of QobCS as well as the compared code search models on two existing large-scale datasets and 50 real queries, and the experimental results demonstrate that QobCS performs better than compared models.
- Making our implementation and the dataset used available.[1]

2. Background

We introduce the background of the code search task from three aspects — the code embedding, the query embedding, and DL-based models in code search practice.

2.1. Code embedding

In DL-based code search models, three textual features were widely used: method name, API sequence, and code tokens. The three features generally were embedded by networks and finally were fused together to represent the code.

First, the words in a feature would be mapped to vectors with the same dimension. Next, concatenate these vectors into a matrix to represent the feature. Then, each feature matrix was processed by a neural network model. Finally, the outputs of three features are fused into a final representative vector. In DeepCS, as in Eq. (1), the code features were processed by two LSTM models and a common multilayer

[1] http://tinyurl.com/4hddsydf

perceptron (MLP) (Gardner, 1998), respectively. Finally, three features were mapped to the same space with the query by a MLP layer. CARLCS-CNN leverages an LSTM model and two CNN models to embed these features. Then they concatenated together for the next module.

$$\boldsymbol{v}_c = M(L_1(F_{name}) + L_2(F_{API}) + M(F_{tokens})) \tag{1}$$

$$V_c = C_1(F_{name}) \oplus C_2(F_{tokens}) \oplus L(F_{API}) \tag{2}$$

Where L, M, and C represent the LSTM, MLP, and CNN modules, respectively. F_{name}, F_{API} and F_{tokens} represent the features mentioned above, respectively.

2.2. Query embedding

Most DL-based code search models took the tokens as the query features. The query tokens are first mapped into vectors. Then, a neural network is applied to process these vectors. DeepCS and CARLCS-CNN use a bidirectional LSTM and a CNN module, respectively.

$$\boldsymbol{v}_q = BiLSTM(F_{query}) \tag{3}$$

$$V_q = C_3(F_{query}) \tag{4}$$

Where BiLSTM and C represent the LSTM, and CNN modules, respectively. F_{query} represents query tokens.

2.3. Deep learning for code search

DL-based models complete the code search task by learning code representation, query representation, and their similarity scores. Some proposed DL-based models apply independent networks to compute a representation for a query or code. The input of these models is not necessarily a code–query pair, because code and query representation calculation can run independently. For these models, the representations of all codes in the code base are pre-computed and stored before given a query. Therefore, for these models, the code search steps are as follows:

(a) First, feed all code into the trained model, obtaining a representative vector for each candidate code.
(b) Next, for a query, obtain the representative vector for the query.
(c) Compute the similarity scores of the query's vector with all candidate code's vectors.
(d) Finally, rank all candidate codes according to their similarity score, getting a recommendation list for the given query.

To further bridge the semantics gap between code and query, researchers have taken account of the potential correlation between natural language and programming language. Therefore, they have applied joint learning networks, such as a co-attention mechanism and its variants, to compute the representation of the code and query. Different from the independent network-based model, the input of joint network-based models must be a code–query pair, because these models compute their representation according to the correlation of the pair. Pairing a code with any query and feeding the pair to these models, the output code representation changes with the query, which is the same as the query. Variability of representation changes the process of code search. For these models, the code search steps are as follows:

(a) At first, given a query, set candidate code–query pairs for each code in the code base.
(b) Then, for each candidate pair, use a trained model to compute their representative vectors.
(c) Compute the similarity score of each candidate representative vector pair.
(d) Finally, rank all candidate codes according to their similarity score, obtaining a recommendation list for the query.

Generally, when obtaining the representative vectors of code and query, DL-based models take cosine similarity as their similarity score, computed as in Eq. (5).

$$sim(\boldsymbol{v}_c, \boldsymbol{v}_q) = \frac{\boldsymbol{v}_c \cdot \boldsymbol{v}_q}{\|\boldsymbol{v}_c\| \cdot \|\boldsymbol{v}_q\|} \tag{5}$$

Cosine similarity is the cosine value of an angle. It can be used to measure the similarity between two vectors in the same vector space model. The theoretical value range is [−1, 1]. Code search models generally represent the correlation between a code–query pair by the similarity score. The higher the correlation is, the more likely the code is to be correct for the query.

3. Model

3.1. Overall framework

The overall framework of QobCS is depicted in Fig. 2, showing how the code search is implemented. The first stage feeds three code textual features, a code structural feature, and a query feature into five independent attention mechanisms, obtaining five feature matrices. The following stage computes the final representative vectors and similarity scores following four steps: (1) It firstly feeds the query feature matrix into an average-pooling layer, directly obtaining the final query representative vector; (2) Then, it feeds the query representative vector and code feature matrices into four independent query-oriented attention mechanisms, obtaining code feature representative vectors; (3) Next, it fuses the four vectors into the final code representative vector; (4) Finally, it computes the cosine similarity, as the similarity score of code and query. In the rest of this paper, we use Stage1 and Stage2 refer to the first stage and the second stage of QobCS.

3.2. Stage1

Stage1 extracts the independent semantics of code and query, obtaining five feature matrices for four code features and a query feature.

3.2.1. Code embedding

We leverage four attention mechanisms on three code textural features, and a code structural feature.

Step-1: Textual Feature Embedding. For a textual feature (e.g., tokens), we build a matrix $E \in \mathbb{R}^{o \times K}$ for initial embedding. The matrix contains o K-dimensional vectors corresponding o words in the feature vocabulary. The initial value of the embedded matrix is a randomly generated uniform distribution from −1 to 1. For every word, this matrix could map every word from its id to a unique embedded vector, thus obtaining an initial feature matrix as follow:

$$[\boldsymbol{w}_1, \ldots, \boldsymbol{w}_n] = embed(\{id_1, id_2, \ldots, id_n\}, E) \tag{6}$$

Then we leverage an independent attention mechanism on the initial feature matrix, to obtain a deeper representation of a code feature. The attention mechanism extracts the semantics of a feature by assigning different weights to different words. A word frequently used in code snippets tend to obtain fewer weights to its embedded vector. Taking the code in Fig. 2 as an example, "is", "if", "return", "false", and "true" frequently exist in both positive code and negative code for any query. Therefore, these words cannot represent the specific semantics of a method. During model training, the attention mechanism learns that these words are useless for any query, so these words with high frequency would be judged as unimportant. On the contrary, the words with low frequency tend to be more helpful in representing the method's functionality, such as "last", "Modified", and "file" in Fig. 2. Therefore, they are assigned greater weights by the attention mechanism.

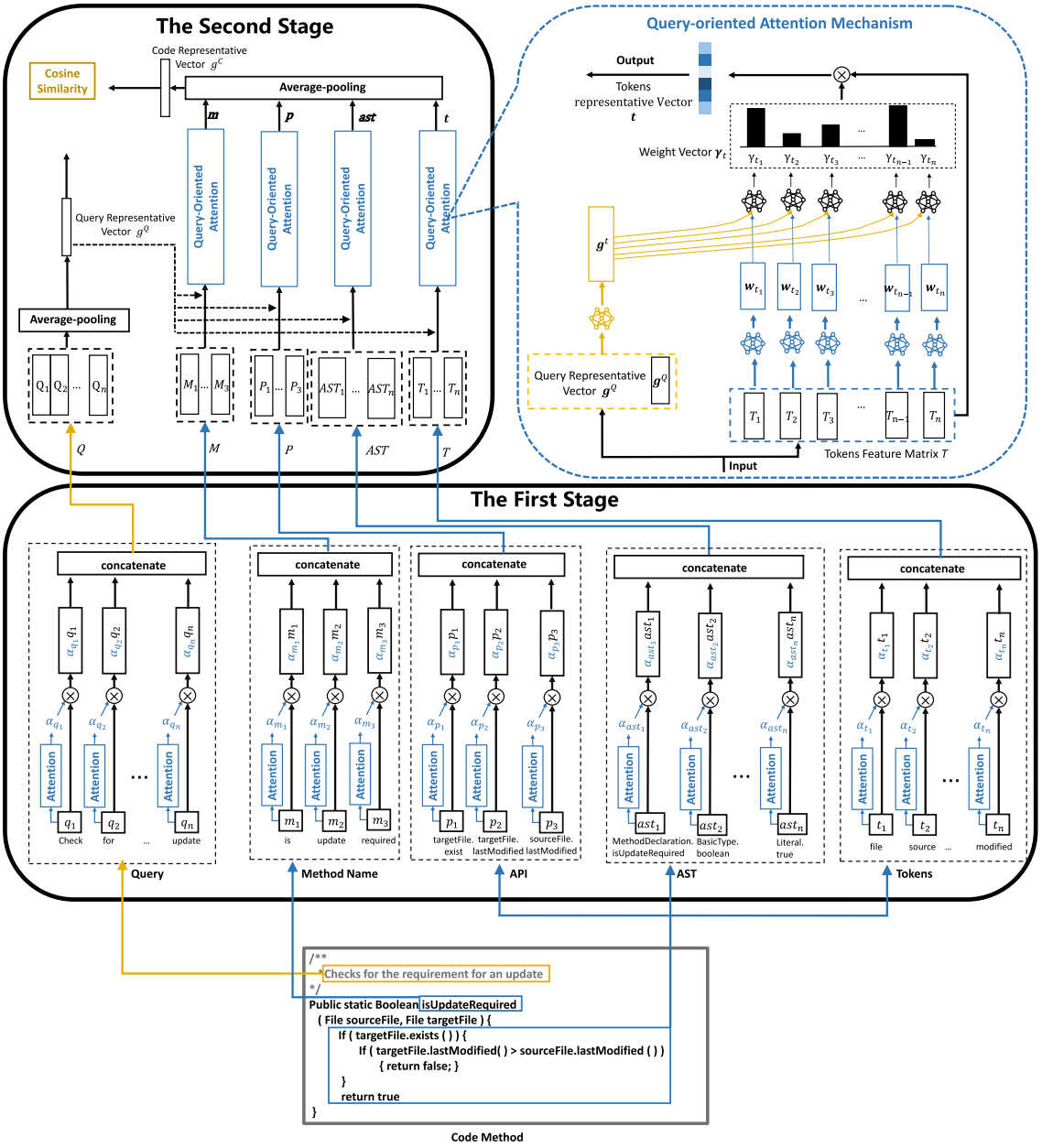


Fig. 2. Overall Framework of QobCS.

Method Name. For the method name code feature, let $\{ id_{m_1}, id_{m_2}, \ldots, id_{m_n} \}$ be a sequence of ids for a method name. We first map the sequence to an initial feature matrix:

$$[\boldsymbol{m}_1, \ldots, \boldsymbol{m}_n] = embed(\{id_{m_1}, id_{m_2}, \ldots, id_{m_n}\}, E^m) \tag{7}$$

where $E^m \in \mathbb{R}^{o^m \times k}$ is the embedding matrix of method name, o^m is the size of the method name vocabulary, and $\boldsymbol{m}_i \in \mathbb{R}^K$ is a k-dimensional word initial vector corresponding to the ith word in a method name.

Then, we feed the method name initial feature matrix into an attention mechanism:

$$\alpha_{m_i} = \frac{exp(\boldsymbol{a}_m \cdot \boldsymbol{m}_i^T)}{\sum_{i=1}^{n} exp(\boldsymbol{a}_m \cdot \boldsymbol{m}_i^T)} \tag{8}$$

Where the α_{m_i} is the weight assigned to the word vector $\boldsymbol{m}_i$. $\boldsymbol{a}_m \in \mathbb{R}^k$ is a trainable attention vector. During model training, $\boldsymbol{a}_m$ is optimized to determine whether the word is important and meaningful. We leverage the softmax layer to limit the value of each weight between 0 and 1, as well as their sum being equal to 1.

Then, we compute the weighted vectors and concatenate them. $M \in \mathbb{R}^{k \times n}$ represents the method name feature matrix, computed as in Eq. (9):

$$M = \left[\alpha_{m_1}\boldsymbol{m}_1, \alpha_{m_2}\boldsymbol{m}_2, \ldots, \alpha_{m_n}\boldsymbol{m}_n\right] \tag{9}$$

Therefore, the feature matrix preserves the main semantics and filters out some noise in the method name. Then, it would be fed into Stage2.

API Sequence. For API sequence, let $\{ id_{p_1}, id_{p_2}, \ldots, id_{p_n} \}$ be a sequence of ids. We map the ids into vectors as Eq. (10), and leverage an attention mechanism as Eqs. (11)–(12). Finally, we obtain the feature matrix $P \in \mathbb{R}^{k \times n}$, which preserves important API information.

$$[\boldsymbol{p}_1, \ldots, \boldsymbol{p}_n] = embed(\{id_{p_1}, id_{p_2}, \ldots, id_{p_n}\}, E^p) \tag{10}$$

$$\alpha_{p_i} = \frac{exp(\boldsymbol{a}_p \cdot \boldsymbol{p}_i^T)}{\sum_{i=1}^{n} exp(\boldsymbol{a}_p \cdot \boldsymbol{p}_i^T)} \tag{11}$$

$$P = \left[\alpha_{p_1}\boldsymbol{p}_1, \alpha_{p_2}\boldsymbol{p}_2, \ldots, \alpha_{p_n}\boldsymbol{p}_n\right] \tag{12}$$

Tokens. For code tokens, let $\{id_{t_1}, id_{t_2}, \ldots, id_{t_n}\}$ be a sequences of ids. We map the ids to vectors as Eq. (13), and leverage an attention mechanism as Eqs. (14)–(15). The final output $T \in \mathbb{R}^{k\times n}$ preserves meaningful and important information and filters out unimportant and meaningless words.

$$[\boldsymbol{t}_1, \ldots, \boldsymbol{t}_n] = embed(\{id_{t_1}, id_{t_2}, \ldots, id_{t_n}\}, E^t) \tag{13}$$

$$\alpha_{t_i} = \frac{exp(\boldsymbol{a}_t \cdot \boldsymbol{t}^T)}{\sum_{i=1}^{n} exp(\boldsymbol{a}_t \cdot \boldsymbol{t}_i^T)} \tag{14}$$

$$T = \left[\alpha_{t_1}\boldsymbol{t}_1, \alpha_{t_2}\boldsymbol{t}_2, \ldots, \alpha_{t_n}\boldsymbol{t}_n\right] \tag{15}$$

Step-2: Structural Feature Embedding. We leverage the abstract syntax tree (AST) to represent the structure of a code. AST is a syntax tree. Every node in the AST represents the type of a line or a keyword in the code snippet. For example, a code line with "for" loop, "if" or a method calling statement would be transformed to a node of loop structure type, conditional judgment structure type, or method call type, respectively. We traverse the AST to obtain all nodes. While some nodes reflect the method's function, others do not. Taking the code in Fig. 2 for example, the node "IfStatement" is a commonly used node in various code snippets and does not contains the semantics of the code, whereas the node "MethodDeclaration" tend to represent the method's functionality. Therefore, for these nodes, we perform an embedding layer and an attention mechanism to get a deeper representation of the code structure.

Firstly, we build vocabulary and embedding matrix for AST. Then, we map the nodes into initial vector embedding. Next, we leverage an attention mechanism to obtain an AST feature matrix, which represents the method's structural information. Given n nodes $\{id_{ast_1}, id_{ast_2}, \ldots, id_{ast_n}\}$, we perform the mapping followed with an attention mechanism as Eqs. (16)–(18), obtaining the feature matrix $AST \in \mathbb{R}^{k\times n}$.

$$[\boldsymbol{ast}_1, \ldots, \boldsymbol{ast}_n] = embed(\{id_{ast_1}, id_{ast_2}, \ldots, id_{ast_n}\}, E^{ast}) \tag{16}$$

$$\alpha_{ast_i} = \frac{exp(\boldsymbol{a}_{ast} \cdot \boldsymbol{ast}_i^T)}{\sum_{i=1}^{n} exp(\boldsymbol{a}_{ast} \cdot \boldsymbol{ast}_i^T)} \tag{17}$$

$$AST = \left[\alpha_{ast_1}\boldsymbol{ast}_1, \alpha_{ast_2}\boldsymbol{ast}_2, \ldots, \alpha_{ast_n}\boldsymbol{ast}_n\right] \tag{18}$$

3.2.2. Query embedding

Query describes the function or intention of the target code snippet. To extract a deeper representation of the query, we first split a query into tokens. Like code tokens, a word frequently used in queries, is not helpful to represent the key semantics, like "how", "to", and "a". Therefore, after embedding words, we leverage an attention mechanism to filter out some useless words. Words with high frequency are assigned with fewer attention weights, while words that tend to represent the intention of the developers are assigned with higher weights. Finally, we obtain the query feature matrix by concatenating the weighted vectors. Given n tokens $\{ id_{q_1}, id_{q_2}, \ldots, id_{q_n} \}$, we perform the mapping followed with an attention mechanism as Eqs. (20)–(21), obtaining the feature matrix $Q \in \mathbb{R}^{k\times n}$.

$$[\boldsymbol{q}_1, \ldots, \boldsymbol{q}_n] = embed(\{id_{q_1}, id_{q_2}, \ldots, id_{q_n}\}, E^q) \tag{19}$$

$$\alpha_{q_i} = \frac{exp(\boldsymbol{a}_q \cdot \boldsymbol{q}_i^T)}{\sum_{i=1}^{n} exp(\boldsymbol{a}_q \cdot \boldsymbol{q}_i^T)} \tag{20}$$

$$Q = \left[\alpha_{q_1}\boldsymbol{q}_1, \alpha_{q_2}\boldsymbol{q}_2, \ldots, \alpha_{q_n}\boldsymbol{q}_n\right] \tag{21}$$

3.3. Stage2

In Stage1, we obtain the five feature matrices $M, P, T, AST, and Q$. These matrices have filtered out unimportant and meaningless information. Then, we feed them into Stage2, which builds the semantics correlation between natural language and programming language, and leverages the correlation as well as the query representation to identify code words related to the query's intention, and finally, outputs the code representative vectors.

3.3.1. Query representation

Firstly, we perform a simple mean-pooling layer on the query feature matrix Q, obtaining the final query representative vector $\boldsymbol{g}^Q \in \mathbb{R}^k$:

$$\boldsymbol{g}^Q = MeanPooling(Q_1, \ldots, Q_q) \tag{22}$$

where Q_i represents the i_{th} vector in feature matrix.

Then, to learn the correlation between query vector $\boldsymbol{g}^Q$ and code feature matrices M, P, T, AST, we apply four independent query-oriented attention mechanisms to them. Based on query-oriented attention mechanisms, the code words which are related to the query vector are largely weighted, and unrelated information in code feature matrices is filtered out. In particular, query-oriented attention mechanism only pays attention to the code feature matrix and computes the final code feature representation, never changing any information in the query.

Method Name. For the method name, let M_i as the i_{th} vector in M output by Stage1. Firstly, to obtain the deeper representation of the query and the i_{th} word in the method name, we apply two full collection layers on M_i and $\boldsymbol{g}^Q$, respectively:

$$\boldsymbol{g}^m = G_m g^Q, \quad \boldsymbol{w}_{mi} = W_m M_i \tag{23}$$

where $G_m \in \mathbb{R}^{k\times k}$ and $W_m \in \mathbb{R}^{k\times k}$ are parameters.

Then, to learn the correlation between $\boldsymbol{g}^m$ and $\boldsymbol{w}_{m_i}$, we compute a union vector of them as follow:

$$\boldsymbol{u}_{m_i} = tanh(\boldsymbol{g}^m + \boldsymbol{w}_{mi}) \tag{24}$$

Where $\boldsymbol{u}_{m_i} \in \mathbb{R}^k$ represents the union representation of query and the i_{th} word in method name. $tanh$ activation layer is to make sure each element of the $\boldsymbol{u}_{m_i}$ in the range of [−1,1].

Since a method name has n words, for vectors$\{M_1, M_2, \ldots, M_n\}$, following above steps, we get n union vectors $\{\boldsymbol{u}_{m_1}, \boldsymbol{u}_{m_2}, \ldots, \boldsymbol{u}_{m_n}\}$. Then, the correlation weight γ_{m_i} for each $\boldsymbol{u}_{m_i}$ is computed as follows:

$$\gamma_{m_i} = \frac{exp(\boldsymbol{a}_m \cdot \boldsymbol{u}_{m_i}^T)}{\sum_{i=1}^{n} exp(\boldsymbol{a}_m \cdot \boldsymbol{u}_{m_i}^T)} \tag{25}$$

Where the correlation vector $\boldsymbol{a}_m \in \mathbb{R}^k$ is a k-dimensional parameter vector. It extracts correlation from union vectors and learns to judge whether the corresponding method name words are related to the query. We use the $softmax$ function to limit the weights in range [0,1], and their sum to be one.

Next, we leverage the weights on the corresponding vectors, and add the weighted vector together, obtaining the final representation for method name $\boldsymbol{m} \in \mathbb{R}^k$:

$$\gamma_m = \left[\gamma_{m_1}, \gamma_{m_2}, \ldots, \gamma_{m_n}\right] \tag{26}$$

$$\boldsymbol{m} = M\gamma_m \tag{27}$$

The information preserved by $\boldsymbol{m}$ is not only meaningful and important, but also related to the query, because query-oriented attention mechanism assigns greater weights to these information which may have a potential correlation to the query. Therefore, for the code search task, compared with M, $\boldsymbol{m}$ can represent the method name semantics better.

For short, we use QOM to refer to a query-oriented attention mechanism (i.e., Eqs. (23)–(27)). Therefore, Eqs. (23)–(27) can be represented as:

$$\boldsymbol{m} = QOM_m(M, \boldsymbol{g}^Q) \tag{28}$$

where QOM_m represents an independent query-oriented attention mechanism for the method name. Its inputs are the method name feature matrix and a query vector; the output is the final method name representative vector.

API Sequence. In QobCS, we take four independent query-oriented attention mechanisms on four code feature matrices, since we think different code features have a unique potential correlation to the query. For API sequence feature matrix P, we feed it and query vector $\boldsymbol{g}^Q$ into an independent query-oriented attention mechanism QOM_p, and obtain the final API sequence representative vector $\boldsymbol{p} \in \mathbb{R}^k$:

$$\boldsymbol{p} = QOM_p(P, \boldsymbol{g}^Q) \tag{29}$$

Considering the query semantics, the query-oriented attention mechanism learns whether an API function is helpful for the implementation of the query. Therefore, the output $\boldsymbol{p}$ only preserves the API sequences, which may have a potential correlation to the query, and filters out unrelated ones.

Tokens. Tokens contain many words used in a method function. However, not every word contributes to the key goal of the code, and perhaps only part of the code corresponds to the goal of the query. Therefore, we use query-oriented attention mechanism to extract the most related part in the code. For Tokens feature matrix T, we feed it and query vector $\boldsymbol{g}^Q$ into an independent query-oriented attention mechanism QOM_t, and obtain the final tokens representative vector $\boldsymbol{t} \in \mathbb{R}^k$:

$$\boldsymbol{t} = QOM_t(T, \boldsymbol{g}^Q) \tag{30}$$

AST Sequence. AST represents the logical structure of a code. We think that perhaps only part of the structure contributes to the goal of the query. For AST sequence, feature matrix AST, we feed it and query vector $\boldsymbol{g}^Q$ into an independent query-oriented attention mechanism QOM_{ast}, and obtain the final tokens representative vector $\boldsymbol{ast} \in \mathbb{R}^k$:

$$\boldsymbol{ast} = QOM_{ast}(AST, \boldsymbol{g}^Q) \tag{31}$$

Feature Fusion. To obtain the final code representative vector $g^C \in \mathbb{R}^k$, we concatenate code feature vectors $\boldsymbol{m}, \boldsymbol{p}, \boldsymbol{t}, \boldsymbol{ast}$ into a matrix C and perform max-pooling on it:

$$C \in \mathbb{R}^{k\times 4} = [\boldsymbol{m}, \boldsymbol{p}, \boldsymbol{t}, \boldsymbol{ast}] \tag{32}$$

$$\boldsymbol{g}^C = MeanPooling(C) \tag{33}$$

We test the attention mechanism, average-pooling layer, and max-pooling layer for fusion, and it turns out that average-pooling shows the best performance.

3.3.2. Similarity score

The final representative vectors g^Q and g^C are in the same vector space. To compute how related the code is to the query, we perform cosine similarity on them:

$$S(\boldsymbol{g}^C, \boldsymbol{g}^Q) = \frac{\boldsymbol{g}^C \cdot \boldsymbol{g}^Q}{\|\boldsymbol{g}^C\| \cdot \|\boldsymbol{g}^Q\|} \tag{34}$$

S represents the cosine similarity, which measures the similarity between two vectors by computing the cosine of the angle between them. Its theoretical value range is $[-1, 1]$. In our model, when the code corresponds to the query's intention, their cosine similarity tends to be greater.

3.4. Cooperation of two stages

This section introduces how QobCS's two attention-based stages can be jointed together and cooperate.

Stage1 is expected to obtain deeper representations of code and query, which contain the main semantics, and have filtered out unimportant words. The query feature matrix output by Stage1 is considered to represent the query's intention. In QobCS's Stage2, we firstly leverage a simple average-pooling layer to transform the matrix into the final query representative vector. The query-oriented attention mechanism's function is to further learn whether a word in code features is related to the query representative vector (i.e., query's intention) as in Eqs. (23)–(27).

C_i is the representation of i_{th} code word output by Stage1, g^Q is the final query representative vector. Before learning their correlation, two full collection layers are applied to obtain a deeper representation, and a union vector C_i'' as follows:

$$\boldsymbol{z}^c = G_c g^Q, \quad C_i' = W_c C_i \tag{35}$$

$$C_i'' = \boldsymbol{z}^c + C_i' \tag{36}$$

C_i'' contains the semantics of both the query's intention and the i_{th} code word. Because of the semantics gap between code and query, before simple addition operation as in Eq. (36), we use two full collection layers as Eq. (35). In the training process, parameter matrices G_c and W_c learn how to help fuse two vectors and obtain a better union vector. Therefore, C_i'' also contains the correlation of C_i and $\boldsymbol{g}^Q$.

The next step is to make use of the correlation to filter code words. In other words, we need to transform C_i'' into a weight γ_{c_i}:

$$\gamma'_{c_i} = C_i'' \cdot \boldsymbol{a}_c \tag{37}$$

$$\left[\gamma_{c_1}, \gamma_{c_2}, \ldots, \gamma_{c_n}\right] = softmax(\left[\gamma'_{c_1}, \gamma'_{c_2}, \ldots, \gamma'_{c_n}\right]) \tag{38}$$

$\boldsymbol{a}_c$ is a parameter vector. It learns to transform the union semantics of the query's intention and a code word, as well as their correlation into a weight γ'_{c_i}, whose value represents how the code word is related to the query. $\gamma'_{c_1}, \gamma'_{c_2}, \ldots, \gamma'_{c_n}$ represents the weights for $C_1, C_2, \ldots, C_n$. We perform softmax on these values to limit the weight from 0 to 1. Finally, C_i is weighted by γ_{c_i}. Therefore, if i_{th} code word is not used to achieve the query's intention, γ_{c_i} would be much less. Finally, the final code representation hardly contains the semantics of C_i. Conversely, if closely related, γ_{c_i} would be larger and C_i's semantics would have a large proportion in the final representation's semantics. Through assigning weights to the query's words in Stage1, QobCS learns the query representative vector. By the cooperation of two-stage, code words get two weights, which represent its relatively importance and the correlation to the query's intention, respectively. In this way, QobCS learns a code representative vector, which is very helpful in judging whether the code achieves a query's intention.

3.5. Model prediction for code search

Since QobCS applied joint networks (i.e., query-oriented attention mechanism), as mentioned in Session 2.3, for each candidate code–query pair, our model computes their representative vectors and similarity scores.

We test the performance of QobCS in a test codebase with n code snippets: given a query, we first get n candidate code–query pairs by pairing all code snippets with the query. We feed these pairs into our model, obtaining similarity scores for every pair. Then, they are sorted in descending order according to their similarity score. Finally, we select the top 10 code snippets to generate a recommendation list for the given query.

Table 1
Vocabulary capacity of code and query features for two datasets.

Dataset	Hu et al.'s dataset	Husain et al.'s dataset
method name	27 177	9895
API sequence	52 892	19 873
code tokens	46 145	14 247
AST	89	89
query tokens	30 001	12 318

3.6. Model optimization

For four code features and the query tokens, we build five vocabularies to map the words to ids. A feature vocabulary contains words with a frequency greater than 5 in a code feature or a query feature of the training set. Words with a frequency of less than 5 were filtered out. In this way, each word can be mapped to an id, and then mapped by our model into a unique vector.

Intuitively speaking, given a query, the corresponding code's representative vector should be close to the query's representative vector, while the other code's representative should be far away. So in practice, when we construct the train set, we provide a positive code and a negative code for a query. A triple $<Q, C^+, C^->$ represent a piece of training data: C^+ is a positive code for query Q, while C^- is a negative one. The data set only provides code–query pairs. To build triples, for every pair in the train set, we randomly selected a code as the C^- from the whole training set, and for every epoch, we re-select a new one. $\langle c^+, q\rangle$ and $\langle c^-, q\rangle$ are the representative vectors compute by QobCS for $\langle C^+, Q\rangle$ and $\langle C^-, Q\rangle$. We feed the final representation triple into the loss function:

$$L(\theta) = \sum_{(Q,C^+,C^-)\in G} max(0, \beta - S(q, c^+) + S(q, c^-)) \tag{39}$$

where the S refer to Eq. (34). θ represents all of trainable parameters. β is a value in a range of [0, 1], and we explore its influence on our model in Section 5. In the training process, with loss decreasing, the $S(c, q^+)$ tends to be higher but does not greater than 1. The $S(c, q^-)$ tends to be smaller but not less than −1. To better understand how the loss function work, we deform as in Eqs. (40)–(41):

$$diff = S(c^+, q) - S(c^-, q) \tag{40}$$

$$L(\theta) = \sum_{(Q,C^+,C^-)\in G} max(0, \beta - diff) \tag{41}$$

where $diff$ represents how much is the correct code's score than the incorrect code. In the training process, when the value of the loss function becomes smaller, the $diff$ would be closer to the margin β. Thus the $S(c^+, q)$ would be much higher than $S(c^-, q)$.

We apply the Adam algorithm (Kingma and Ba, 2014) for training. The five attention mechanisms of Stage1 learn to compute different weights for words in features, and then output the feature matrices. In Stage2 of QobCS, four query-oriented attention mechanisms learn to weight vectors to weight feature matrices and get the final code representative vector. With Adam optimization training our model, all trainable parameters θ in QobCS are optimized following the gradient. Finally, our model learns to get better representative vectors for code and query.

3.7. Implementation details

The parameter setting and experimental environment of the QobCS are as follows: We train the model with a batch size of 256. The word embedding size for all word vectors is set to 100 following our previous work (Xu et al., 2021). The vocabulary capacity for two datasets is as shown in Table 1. we train and test QobCS in Python 3.5 with tensorflow 1.7.0, and keras 2.1.6.

4. Experiment setup

4.1. Research questions

We evaluated the performance of QobCS and four code search models for the following five research questions.

RQ1. Is the performance of QobCS better than that of the compared code search models?

The objective of this RQ is to investigate whether the proposed model QobCS outperforms the compared code search models described in Section 4.3.

RQ2. How does the proposed query-oriented attention mechanism contribute to the code search?

The objective of this RQ is to analyze the performance of query-oriented attention mechanism on the semantic gap of code search, and whether it outperforms the co-attention mechanism. For this goal, we run variants of QobCS and compared models to make comparisons.

RQ3. How do the code features contribute to the code search?

QobCS leverages four features to represent a code method, including method name, API sequence, tokens, and AST. The four features help our model capture the textual and structural semantic information of the source code. This experiment analyzes impacts of these features on model effectiveness, and more to investigate whether using them together in code search is the best choice. In addition, to explore the effectiveness of structural features, we run variants of CARLCS-CNN and UNIF, in which four features are applied.

RQ4. Does QobCS have desirable performance in terms of efficiency?

RQ4 tests if QobCS can significantly reduce computation resources compared to the baseline models.

RQ5. How does the margin of the loss function impact the model performance?

As mentioned in Section 3.6, the loss function helps the similarity score of the correct code–query pair to be β higher than the incorrect one. RQ5 explores how the parameter β influences the effectiveness of QobCS.

4.2. Datasets

We trained and tested our model on two existing datasets. Both datasets were collected from GitHub's Java repositories, and the queries were extracted from Javadoc. Hu et al.'s (Hu et al., 2019) dataset[2] contains 485k code–query pairs. Its repositories were created in 2015 and 2016 and with more than 10 stars. Husain et al.'s (Husain et al., 2019) dataset[3] contains 542k Java code with queries. In the rest of the paper, we use dataset1 and dataset2 to refer to Hu et al.'s dataset and Husain et al.'s datasets, respectively.

To explore how QobCS performs for real queries, we test our model on 50 real queries, which are provided by Gu et al. (2018) and were extracted from the top 50 voted Java programming questions in Stack Overflow.

[2] https://github.com/xing-hu/EMSE-DeepCom
[3] https://github.com/github/CodeSearchNet

4.3. Baselines

DeepCS. DeepCS proposed by Gu et al. (2018) is one of the state-of-the-art models. It first obtains embedding of code and query. Then, it computes the cosine similarity of the embedding for ranking. We run the DeepCS with the source model code shared by Gu et al. on the GitHub.[4]

CARLCS-CNN. CARLCS-CNN proposed by Shuai et al. (2020), is joint embedding-based code search model. It leverages CNN and LSTM for individual embedding and then leverages a co-attention mechanism to learn final representations for code and query.[5] In the following sections, we use CARLCS in brief.

UNIF. UNIF proposed by Cambronero et al. (2019) is a state-of-the-art supervised code search model. It produces the code representative vector by using a learned attention-based weighting assignment scheme and then combining the weighted embedding of words. It computes the average vector from the token embeddings of query to represent the query representative vector.

TabCS. A state-of-the-art supervised code search model TabCS proposed in our preliminary work (Xu et al., 2021). It leverages attention mechanisms followed with co-attention mechanism to achieve code search.

4.4. Evaluation metrics

For evaluation, on both datasets, we randomly selected 10k code–query pairs as test set, while the rest 475 k and 532 k pairs are applied for training. In testing, we took all code snippets in test set as candidate code. Models return the top-10 candidate code recommended list for each test query. We leverage SuccessRate, MRR and NDCG as matrices. We test models 10 times, and every time we re-divide testing pairs and training pairs in order to suppress the effect of randomness. The final results are the average value from the ten times.

SuccessRate@k (R@k): The percentage of queries for which the ground-truth code method is available in the top-k ranked list. For a query set Q, the calculation of SuccessRate@k: $R@k = |Q|^{-1} \sum_{i=1}^{|Q|} \sigma_k(Q_i)$. σ_k is an indicator function. If the top-k code snippets in the recommendation list include the ground-truth code of ith query (Q_i), it returns 1; Otherwise, it returns 0. Following our prior work (Xu et al., 2021), we evaluate on three R@ks with k at 1, 5, 10 (i.e., $R@1$, $R@5$, and $R@10$), respectively.

Mean Reciprocal Rank (MRR): The average of the reciprocal of ranks in the top-10 of all queries. For a query set Q, the calculation of MRR: $|Q|^{-1} \sum_{i=1}^{|Q|} \sigma_k(Rank_{Q_i})$. $Rank_{Q_i}$ is the rank of the ground-truth code. If $Rank_{Q_i}$ no more than k, σ_k return $Rank_{Q_i}^{-1}$. Otherwise, it returns 0. Following our prior work (Xu et al., 2021), we evaluate our model on MRR with k of 10.

Normalized Discounted Cumulative Gain (NDCG): correlation between the ranking result of code snippets and the query. The calculation of NDCG: $|Q|^{-1} \sum_{i=1}^{|Q|} \frac{2^{r_i}-1}{log_2(i+1)}$, where Q is a set containing 10k queries from test set, as referred to in Section 4.2; and r_i is the relevance score of the top-k code snippets at position i. We set the ground-truth code's relevance score as 1, and others are set as 0. The results of high correlation affect the final NDCG score more, while the results of general correlation affect it less. The NDCG score could be higher if the ground-truth code with high correlation ranked in top-10.

[4] https://github.com/guxd/deep-code-searh
[5] https://github.com/cqu-isse/CARLCS-CNN

Table 2
Effectiveness on dataset1.

Model	R@1	R@5	R@10	MRR	NDCG
DeepCS	0.311	0.503	0.593	0.335	0.387
CARLCS	0.493	0.651	0.725	0.485	0.465
UNIF	0.528	0.678	0.754	0.516	0.482
TabCS	0.576	0.732	0.802	0.562	0.617
QobCS	**0.720**	**0.820**	**0.865**	**0.701**	**0.741**

Table 3
Effectiveness on dataset2.

Model	R@1	R@5	R@10	MRR	NDCG
DeepCS	0.294	0.440	0.529	0.307	0.371
CARLCS	0.413	0.557	0.633	0.409	0.455
UNIF	0.420	0.556	0.624	0.419	0.471
TabCS	0.522	0.661	0.728	0.513	0.560
QobCS	**0.605**	**0.723**	**0.776**	**0.595**	**0.638**

5. Results

5.1. RQ1: Is the performance of QobCS better than that of the compared code search models?

We explore the effectiveness of QobCS and baselines described in Section 4.3. We train and test models on two data sets described in Sections Section 4.2.

Table 2 shows the performance on dataset1. QobCS achieves an MRR of 0.701, a NDCG of 0.741, and R@1/R@5/R@10 of 0.720/0.820/0.865. QobCS outperforms DeepCS, CARLCS, UNIF, and TabCS by 109.25%, 44.54%, 35.85%, and 24.73% in terms of MRR; by 91.47%, 59.35%, 53.73%, and 20.10% in terms of NDCG; by 131.51%/63.02%/45.87%, 46.04%/25.96%/19.31%, 36.36%/20.94%/14.72%, and 25.00%/12.02%/7.86% in terms of R@1/R@5/R@10, respectively.

For dataset2, Table 3 shows the evaluation results. QobCS achieves an MRR of 0.595, a NDCG of 0.638 and R@1/R@5/R@10 of 0.605/0.723/0.776. QobCS outperforms DeepCS, CARLCS, UNIF, and TabCS by 93.81%, 45.48%, 42.00%, and 15.98% in terms of MRR; by 71.96%, 40.22%, 35.45%, and 13.93% in terms of NDCG; by 105.78%/64.32%/46.69%, 46.49%/29.80%/22.59%, 44.05%/30.04%/24.36%, and 15.90%/9.38%/6.59% in terms of R@1/R@5/R@10, respectively.

Furthermore, we evaluate the Wilcoxon signed-rank test ($p < 0.05$) (Wilcoxon, 1992) for comparison of MRR between QobCS and baselines. The p values for QobCS with baselines are less than 0.05. They indicate that the improvements of QobCS compared with the baselines are statistically significant.

> **Result 1:** *QobCS is significantly better than the compared code search models in terms of effectiveness.*

5.2. RQ2. How does the proposed query-oriented attention mechanism contribute to the code search?

To investigate the effectiveness of the query-oriented attention mechanism on the semantic gap between code and query, we run the variants of DeepCS and UNIF (i.e., DeepCS-Q, and UNIF-Q) on two datasets. In DeepCS-Q and UNIF-Q, the query-oriented attention mechanism is applied after their embedding networks. In addition, to compare the effectiveness of the query-oriented attention mechanism and co-attention mechanism on correlation learning, we run the variants of CARLCS, TabCS, and QobCS (i.e.,CARLCS-Q, TabCS-Q, and QobCS-C). In CARLCS-Q and TabCS-Q, co-attention mechanism is replaced by the query-oriented attention mechanism. In QobCS-C, the query-oriented attention mechanism is replaced by co-attention.

Table 4
Effectiveness of models with/without query-oriented attention mechanism on dataset1.

Model	R@1	R@5	R@10	MRR
DeepCS	0.331	0.503	0.593	0.335
DeepCS-Q	**0.392**	**0.563**	**0.653**	**0.390**
UNIF	0.528	0.678	0.754	0.516
UNIF-Q	**0.568**	**0.728**	**0.798**	**0.558**
CARLCS	0.493	0.651	0.725	0.485
CARLCS-Q	**0.542**	**0.696**	**0.765**	**0.530**
TabCS	0.576	0.732	0.802	0.562
TabCS-Q	**0.621**	**0.765**	**0.826**	**0.607**
QobCS-C	0.676	0.796	0.850	0.659
QobCS	**0.720**	**0.820**	**0.865**	**0.701**

Table 5
Effectiveness of models with/without query-oriented attention mechanism on dataset2.

Model	R@1	R@5	R@10	MRR
DeepCS	0.294	0.440	0.529	0.307
DeepCS-Q	**0.356**	**0.506**	**0.590**	**0.354**
UNIF	0.420	0.556	0.624	0.419
UNIF-Q	**0.435**	**0.569**	**0.641**	**0.430**
CARLCS	0.413	0.557	0.633	0.409
CARLCS-Q	**0.488**	**0.594**	**0.659**	**0.440**
TabCS	0.522	0.661	0.728	0.513
TabCS-Q	**0.543**	**0.685**	**0.745**	**0.535**
QobCS-C	0.587	0.719	0.775	0.577
QobCS	**0.605**	**0.723**	**0.776**	**0.595**

Table 4 shows the comparison of baselines, QobCS, and their variants on dataset1. We can observe that in terms of R@1/R@5/R@10 and MRR, DeepCS-Q improves DeepCS by 19.43%/11.93%/10.12%, and 16.42%; UNIF-Q improves UNIF by 7.58%/7.37%/5.84%, and 8.14%; CARLCS-Q improves CARLCS by 9.94%/6.91%/5.52%, and 9.28%; TabCS-Q improves TabCS by 7.81%/4.51%/2.99%, and 8.01%; QobCS-C decreases QobCS by 6.11%/2.93%/1.73%, and 5.99%;

Table 5 shows the comparison results on dataset2. We can observe that in terms of R@1/R@5/R@10 and MRR, DeepCS-Q improves DeepCS by 21.09%/15.00%/11.53%, and 15.31%; UNIF-Q improves UNIF by 3.57%/2.34%/2.72%, and 2.63%; CARLCS-Q improves CARLCS by 18.16%/6.83%/4.11%, and 5.01%; TabCS-Q improves TabCS by 4.02%/6.36%/3.57%, and 4.29%; QobCS-C decreases QobCS by 2.98%/0.55%/ 0.13%, and 3.03%.

These results imply that the query-oriented attention mechanism is effective for DeepCS and UNIF. For CARLCS, TabCS and QobCS, the query-oriented attention mechanism has clear advantages over the co-attention mechanism.

> **Result 2:** *The proposed query-oriented attention mechanism shows good performance on building code and query's correlation in the code search task.*

5.3. RQ3. How do textual and structural features contribute to the code search?

To investigate the effectiveness of the code features applied in QobCS, we do experiments on four variants of QobCS. Each variant removes a feature, using three other features to represent a code. To explore the effectiveness of the AST, we run the variants of CARLCS and UNIF (i.e., CARLCS (M+P+T+A), and UNIF (M+P+T+A)) on two datasets. Four features mentioned above are applied in CARLCS (M+P+T+A), and it applies a CNN network to embed the newly added AST sequence. UNIF (M+P+T+A) takes four code features, which are embedded in individual attention mechanisms and merge them into the final code representation.

Table 6
Effectiveness of three textual features (i.e., method name (M), tokens (T), and API sequence (P)) and a structural feature (i.e., AST (A)) on dataset1.

Model	R@1	R@5	R@10	MRR
CARLCS	0.493	0.651	0.725	0.485
CARLCS (M+P+T+A)	**0.520**	**0.678**	**0.750**	**0.511**
UNIF	0.528	0.678	0.754	0.516
UNIF (M+P+T+A)	**0.541**	**0.701**	**0.773**	**0.533**
QobCS (P+T+A)	0.663	0.774	0.823	0.645
QobCS (M+T+A)	0.682	0.797	0.845	0.665
QobCS (M+P+A)	0.668	0.779	0.825	0.654
QobCS (M+P+T)	0.707	0.812	0.858	0.690
QobCS (M+P+T+A)	**0.720**	**0.820**	**0.865**	**0.701**

Table 7
Effectiveness of three textual features (i.e., method name (M), tokens (T), and API sequence (P)) and a structural feature (i.e., AST (A)) on dataset2.

Model	R@1	R@5	R@10	MRR
CARLCS	0.413	0.557	0.633	0.409
CARLCS (M+P+T+A)	**0.439**	**0.580**	**0.656**	**0.426**
UNIF	0.420	0.566	0.624	0.419
UNIF (M+P+T+A)	**0.458**	**0.601**	**0.674**	**0.455**
QobCS (P+T+A)	0.531	0.658	0.719	0.521
QobCS (M+T+A)	0.600	0.716	0.771	0.587
QobCS (M+P+A)	0.569	0.690	0.746	0.558
QobCS (M+P+T)	0.601	0.716	0.767	0.585
QobCS (M+P+T+A)	**0.605**	**0.723**	**0.776**	**0.595**

We train and test models on dataset1. Table 6 shows the evaluation results. In terms of R@1/R@5/R@10 and MRR, CARLCS (M+P+T+A) improves CARLCS by 5.48%/4.15%/3.45%, and 5.36%, and UNIF (M+P+T+A) outperforms UNIF by 2.46%/3.39%/2.52%, and 3.29%. By removing the method name, API sequence, tokens and AST on QobCS, the MRR decreases by 8.68%, 5.41%, 7.19% and 1.59%, respectively; the R@1/R@5/R@10 decreases by 7.92%/5.94%/5.10%, 5.28%/2.89%/2.37%, 7.22%/ 5.26%/4.85%, and 1.81%/0.99%/ 0.82%, respectively.

Table 7 shows the evaluation results on dataset2. In terms of R@1/R@5/R@10 and MRR, CARLCS (M+P+T+A) improves CARLCS by 6.30%/4.13%/3.63%, and UNIF (M+P+T+A) improves UNIF by 9.05%/6.18%/8.01%, and 8.59%. By removing the method name, API sequence, tokens and AST on QobCS, the MRR decreases by 14.20%, 1.36%, 6.63% and 1.71%, respectively; the R@1/R@5/R@10 decreases by 12.23%/9.88%/7.93%, 0.83%/0.98%/0.65%, 5.95%/ 4.78%/ 4.02%, and 0.66%/0.98%/1.17%, respectively.

These results show that the code structural feature contribute to the effectiveness in code search. And in QobCS, the four features are all effective and do not conflict with each other.

> **Result 3:** *The structural feature is effective in code search task.*

5.4. Does QobCS have good performance in terms of efficiency?

We train and test QobCS and baselines on an Nvidia Titan V GPU with 12 GB of memory.

As shown in Tables 8 and 9, in model testing, QobCS is slightly slower than CARLCS. But compared with CARLCS (M+P+T+A), we find that QobCS is quicker, which means when using textual and structural features, QobCS is quicker for testing. UNIF (M+P+T+A)'s efficiency on training and testing is slightly quicker than QobCS, because UNIF performs independent embedding, without considering the correlation between code and query.

These results imply that the attention mechanism-based search models — TabCS, QobCS and UNIF, are quicker in training and testing.

Table 8
Time cost for model training and testing on dataset1. Training column shows the time cost for 300 training epoch by models. The test column shows the average time cost for searching for a query on test set.

Model	Training	Testing
DeepCS	36.6 h	1.1s/query
CARLCS	12.2 h	0.4s/query
UNIF	2.8 h	0.2s/query
CARLCS (M+P+T+A)	16.6 h	0.7s/query
UNIF (M+P+T+A)	5.0 h	0.4s/query
TabCS	5.1 h	0.4s/query
QobCS	6.3 h	0.6 s/query

Table 9
Time cost for model training and testing on dataset2. Training column shows the time cost for 300 training epoch by models. The test column shows the average time cost for searching for a query on test set.

Model	Training	Testing
DeepCS	34.1 h	0.9s/query
CARLCS	10.7 h	0.4s/query
UNIF	1.7 h	0.2s/query
CARLCS (M+P+T+A)	13.2 h	0.6s/query
UNIF (M+P+T+A)	3.8 h	0.3s/query
TabCS	3.9 h	0.4s/query
QobCS	5.8 h	0.5s/query

DeepCS, CARCLS, and CARCLS (M+P+T+A) are slower because of their complex network structure (Yin et al., 2017). UNIF and TabCS are slightly quicker than QobCS, but we think that it is worthwhile to exchange a little efficiency loss for effect improvement.

Result 4: *QobCS shows desirable performance in terms of efficiency.*

5.5. RQ5. How does the margin of the loss function impact the model performance?

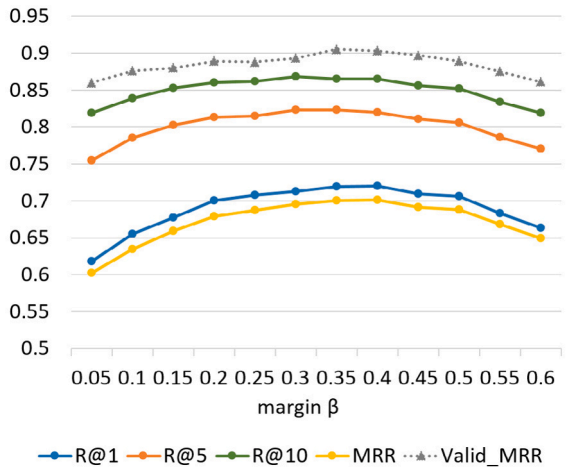
We demonstrate the margin β tuning on two datasets for QobCS, respectively. In particular, the β is set from 0.05 to 0.6 with a step size of 0.05. For each value of β, we select the model with the best performance on the validation set, and then leverage the test set to the selected model, obtaining the final results on R@1/R@5/R@10, and MRR. In addition, we record the ValidMRR on the validation set.

In Fig. 3(a), we can observe that for dataset1, setting β to 0.4 makes QobCS achieve the best performance. In Fig. 3(b), we can observe that for dataset2, setting β to 0.25 makes QobCS achieve the best performance. For both models on two datasets, the tendency of ValidMRR is similar. It means it is reliable to set β according to ValidMRR.

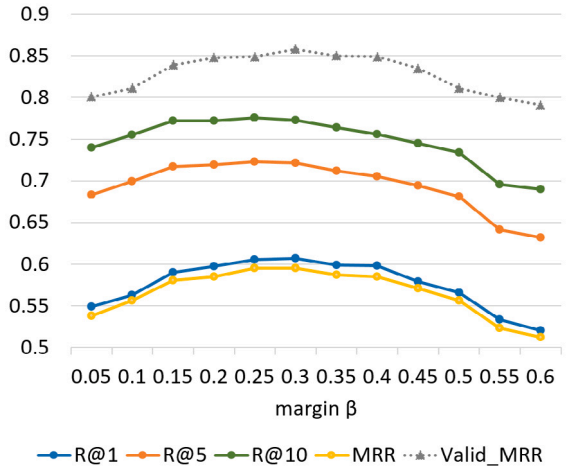
Result 5: *Margin of loss function largely affects the performance of QobCS. Setting the value of the margin considering the ValidMRR of the valid set is feasible.*

6. Discussion

This section first discusses performance of QobCS. Sections 6.1 and 6.2 analyze the effectiveness and efficiency of QobCS. Section 6.3 describes how QobCS works with the pre-trained model. Section 6.4 analyzes the QobCS's limits. Section 6.5 discusses the threats to model validity.



(a) Effectiveness comparison of QobCS on margin β from 0.05 to 0.6 in terms of R@1/R@5/R@10, MRR, and ValidMRR on dataset1.



(b) Effectiveness comparison of QobCS on margin β from 0.05 to 0.6 in terms of R@1/R@5/R@10, MRR, and ValidMRR on dataset2.

Fig. 3. Effectiveness comparison of QobCS on margin β on two dataset.

6.1. Why does QobCS work?

We discuss the substantial advantages of QobCS for two reasons — the cooperation of QobCS's two stages, and query-oriented attention mechanism.

Two-Stage Attention Mechanism. Fig. 4 shows the retrieved results for the query "convert a word to uppercase". Code B and Code C are incorrect for the query, and are recommended by CARCLS and UNIF. Code A is correct and is recommended by QobCS. Its Stage1 uses an attention mechanism to filter out unimportant or meaningless words in code and query. Stage2 uses the query-oriented attention mechanism to search code for related information to the query's main intention. As shown in Fig. 4(a), QobCS finds the keywords (i.e.,"convert", "word", and "return word") and the API (i.e., "word.toUpperCase") are closely related to the query's intention (i.e., "convert", "word", "uppercase"). In this way, QobCS learns the code representative vector, which is highly closed to the query representative vector. As shown in Fig. 4(b), Code B has less related information. In Fig. 4(c), Code C has a related

```
//Query: convert a word to uppercase
//Code A:
protected String convertWord ( String word )
{     word = word.toUpperCase ( ) ;
      if ( ignoreList . contains ( word ) ) {   return null ; }
      return word ;
}
```

(a) The code recommended by QobCS.

```
//Query: convert a word to uppercase
//Code B:
public static boolean isUpperCase ( char c )
 {
      return isUpperCase ( ( int ) c ) ;
 }
```

(b) The code recommended by CARLCS.

```
//Query: convert a word to uppercase
//Code C:
public boolean isKeyword ( String s )
{
      return m_Keywords.contains (  s.toUpperCase ( ) ) ;
}
```

(c) The code recommended by UNIF.

Fig. 4. The recommended codes for the query "convert a word to uppercase".

API, but it contains much unrelated information. Therefore, compared with Code A, the representative vectors of Code B and Code C are far from the query's representative vector. Finally, QobCS correctly put the code A first, which is the ground-truth code.

Moreover, to explore how the two-stage attention mechanism promotes code search, we analyze the weights assigned to a code–query pair, which has been shown in Fig. 1. The histograms in Fig. 5 show the weights assigned to the query and code tokens in two stages. As shown in Fig. 5(a), in Stage1, the query word"two" and the code tokens "system", and "length" get less weights. Because they are frequently used, QobCS assumes that they contribute very little to represent the query's intention or the code functionality. Therefore, the other query words (i.e., "concatenate" and "arrays") and the other code tokens ("arraycopy","values" and "concatenate") get higher weights. Then, in stage2 as shown in Fig. 5(b), the code tokens "arraycopy" and "concatenate" received weights of 0.44 and 0.114, higher than the weight of "values". This is because QobCS found the former corresponds to the query's intention, but the latter does not. Finally, the weight assignment in two stages would be used for the model to predict how the code snippet corresponds to the query's intention.

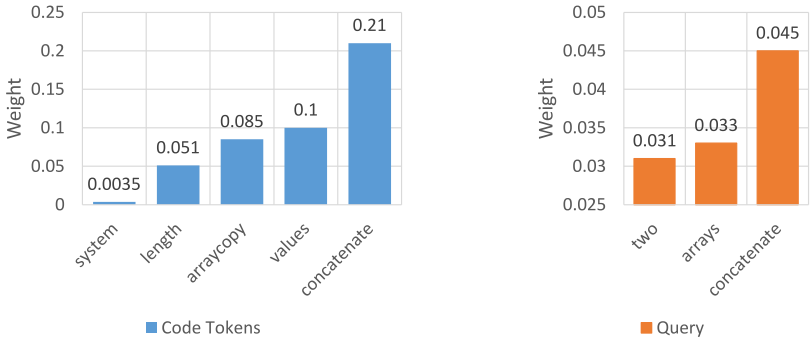
Query-Oriented Attention Mechanism. In Table 10, Code A and D are the candidate code for the query "convert a word to uppercase". Code A is the ground-truth code and Code D is wrong. As shown in Table 10, QobCS correctly gives Code A a higher score than Code D, but QobCS-C gives Code D a higher score than Code A. By analyzing the weight distribution, we explore why query-oriented attention mechanism is more effective than co-attention mechanism.

From Table 10, We can notice that the query's weights change with the candidate code for QobCS-C. Given Code D, QobCS-C assigns more attention weights on the query word (i.e., "uppercase") since in Code D, words with the same semantics can be found. However, for other important query words (i.e., "convert", and "word"), since there is no related information in code D, they are filtered out. Similarly, in Code D, QobCS-C focuses more on these related words (i.e., "case", and "return"). Finally, since QobCS-C focuses on only part of the query's intention, it believes that code D is most likely correct. Similarly, for Code A, QobCS-C focuses on related query and code words (i.e., "convert", "word", and "case"). Thus, QobCS-C believes that Code A may also be correct. To sum up, it is obvious that the co-attention mechanism may distort or one-sided the intention of the query, resulting in QobCS-C being unable to distinguish whether code meets all or part of the query's demands. Therefore, Code A and D get the closed similarity scores, and the score of the wrong Code D is a little higher.

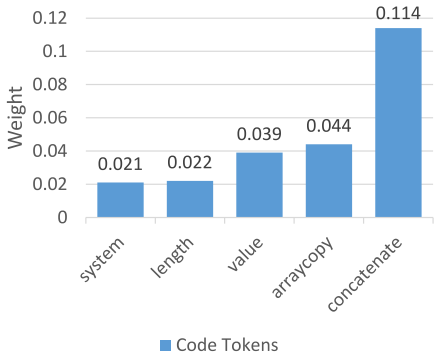
Compared with the co-attention mechanism, the query-oriented attention mechanism tends to save more important information about the query. As shown in Table 10, given Code A or Code D, the weights assigned to the query do not change. Important words (i.e., "convert", "word", and"uppercase") are all assigned with higher weights. In addition, QobCS assigns high attention weights to the words (i.e., "convert", "word", and "upper" in Code A, and "case" in Code D) which is related to the query's weighted information. Therefore, QobCS accurately distinguishes Code A and Code D. Code A meets all of the query's demands and gets a similarity score of 0.6950; Code D meets part of the demands and obtains 0.6407, much smaller than Code A. This result implies that the query-oriented attention mechanism shows desirable performance on the code search task. It makes up for the defects of the co-attention mechanism, which tends to ignore query's semantics.

6.2. *Why is QobCS fast?*

QobCS's network is simple and quick because of the two attention-based stages. In both stages, the embedding of each vector is only changed by a weight, and the output representative vector is the weighted summation of vectors. Therefore, the network is very simple. In addition, the trainable parameters of QobCS are less than common CNN-based or LSTM-based models, so QobCS greatly saves time consumption.



(a) Weights assigned to code tokens and query in Stage1.



(b) Weights assigned to code tokens in Stage2.

Fig. 5. Weights assignment for the code–query pair in Fig. 1.

Table 10
The similarity score and weight distribution on Code A, Code D and a query computed by QobCS and QobCS-C for the query "convert a word to uppercase". Code A is the ground-truth code and Code D is wrong. For both code and query as shown, the darker the color, the more weight is assigned to the word by the model.

Model	QobCS-C	QobCS

Code A	Query: convert a word to uppercase
	Protected String convertWord (String word)
	{
	word = word . toUpperCase ( ) ;
	if (ignoreList . contains (word)) {
	return null ; }
	return word ;
	}
	
	Similar score: 0.7481
		Query: convert a word to uppercase
		Protected String convertWord (String word)
		{
		word = word . toUpperCase ( ) ;
		if (ignoreList . Contains (word)) {
		return null ; }
		return word ;
		}
		
		Similar score: 0.6950

Code D	Query: convert a word to uppercase
	Public static Boolean isUpperCase (char c)
	{
	return isUpperCase ((int) c) ;
	}
	
	Similar score: 0.7576
		Query: convert a word to uppercase
		Public satic Boolean isUpperCase (char c)
		{
		return isUpperCase ((int) c) ;
		}
		
		Similar score: 0.6407

6.3. Performance with pre-trained models

Recently, pre-trained code search models have become popular. In order to verify the performance of QobCS combined with the pre-trained model, we fine-tune the CodeBERT with the code search task using the public pre-trained model. In addition, we build a variant of QobCS — QobCS-cb in which the embedding process is replaced with CodeBERT (Feng et al., 2020).

As Table 11 shown, QobCS-cb outperforms CodeBERT by 1.31%, and 1.32%/0.92%/0.44% in terms of MRR, and R@1/R@5/R@10, respectively; In QobCS-cb, the CodeBERT could map the code and query into the same space. The following query-oriented attention mechanism could compute a deeper code representation according to the current

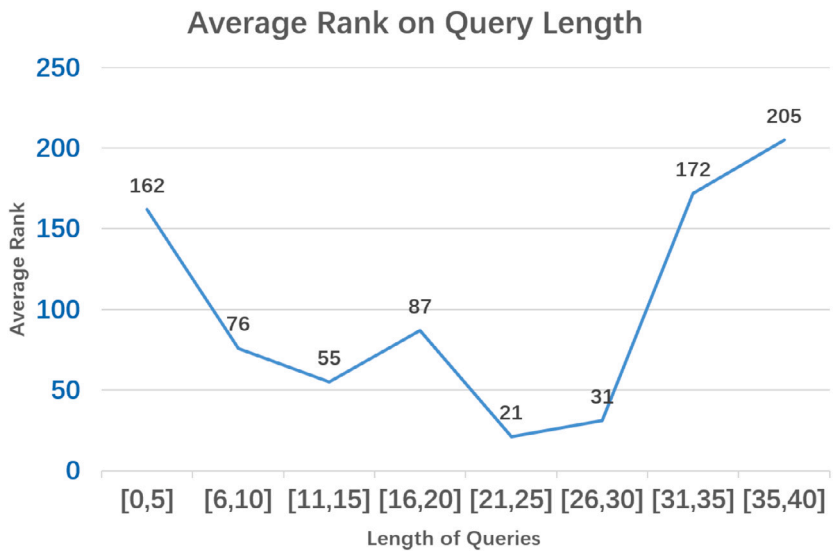


Fig. 6. Average Rank of ground-truth code for query with multiple length.

Q1: convert a word to uppercase

Q2: return content buffer start this slice ending starting end position represents at new buffer

Fig. 7. Weight assignment on two query computed by QobCS on dataset2.

Table 11
Performance of CodeBERT and QobCS-cb on dataset1.

Model	R@1	R@5	R@10	MRR
CodeBERT	0.680	0.870	0.913	0.761
QobCS-cb	0.689	0.878	0.917	0.771

query, which is more useful for matching the query to select which code is closer to the query. Therefore, QobCS-cb shows better performance than CodeBERT.

6.4. Limitations of QobCS

We make statistics on the results of queries with different lengths. The results are shown in Fig. 6. As shown, we find the length of the query has a great influence on the model performance.

Long and Complex Query. As shown in Fig. 6, QobCS shows poor performance on queries with more than 30 words on dataset2 (Husain et al., 2019). From Table 12, for the real complex queries (i.e., NO.14, NO.22, NO.46, and NO.47), our model cannot search out the correct code snippets in the top-10. To further analysis of the specific reason, we compare the weight distribution of a long complex query and a common-length query. As shown in Fig. 7, compared with Q1, the weights of words in Q2 show more smooth distribution, which is caused by weights normalization in the attention mechanism. On a long query, the smooth distribution of weights makes QobCS extract semantics in a way similar to simple crude addition. Therefore, for long and complex queries, QobCS cannot accurately obtain the semantics of the query.

Simple and Short Query. In Fig. 6, QobCS perform poorly on queries with less than 5 words on Husain et al.'s dataset (Husain et al., 2019). Too short queries may not contain complete intention, so our model cannot extract out the full intention from the query without clear intention. Therefore, QobCS cannot correctly rank candidate code because of incomplete query semantics. Query reformulation tools can solve this problem, and break through the limitation of the code search model on low-quality queries, which is considered in our research direction in the next step.

Real Query. In our dataset, a comment is used as a query, but they are different text entities. Compared to comments, real queries tend to be more poor and short. As shown in Fig. 6, QobCS shows poor performance on short queries. To explore the performance of QobCS on real queries, we make a manual verification on 50 real queries provided by Gu et al. (2018). Models are trained on dataset2. We evaluate models on a code base containing all candidate code in dataset1.

As shown in Table 12, 31/27/22 queries can find the correct code snippets in the top-10 code snippets recommended by QobCS, TabCS and DeepCS, respectively; 18/9/6 queries' correct code snippets ranked first by QobCS, TabCS, and DeepCS. These results show that our model has better performance on real queries than baselines TabCS and DeepCS.

Through the analysis results, we find when the query contains some adjectives (e.g., "the simplest" in NO.12, "the fastest" in NO.14, and "the best" in NO.47), our model as well as baselines, cannot find the correct code snippets. This is caused by the difference between comments and real queries. Comments always only describe a method's function, but in practice, real inquirers sometimes add adjectives or details into their queries to narrow the target range. However, these words cannot be understood by our models and also make noise.

In addition, we find for queries (e.g., NO.6, NO.9, and NO.10) that only one line of code or a few words could answer, models cannot find any results. Because sometimes, the intention of the inquirers is a knowledge point, the correct answer can be as small as one keyword, and can be as large as one class. But comments we extracted only focus on a method, which limits the models' ability.

6.5. Threats to model validity

We extract comments from javadocs as queries. Compared to comments, the quality and expression of real queries are more diverse depending on the queriers. Therefore, the difference between the two entities may affect performance. The natural language of the query is limited to English. The programming language of code is limited in java on the method level. In this paper, we choose a simple method for representing code structure. In the near future, we plan to apply more useful code structure representations. Some parameters in QobCS, such as the max length of code tokens and query tokens, as well as the dimension of word embeddings, are set by default, which may effect the models' performance and not apply to other data sets.

Table 12
Real Queries and Evaluation Results on DeepCS, TabCS and QobCS. (NF: Not Found within the top 10 returned results.)

NO.	Query	DeepCS	TabCS	QobCS
1	convert an inputstream to a string	1	1	1
2	create arraylist from array	NF	NF	1
3	iterate through a hashmap	NF	NF	NF
4	generating random integers in a specific range	3	1	1
5	converting string to int in java	NF	8	NF
6	initialization of an array in one line	NF	NF	NF
7	how can I test if an array contains a certain value	2	1	10
8	lookup enum by string value	2	1	1
9	breaking out of nested loops in java	NF	NF	NF
10	how to declare an array	NF	NF	NF
11	how to generate a random alpha-numeric string	2	3	1
12	what is the simplest way to print a java array	NF	NF	NF
13	sort a map by values	1	1	1
14	fastest way to determine if an integer's square root is an integer	NF	NF	NF
15	how can I concatenate two arrays in java	NF	1	1
16	how do I create a java string from the contents of a file	5	NF	4
17	how can I convert a stack trace to a string	6	4	2
18	how do I compare strings in java	4	4	1
19	how to split a string in java	1	NF	1
20	how to create a file and write to a file in java	7	2	2
21	how can I initialize a static map	NF	NF	NF
22	iterating through a collection, avoiding concurrent modification exception when removing in loop	NF	NF	NF
23	how can I generate an md5 hash	3	8	7
24	get current stack trace in java	NF	7	1
25	sort arraylist of custom objects by property	NF	NF	NF
26	how to round a number to n decimal places in java	3	3	2
27	how can I pad an integers with zeros on the left	NF	NF	NF
28	how to create a generic array in java	NF	NF	NF
29	reading a plain text file in java	NF	7	3
30	a for loop to iterate over enum in java	NF	NF	NF
31	check if at least two out of three booleans are true	NF	NF	NF
32	how do I convert from int to string	6	2	5
33	how to convert a char to a string in java	NF	NF	NF
34	how do I check if a file exists in java	3	2	1
35	java string to date conversion	1	6	2
36	convert inputstream to byte array in java	1	1	1
37	how to check if a string is numeric in java	2	1	2
38	how do I copy an object in java	NF	4	1
39	how do I time a method's execution in java	NF	9	1
40	how to read a large text file line by line using java	8	NF	1
41	how to make a new list in java	NF	NF	4
42	how to append text to an existing file in java	NF	2	1
43	converting iso 8601-compliant string to date	NF	1	1
44	what is the best way to filter a java collection	3	7	2
45	removing whitespace from strings in java	1	5	2
46	how do I split a string with any whitespace chars as delimiters	NF	NF	NF
47	in java, what is the best way to determine the size of an object	NF	NF	NF
48	how do I invoke a java method when given the method name as a string	6	3	1
49	how do I get a platform dependent new line character	NF	NF	NF
50	how to convert a map to list in java	NF	NF	NF

7. Related work

7.1. Code search

Code search is a hot topic in software engineering. In the early stage of the development of code search, Mica (Stylos and Myers, 2006) and Assieme (Hoffmann et al., 2007) input queries into search engines, and then extract code from the output results. The search base of search engines contains many other contents besides the code, which greatly affects the performance of code search. Therefore, researchers become to construct a code base, and leverage the IR technique to explore code search tools (Lv et al., 2015)(Liu et al., 2021). In the beginning, they record the common words in the query and code to compute a relevance score, and then sort the code in the code base. For example, CodeHow (Lv et al., 2015) considers the impact of text similarity of code and query into code search. Liu et al. (2021) proposed an IR-based code search model CodeMatcher, which understands irrelevant/noisy keywords and captures sequential relationships between words in query and code. However, IR-based models only take natural language and programming language as the combination of characters or words,

ignoring their semantics. In recent years, with the rapid development of deep learning) (DL) technology, researchers began to use deep learning to extract the deep semantics of code and query (Gu et al., 2018; Shuai et al., 2020; Cambronero et al., 2019; Cheng and Kuang, 2022). Gu et al. (2018) first proposed DL-based model named DeepCS. It builds deep learning networks to map the code and query into the same vector space, then compute the distance for ranking all candidate code in the code base. For deeper embedding, Shuai et al. (2020) proposed CARLCS-CNN, which applied a co-attention mechanism to build the natural language and programming language's correlation. UNIF (Cambronero et al., 2019) leverages the attention mechanism, which is simpler compared to most DL-based code search model. Cheng and Kuang (2022) proposed CSRS, which leverages relevance matching and semantic matching to build more complete correlation. Since DL-based models are usually slower than IR-based models, researchers proposed to combine IR techniques and DL techniques, to guarantee both effectiveness and efficiency. Hu et al. (2023) proposed TOSS, which combines the advantages of IR-based, bi-encoder, and cross-encoder models.

To further identify the correlation between code and query, many tools (Wan et al., 2019; Gu et al., 2020; Gang et al., 0000; Zeng et al., 2021; Ling et al., 2021) have been proposed to explore the effectiveness code structure in the code search task. For example, Wan et al. (2019) proposed MMAN, which uses TREE-LSTM and GGNN to embed AST and CFG for the code search task. Zeng et al. (2021) proposed DEGRAPHCS, which transfers source code into variable-based flow graphs to model code semantics more precisely. Gu et al. (2020) proposed CRaDLe, which fuses the dependency and semantic information from code together to obtain the final representation. Ling et al. (2021) proposed DGMS, which leverage graph neural networks to learn both query's structure and code's structure. In addition, Feng et al. (2020) proposed a transformer-based pre-trained model CodeBert. It learns general-purpose representation and shows good performance in code search. Many works leveraging pre-trained codebert were proposed (Guo et al., 2021; Hu et al., 2023).

Some researchers (Liu et al., 2019; Eberhart and Mcmillan, 2022; Sun et al., 2022; Wang et al., 2022) explored to high-quality data in code search task. Eberhart and Mcmillan (2022) proposed to use information extracted from method names and comments to generate a query of good quality. Sun et al. (2022) proposed a data cleaning framework build high-quality training datasets. Wang et al. (2022) leverages reinforcement learning to generate semantically enriched queries for code search.

7.2. Code structure

Different from natural language, code is written by programming languages (e.g., Python, Java, C), which are strictly structured. Code Structure (e.g., Abstract syntax tree (Graph, 2015), Control Flow Graph, and Data Flow Graph) is widely used in code search and code representation. Abstract Syntax Tree (AST) (Graph, 2015) represents the syntax structure of a code snippet by a tree, and each node on the tree represents a structure or logic used in the code. AST is widely used in code search to obtain the deeper representation of code structure (Zhang et al., 2020; Alon et al., 2019, 2018; Zhang et al., 2019; Zügner et al., 2021; Wang et al., 2020).

Alon et al. (2019, 2018) used AST to represent snippets of code as continuously distributed vectors. Zhang et al. (2019) split each large AST into a sequence of small statement trees, and encode the statement trees to vectors to represent a code. Zügner et al. (2021) extracted relative pairwise distances from AST to learn code representation. Wang et al. (2020) proposed a code representation by dynamically composing different neural network units into tree structures based on the input AST. Tang et al. (2021) proposed to encode ast bode by ancestor-descendent nodes and sibling nodes. Tao et al. (2023) uses self-attention networks on the AST to obtain deeper representation of code. Xu et al. (2023) proposed eXtreme Abstract Syntax Tree (AST)-based Neural Network for source code representation for industrial practice.

Control Flow Graph (CFG) is an abstract representation of a code block or a program. It represents the possible flow direction of all basic blocks in a process in the form of a graph. Wan et al. (2019) proposed using CFG for the code structure features in the code search task. In addition to AST and CFG, Data Flow Graph (DFG) also shows good performance with many problems of software engineering. Gu et al. (2020) extracted Data Dependency from DFG to help represent a code.

7.3. Attention mechanism

The attention mechanism is a special structure embedded in machine learning models, which is used to automatically learn and calculate the contribution of input data to output data. The attention mechanism popular in software engineering and shows good performance in learning semantics. UNIF (Cambronero et al., 2019) is the first attention-based code search model. The experiment results shows that attention mechanism could largely improve code search.

To learn the correlation between natural language and programming language, researchers applied many variants of attention mechanism to jointly embed code and query (Jing et al., 2018; Tay et al., 2018). Ling et al. (2021) proposed code search model DGMS, which uses cross-attention mechanisms to learn the correlation between the query's structure and the code's structure. Our model leverages the query-oriented attention mechanism to build the semantics correlation and leverage the given query to compute the code final representation. Moreover, our experimental results indicate that combining two kinds of attention mechanisms (i.e., the attention mechanism followed by the query-oriented attention mechanism) are valuable for the code search task.

8. Conclusion and future work

In this paper, we proposed a query-oriented code search model named QobCS, which leverages attention mechanisms followed by the query-oriented attention mechanism to learn the final representation of code and query. The evaluation results on two datasets show that QobCS outperforms the compared models. The experimental results indicate that the query-oriented attention mechanism has good performance on the semantic gap between code and query, and the combination of attention mechanism and query-oriented attention mechanism is valuable for code search.

In the future, we will explore better models and methods to extract the code structural information, and develop a code search tool for QobCS. In addition, we decide to apply query reformulation to improve the code search model's performance on real queries.

CRediT authorship contribution statement

Huanhuan Yang: Conceptualization, Investigation, Methodology, Validation, Writing – original draft. **Ling Xu:** Conceptualization, Project administration, Resources, Writing – review & editing. **Chao Liu:** Conceptualization, Writing – review & editing. **Luwen Huangfu:** Project administration, Supervision, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

I have share the link in the paper.

Acknowledgments

This work was supported in part by the National Key Research and Development Project (No. 2018YFB2101200), the Fundamental Research Funds for the Central Universities (No. 2019CDYGYB014 and No. 2020CDCGRJ037), the National Nature Science Foundation of China (No. 62002034) and the National Defense Basic Scientific Research Program (No. WDZC20205500308).

References

Alon, U., Levy, O., Yahav, E., 2018. code2seq: Generating sequences from structured representations of code.

Alon, U., Zilberstein, M., Levy, O., Yahav, E., 2019. Code2vec: Learning distributed representations of code. Proc. ACM Program. Lang. 3 (POPL), http://dx.doi.org/10.1145/3290353.

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

Cheng, Y., Kuang, L., 2022. CSRS: Code search with relevance matching and semantic matching. In: Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension. ICPC '22, Association for Computing Machinery, New York, NY, USA, pp. 533–542. http://dx.doi.org/10.1145/3524610.3527889.

Eberhart, Z., Mcmillan, C., 2022. Generating clarifying questions for query refinement in source code search.

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., 2020. CodeBERT: A pre-trained model for programming and natural languages.

Gang, H.A., Min, P.A., Yz, B., Qx, A., My, A., 0000. Neural joint attention code search over structure embeddings for software Q&A sites - ScienceDirect, . Syst. Softw. 170.

Gardner, M.W., 1998. Artificial neural networks (the multilayer perceptron)—a review of applications in the atmospheric sciences. Atmos. Environ. 32.

Graph, A.S., 2015. Abstract syntax tree.

Gu, W., Li, Z., Gao, C., Wang, C., Zhang, H., Xu, Z., Lyu, M.R., 2020. CRaDLe: Deep code retrieval based on semantic dependency learning.

Gu, X., Zhang, H., Kim, S., 2018. Deep code search. In: 2018 IEEE/ACM 40th International Conference on Software Engineering. ICSE, IEEE, pp. 933–944.

Guo, D., Ren, S., Lu, S., Feng, Z., Tang, D., Liu, S., Zhou, L., Duan, N., Svyatkovskiy, A., Fu, S., 2021. GraphCodeBERT: Pre-training code representations with data flow. In: International Conference on Learning Representations.

Hoffmann, R., Fogarty, J., Weld, D.S., 2007. Assieme: finding and leveraging implicit references in a web search interface for programmers. In: Proceedings of the 20th Annual ACM Symposium on User Interface Software and Technology. ACM, pp. 13–22.

Hu, X., Li, G., Xia, X., Lo, D., Jin, Z., 2019. Deep code comment generation with hybrid lexical and syntactical information. Empir. Softw. Eng. 1–39.

Hu, F., Wang, Y., Du, L., Li, X., Zhang, H., Han, S., Zhang, D., 2023. Revisiting code search in a two-stage paradigm. In: Proceedings of the Sixteenth ACM International Conference on Web Search and Data Mining. WSDM '23, Association for Computing Machinery, New York, NY, USA, pp. 994–1002. http://dx.doi.org/10.1145/3539597.3570383.

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

Jing, Y., Yuhang, L., Weifeng, Z., Zengchang, Q., Yanbing, L., Yue, H., 2018. Learning cross-modal correlations by exploring inter-word semantics and stacked co-attention. Pattern Recognit. Lett. S0167865518304380–.

Kingma, D., Ba, J., 2014. Adam: A method for stochastic optimization. Comput. Sci.

Ling, X., Wu, L., Wang, S., Pan, G., Ji, S., 2021. Deep graph matching and searching for semantic code retrieval. ACM Trans. Knowl. Discov. Data 15 (5), 1–21.

Liu, J., Kim, S., Murali, V., Chaudhuri, S., Chandra, S., 2019. Neural query expansion for code search.

Liu, C., Xia, X., Lo, D., Liu, Z., Hassan, A.E., Li, S., 2021. CodeMatcher: Searching code based on sequential semantics of important query words. ACM Trans. Softw. Eng. Methodol. (TOSEM).

Lv, F., Zhang, H., Lou, J., Wang, S., Zhang, D., Zhao, J., 2015. CodeHow: Effective code search based on API understanding and extended boolean model (E). pp. 260–270.

Shuai, J., Xu, L., Liu, C., Yan, M., Xia, X., Lei, Y., 2020. Improving code search with co-attentive representation learning.

Stylos, J., Myers, B.A., 2006. Mica: A web-search tool for finding API components and examples. In: 2006 IEEE Symposium on Visual Languages and Human-Centric Computing. VL/HCC 2006, 4-8 September 2006, Brighton, UK.

Sun, Z., Li, L., Liu, Y., Du, X., Li, L., 2022. On the importance of building high-quality training datasets for neural code search.

Sundermeyer, M., Schlüter, R., Ney, H., 2012. LSTM neural networks for language modeling. In: Thirteenth Annual Conference of the International Speech Communication Association.

Tang, Z., Li, C., Ge, J., Shen, X., Zhu, Z., Luo, B., 2021. AST-transformer: Encoding abstract syntax trees efficiently for code summarization. In: 2021 36th IEEE/ACM International Conference on Automated Software Engineering. ASE, pp. 1193–1195. http://dx.doi.org/10.1109/ASE51524.2021.9678882.

Tao, C., Lin, K., Huang, Z., Sun, X., 2023. CRAM: Code recommendation with programming context based on self-attention mechanism. IEEE Trans. Reliab. 72 (1), 302–316. http://dx.doi.org/10.1109/TR.2022.3171309.

Tay, Y., Luu, A.T., Hui, S.C., 2018. Multi-pointer co-attention networks for recommendation. In: The 24th ACM SIGKDD International Conference.

Wan, Y., Shu, J., Sui, Y., Xu, G., Zhao, Z., Wu, J., Yu, P.S., 2019. Multi-modal attention network learning for semantic source code retrieval.

Wang, W., Li, G., Shen, S., Xia, X., Jin, Z., 2020. Modular tree network for source code representation learning. ACM Trans. Softw. Eng. Methodol. 29 (4), http://dx.doi.org/10.1145/3409331.

Wang, C., Nong, Z., Gao, C., Li, Z., Zeng, J., Xing, Z., Liu, Y., 2022. Enriching query semantics for code search with reinforcement learning. Neural Netw. 145, 22–32. http://dx.doi.org/10.1016/j.neunet.2021.09.025, URL https://www.sciencedirect.com/science/article/pii/S0893608021003877.

Wilcoxon, F., 1992. Individual comparisons by ranking methods. In: Breakthroughs in Statistics. Springer, pp. 196–202.

Xu, L., Yang, H., Liu, C., Shuai, J., Xu, Z., 2021. Two-stage attention-based model for code search with textual and structural features. In: 2021 IEEE International Conference on Software Analysis, Evolution and Reengineering. SANER.

Xu, Z., Zhou, M., Zhao, X., Chen, Y., Cheng, X., Zhang, H., 2023. xASTNN: Improved code representations for industrial practice. http://dx.doi.org/10.48550/arXiv.2303.07104, CoRR abs/2303.07104. arXiv:2303.07104.

Yin, W., Kann, K., Yu, M., Schütze, H., 2017. Comparative study of CNN and RNN for natural language processing. arXiv preprint arXiv:1702.01923.

Zeng, C., Yu, Y., Li, S., Xia, X., Wang, Z., Geng, M., Xiao, B., Dong, W., Liao, X., 2021. deGraphCS: Embedding variable-based flow graph for neural code search.

Zhang, J., Wang, X., Zhang, H., Sun, H., Liu, X., 2020. Retrieval-based neural source code summarization. In: International Conference on Software Engineering. p. 1.

Zhang, J., Wang, X., Zhang, H., Sun, H., Wang, K., Liu, X., 2019. A novel neural source code representation based on abstract syntax tree. In: 2019 IEEE/ACM 41st International Conference on Software Engineering. ICSE, pp. 783–794. http://dx.doi.org/10.1109/ICSE.2019.00086.

Zügner, D., Kirschstein, T., Catasta, M., Leskovec, J., Günnemann, S., 2021. Language-agnostic representation learning of source code from structure and context.

Huanhuan Yang is a master student in the School of Big Data & Software Engineering, Chongqing University, China. Her research interests include intelligent software engineering, code search.

Ling Xu is an Associate Professor at the School of Big Data & Software Engineering, Chongqing University, China. She received her B.S. degree in Hefei University of Technology in 1998, and her M.S. degree in software engineering in 2004. She received her Ph.D. degree in Computer Application from Chongqing University, P.R. China in 2009. Her research interests include mining software repositories, bug reduction, and localization

Chao Liu is an Associate Professor at the School of Big Data & Software Engineering, Chongqing University, China. He received her Ph.D. degree in Chongqing University in 2018. His research interests include Big data and software intelligence, artificial intelligence, natural language processing

Luwen Huangfu is an Assistant Professor of management information systems with San Diego State University, USA. She received the B.S. degree in software engineering from Chongqing University, China, the M.S. degree in computer science from the Chinese Academy of Sciences, and the Ph.D. degree in management information systems from the University of Arizona. Her research interests include business analytics, text mining, data mining, artificial intelligence, and healthcare management.