Results in Engineering 27 (2025) 105979

Contents lists available at ScienceDirect

Results in Engineering

journal homepage: www.sciencedirect.com/journal/results-in-engineering

Research paper

Enhancing code search through query expansion: A fusion of LSTM with GloVe and BERT model (ECSQE)

Nazia Bibi [a,*], Muhammad Usman Tariq [b], Zabeeh Ullah [a], Muhammad Babar [c], Zahid Khan [d]

[a] Department of Computer Software Engineering, National University of Sciences and Technology, Islamabad, Pakistan
[b] Department of Marketing, Operations, and Information Systems, College of Business, Abu Dhabi University, Abu Dhabi, United Arab Emirates
[c] Robotics and Internet of Things Lab, Prince Sultan University, Riyadh, Saudi Arabia
[d] Robotics and Internet of Things Lab, Prince Sultan University, Riyadh, Saudi Arabia

ARTICLE INFO

Keywords:
Codesearchnet
Natural language
Deep neural networks
Semantic matching
LSTM
GloVe
BERT
Source code selection
Source code reuse
Recommendation system

ABSTRACT

In software engineering efficient code retrieval is essential for developers to quickly access relevant code snippets from vast repositories. This study focuses on enhancing code search through query expansion, leveraging advanced word embedding techniques such as GloVe and BERT to improve search accuracy and relevance. The main objective is to assess how the proposed model can outperform traditional models in terms of evaluation metrics by expanding query terms based on contextual understanding. The proposed model was evaluated using three query datasets to enable a comprehensive performance comparison with baseline models, including UNIF, CNN-CS and DeepCS. The results demonstrate that the proposed model significantly enhances code search effectiveness by generating more contextually relevant queries, which in turn leads to improved retrieval outcomes. Nonetheless, challenges such as language dependency and data diversity, were identified. The study concludes by emphasizing the potential of the proposed model to transform code search while highlighting areas for future work including expanding the model's applicability across more languages and addressing scalability issues for larger datasets.

1. Introduction

Searching within existing source code repositories for relevant code snippets has been a critical task in Software Engineering for quite some time [1]. Various approaches have been proposed to tackle this challenge. More recently, neural methods have been integrated, such as Neural Code Search (NCS) developed by Facebook, which computes continuous vector embeddings of programs at the method-level granularity. However, code search history logs revealed that shorter queries often led to less successful search sessions, requiring more query reformulations and increased time browsing results. This suggests that relying solely on NCS with short queries may not lead to productive outcomes [2].

Query expansion is a method that is frequently applied to improve keyword-based searches in Information Retrieval systems. This is usually done by making the given query more informative through the addition of relevant words to it [3,4]. This maneuvering is a powerful method that helps developers discover the most pertinent code snippets, libraries, or documents to perform their projects successfully. Platforms like GitHub help developers find code examples, learn from existing code, or solve similar problems. The motivation for developing advanced query and code search techniques stems from the challenges developers encounter when formulating complete and effective queries—particularly when they lack knowledge of relevant definitions, libraries, or project-specific processes. Various Natural Language Processing (NLP) and Information Retrieval (IR) techniques are specifically designed toward query expansion in code search [5]. These techniques allow to bridge the gap between the search need of a developer and the pieces of code available by enriching the initial query with a set of words or phrases that have a high probability of giving better results for the search requirement. Query expansion techniques enhance the effectiveness of code search results by adding relevant terms to the original query, thereby making the software development process more efficient and productive [3].

Existing code search query expansion techniques exhibit several notable gaps: (i) Current methods generally overlook domain specific knowledge and context required to broaden code search queries. They

* Corresponding author.
E-mail addresses: nazia.bibi@mcs.nust.edu.pk (N. Bibi), Muhammad.kazi@adu.ac.ae (M.U. Tariq), zabeeh.phdcse@students.mcs.edu.pk (Z. Ullah), mbabar@psu.edu.sa (M. Babar), zskhan@psu.edu.sa (Z. Khan).

https://doi.org/10.1016/j.rineng.2025.105979
Received 18 January 2025; Received in revised form 11 June 2025; Accepted 25 June 2025
Available online 3 July 2025

heavily rely on mechanisms like stemming and synonym expansion that may not be sufficiently sensitive to the complexities of code and programming languages [6]; (ii) Existing methods unable to capture semantically rich linguistic relationships and code structures such as function calls and data structure matching (variable types). Conventional text-based approaches often find it challenging to handle the intricate structures and patterns found in code [7]; (iii) There is no strong evaluation framework currently exists for measuring the success of query expansion technique over various code repositories and programming languages. Similarly, many popular platforms such as Stack Overflow suffer from similar limitations, which prevents efficient code search [8]; (iv) Most studies focus on a single code example or a limited set of examples in a specific programming language, which makes generalization challenging. To fully evaluate a query expansion approach, it is also essential to consider a broader range of programming paradigms and repositories [9].

In the current code search query expansion techniques [10], it is seen that there lies a failure to bridge the gap. To address this problem, "Enhancing Code Search through Query Expansion" (ECSQE) model is proposed. In the ECSQE model, all the advanced deep learning methods like LSTM, GloVe and BERT are used to address primary failures. During the code search operation, the training scheme enables the ECSQE model to learn and comprehend how to pair code snippets more coherently to queries. Compared to the traditional methods that utilize only text-based approaches, the ECSQE model integrates code structures and relationships at the semantic level. This boosts the ability to bring and match queries such that they best match relevant code snippets with high accuracy. Additionally, algorithms like LSTM, GloVe and BERT are included in the ECSQE model making it important to cope with the variety of programming languages and their structure. This multi-facet approach improves the retrieval of relevant code and offers generalizability across several programming languages and contexts. The ECSQE model introduces a major step forward in the code search field due to its more thorough and effective way of dealing with existing query expansion techniques. The ECSQE model introduces several critical improvements to improve the effectiveness of code search from query expansion. Our main contributions are as follows:

- ECSQE model supports different types of programming languages. This is a multi-aspect application that not only retrieves relevant code but also generalizes the model across different types of programming.
- The ECSQE model fills existing gaps by incorporating domain-specific knowledge and context requirements necessary for expanding code search queries. The proposed model ensures that the expansion occurs towards more aligned queries with the complexities of the code and the programming language.
- The performance of the suggested ECSQE model is evaluated on the benchmarked "CodeSearchNet" dataset that consists of code snippets from a range of programming languages. The results are compared with available state-of-the-art baseline models to determine how well the proposed model operates.
- The proposed ECSQE model expands the queries of the user by automatically including relevant terms and improves the code search results. This will help developers to find the most relevant code snippets for their programming work.

The organization of the paper is structured as follows. Section 2 provides the Literature Review, examining related work and studies. Section 3 outlines the Research Methodology adopted for the study. Section 4 describes the Architecture of the Proposed ECSQE Model, detailing how the model is structured. Section 5 presents the Experimental Design, explaining the setup for testing and evaluating the ECSQE model. Section 6 reports the Results of the ECSQE Model, providing a detailed performance analysis. Section 7 discusses a Case Study to showcase the application of the model. Section 8 presents a comprehensive Discussion, addressing the advantages, limitations and potential improvements of the approach. Finally, Section 9 concludes the paper with Conclusion and Future Work, summarizing the findings and outlining potential directions for further research.

2. Literature review

In the domain of source code retrieval [11], query expansion plays a pivotal role in bridging the gap between the user's natural language queries and the technical language of source code. By enhancing the original query with additional terms or phrases, query expansion techniques aim to improve the search engine's ability to retrieve relevant code snippets. These techniques vary in approach and application, each seeks to address the inherent challenge of accurately interpreting and matching user queries to the appropriate code [12]. However, they also present unique limitations, ranging from the introduction of irrelevant terms to challenges in adapting to evolving technical terminologies [13]. In the following literature review, various query expansion techniques are explored along with their limitations and how the proposed model seeks to overcome these challenges, thereby enhancing the effectiveness of code search systems.

2.1. Query expansion techniques

- Synonym expansion [14] is the most elementary query expansion methodology that includes synonyms or close variants of the original query. This approach aims to broaden the search scope by capturing various expressions of a concept. However, a key drawback to this is its tendency to allow irrelevant terms especially due to the polysemous problem where words acquire more than one meaning. This leads to a loss in accuracy, mainly because the extended query may return an extremely high range of irrelevant results. The proposed ECSQE model addresses this problem by including the BERT model which is superior to understanding the word context in a given query. By identifying the specific meaning of a term within its context, the model aims to reduce the likelihood of irrelevant query expansions, thereby preserving the precision of the retrieved results.
- Semantic models like GloVe and Word2Vec find words or phrases that are similar in meaning to the ones given in the query [15,16]. These models are capable of capturing contextual nuances but not very robust especially in adapting with the latest terminologies of rapidly evolving fields. At times, terms may not even depict present usage and evolving concepts especially in technology and programming. Our proposed ECSQE model overcomes it with the incorporation of the BERT model. BERT model is known for its strong deep learning capabilities as well as contextual awareness. BERT's dynamic understanding of language nuances and ability to adapt toward newer terminologies enhance the model's effectiveness in the current and evolving context with a more robust query expansion.
- API-based expansion [17,18] focuses on the addition of pertinent API calls or library functions to the queries. This is very applicable in source code retrieval as it directly relates queries with possible code implementations. Nevertheless, having API terms added without prior consideration may result in information overload, which dilutes the precision of the retrieved search results [19]. This is reduced through the LSTM layer and attention mechanism within our proposed model of ECSQE, which emphasize contextually relevant APIs in the expansion process. By focusing on the most relevant APIs and library functions, the query expansion is balanced making it focused and effective.
- Pattern-based expansion [20] recognizes and augments queries with common programming patterns and idioms. This approach is especially useful for locating code that follows specific design patterns or programming constructs. The main limitation of this approach is the risk of overlooking less common yet relevant patterns,

which can result in incomplete search results. Our proposed ECSQE model by coupling BERT and LSTM provides a detailed insight into several coding patterns. This integration enables the model to accept and include a wider range of patterns in the query expansion process, thereby enriching the search results with a more inclusive set of relevant code snippets.

- Inferring user intent [21–23] to expand queries involves deducing the underlying purpose or task behind a query and adding related terms. This technique tailors search results to what users are likely looking for and ultimately enhance relevancy. However, this technique faces the challenge of accurately interpreting user intent which leads to irrelevant query expansions. Our proposed ECSQE model's attention mechanism plays a crucial role by focusing on key aspects of the query to derive a more accurate inference of user intent. This targeted approach ensures that the expansion is closely aligned with the user's actual needs, thereby improving the overall relevance of the search results.
- Ontology-based expansion [24,25] relies on an already structured representation of knowledge, for instance a database or framework of interrelated concepts in order to develop further related concepts and terminologies of enrichment for the query. This way of enrichment is supposed to introduce all relevant terminologies into the query by creating relevant concepts. Nevertheless, its applicative scope remains narrow since it is not possible to cover all the possible types of queries. This limitation compensated by the proposed ECSQE model with best exploitation of both contextual embedding and LSTM that captures the rich nuances in concepts and relationships. This will ensure that the query expansion encompasses a wider range of relevant terms and moves beyond the boundaries of predefined ontologies.
- The use of the feedback loop in query expansion [26,27] applies data from past searches to expand queries for use in future interactions. It tailors search experience based on the aggregated user behavior and preferences. This is the limitation of this method because it relies only on historical data that may not always rich or relevant. The proposed ECSQE model trained on diversified data that permit dynamic adaptation of the query expansion process, making sure that it remains responsive to current search contexts and evolving user needs, thus improving effectiveness and accuracy.
- Cross-language expansion [28,29] involves incorporating terms from multiple programming languages during query expansion. This approach recognizes that code functionality often extends beyond the implementation details of any single language. It is particularly useful where multiple developers work on different languages. However, it can introduce irrelevant terms if not properly controlled. The ECSQE model addresses the challenge using deep learning that discerns relevance across languages. The proposed model generates relevant cross-language terms so that the specificity and relevance of the search results can be kept intact.
- Recent advancements in the use of large language models (LLMs) for code summarization have gained significant attention [30]. These studies provide valuable insights into the capabilities of LLMs. However, several key limitations remain. Many works are limited to a single prompting strategy, such as zero-shot or few-shot prompting [31–33]. There is little exploration of more advanced techniques like chain-of-thought prompting. In addition, the impact of different model configurations and parameter settings on LLM performance is rarely examined. This results in a limited understanding of their generalizability and robustness. Another major shortcoming lies in the evaluation methodology. Most studies rely on conventional metrics such as BLEU, METEOR, ROUGE-L, or SentenceBERT-based cosine similarity. These metrics may not be well-suited for evaluating the expressive and detailed outputs generated by LLMs [30].

Each of these query expansion techniques offers distinct advantages for enhancing code search but they also come with their own set of limitations. To address these limitations, the proposed ECSQE model adopts a deep learning-based architecture that leverages GloVe and BERT embeddings, an LSTM layer and an attention mechanism to enrich natural language queries with contextually relevant API calls and library functions. This structured approach reduces the reliance on prompt engineering and introduces explicit control over model behavior through interpretable components. By focusing on semantically meaningful query expansion, ECSQE enhances retrieval precision and relevance, thus providing a robust alternative to current LLM-based summarization techniques. In doing so, the model bridges gaps related to prompting diversity, parameter transparency and evaluation reliability in the code summarization domain.

2.2. Query reformulation approaches

Effective query reformulation is crucial in enhancing the retrieval of relevant information. This section explores various methods, their limitations and proposed improvements.

- **Conquer and Keyword Co-occurrence** Singh et al. [34], Singh and Kumar [35] utilizes natural language techniques to identify sets of co-occurring keywords among query results, aiding in query refinement. However, its primary focus on keyword co-occurrence can potentially miss the broader context or semantic relationships. Our proposed ECSQE model goes beyond mere co-occurrence by leveraging semantic and contextual analysis by offering a more comprehensive query expansion.
- **WordNet for Synonym Expansion** Gong et al. [36] employs WordNet for expanding queries with synonyms, presenting a classic approach. The limitation of this method is that synonym-based expansion can introduce irrelevant terms due to polysemy. Our proposed ECSQE model solution incorporates contextual understanding through BERT, which helps discern specific meanings, thereby mitigating the risk of irrelevant expansions.
- **Refoqus and Past Query Analysis** Haiduc et al. [37] trains a classifier on past queries and results to recommend reformulation strategies. This approach heavily relies on historical data, which may not reflect current or evolving coding practices. Our proposed ECSQE model solution addresses this limitation by utilizing user queries and a broader corpus for query expansion that ensures that the expansions are current and accurate.

3. Research methodology

The research methodology for the ECSQE model is structured around several key stages to ensure a robust, data-driven approach in improving code search as shown in Fig. 1. The methodology involves the design, development and evaluation of the ECSQE model that incorporates deep learning-based query expansion techniques using word embeddings such as GloVe and BERT. Below is an overview of the steps involved.

- ***Step1: Data Collection-*** The first phase includes collecting relevant a dataset from publicly available code sources. The "CodeSearchNet" dataset is selected. This dataset comprises a variety of code query pairs but the suggested model uses Python and Java query code pairs. Furthermore, three query datasets are used to assess the model's performance. Pre-processing is done on datasets to ensure uniform structure and quality.
- ***Step 2: Word Embedding Techniques-*** To extend the query terms, two pre-trained word embedding models: the GloVe and BERT models were used. GloVe is used for the fact that it can express semantic similarity between the two words in a vector of a given size. Unlike other original word embeddings models, BERT has a context-based

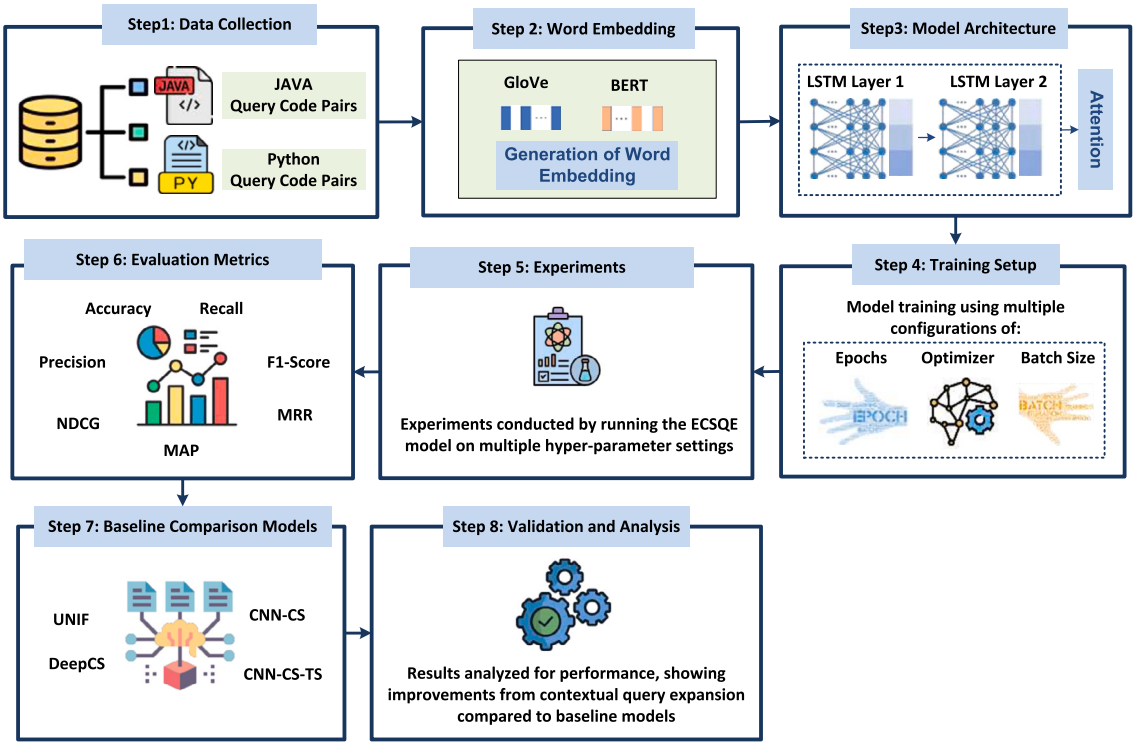


Fig. 1. Methodology.

word representation technique and highly parameterized elements. The conjunction of these embeddings enables ECSQE to produce richer query expansions to ensure the potential of retrieving highly relevant code snippets.

- ***Step3: Model Architecture-*** The architecture of ECSQE integrates the strengths of GloVe and BERT with a Long Short-Term Memory (LSTM) network to handle sequential relationships in the code queries. Additionally, an attention mechanism is incorporated to focus on the most important parts of the query during the expansion process. This helps the model to weigh critical query terms higher than less relevant ones, improving the precision of retrieval. LSTM helps capture long-term dependencies within the query, while Attention mechanisms enhance the relevance of the expansion by highlighting significant parts of the code and query context.
- ***Step 4: Training Setup-*** The model is trained using multiple configurations of batch size, epochs and optimization techniques [38]. Hyperparameters are tuned based on performance metrics such as accuracy, precision, recall and F1-score. The training involved minimizing a loss function that accounts for the relevance of retrieved code snippets relative to the expanded query.
- ***Step 5: Experimental Setup-*** Experiments are conducted by running the ECSQE model on multiple query datasets. These datasets consisted of varying query complexities to test the generalizability and scalability of the model. Each model variant (GloVe+LSTM, BERT+LSTM, etc.) is evaluated for performance across different query sets to analyze how well the query expansion techniques enhanced code retrieval.
- ***Step 6: Evaluation Metrics-*** The ECSQE model performance is computed using a variety of metrics to ensure a comprehensive assessment of its performance. The evaluation metrics include Accuracy, Precision, Recall, F1-Score, Mean Reciprocal Rank (MRR), Normalized Discounted Cumulative Gain (NDCG), etc.
- ***Step 7: Baseline Comparison Models-***The results of the evaluation metrics are then used for comparison against several baseline models, including UNIF, CNN-CS, DeepCS etc [39]. These baseline models are selected to provide a comprehensive benchmark for assessing the improvement brought by the ECSQE model.
- ***Step 8: Validation and Analysis-*** Comparative analysis is conducted between the ECSQE model and baseline models to highlight the improvements achieved through contextual query expansion.

4. Architecture of proposed ECSQE model

Fig. 2 illustrates the proposed architecture for the ECSQE model. This section provides a detailed discussion of the ECSQE model.

4.1. Input processing layer

The user provides a query in natural language to search for relevant code snippets as shown in Equation (1). Initially, this layer processes a typically short text query. ECSQE model is built to decipher queries in different programming languages, from function calls to more abstract programming concepts.

$$Query = (NL_{Text}) \tag{1}$$

4.2. Tokenization and preprocessing

Equation (2) tokenizes the natural language query to prepare it for embedding. To clean and standardize the input query for further processing. This process involves converting all text to lowercase to ensure consistency, removing special characters and punctuation (unless they are significant in a programming context) and tokenizing the query into individual words or tokens. This step is crucial for both the GloVe and BERT models to work effectively.

$$Tokens = Tokenize(Query) \tag{2}$$

4.3. GloVe embeddings layer

This layer's purpose is to transform query tokens into vector representations by utilizing pre-trained GloVe embeddings [40]. GloVe

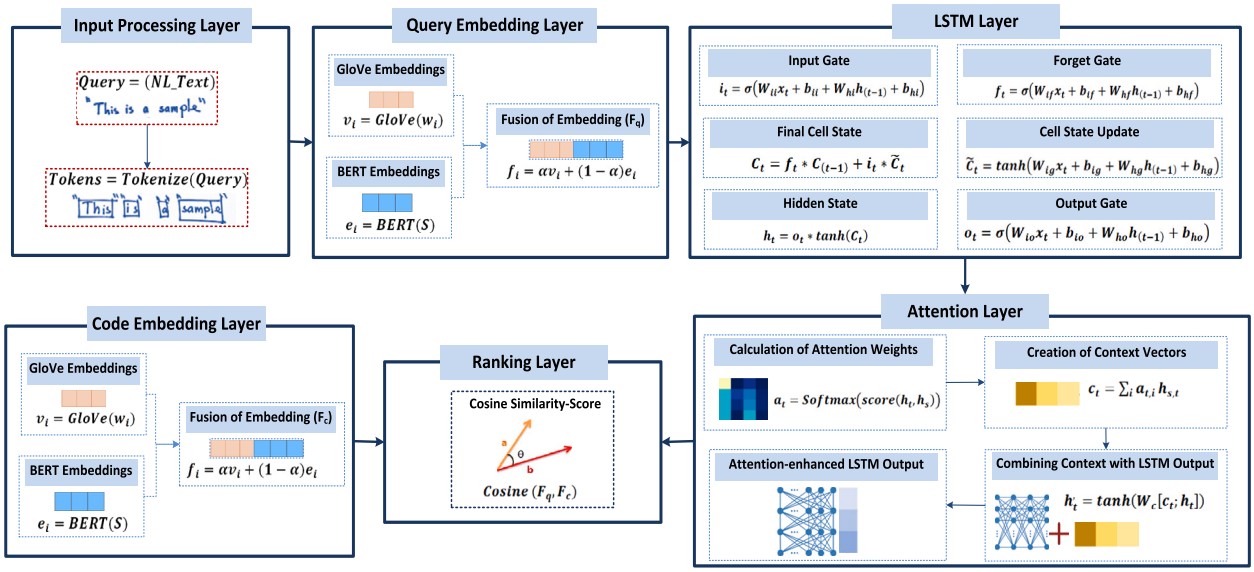


Fig. 2. Proposed Architecture for ECSQE Model.

embeddings are developed from extensive text corpora and excel at understanding the semantic relationships between words. Each token from the query is linked to a specific high-dimensional vector [41]. These vectors effectively capture both the syntactic and semantic subtleties in the text, as demonstrated in Equation (3).

$$v_i = GloVe(w_i) \tag{3}$$

Each word w_i in the query is mapped to a vector v_i using the GloVe embedding function.

4.4. BERT embeddings layer

The purpose of this layer is to provide context-aware embeddings for the query tokens. Unlike GloVe, BERT considers the entire query at once, allowing it to understand the context around each word [42,43]. This results in embeddings that are sensitive to the order of words and their contextual meanings, which is particularly useful for understanding the intent behind a query as shown in Equation (4).

$$e_i = BERT(S) \tag{4}$$

The entire query string S is input into the BERT model, which outputs a set of embeddings e_i for each token in the query.

4.5. Fusion layer (Fq)

This layer the synergies GloVe and BERT embeddings effectively. It may be a simple concatenation of the vectors or more sophisticated methods like weighted averaging or neural network-based fusion. The ultimate aim is to generate a compact, integrated expression of the query which is both the general semantic meaning (from GloVe) and the context-specific nuances (from BERT) as depicted in Equation (5).

$$f_i = \alpha v_i + (1+\alpha)e_i \tag{5}$$

Where

- f_i: This represents the final embedding for the query after fusion.
- v_i: This is the GloVe embedding for the query.
- e_i: This is the BERT embedding for the query.
- α: This is a weighting factor that controls the importance of each embedding method in the final fusion. When α is high, GloVe embeddings will have more influence. When α is low, BERT embeddings will have more influence.
- $(1+\alpha)$: This ensures a balanced scaling between GloVe and BERT embeddings in the fusion process.

It is a straightforward simple weighted average of GloVe and BERT embeddings, where α is a weighting factor that determines the relative importance of GloVe versus BERT embeddings. This layer enables the model to utilize the unique strengths of both embeddings. GloVe diffusion global words co-occurrence pattern of the entire vocabulary and hence gives general semantic meaning. BERT shows more subtle meanings, nuanced, context-specific meanings. The α factor can be tuned such that the model can choose to emphasize either general meanings or context-specific meanings that are more relevant to a particular application.

4.5.1. Optimization of the fusion weight parameter (α)

The fusion parameter α plays a pivotal role in modulating the balance between GloVe and BERT embeddings within the fusion layer. This scalar determines the relative influence of general-purpose semantic information (captured by GloVe) and context-specific nuances (captured by BERT) in the final query representation. To ensure the optimal combination of these two complementary embedding types, the parameter α is treated as a tunable hyperparameter. During the model development phase, a systematic grid search or random search is performed over a predefined range of α values (e.g., 0.1 to 5.0). Each candidate value is evaluated based on the model's performance on a held-out validation set, to maximize accuracy or another task-relevant metric, such as mean reciprocal rank (MRR) or top-k retrieval performance.

In addition to fixed tuning, an adaptive variant is also considered, wherein α is treated as a learnable parameter that is updated through backpropagation during training. This approach allows the model to dynamically adjust the semantic-contextual trade-off based on the characteristics of the data and the task. For instance, if a query benefits more from contextual understanding, the model can down-weight the GloVe component, and vice versa. This dynamic adjustment enhances the expressiveness and flexibility of the final embedding, enabling the model to generalize more effectively across diverse query types. Ultimately, the optimization of α ensures that the fusion layer leverages the strengths of both embedding sources in a balanced and task-sensitive manner.

4.6. LSTM layer

The purpose of this layer is to capture the sequential nature and long-term dependencies within the query. LSTM networks are adept at

processing sequences of data, making them suitable for understanding the order and flow of words in a query. This step further refines the query representation, taking into account the sequence in which terms appear, which is crucial for accurately interpreting the user's intent. The LSTM layer enhances code search by supporting query expansion, particularly when integrated with GloVe and BERT models.

- **Input Gate:** In the context of code search, the input gate controls which parts of the user's query (represented by x_t are important to retain in the cell state as shown in Equation (6). This is crucial for determining the significance of each word or phrase in the query regarding the code-related context. For example, specific programming keywords or function names might be given more importance.

$$i_t = \sigma(W_{ii}x_t + bii + W_{hi}h_{(t-1)} + b_{hi}) \tag{6}$$

Where
  - i_t: Input gate vector at time step t. Determines the degree to which new information is allowed into the cell state.
  - W_{ii}: Weight matrix for input gate applied to the input vector.
  - x_t: Input vector at time step t, representing the current token or word from the query.
  - b_{ii}: Bias vector for input gate applied to the input vector.
  - W_{hi}: Weight matrix for input gate applied to the hidden state.
  - $h_{(t-1)}$: Hidden state vector from the previous time step
  - $(t-1)$ carrying information from previous tokens in the query.
  - b_{hi}: Bias vector for input gate applied to the hidden state.
  - σ: Sigmoid activation function, mapping the input to a range between 0 and 1.
- **Forget Gate:** The forget gate's role is particularly important in the dynamic context of programming queries, where certain information might become irrelevant as the query progresses as shown in Equation (7). For instance, if the query initially focuses on a general concept but then specifies a particular programming language, the forget gate helps in discarding the less relevant general context.

$$f_t = \sigma(W_{if}x_t + b_{if} + W_{hf}h_{(t-1)} + b_{hf}) \tag{7}$$

  - f_t: Forget gate vector at time step t. Decides what information to discard from the cell state.
  - W_{if}, b_{if}, W_{hf}, b_{hf}: Similar to the input gate, these are weight matrices and bias vectors, but for the forget gate.
- **Cell State Update:** This stage creates candidate values for updating the cell state, which is essential for adding new context from the query as shown in Equation (8). In code search, this means integrating new terms or concepts from the query that are relevant to the search.

$$\tilde{C}_t = tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{(t-1)} + b_{hg}) \tag{8}$$

Where
  - $\tilde{C}_t$: Vector of candidate values for updating the cell state at time step.
  - W_{ig}, b_{ig}, W_{hg}, b_{hg}: Weight matrices and bias vectors for generating the candidate values.
  - $tanh$: Hyperbolic tangent activation function, providing a range between -1 and 1.
- **Final Cell State:** The final cell state is a blend of the old state and new information as shown in Equation (9). For code search, this helps maintain a balance between the initial part of the query and any new information added later, ensuring a comprehensive understanding of the user's intent.

$$C_t = f_t * C_{(t-1)} + i_t * \tilde{C}_t \tag{9}$$

Where
  - C_t: Cell state vector at time step t. Represents the 'memory' of the LSTM unit.
  - $C_{(t-1)}$: Vector representing the Cell state from the previous time step.
- **Output Gate:** This gate decides which parts of the cell state will be used to generate the output (or hidden state) as shown in Equation (10). In the context of code search, this means deciding which aspects of the query's context are crucial for the current step in searching for code.

$$O_t = \sigma(W_{io}x_t + b_{io} + W_{ho}h_{(t-1)} + b_{ho}) \tag{10}$$

Where
  - o_t: Output gate vector at time step t. Controls the parts of the cell state to be output.
  - W_{io}, b_{io}, W_{ho}, b_{ho}: Weight matrices and bias vectors for the output gate.
- **Hidden State:** The hidden state represents the current understanding of the query. It's used for the next steps in the process, such as interfacing with the code search engine as shown in Equation (11). This state carries the distilled information of the query, which is essential for retrieving the most relevant code snippets.

$$h_t = O_t * tanh(C_t) \tag{11}$$

Where
  - h_t: Hidden state vector at time step t. Represents the current output of the LSTM unit, which will be used for subsequent processing steps in the code search.

    In the context of enhancing code search, these components of the LSTM layer work together to process the query sequentially. Each word or token in the query is represented by x_t and the LSTM gates decide how this information is stored, updated and used, influenced by the context provided by previous tokens (captured in $h_((t-1))$ and $C_{(t-1)}$). This mechanism allows the model to understand and retain the essential parts of the query, improving the relevance and accuracy of the code search results.

4.7. Integrating an attention mechanism with LSTM for query expansion

For code search, the proposed model combines an attention mechanism alongside LSTM for further expanding the query whereas the attention mechanism is supposed to focus on the important parts of the query so that the appropriate code can be found.

4.7.1. LSTM processes the query

The tokenized query is fed into the LSTM layer. The query is tokenized using GloVe and BERT word embedding techniques. LSTM analyzes hidden states (h_t) which are a representation of the contextual information of each word of query. This approach grasps an idea of the sequence and context of the query and is thus able to find relationships between words and their impact on coding concepts.

4.7.2. Calculation of attention weights

At each step (i.e. step corresponding to each word (or token) of a query processed by the LSTM), the attention mechanism calculates the set of attention weights for all words (or tokens) from the vocabulary. This is done by comparing the current hidden state of the LSTM with all previous hidden states. The attention weights (a_t) are typically calculated using a softmax function over a scoring function that measures the alignment between the current hidden state (h_t) and each of the previous hidden states (h_s) as shown in Equation (12).

$$a_t = Softmax(score(h_t, h_s)) \tag{12}$$

Equation (12) at each step of the input sequence determines which parts of the query are most relevant. The input sequence determines how many points should be accrued along each path. For instance, when a query involves some specific programming function, the attention mech-

anism learns how to pay more attention to a piece of query that contains this information.

4.7.3. Creation of context vectors

Using the attention weights, the model forms a context vector (c_t) for each time step by taking a weighted sum of all the hidden states as shown in Equation (13). The ECSQE model utilizes attention weights to form a context vector (c_t) for each time step by taking a weighted sum of all the hidden states as shown in Equation (13).

$$c_t = \sum_i a_{(t,i)} h_{(s,t)} \tag{13}$$

The context vector preserves the summary of the query till the current word and meanwhile, the most relevant parts of the query for code search are highlighted.

4.7.4. Combining context with LSTM output

The context vector is integrated with the current hidden state of the LSTM to create an improved representation of the query at each time step, as illustrated in Equation (14). Doing this prevents the output at each time step of LSTM from only referencing the sequence up to that point, but also focuses on the most relevant parts of the query when searching for code.

$$h'_t = tanh(W_c[c_t; h_t]) \tag{14}$$

4.7.5. Query expansion against source code

The attention-enhanced LSTM further improves query representation to produce output that matches the source code. Embedding and encoding methods are applied to source code representations to compare code snippets versus queries. These attention-oriented search algorithms focus on the most relevant code snippets to the critical aspects of the query indicated by the attention mechanism. It encompasses specific terms patterns or concepts within programming that match with a query. The enhanced code search is accomplished by the mechanisms outlined below:

- By integrating attention with LSTM, the model can focus on the most critical part of a user's query instead of delivering equal importance to all words.
- This is effective in handling complex queries that contain specific keywords or phrases that are important for the retrieval of relevant code.
- The ability of the model to focus or 'attend' on these parts of the query helps in effective query expansion. The ECSQE model better understands what the user intends (the context) and thus results in more accurate and relevant search results for the code database.

The attention method facilitates the query expansion procedure and facilitates the model to focus on the most important parts of the query. This gives more relevant results as it targets the user's query and helps improve performance in code search as each result will answer a specific context and need of the user's query.

4.7.6. Code representation

This GloVe Code Embedding Layer takes code snippets and maps them to GloVe word embeddings to capture the semantic meaning of the code snippet. Meanwhile, the Fine-Tuned BERT Model obtains contextualized embeddings for the code snippets while considering the surrounding context.

4.7.7. Fusion of code representations(Fc)

This Fusion Layer utilizes the GloVe-based code embeddings and the contextualized BERT-based code embeddings to create a fused code representation.

4.7.8. Similarity scoring

To measure the similarity evaluation metrics like cosine similarity are used. The similarity between the enhanced query representation and the fused code representation is measured by this metric.

4.7.9. Ranking mechanism

Our ranking mechanism uses a scoring system to compare the similarity between enriched query representations to a set of different code snippets. Our query representation is constructed from combining GloVe and LSTM embeddings and our code representations are fused with the help of both GloVe and BERT embeddings. Equation (15) shows how cosine similarity is used to measure the vector closeness of the query vectors to the code snippet vectors.

Cosine similarity computes the angular difference between vectors. The higher similarity score is associated with greater relevance between the query and the code snippet. Code snippets are ranked in descending order according to their cosine similarity scores showing higher score code snippets are deemed more relevant to the expanded query. The prioritized code snippets are more likely to meet a user's search criteria due to their ranking. In essence, the model improves code search efficiency by combining the contextual and semantic capabilities of GloVe, LSTM and BERT embeddings. The suggested model also yields more correct and relevant results. The cosine similarity is calculated using the following Equation (15).

$$Cosine(F_q, F_c) \tag{15}$$

Where F_q represents the vector for the query and F_c represents the vector for the code snippet.

5. Experimental design

This section formulates the research questions, discusses comparison baseline models and defines evaluation metrics to assess model accuracy.

5.1. Research questions formulation

The research questions (RQs) outlined focus on evaluating various aspects of the proposed ECSQE model to enhance code search through query expansion. Each question targets a different dimension of the model's performance and operational characteristics:

RQ1: How does the proposed ECSQE model perform in terms of evaluation metrics compared to traditional models?

This question examines how the proposed ECSQE model stacks up against traditional models in terms of core performance metrics (accuracy, precision, recall, F1 score, MAP, MRR and NDCG). The aim is to assess whether the ECSQE model offers any significant improvements in identifying and ranking relevant code snippets when compared to older established models.

RQ2: What impact do the different embeddings (GloVe and BERT) have on the performance of the proposed model?

This research question focuses influence of different embeddings on the ECSQE model. This question seeks to understand how each type of embedding affects the model's ability to process and understand the semantics of code queries potentially impacting the overall effectiveness of the search results.

RQ3: What are the training and query response times of the proposed ECSQE model compared to existing baseline models? This question compares the ECSQE model's efficiency in terms of training time and query response time against existing baseline models. It is crucial to evaluate the practicality of the ECSQE model in a real-world environment where response time can be as critical as accuracy.

RQ4: How does the training setup (batch size, epochs, optimizer) influence the performance of the proposed model? The performance of the ECSQE model under different training setup with varying the batch

Table 1
Dataset used in Query Expansion for Source Code Retrieval.

Dataset	Programming Languages	Size	Training Data	Testing Data	Validation Data
CodeSearchNet	Python	1,418,606	1,274,986	79,787	63,833
	Java	1,560,804	1,402,976	87,682	70,146

size, the number of training epochs and the choice of optimizer is analyzed in this research question The goal is to determine the optimal training configuration that maximizes the model's effectiveness without undue computational cost.

RQ5: What are the benefits and limitations of using attention mechanisms in enhancing code search through query expansion? This question explores the use of attention mechanisms within the ECSQE model particularly how they contribute to improving the model's ability to focus on relevant parts of the query during expansion. It also addresses the potential limitations or downsides of incorporating such mechanisms like increased computational demand or overfitting.

These research questions are designed to provide a comprehensive evaluation of the ECSQE model from multiple angles. This multifaceted analysis helps in understanding not only the strengths and capabilities of the ECSQE model but also its practical limitations and areas for future improvement.

5.2. Datasets

The CodeSearchNet dataset [44,45] is used in this work to evaluate and experiment with query expansion in source code retrieval. The CodeSearchNet dataset is a popular benchmark for code search and retrieval [46]. Husain et al. [46] promote its application in cross-lingual code retrieval. This dataset includes a large number of Code Snippets from various GitHub open-source sources. It includes Python, Java, JavaScript, C++ and a few more. Each code sample in the dataset is also accompanied by pertinent metadata, such as its originating source, what the code performs and other related factors. The CodeSearchNet dataset is evaluated for multiple query expansion strategies for the accuracy of source code retrieval. Preprocessing on the dataset is performed in order to create an efficient indexing system as well as extract the most relevant features in the dataset. This made the processes of evaluation and retrieval very easy. Although the CodeSearchNet dataset is a decent starting point for research into source code retrieval but it is important to keep in mind that the dataset may be susceptible to some bias and limitation: for example, the likelihood that some programming languages or repositories will be disproportionately overrepresented in the dataset. However, these factors should be taken into account when the findings of this study are interpreted and generalized.

Table 1 presents the dataset split used for source code retrieval experiments with the CodeSearchNet dataset. CodeSearchNet contains a vast repository of source code in various programming languages. This research utilizes Python and Java code snippets. Table 1 details the number of samples available for each language and how they were partitioned into training, validation and test sets. For Python, a total of 1,418,606 samples were available. Of these, 1,274,986 were allocated for model training, 63,833 were used for validation and hyperparameter tuning during training and the remaining 79,787 were reserved for final model evaluation. For Java, 1,560,804 samples were available with 1,402,976 used for training, 70,146 for validation and 87,682 for testing. The CodeSearchNet dataset exceed 5 GB. Therefore, a subset was utilized to optimize computational resources for this research. We carefully split the dataset to ensure fair evaluation and prevent overfitting during model training and development.

The preprocessing phase used several key steps to transform raw code snippets into a suitable format for training and evaluation. First, tokenization was applied to divide the source code into individual tokens. Next, stemming reduced each token to its root form, simplifying the vocabulary by grouping tokens with similar meanings. Stopword removal was also conducted to eliminate common words that were unlikely to contribute meaningfully to code understanding. Pre-trained word embeddings were then used to embed the preprocessed code snippets, capturing the semantic relationships between different code elements. Additionally, padding and truncation strategies were implemented to manage variable-length queries, ensuring consistency in input data. This rigorous preprocessing approach ensured that the models received structured, semantically-rich input, facilitating more accurate code retrieval across diverse programming contexts.

5.3. Benchmark queries

The evaluation of the proposed approach is done by utilizing three benchmark available query sets.

- $QuerySet_{50}$: Gu et al. [47] manually compiled a query set extracted from Stack Overflow through a systematic process. This set includes the top 50 most upvoted Java programming questions that adhere to three criteria: (1) specificity, where questions pertain to distinct programming tasks, such as "How can I concatenate two arrays in Java?"; (2) adequacy of answers, requiring accepted answers to contain at least one code snippet; and (3) non-duplication, ensuring questions are not duplicates of others within the same collection.
- $QuerySet_{99}$: Husain et al. [46] developed a query set containing 99 queries extracted from the CodeSearchNet challenge dataset. These 99 queries are frequently used search queries on Bing, characterized by technical keywords and high clickthrough rates to code content.
- $QuerySet_{25}$: After conducting their respective investigations, Mishne et al. [48] and Keivanloo et al. [49] developed a set of 25 queries. It's worth noting that this query set originally relied on API names as the input for code searches. However, our study centers around on queries presented in natural language. As a result, we have adopted the natural language descriptions from this query set as inputs for our model.

5.4. Baselines

The baseline models were re-implemented and re-trained under the same experimental conditions to ensure a fair comparison with the proposed CSQE model. Specifically, the DeepCS model was reconstructed using the source code available on GitHub, allowing replication of its architecture and training procedures. For the other baselines, including CNN-CS, UNIF, CNN-CS-TS, and UNIF-TS, the implementations followed the original methodologies while ensuring consistent input representations and training setups. This uniform re-training approach ensures that all models are evaluated under identical conditions, thereby enabling a reliable assessment of comparative performance. The following baselines are used to compare the performance of the proposed CSQE model.

- **DeepCS.** This model is developed by Gu et al. [50] that employs Recurrent Neural Networks (RNN) and Multilayer Perceptron (MLP) networks to process method names, API sequences and code tokens, generating a vector representation of the code. Moreover, it utilizes RNN to create a vector representation for queries. To confirm and replicate the findings, we reconstructed DeepCS using the source code available on GitHub.[1]

[1] https://github.com/guxd/deep-code-search.

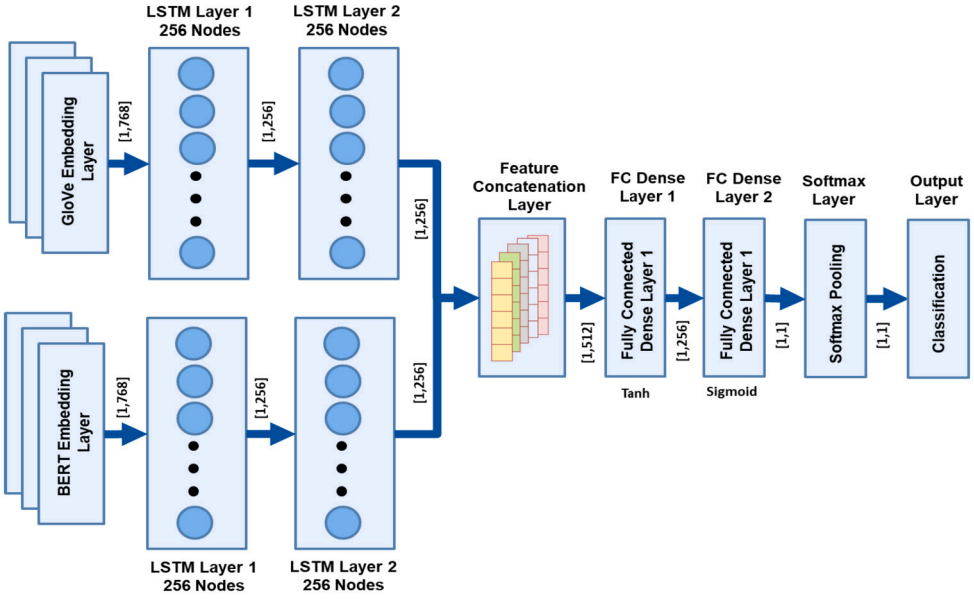


Fig. 3. ECSQE Model Network Structure.

- **CNN-CS.** A prominent deep learning-powered code search model is CNN-CS., presented by Shuai et al. [51]. CNN-CS incorporates CNN and LSTM networks, along with an attention mechanism [52]. This model necessitates creating unified code and query representations. This is accomplished by employing separate embedding processes for each, ensuring that both elements are effectively captured and related to each other in the representation space.
- **UNIF.** Cambronero et al. [53] introduced an innovative supervised code search model known as UNIF. This model employs an advanced attention-based weighting mechanism to integrate the embeddings of each token, resulting in a comprehensive embedded code sentence vector. Additionally, the description sentence embedding is generated by averaging the query embeddings, providing a cohesive representation of the query.
- **CNN-CS-TS**. This model includes structural code element, AST Shuai et al. [51]. This model takes in four code features as input which are: (i) API, (ii) the name of the method on which the request is made, (iii) the AST sequence and (iv) the code tokens. As with other features, AST is processed by the CNN network and then combined with the code vector.
- **UNIF-TS.** AST is incorporated in the existing UNIF model [54]. It investigates whether ASTs can improve code search effectiveness by also offering a more structured and detailed representation of the code.

5.5. Experimental settings

The ECSQE model is evaluated on a subset of the CodeSearchNet, a dataset of millions of code snippets containing multiple programming languages. In particular, for this research, we concentrated on two languages Python and Java. To process sequential dependencies and improve relevance to query and code elements, the attention and embeddings (BERT GloVe) serve as winning combinations together with an LSTM layer within a hybrid model of the ECSQE model.

5.6. Hyperparameter setting

The model is trained using an Adam optimizer with a learning rate of 0.001 which was chosen carefully to provide maximum performance. The model was trained for 20 epochs with a batch size of 32. A model with higher predictive accuracy was achieved by minimizing the distance between predictions and actual results using a cross-entropy loss function. By performing the architecture with two LSTM layers, having 64 hidden units each, the model was able to capture complex sequential dependencies. Since multi-class classification tasks are concerned so we chose the Softmax function for the output layer. Softmax acts as a means to convert logits to probabilities which makes it an excellent tool for classifying queries to multiple classes or categories/types for code snippets. It was executed on an NVIDIA Tesla V100 GPU with 64 GB of RAM and Intel Xeon processor which helps in efficient computing as well as faster training time.

Table 2
ECSQE model classifier architecture.

Layer	Input Shape	Output Shape	# Parameters
GloVe Embedding	[1, 768]	[1, 768]	-
BERT Embedding	[1, 768]	[1, 768]	-
LSTM-1 (256 units)	[1, 256]	[1, 256]	-
LSTM-2 (256 units)	[1, 256]	[1, 256]	-
Feature Concatenation	[1, 256+256]	[1, 512]	-
Dense Layer-1	[1, 512]	[1, 256]	1,180,416
Tanh Activation	[1, 256]	[1, 256]	-
Dense Layer-2	[1, 256]	[1, 1]	1538
Sigmoid Activation	[1, 1]	[1, 1]	-
Softmax Pooling	[1, 1]	[1, 1]	-

5.7. Network structure of ECSQE model

Table 2 presents the architecture of the ECSQE model, detailing the layers used, their input and output dimensions and the number of trainable parameters.

Fig. 3 shows the network structure of the ECSQE model. GloVe Embedding layer takes an input of size [1, 768], which corresponds to the pre-trained GloVe embedding vector size. It outputs the same dimension [1, 768], as it only provides the embeddings, without any learnable parameters at this stage. Similar to the GloVe embedding layer, this BERT embedding layer processes the input queries and code snippets with BERT embeddings. It transforms the input into a [1, 768] dimensional space, where 768 refers to the size of the embedding. Like GloVe, this layer doesn't include trainable parameters but provides meaningful contextual embeddings for downstream processing. The LSTM-1 (256 units) layer introduces the first LSTM network with 256 units. It takes the [1, 768] BERT embedding input and reduces it to [1, 256] to capture sequence patterns while reducing dimensionality. LSTMs are particularly

effective for processing sequential data, such as text or code, as they remember important dependencies across sequences. The Second LSTM-2 (256 units) layer also with 256 units processes the output of the first LSTM layer. It again processes a sequence with shape [1, 256] to capture deeper sequential features, refining the representations from the previous LSTM. The Feature Concatenation layer concatenates both the vectors generated by GloVe and BERT embeddings. The combined vector has an input size of [1, 512], derived from the sum of the two embedding sizes (256+256). This feature concatenation enriches the model by leveraging both embeddings for a more comprehensive understanding of the data. The Dense Layer-1 is the fully connected layer that takes the concatenated features of size [1, 512] and projects them down to [1, 256] using 1,180,416 trainable parameters. This layer helps the model learn the relationship between features and improves its ability to generalize. The Tanh activation function introduces non-linearity into the model, which helps it learn complex patterns in the data. It takes the output of the Dense Layer-1 ([1, 256]) and returns the same size [1, 256] while mapping the output values between -1 and 1. The Dense Layer-2 is another fully connected layer but with a single output [1, 1]. It reduces the dimensionality further using 1538 parameters. This layer compresses the input information into a single value used for binary classification or regression tasks. The Sigmoid activation function covets the output of the Dense Layer-2. This activation function converts the output into probability values ranging from 0 to 1. The ECSQE model outputs a probability for which the code snippet matches the query. After that, a Softmax pooling layer is used to normalize the output to 1 enabling the ECSQE model to classify multi-class. This mechanism for ranking code snippets is effective because the highest-ranked snippets will get the highest probabilities.

To make the ECSQE model more powerful, GloVe and BERT embedding are used. These combined embeddings are then passed to the LSTM for sequential processing. The dense layers compress the extracted sequential dependencies of the embeddings using two LSTM layers. The last layers of the neural network are the sigmoid and softmax layers. These layers transform logits into probabilities related to classification. The presented architecture allows the model to quickly and precisely learn to rank relevant code snippets to perform well at code search tasks.

5.8. Evaluation metrics

ECSQE model outcome is analyzed against a set of standard metrics outlined in the literature [55–57]. These metrics help define the effectiveness of the algorithm and allow the users to select the system that will meet their needs. The metrics we use in our work for clone detection and retrieval are described below. For clone detection, we use precision, recall, accuracy and F1-score as the metrics for comparing the performance of the existing works with ours. These metrics are defined below:

- **Accuracy:** Accuracy Wang et al. [58] measures the proportion of correctly identified relevant code snippets out of all predictions made by the model. It reflects the overall correctness of the model's predictions. Mathematically, accuracy is defined using Equation (16):

$$Accuracy = = \frac{Number\ of\ Correct\ Predictions}{Total\ Number\ of\ Predictions} = \frac{TP+TN}{TP+TN+FP+FN} \tag{16}$$

  where TP is True Positives, TN is True Negatives, FP is False Positives and FN is False Negatives.
- **Precision:** Precision is the ratio of correctly retrieved relevant code snippets (true positives) to the total number of snippets retrieved by the model. High precision indicates that the model retrieves more relevant snippets with fewer irrelevant ones. Precision is calculated using Equation (17).

$$Precision = \frac{TP}{TP+FP} \tag{17}$$

  This metric helps in understanding how many of the retrieved results are actually relevant.
- **Recall:** Recall Hu et al. [59] (or Sensitivity) quantifies the ability of the model to retrieve all relevant code snippets from the dataset. It is the ratio of correctly retrieved relevant snippets to the total number of relevant snippets in the dataset and is calculated using Equation (18).

$$Recall = \frac{TP}{TP+FN} \tag{18}$$

  Recall measures the model's effectiveness in finding all relevant instances.
- **F1-Score:** The F1-Score [60] is the harmonic mean of Precision and Recall. It provides a balanced measure that takes both precision and recall into account, particularly useful when the distribution of relevant and irrelevant snippets is uneven. The F1-Score is calculated using Equation (19).

$$F1-Score = 2 \times \frac{Precision \times Recall}{Precision + Recall} \tag{19}$$

  This score is especially valuable when a balance between precision and recall is desired.
- **Mean Reciprocal Rank (MRR):** MRR [61] is a measure of how well the model ranks the first relevant code snippet. It is the average of the reciprocal ranks of the first relevant snippet for each query and is calculated using Equation (20).

$$MRR = \frac{1}{|i|}\sum_{i=1}^{|Q|}\frac{1}{Rank_i} \tag{20}$$

  where Q is the set of queries and rank $rank_i$ is the rank position of the first relevant document for the i^{th} query. A higher MRR indicates that relevant snippets are ranked higher in the results list.
- **Mean Average Precision (MAP):** MAP [62] assesses the precision across all relevant snippets for each query and averages these values over all queries. It reflects the model's ability to rank relevant snippets higher across multiple queries. MAP is calculated using Equation (21).

$$MRR = \frac{1}{|i|}\sum_{i=1}^{|Q|}\frac{1}{AP_i} \tag{21}$$

  where AP_i is the average precision for the i^{th} query. The Average Precision (AP) for each query is calculated using Equation (22).

$$AP = \frac{\sum_{k=1}^{n} P(k) \times \mathrm{rel}(k)}{\text{Number of Relevant Documents}} \tag{22}$$

  where $P(k)$ is the precision at cut-off k in the list and $rel(k)$ is an indicator function equaling 1 if the item at rank k is relevant.
- **Normalized Discounted Cumulative Gain (NDCG):** NDCG [63] measures the ranking quality of the retrieved code snippets by considering the position of relevant snippets in the ranked list as shown in Equation (23). It discounts the importance of relevant snippets that appear lower in the list. NDCG is normalized based on the ideal ranking order, making it easier to compare across different datasets or models.

$$NDCG = \frac{DCG}{IDCG} \tag{23}$$

  where DCG (Discounted Cumulative Gain) is calculated using Equation (24).

$$\mathrm{DCG} = \sum_{i=1}^{p}\frac{2^{\mathrm{rel}_i}-1}{\log_2(i+1)} \tag{24}$$

  IDCG (Ideal Discounted Cumulative Gain) is the maximum possible DCG, representing the ideal ordering of results.

Table 3
ECSQE Comparison with Existing Baseline for Java Language.

	Model	Acuracy	Precison	Recall	F1-Score	MRR	MAP	NDCG
Query Set 99	UNIF	0.520	0.560	0.580	0.540	0.530	0.550	0.560
	CNN-CS	0.560	0.610	0.620	0.590	0.550	0.570	0.580
	DeepCS	0.500	0.540	0.550	0.530	0.510	0.530	0.540
	CNN-CS-TS	0.610	0.660	0.680	0.640	0.580	0.600	0.610
	ECSQE	**0.830**	**0.860**	**0.870**	**0.850**	**0.840**	**0.860**	**0.870**
Query Set 50	UNIF	0.530	0.570	0.590	0.550	0.540	0.560	0.570
	CNN-CS	0.570	0.620	0.630	0.600	0.560	0.580	0.590
	DeepCS	0.510	0.550	0.560	0.540	0.520	0.540	0.550
	CNN-CS-TS	0.620	0.670	0.690	0.650	0.590	0.610	0.620
	ECSQE	**0.840**	**0.870**	**0.880**	**0.860**	**0.850**	**0.870**	**0.880**
Query Set 25	UNIF	0.540	0.580	0.600	0.560	0.550	0.570	0.580
	CNN-CS	0.580	0.630	0.640	0.610	0.570	0.590	0.600
	DeepCS	0.520	0.560	0.570	0.550	0.530	0.550	0.560
	CNN-CS-TS	0.630	0.680	0.700	0.660	0.600	0.620	0.630
	ECSQE	**0.850**	**0.880**	**0.890**	**0.870**	**0.860**	**0.880**	**0.890**

6. Results of CSQE model

The results of the CSQE model in terms of the research questions (RQs) are discussed in this section.

6.1. RQ1. How does the proposed ECSQE model perform in terms of evaluation metrics compared to traditional models?

The Table 3 presents a comparison of various models including the proposed ECSQE mode across three different query sets (99, 50 and 25). The performance metrics include Accuracy, Precision, Recall, F1-Score, MRR (Mean Reciprocal Rank), MAP (Mean Average Precision) and NDCG (Normalized Discounted Cumulative Gain).

For $QuerySet_{99}$, the UNIF model shows the lowest performance among all models with an Accuracy of 0.520 and an F1-Score of 0.540. These relatively low scores suggest that UNIF struggles to accurately retrieve relevant results from the larger query set. The CNN-CS model performs better than UNIF with an Accuracy of 0.560 and F1-Score of 0.590 indicating a moderate improvement in handling code search queries. However, DeepCS performs worse than CNN-CS with an Accuracy of 0.500 and F1-Score of 0.530, highlighting potential challenges with this larger query set. Conversely, the CNN-CS-TS variant shows significant improvement, achieving an accuracy of 0.610 and an F1-Score of 0.640, demonstrating better generalization and robustness in retrieving relevant results. The proposed model, ECSQE, surpasses all other models with an accuracy of 0.830 and an F1-Score of 0.850, indicating a strong capability to handle complex and larger queries effectively. Its high precision, recall and other metrics further underscore its effectiveness.

When analyzing **QuerySet50**, UNIF's performance sees a slight improvement, reaching an accuracy of 0.530 and an F1-Score of 0.550, yet it still lags behind other models. CNN-CS performs better than UNIF, with an accuracy of 0.570 and an F1-Score of 0.600, showing enhanced results with a moderate query set size. Conversely, DeepCS continues to struggle, achieving an accuracy of 0.510 and an F1-Score of 0.540, indicating challenges with smaller query sets. On the other hand, the CNN-CS-TS model shows further improvement, attaining an accuracy of 0.620 and an F1-Score of 0.650, suggesting effective retrieval with moderate query sets. ECSQE once again leads the pack, with an accuracy of 0.840 and an F1-Score of 0.860. Its strong performance across all metrics indicates it is well-suited for handling query sets of varying sizes.

For $QuerySet_{25}$, UNIF's performance in the smallest query set is better than its performance in larger sets, with an Accuracy of 0.540 and F1-Score of 0.560. However, it still trails behind the other models. CNN-CS continues to show improvements over UNIF with an Accuracy of 0.580 and F1-Score of 0.600 indicating its robustness even with smaller query sets. DeepCS again underperforms with an Accuracy of 0.520 and F1-Score of 0.550 suggesting consistent challenges across different query set sizes. The CNN-CS-TS model performs well with an Accuracy of 0.630 and an F1-Score of 0.660 demonstrating its reliability in handling small query sets effectively. Finally, ECSQE remains the top-performing model with an Accuracy of 0.850 and an F1-Score of 0.870 demonstrating its greater ability to retrieve relevant results even from smaller query sets.

Result 1(a): *Across all query sets (99, 50 and 25), ECSQE consistently surpasses the other models. Its high accuracy, precision, recall and F1-Score indicate its exceptional effectiveness in retrieving relevant code from various query set sizes. The proposed model's performance is robust, adaptable, and well-suited for handling different query complexities.*

Table 4 shows a comparative analysis of the ECSQE model against several existing baseline models (UNIF, CNN-CS, DeepCS and CNN-CS-TS) across three different query sets (99, 50 and 25) in the context of Python language code retrieval. The performance metrics evaluated include Accuracy, Precision, Recall, F1-Score, MRR, MAP and NDCG (Normalized Discounted Cumulative Gain).

For $QuerySet_{99}$, the UNIF model achieves an Accuracy of 0.68 with modest Precision and Recall of 0.70 and 0.72 respectively indicating balanced but limited overall performance. CNN-CS performs slightly better with an Accuracy of 0.72 and F1-Score of 0.74 showing improvement in csapturing relevant code snippets. However, DeepCS underperforms with the lowest scores across all metrics, including an Accuracy of 0.65 and F1-Score of 0.67 suggesting it struggles with the complexity or size of this query set. CNN-CS-TS with an Accuracy of 0.75 and F1-Score of 0.77 demonstrates better capability in handling larger and more complex queries effectively. The proposed ECSQE model significantly outperforms all the baseline models with an Accuracy of 0.90 and F1-Score of 0.91 along with high Precision and Recall scores of 0.91 and 0.92 respectively indicating its precision in predictions and comprehensive retrieval of relevant results. The high MRR, MAP and NDCG scores further reinforce ECSQE's superiority in ranking and relevance.

For $QuerySet_{50}$, UNIF shows slight improvement with an Accuracy of 0.70 andF1-Score of 0.72 but remains one of the weaker models. CNN-CS performs better achieving an Accuracy of 0.74 and F1-Score of 0.75, indicating good balance between precision and recall for a moderate query set. However, DeepCS again underperforms with an Accuracy of 0.68 and F1-Score of 0.69 suggesting consistent limitations across different query sizes. CNN-CS-TS with an Accuracy of 0.77 and F1-Score of 0.78 shows strong performance in managing medium-sized query sets. The ECSQE model once again excels topping the performance metrics with an Accuracy of 0.91 and F1-Score of 0.92. Its Precision and Recall scores are above 0.92 demonstrate its effectiveness in handling a moderate number of queries with consistently providing accurate and relevant results.

Table 4
ECSQE Comparison with Existing Baseline for Python Language.

	Model	Accuracy	Precision	Recall	F1-Score	MRR	MAP	NDCG
Query Set 99	UNIF	0.68	0.70	0.72	0.71	0.69	0.71	0.72
	CNN-CS	0.72	0.74	0.75	0.74	0.73	0.74	0.75
	DeepCS	0.65	0.67	0.68	0.67	0.66	0.67	0.68
	CNN-CS-TS	0.75	0.77	0.78	0.77	0.76	0.77	0.78
	ECSQE	**0.90**	**0.91**	**0.92**	**0.91**	**0.90**	**0.91**	**0.92**
Query Set 50	UNIF	0.70	0.71	0.73	0.72	0.70	0.71	0.73
	CNN-CS	0.74	0.75	0.76	0.75	0.74	0.75	0.76
	DeepCS	0.68	0.69	0.70	0.69	0.68	0.69	0.70
	CNN-CS-TS	0.77	0.78	0.79	0.78	0.77	0.78	0.79
	ECSQE	**0.91**	**0.92**	**0.93**	**0.92**	**0.91**	**0.92**	**0.93**
Query Set 25	UNIF	0.72	0.73	0.74	0.73	0.72	0.73	0.74
	CNN-CS	0.76	0.77	0.78	0.77	0.76	0.77	0.78
	DeepCS	0.70	0.71	0.72	0.71	0.70	0.71	0.72
	CNN-CS-TS	0.79	0.80	0.81	0.80	0.79	0.80	0.81
	ECSQE	**0.92**	**0.93**	**0.94**	**0.93**	**0.92**	**0.93**	**0.94**

Table 5
ECSQE Model Variants Performance for Java Language.

Model	Accuracy	Precision	Recall	F1-Score	MRR	MAP	NDCG
ECSQE (Glove+LSTM)	0.82	0.84	0.85	0.83	0.81	0.82	0.83
ECSQE (Bert+LSTM)	0.85	0.87	0.88	0.86	0.84	0.85	0.86
ECSQE (Glove+Bert+LSTM)	0.87	0.89	0.90	0.88	0.86	0.87	0.88
ECSQE (Glove+Bert+LSTM+Att)	0.90	0.92	0.93	0.91	0.89	0.90	0.91

For $QuerySet_{25}$, the UNIF model performs better with an accuracy of 0.72 and an F1-Score of 0.73, but it is still outperformed by more advanced models. CNN-CS achieves an accuracy of 0.76 and an F1-Score of 0.75, demonstrating its capability to handle smaller query sets. However, DeepCS continues to struggle, showing the lowest performance with an accuracy of 0.70 and an F1-Score of 0.71. CNN-CS-TS performs well with an Accuracy of 0.79 and F1-Score of 0.80 showing strong capability in handling smaller and more straightforward queries effectively. The ECSQE model consistently leads with the highest Accuracy of 0.92 and F1-Score of 0.93. The Precision and Recall scores of 0.93 and 0.94 highlight its exceptional ability to retrieve relevant code snippets even in a smaller query set. The high MRR, MAP and NDCG scores confirm its effectiveness in ranking and overall relevance.

Result 1(b): *Overall, across all query sets the ECSQE model consistently demonstrates higher performance compared to the other baseline models. It achieves the highest scores in all key metrics indicating that it is particularly well-suited for code search tasks, regardless of the query set size. The ECSQE model's architecture likely leveraging advanced techniques such as query expansion and attention mechanisms allows it to deliver precise, relevant and highly ranked results across different query complexities.*

6.2. RQ2. What impact do the different embeddings (GloVe and BERT) have on the performance of the proposed model?

Table 5 shows evaluation results for different variants of the proposed ECSQE model for the Java language. As the complexity of the model increases, the performance improves across all metrics. The base model which uses GloVe embeddings with an LSTM achieves an accuracy of 0.82 with precision and recall values of 0.84 and 0.85, respectively. However, when BERT embeddings are introduced in the model (ECSQE (Bert+LSTM)), a noticeable improvement occurs. The accuracy increases to 0.85 and both precision and recall rise to 0.87 and 0.88, respectively. This indicates that BERT embeddings capture rich contextual information that contributes to a more accurate and reliable model.

Further enhancements are seen when both GloVe and BERT embeddings are combined in the ECSQE (GloVe+Bert+LSTM) model. The accuracy, precision and recall climb to 0.87, 0.89 and 0.90, respectively. This suggests that the combined use of these embeddings allows the model to capture complementary information leading to better performance. The most advanced model that includes an attention mechanism (ECSQE (GloVe+Bert+LSTM+Att)) achieves the highest accuracy of 0.90 with precision and recall reaching 0.92 and 0.93, respectively. The attention mechanism likely helps the model focus on the most relevant parts of the input sequence to further refine its predictions.

Moreover, the F1-Score, MRR, MAP and NDCG metrics exhibit similar upward trends. The F1-Score improves from 0.83 in the base model to 0.91 in the most advanced model, reflecting a better balance between precision and recall. The ranking metrics (MRR, MAP, NDCG) also show substantial improvements with the best scoring of 0.91, 0.89 and 0.91, respectively. These results indicate that the advanced ECSQE model not only predicts relevant code snippets more accurately but also ranks them higher in the list of search results. The results for the Python language query set mirror the trends observed in the Java language with performance improving as the model becomes more sophisticated as shown in Table 6. The base model (ECSQE (GloVe+LSTM)) starts with an accuracy of 0.81, a precision of 0.83 and a recall of 0.84. When BERT embeddings replace GloVe in the ECSQE (Bert+LSTM) model, the accuracy increases to 0.84 whereas the precision and recall rise to 0.86 and 0.87, respectively. This confirms the effectiveness of BERT embeddings in improving model performance by capturing deeper contextual nuances in the code.

When both GloVe and BERT embeddings are combined (ECSQE (GloVe+Bert+LSTM)), the model achieves an accuracy of 0.87, with a precision of 0.89 and a recall of 0.88. The addition of the attention mechanism in the ECSQE (GloVe+Bert+LSTM+Att) model further enhances the performance, pushing the accuracy to 0.89 and the precision and recall to 0.91 and 0.92, respectively. The improvements across these metrics suggest that the attention mechanism helps the model better identify and prioritize the most relevant parts of the input leading to more accurate predictions.

The F1-Score follows a similar pattern and increases from 0.82 in the base model to 0.90 in the most advanced model. This consistent improvement indicates that the model becomes better at balancing precision and recall as it incorporates more sophisticated components. The ranking metrics (MRR, MAP, NDCG) also show significant improvements with the advanced model scoring 0.88, 0.89 and 0.90, respec-

Table 6
ECSQE Model Variants Performance for Python Language.

Model	Accuracy	Precision	Recall	F1-Score	MRR	MAP	NDCG
ECSQE (Glove+LSTM)	0.81	0.83	0.84	0.82	0.80	0.81	0.82
ECSQE (Bert+LSTM)	0.84	0.86	0.87	0.85	0.83	0.84	0.85
ECSQE (Glove+Bert+LSTM)	0.86	0.88	0.89	0.87	0.85	0.86	0.87
ECSQE (Glove+Bert+LSTM+Att)	0.89	0.91	0.92	0.90	0.88	0.89	0.90

Table 7
Training Time Cost for ECSQE Model in Comparison with existing baseline.

Model	Training	Query Time (sec)
UNIF	3.2 hours	0.3
CNN-CS	13.2 hours	0.5
DeepCS	35.6 hours	1.1
CNN-CS -TS	15.7 hours	0.6
ECSQE	**6.2 hours**	**0.4**

tively. These results suggest that the advanced ECSQE model not only excels at identifying relevant Python code snippets but also ranks them more effectively ensuring that the most relevant results appear at the top.

Result 2: *The evaluation confirms that introducing BERT embeddings and an attention mechanism helps the ECSQE model improve the ranking of Java and Python code search. BERT embeddings considerably enhance the results by increasing the precision and recall. The combination of both GloVe and BERT embeddings increases the overall performance across all metrics showing that the different embeddings techniques contain entirely different information. The highest performance proves the significance of the attention mechanism to focus on important regions of the input sequence. Improvements in all aspects also depict the fact that the advanced ECSQE model is reliable and efficient and makes it a perfect candidate to undertake code search in other languages too.*

6.3. RQ3. What are the training and query response times of the proposed ECSQE model compared to existing baseline models?

The Table 7 compares the training time and query response time of the proposed ECSQE model against several existing baseline models: UNIF, CNN-CS, DeepCS and CNN-CS-TS. These metrics are crucial for evaluating the model's efficiency and practicality in real-world applications.

Regarding **training time**, UNIF has the shortest training duration at 3.2 hours making it the fastest model to train among the baselines. CNN-CS requires 13.2 hours for training indicating a moderate increase in complexity compared to UNIF. DeepCS has the longest training time at 35.6 hours reflecting its high computational cost and complexity. CNN-CS-TS requires 15.7 hours to train that is slightly more than CNN-CS but less than DeepCS. The proposed ECSQE model takes 6.2 hours for training. This is considerably shorter than CNN-CS, DeepCS and CNN-CS-TS but longer than UNIF suggesting that ECSQE achieves a good balance between complexity and training efficiency, making it more feasible for practical use without the extensive training time required by DeepCS.

In terms of **query response time**, UNIF again proves to be the most efficient with the fastest response time at 0.3 seconds making it highly responsive for query processing. CNN-CS has a slightly slower query time of 0.5 seconds. DeepCS reflecting its complexity and has the longest query response time of 1.1 seconds that could be a drawback in real-time applications. CNN-CS-TS has a query time of 0.6 seconds that is moderate but still slower than UNIF and CNN-CS. The proposed ECSQE model has a query response time of 0.4 seconds placing it second only to UNIF in terms of speed. This quick response time combined with its relatively shorter training time that suggests ECSQE model is efficient both in training and real-time application.

Result 3: *The ECSQE model demonstrates a strong balance between training time and query response time. While it is not as fast as UNIF in training. ECSQE model significantly reduces the training time compared to more complex models like DeepCS and CNN-CS-TS. In terms of query response time, ECSQE model is highly efficient and faster than most baselines. This makes ECSQE model a highly practical model offering robust performance improvements without the prohibitive training and query costs associated with some of the more complex baseline models.*

6.4. RQ4. How does the training setup (batch size, epochs, optimizer) influence the performance of the proposed model?

The Table 8 reveals how different batch sizes impact the performance of the ECSQE model. As the batch size increases, the model's accuracy generally improves peaking at a batch size of 64 with an accuracy of 0.80. Precision, recall and F1-Score also follow this trend reaching their highest values of 0.83, 0.84 and 0.83 respectively at a batch size of 64. This suggests that a moderate batch size allows the model to learn effectively without the noise of too small or too large updates. However, when the batch size is further increased to 128 the performance slightly decreases across all metrics indicating that too large of a batch size might lead to less effective updates possibly due to averaging out the gradient too much. The ranking metrics, MRR, MAP and NDCG, follow a similar pattern with the best performance also observed at a batch size of 64.

Table 9 illustrates how varying the number of training epochs affects the ECSQE model's performance. Initially as the number of epochs increases from 10 to 30, there is a significant improvement in all metrics. The accuracy increases from 0.72 to 0.82, while precision and recall improve from 0.75 and 0.76 to 0.85 and 0.86 respectively. The F1-Score, MRR, MAP and NDCG also see substantial gains during this period indicating that the model is effectively learning and refining its predictions with more training. However, after 30 epochs, the performance plateaus or slightly decreases when training continues to 40 epochs. This suggests that after a certain point, the model might start overfitting to the training data, leading to a small decline in performance on the evaluation metrics.

Table 10 compares the effects of different optimizers on the ECSQE model's performance. Among the four optimizers (SGD, Adam, RMSprop and Adagrad) Adam shows the best overall performance with an accuracy of 0.82, precision of 0.85, recall of 0.86 and F1-Score of 0.84. The ranking metrics (MRR, MAP, NDCG) also favor Adam indicating that this optimizer helps the model converge more effectively to a better local minimum. RMSprop also performs well particularly in precision and recall but slightly lags behind Adam in most metrics. SGD, on the other hand, has the lowest performance across all metrics with an accuracy of 0.75 and F1-Score of 0.77 suggesting that it might not be the most suitable optimizer for the proposed ECSQE model. Adagrad performs moderately better than SGD but not as well as Adam or RMSprop which indicates it might be useful in some contexts but is not the best choice overall for this specific model.

Result 4: *The optimal configuration for the ECSQE model is achieved using the Adam optimizer, trained over 30 epochs with a batch size of 64. This setup delivers the highest performance across all evaluation metrics including accuracy, precision, recall, F1-Score and ranking measures like MRR, MAP and NDCG. Selecting these parameters ensures efficient learning and effective convergence that maximizes the model's capability for accurate and reliable code search results.*

Table 8
Influence of Batch Size on ECSQE Model Performance.

Batch Size	Accuracy	Precision	Recall	F1-Score	MRR	MAP	NDCG
16	0.75	0.78	0.80	0.79	0.77	0.76	0.78
32	0.78	0.80	0.82	0.81	0.79	0.78	0.80
64	0.80	0.83	0.84	0.83	0.81	0.80	0.82
128	0.77	0.80	0.81	0.79	0.78	0.77	0.79

Table 9
Influence of Epochs on ECSQE Model Performance.

Epochs	Accuracy	Precision	Recall	F1-Score	MRR	MAP	NDCG
10	0.72	0.75	0.76	0.74	0.73	0.72	0.74
20	0.78	0.80	0.82	0.80	0.79	0.78	0.80
30	0.82	0.85	0.86	0.84	0.83	0.82	0.84
40	0.80	0.82	0.83	0.81	0.80	0.79	0.81

Table 10
Influence of Optimizer on ECSQE Model Performance.

Optimizer	Accuracy	Precision	Recall	F1-Score	MRR	MAP	NDCG
SGD	0.75	0.77	0.79	0.77	0.76	0.75	0.77
Adam	0.82	0.85	0.86	0.84	0.83	0.82	0.84
RMSprop	0.80	0.83	0.84	0.82	0.81	0.80	0.82
Adagrad	0.78	0.81	0.82	0.80	0.79	0.78	0.80

Table 11
ECSQE Model Performance Comparison of Different Optimizers with and without Attention Mechanisms (ATT Represents with attention and W/ATT Representation.

Optimizer	Accuracy	Precision	Recall	F1-Score	MRR	MAP	NDCG
SGD-ATT	0.78	0.80	0.82	0.80	0.79	0.78	0.80
SGD- W/ATT	0.75	0.77	0.79	0.77	0.76	0.75	0.77
Adam-ATT	0.85	0.88	0.89	0.87	0.86	0.85	0.87
Adam- W/ATT	0.82	0.85	0.86	0.84	0.83	0.82	0.84
RMSprop-ATT	0.83	0.86	0.87	0.85	0.84	0.83	0.85
RMSprop- W/ATT	0.80	0.83	0.84	0.82	0.81	0.80	0.82
Adagrad-ATT	0.81	0.84	0.85	0.83	0.82	0.81	0.83
Adagrad- W/ATT	0.78	0.81	0.82	0.80	0.79	0.78	0.80

6.5. RQ5. What are the benefits and limitations of using attention mechanisms in enhancing code search through query expansion?

Table 11 presents a comparison of the ECSQE model's performance using different optimizers with and without attention mechanisms (denoted as ATT and W/ATT respectively). The results illustrate the impact of incorporating attention mechanisms on various performance metrics including accuracy, precision, recall, F1-Score and ranking metrics such as MRR, MAP and NDCG.

The **SGD optimizer** [64] with attention (SGD-ATT) reduces the training loss significantly compared to SGD optimizer without attention (SGD-W/ATT). In particular, accuracy rises from 0.75 to 0.78 and F1-Score from 0.77 to 0.80. MRR, MAP and NDCG also have slight improvements in ranking metrics. This shows that attention mechanisms can influence the model to be more focused on more salient parts of the data. The attention mechanism help improve the performance of the SGD optimizer.

The **Adam optimizer** [65] with attention (Adam-ATT) achieves a substantial performance as compared with Adam optimizer without attention (Adam- W/ATT). The Adam-ATT achieves the highest accuracy (0.85) and F1-Score (0.87) for all configurations and dramatically outperforms on precision, recall and ranking metrics as compared with Adam-W/ATT. The results demonstrate that the Adam optimizer together with attention mechanisms improves the model's ability to accurately match queries with relevant code snippets.

We observed that **RMSprop optimizer** combined with attention mechanisms (RMSprop-ATT) leads to improved model performance. We observe increases in accuracy, precision, recall and F-Score compared to RMSprop without attention (RMSprop-W/ATT). The ranking metrics additionally become better with attention mechanisms indicating their positive contribution to the model's ability to rank more relevant code snippets.

The attention mechanisms with **Adagrad optimizer** [64] (Adagrad-ATT) improve all performance measures relative to using Adagrad without attention (Adagrad-W/ATT). The improvements achieved with the Adagrad optimizer are not as significant as those with Adam and RMSprop but the result shows gains in accuracy, F1-Score and ranking metrics indicate the value of attention mechanisms.

The attention mechanisms with Adagrad optimizer (Adagrad-ATT) improve all performance measures relative to using Adagrad without attention (Adagrad-W/ATT). The improvements achieved with Adagrad optimizer are not as significant as those with Adam and RMSprop but the result shows gains in accuracy, F1-Score and ranking metrics indicate the value of attention mechanisms.

Result 5: *Incorporating attention mechanisms generally enhances the performance of the ECSQE model across all tested optimizers. The improvements are particularly significant with the Adam optimizer which shows the highest overall performance when combined with attention mechanisms. However, while the benefits of using attention mechanisms are evident but the extent of improvement varies depending on the optimizer. With Adam showing the most substantial gains followed by RMSprop and Adagrad. Despite these ben-*

```
Query: "Sort a list of numbers in descending order"
```

Python Code Snippet

```python
def sort_descending(numbers):
    # Sorts a list of numbers in descending order
    sorted_numbers = sorted(numbers, reverse=True)
    return sorted_numbers

numbers = [5, 3, 9, 1, 10]
print(sort_descending(numbers))
```

Java Code Snippet

```java
import java.util.Arrays;
import java.util.Collections;
import java.util.List;

public class SortListDescending {
  public static void main(String[] args) {
   List<Integer> numbers = Arrays.asList(5, 3, 9, 1, 10);
        Collections.sort(numbers,
Collections.reverseOrder());
        System.out.println(numbers);
    }
}
```

Fig. 4. Query and Corresponding Code Snippets in Python and Java for Sorting a List of Numbers in Descending Order.

efits, attention mechanisms may add computational complexity and may not always result in large improvements depending on the base optimizer used.

7. Case study

In this case study, we examine how different models including BERT, GloVe and the proposed ECSQE model process a specific query to retrieve relevant code snippets. By focusing on the query "Sort a list of numbers in descending order," we can observe how each model interacts with and evaluates the query terms relative to the code snippets. Heat maps for each model are presented to visually represent the interaction between the query and the code, offering insights into how each model assigns importance to various words and code elements. These visualizations help us to understand the relevant evaluation mechanisms of the models. Fig. 4 displays the query alongside relevant code snippets in Python and Java.

7.1. Query-code heat maps (Python language)

The heat maps shown in Fig. 5 illustrate the interaction between a query and Python code tokens using two different embedding techniques: GloVe and BERT. These visualizations help to compare how each embedding method captures the semantic similarities between elements of a search query and corresponding code tokens.

Fig. 5a the GloVe heat map exhibits generally low variation in similarity scores with many cells showing zero or near-zero values. This pattern suggests that GloVe is a static word embedding model that might not effectively capture the contextual nuances essential for tasks that require a deep understanding of context such as code search. However, certain tokens like "sort," "a," and "order" show some higher scores with specific Python code tokens such as "return" and "sorted." These instances indicate a basic level of semantic recognition by GloVe though its performance is limited due to the static nature of the embeddings.

In contrast, the BERT heat map shows a richer variety of higher similarity scores indicating that BERT's context-aware embeddings provide a more nuanced understanding of the relationships between the query and the code tokens as shown in Fig. 5b. Notably, the query token "numbers" displays strong similarity scores with multiple Python code tokens including "sorted(numbers)." This highlights BERT's ability to recognize the relevance of context in which terms are used within both the query and the code. Overall, BERT demonstrates a superior capability to discern contextual meanings which likely translates to better performance in matching and retrieving relevant code snippets.

7.2. Query-code heat maps (Java language)

Fig. 6 represents heat maps that compare the interaction between query tokens and Java code tokens using two different embedding techniques (GloVe and BERT). Each map gives insights into how each embedding model captures semantic relationships in the context of a code search query. Fig. 6a represents the GloVe heat map that displays similarity scores that focused only on the first Java code token ("import") with all other tokens showing zero interaction. This pattern indicates a limited capacity of GloVe embeddings to capture nuanced or context-specific relationships necessary for accurate code retrieval whereas some query tokens like "order" and "list" have higher scores (0.43 and 0.34, respectively). These scores are confined to the interaction with "import," suggesting that GloVe recognizes some semantic relationships based on its training corpus but fails to link these tokens effectively with the broader Java code context.

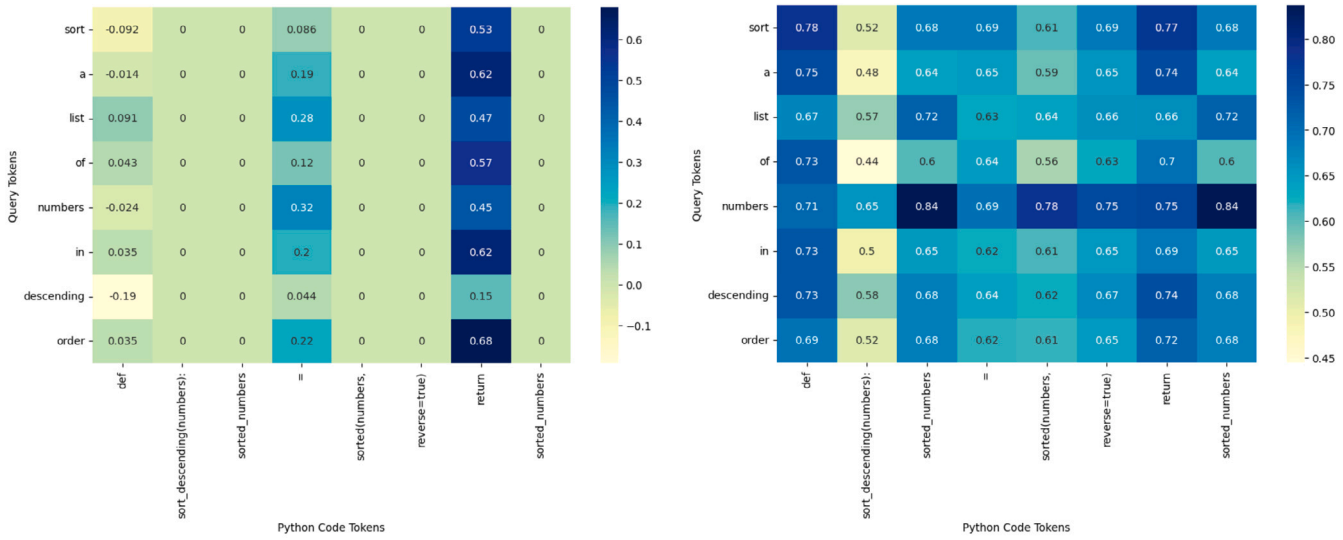
In contrast, the BERT heat map shows a much richer and more varied set of similarity scores across different Java code tokens as shown in Fig. 6b. This broad range of substantial scores suggests that BERT's context-aware embeddings are more adept at discerning the relevance of code tokens in relation to the query. The interaction is particularly strong for the query token "numbers," which achieves a peak similarity score of 0.8 with "Collections.sort(numbers)." This strong contextual alignment indicates that BERT effectively maps query semantics to relevant parts of the Java code, demonstrating its robust capability in handling tasks where the context significantly influences relevance.

7.3. ECSQE model heat maps for query-code pairs (Java and Python language)

The heat maps in Fig. 7 demonstrate the ECSQE model's interactions between query tokens and code tokens for Java and Python displaying how effectively the model captures semantic relationships crucial for code retrieval tasks. These visualizations reflect the model's capability to enhance the search functionality by understanding both the syntactic and semantic nuances of code.

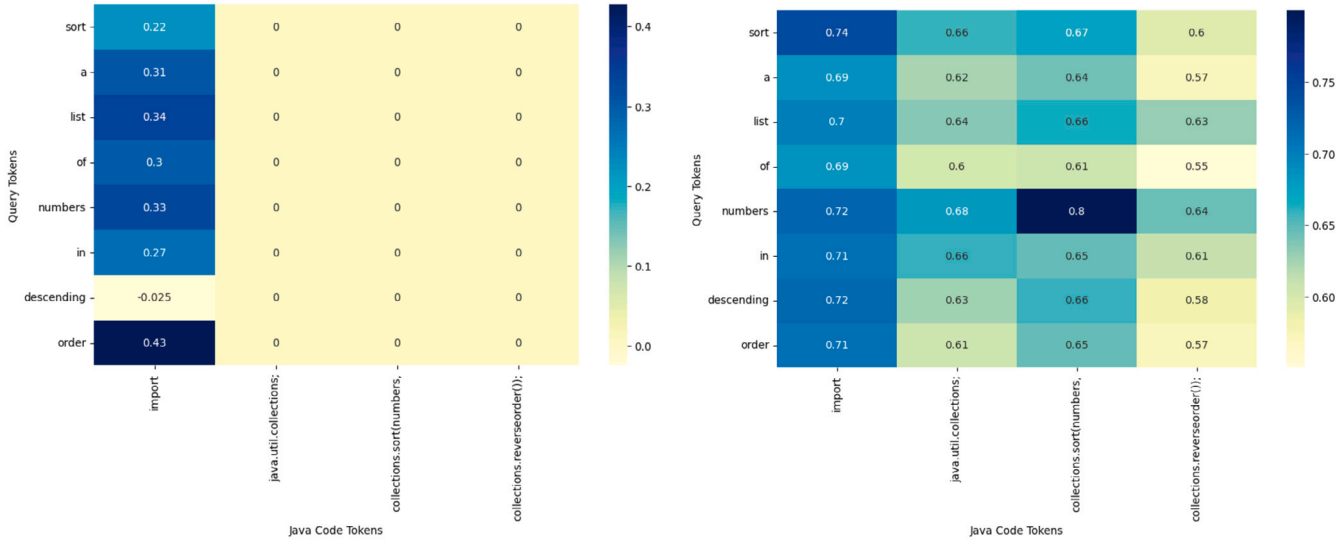
Heat map in Fig. 7a exhibits high similarity scores across most query-code token pairs often reaching the maximum of 1.00 where as many others hovering around 0.73 to 0.76. This high uniformity and consistency across the board suggest a robust understanding of code semantics that is uniformly applied across different types of tokens. Unlike the GloVe model that showed sparse and focused interactions, the ECSQE model ensures a broad interaction landscape where almost every query token is significantly similar to multiple code tokens indicating an effective capture of the code's syntactic and semantic relationships.

Fig. 7b represents Python interaction heat map that demonstrates strong and context-sensitive interactions particularly emphasizing key



(a) GloVe - Query and Python Code Token Interaction Heatmap (b) BERT - Query and Python Code Token Interaction Heatmap

Fig. 5. Glove and BERT heat maps for Python Language Query Code pair.



(a) GloVe - Query and Java Code Token Interaction Heatmap (b) BERT - Query and Java Code Token Interaction Heatmap

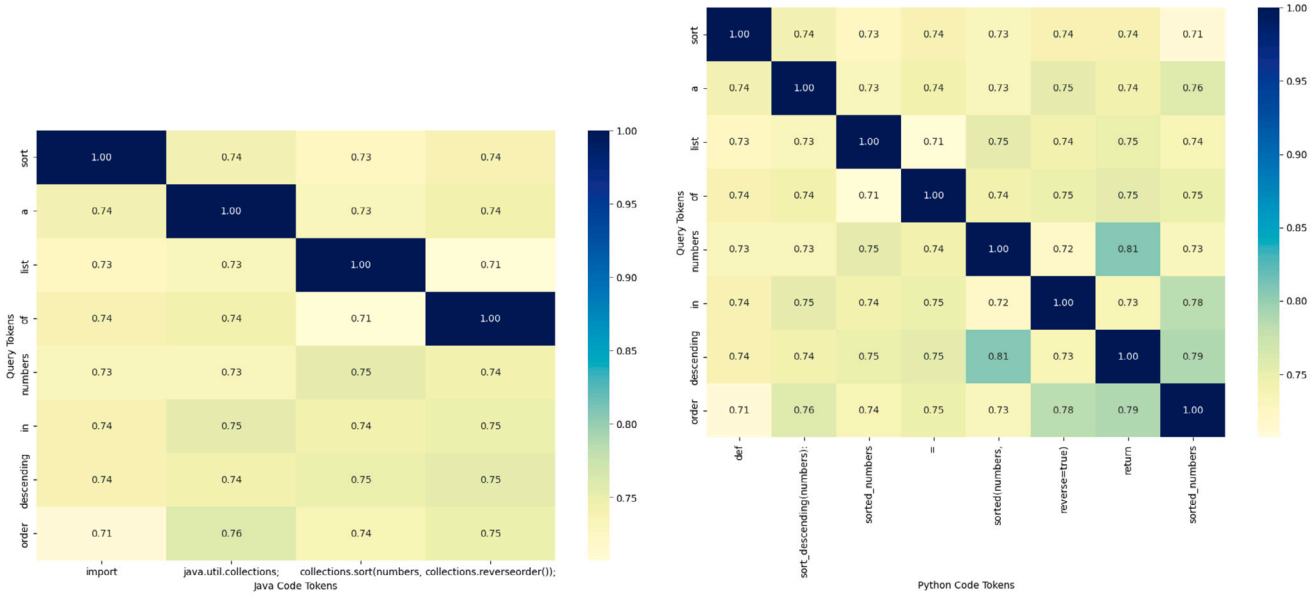
Fig. 6. Glove and BERT heat maps for Java Language Query Code pair.

tokens such as “sort,” “numbers,” and “reverse.” The model not only displays strong similarity scores but also adjusts these scores based on contextual relevance such as higher scores for “numbers” with “sorted(numbers)” and “reverse.” This demonstrates the ability of the model to find the most applicable tokens for each query and most notably identify important aspects of the code corresponding to the query.

The ECSQE model provides more contextual awareness as compared to GloVe that provides static representations and hence find it difficult to provide varying semantic roles of a token in different contexts. ECSQE model differs from Glove because it dynamically adapts each query-code interaction to the specifics of interaction and provides a more semantic mapping across a larger range of tokens. The dynamic adaptation of ECSQE enables it to overcome the traditional limitations of static embeddings by handling each query and code pair differently.

The ECSQE model potentially improves focus on code-specific elements compared to BERT. However, BERT is good at contextual sensitivity. ECSQE can make it better by tuning most specifically to code-related contexts and nuances by integrating domain-specific adjustments to improve code retrieval performance. To meet programming languages’ structural and syntactic needs, the ECSQE model narrows semantic depth by using BERT. It also leads to broader syntactic understanding to extend their capabilities to programming language beyond typical natural language processing.

Most importantly, the ECSQE model is built on ideas from both GloVe and BERT embeddings making the model suitable for code search problems. By combining these two effective embedding methods (GloVe and BERT), ECSQE not only expresses the entire range of semantic and syntactic features but also improves its flexibility and effectiveness in associating query intent with potential code samples. This combination assures ECSQE provides a better view of both the semantic and structural aspects of code as well as affirms the efficiency of the approach in various forms of programming queries. This ability makes the ECSQE model a better tool for code search applications.



(a) ECSQE Model - Query and Java Code Token Interaction Heatmap

(b) ECSQE Model - Query and Python Code Token Interaction Heatmap

Fig. 7. ECSQE model heat maps for Java and python Query Code pair.

8. Discussion

8.1. Why does CSQE model work?

ECSQE model is effective because it applies state-of-the-art NLP techniques to enhance code snippet search relevancy. ECSQE model improves code search by semantically enriching the query intent by adding terms to the initial query. This expansion tends to include additional code possibilities that do not explicitly match the query terms but have contextual relevance. The model employs a fine-grained relevance assessment that considers both terms of the query and the structure of the code, thus retrieving code snippets that more adequately correspond to the needs of the developer. The ECSQE model focuses on the combination of query expansion and effective ranking, which allows to perform better at identifying and retrieving code snippets to fulfill the user's query intentions than earlier models.

8.2. Why is ECSQE model fast?

The ECSQE model is fast due to the optimization carried out on the query expansion and retrieval mechanism. As opposed to some traditional models that will scan an entire codebase, the ECSQE model is selective in expanding queries in a targeted approach by having a strong relevance that reduces the search space. In addition, pre-trained embeddings (such as GloVe or BERT) are employed to efficiently understand natural language queries. In addition, the design strategy of ECSQE considers scalability so that regardless of the size of the code base, the performance should be relatively good with minimal degradation in speed. Our code is available at https://github.com/nazia-phd/ECSQE.

8.3. Validity threats to CSQE model

Validity threats to the ECSQE model that could affect its generalizability and effectiveness are as follows:

- **Limited Language Testing:** The model has only been tested on Python and Java, which raises concerns about its ability to generalize across other programming languages with different syntactical rules and conventions.
- **Query Test Set:** The model performance is tested using only three query datasets which may limit its ability to generalize to a wider range of queries or programming languages. Future evaluations with more diverse and larger datasets are necessary to ensure its robustness across various domains.
- **Limited Word Embedding Techniques:** The ECSQE model relies on only two embedding techniques, GloVe and BERT, which may limit its generalizability. Expanding to additional embedding methods in the future could enhance its performance across diverse queries and programming languages.

9. Conclusion and future work

The ECSQE model is introduced in this study which improves code search through the use of query expansion methods and attention mechanisms. The ECSQE model is effective for code search because it uses GloVe and BERT embeddings in conjunction to provide more semantic awareness and flexibility to adapt to many different code-related questions. This model seriously improves the accuracy and speed of code retrieval procedures to suit the needs of the complicated nature of today's programming venues. The experiments showed considerable improvements in accuracy, precision and recall over the baseline models, especially when using more sophisticated embeddings such as BERT and GloVe along with LSTM and attention mechanism in the proposed ECSQE model.

Future work will focus on addressing the identified validity threats, such as exploring additional programming languages beyond Python and Java, expanding the diversity of training datasets and incorporating more word embedding techniques. Additionally, scalability will be a key focus with efforts to optimize ECSQE for handling larger codebases and queries that are more complex efficiently.

CRediT authorship contribution statement

Nazia Bibi: Writing – review & editing, Writing – original draft, Validation, Software, Methodology, Investigation, Formal analysis, Data curation, Conceptualization. **Muhammad Usman Tariq:** Writing – review & editing, Validation, Supervision, Project administration. **Zabeeh**

Ullah: Writing – review & editing, Validation, Supervision, Project administration. **Muhammad Babar:** Writing – review & editing, Validation, Supervision, Project administration. **Zahid Khan:** Writing – review & editing, Validation, Supervision, Project administration.

Declaration of competing interest

Authors have no conflict of interest to disclose.

Acknowledgement

The authors would like to acknowledge the support of Prince Sultan University for the payment of Article Processing Charges for this research work.

Data availability

Data will be made available on request.

References

[1] Z. Anwar, N. Bibi, A. Ahsan, The future of software engineering: a survey, ACM SIGSOFT Softw. Eng. Notes 39 (2014) 1–3.
[2] J. Liu, S. Kim, V. Murali, S. Chaudhuri, S. Chandra, Neural query expansion for code search, in: Proceedings of the 3rd ACM Sigplan International Workshop on Machine Learning and Programming Languages, 2019, pp. 29–37.
[3] L. Massai, Evaluation of semantic relations impact in query expansion-based retrieval systems, Knowl.-Based Syst. 283 (2024) 111183.
[4] N. Bibi, T. Rana, A. Maqbool, et al., A pragmatic decision making mechanism for reusable source code snippets retrieval using semantic approach, in: 2022 19th International Bhurban Conference on Applied Sciences and Technology (IBCAST), IEEE, 2022, pp. 231–236.
[5] M. Imran, S.H. Qureshi, A.H. Qureshi, N. Almusharraf, Classification of English words into grammatical notations using deep learning technique, Information 15 (2024) 801.
[6] Q. Huang, Y. Yang, M. Cheng, Deep learning the semantics of change sequences for query expansion, Softw. Pract. Exp. 49 (2019) 1600–1617.
[7] C. Wang, Z. Nong, C. Gao, Z. Li, J. Zeng, Z. Xing, Y. Liu, Enriching query semantics for code search with reinforcement learning, Neural Netw. 145 (2022) 22–32.
[8] L. Wang, N. Yang, F. Wei, Query2doc: query expansion with large language models, arXiv preprint, arXiv:2303.07678, 2023.
[9] Z. Zheng, K. Hui, B. He, X. Han, L. Sun, A. Yates, Contextualized query expansion via unsupervised chunk selection for text retrieval, Inf. Process. Manag. 58 (2021) 102672.
[10] N. Bibi, A. Maqbool, T. Rana, Exploring the effectiveness of joint bi-lstm-gnn vs chatgpt-llm in source code retrieval, in: 2023 20th International Bhurban Conference on Applied Sciences and Technology (IBCAST), IEEE, 2023, pp. 321–326.
[11] J. Wang, J.X. Huang, X. Tu, J. Wang, A.J. Huang, M.T.R. Laskar, A. Bhuiyan, Utilizing bert for information retrieval: survey, applications, resources, and challenges, ACM Comput. Surv. 56 (2024) 1–33.
[12] M. Younas, D.A.S. El-Dakhs, Y. Jiang, A comprehensive systematic review of ai-driven approaches to self-directed learning, IEEE Access (2025).
[13] H.N.T. Al-Azzawi, A. Ghandour, H. Ali, A.T. Azar, N. Althuniyan, I.K. Ibraheem, Y.I. Hammadi, A.J. Humaidi, Z. Haider, S. Ahmed, Utilization of a deep convolutional neural network for the binary classification of chest X-ray pneumonia, Eng. Technol. Appl. Sci. Res. 15 (2025) 20471–20483.
[14] E. Niazmand, Enhancing query answer completeness with query expansion based on synonym predicates, in: Companion Proceedings of the Web Conference 2022, 2022, pp. 354–358.
[15] Y.H. Farhan, S.A. Mohd Noah, M. Mohd, J. Atwan, Word-embedding-based query expansion: incorporating deep averaging networks in Arabic document retrieval, J. Inf. Sci. 49 (2023) 1168–1186.
[16] N. Nagpal, Query expansion for information retrieval using word embeddings: a comparative study, in: 2022 2nd International Conference on Technological Advancements in Computational Sciences (ICTACS), IEEE, 2022, pp. 289–293.
[17] I.C. Irsan, T. Zhang, F. Thung, K. Kim, D. Lo, Picaso: enhancing api recommendations with relevant stack overflow posts, in: 2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR), IEEE, 2023, pp. 92–103.
[18] F. Zhang, H. Niu, I. Keivanloo, Y. Zou, Expanding queries for code search using semantically related api class-names, IEEE Trans. Softw. Eng. 44 (2017) 1070–1082.
[19] Q. Zou, C. Zhang, Query expansion via learning change sequences, Int. J. Knowl. Based Intell. Eng. Syst. 24 (2020) 95–105.
[20] Y. Gao, Y. Xu, Y. Li, Pattern-based topic modelling for query expansion, in: AusDM, 2014, pp. 165–173.
[21] D. Jiang, L. Yang, Query intent inference via search engine log, Knowl. Inf. Syst. 49 (2016) 661–685.
[22] P. Ren, Z. Chen, J. Ma, S. Wang, Z. Zhang, Z. Ren, Mining and ranking users' intents behind queries, Inf. Retr. J. 18 (2015) 504–529.
[23] H. Wu, Y. Yang, Code search based on alteration intent, IEEE Access 7 (2019) 56796–56802.
[24] S. Jain, K. Seeja, R. Jindal, A fuzzy ontology framework in information retrieval using semantic query expansion, Int. J. Inf. Manag. Data Insights 1 (2021) 100009.
[25] O. Corcho, F. Priyatna, D. Chaves-Fraga, Towards a new generation of ontology based data access, Semant. Web 11 (2020) 153–160.
[26] Y. Wang, H. Huang, C. Feng, Query expansion based on a feedback concept model for microblog retrieval, in: Proceedings of the 26th International Conference on World Wide Web, 2017, pp. 559–568.
[27] J. Singh, A. Sharan, Relevance feedback-based query expansion model using ranks combining and word2vec approach, IETE J. Res. 62 (2016) 591–604.
[28] B. Chen, Z. Abedjan, Interactive cross-language code retrieval with auto-encoders, in: 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, 2021, pp. 167–178.
[29] B. Xu, Z. Xing, X. Xia, D. Lo, Q. Wang, S. Li, Domain-specific cross-language relevant question retrieval, in: Proceedings of the 13th International Conference on Mining Software Repositories, 2016, pp. 413–424.
[30] W. Sun, C. Fang, Y. You, Y. Miao, Y. Liu, Y. Li, G. Deng, S. Huang, Y. Chen, Q. Zhang, et al., Automatic code summarization via chatgpt: how far are we?, arXiv preprint, arXiv:2305.12865, 2023.
[31] M. Geng, S. Wang, D. Dong, H. Wang, G. Li, Z. Jin, X. Mao, X. Liao, Large language models are few-shot summarizers: multi-intent comment generation via in-context learning, in: Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, 2024, pp. 1–13.
[32] S. Gao, W. Mao, C. Gao, L. Li, X. Hu, X. Xia, M.R. Lyu, Learning in the wild: towards leveraging unlabeled data for effectively tuning pre-trained code models, in: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, 2024, pp. 1–13.
[33] S. Gao, X.-C. Wen, C. Gao, W. Wang, H. Zhang, M.R. Lyu, What makes good in-context demonstrations for code intelligence tasks with llms?, in: 2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, 2023, pp. 761–773.
[34] J. Singh, A. Sharan, M. Saini, Term co-occurrence and context window-based combined approach for query expansion with the semantic notion of terms, Int. J. Web Sci. 3 (2017) 32–57.
[35] J. Singh, R. Kumar, Lexical co-occurrence and contextual window-based approach with semantic similarity for query expansion, Int. J. Intell. Inf. Technol. (IJIIT) 13 (2017) 57–78.
[36] Z. Gong, C.W. Cheang, U. Leong Hou, Web query expansion by wordnet, in: Database and Expert Systems Applications: 16th International Conference, Proceedings, DEXA 2005, Copenhagen, Denmark, August 22–26, 2005, vol. 16, Springer, 2005, pp. 166–175.
[37] S. Haiduc, G. De Rosa, G. Bavota, R. Oliveto, A. De Lucia, A. Marcus, Query quality prediction and reformulation for source code search: the refoqus tool, in: 2013 35th International Conference on Software Engineering (ICSE), IEEE, 2013, pp. 1307–1310.
[38] N.A. Kamal, M. ElSobky, A.M. Ibrahim, Z. Haider, Deep learning optimisation for spatial wind power forecasting: a data driven approach to grid stability enhancement, Int. J. Autom. Control 19 (2025) 188–212.
[39] H. Khan, M. Abdel-Aty, D. Almutairi, J. Gómez-Aguilar, J. Alzabut, Artificial intelligence neural networking for data clustering of carbon dioxide model, Ain Shams Eng. J. 16 (2025) 103460.
[40] Y. Xie, J. Lin, H. Dong, L. Zhang, Z. Wu, Survey of code search based on deep learning, ACM Trans. Softw. Eng. Methodol. 33 (2023) 1–42.
[41] T. Sultan, M.A.T. Rony, M.S. Islam, S. Alshathri, W. El-Shafai, Sumgpt: a novel multimodal framework for radiology report summarization to improve clinical performance, IEEE Access (2025).
[42] F.J. Peña, A.L. Gonzalez, S. Pashami, A. Al-Shishtawy, A.H. Payberah, Siambert: Siamese bert-based code search, in: 2022 Swedish Artificial Intelligence Society Workshop (SAIS), IEEE, 2022, pp. 1–7.
[43] N. Bibi, N. Choudhry, K. Khurshid, I. Rao, Enhancing code search through query expansion: a fusion of lstm with glove and bert model (csqe), in: International Conference on Asia Pacific Advanced Network, Springer, 2024, pp. 79–101.
[44] C. Wu, M. Yan, Learning deep semantic model for code search using codesearchnet corpus, arXiv preprint, arXiv:2201.11313, 2022.
[45] J. Chen, X. Hu, Z. Li, C. Gao, X. Xia, D. Lo, Code search is all you need? Improving code suggestions with code search, in: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, 2024, pp. 1–13.
[46] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, M. Brockschmidt, Codesearchnet challenge: evaluating the state of semantic code search, arXiv preprint, arXiv:1909.09436, 2019.
[47] X. Gu, H. Zhang, S. Kim, Deep code search, in: 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE), IEEE, Gothenburg, Sweden, 2018, pp. 933–944.
[48] A. Mishne, S. Shoham, E. Yahav, Typestate-based semantic code search over partial programs, in: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, 2012, pp. 997–1016.
[49] I. Keivanloo, J. Rilling, Y. Zou, Spotting working code examples, in: Proceedings of the 36th International Conference on Software Engineering, 2014, pp. 664–675.

[50] H. Yu, Y. Zhang, Y. Zhao, B. Zhang, Incorporating code structure and quality in deep code search, Appl. Sci. 12 (2022) 2051.
[51] J. Shuai, L. Xu, C. Liu, M. Yan, X. Xia, Y. Lei, Improving code search with co-attentive representation learning, in: Proceedings of the 28th International Conference on Program Comprehension, 2020, pp. 196–207.
[52] A. Saeed, K. Shehzad, S. Ahmed, A.T. Azar, Eg-van: a global and local attention based dual-branch ensemble network with advanced color balancing for multi-class skin cancer recognition, IEEE Access (2025).
[53] J. Cambronero, H. Li, S. Kim, K. Sen, S. Chandra, When deep learning met code search, in: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2019, pp. 964–974.
[54] L. Xu, H. Yang, C. Liu, J. Shuai, M. Yan, Y. Lei, Z. Xu, Two-stage attention-based model for code search with textual and structural features, in: 2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, 2021, pp. 342–353.
[55] Y. Meng, L. Liu, A deep learning approach for a source code detection model using self-attention, Complexity 2020 (2020) 1–15.
[56] N. Bibi, A. Maqbool, T. Rana, F. Afzal, A. Akgül, S.M. El Din, Enhancing semantic code search with deep graph matching, IEEE Access (2023).
[57] T. Zhu, Z. Li, M. Pan, C. Shi, T. Zhang, Y. Pei, X. Li, Deep is better? An empirical comparison of information retrieval and deep learning approaches to code summarization, ACM Trans. Softw. Eng. Methodol. 33 (2024) 1–37.
[58] W. Wang, Y. Wang, S. Joty, S.C. Hoi, Rap-gen: retrieval-augmented patch generation with codet5 for automatic program repair, in: Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2023, pp. 146–158.
[59] F. Hu, Y. Wang, L. Du, X. Li, H. Zhang, S. Han, D. Zhang, Revisiting code search in a two-stage paradigm, in: Proceedings of the Sixteenth ACM International Conference on Web Search and Data Mining, 2023, pp. 994–1002.
[60] J.P. Meher, S. Biswas, R. Mall, Deep learning-based software bug classification, Inf. Softw. Technol. 166 (2024) 107350.
[61] C. França, R.C. Lima, C. Andrade, W. Cunha, P.O.V. de Melo, B. Ribeiro-Neto, L. Rocha, R.L. Santos, A.S. Pagano, M.A. Gonçalves, On representation learning-based methods for effective, efficient, and scalable code retrieval, Neurocomputing 600 (2024) 128172.
[62] N. Bibi, T. Rana, A. Maqbool, Bi-directional lstm-based search engine for source code retrieval, in: 2023 International Conference on Communication Technologies (ComTech), IEEE, 2023, pp. 87–92.
[63] N. Bibi, A. Maqbool, T. Rana, F. Afzal, A.A. Khan, C2b: a semantic source code retrieval model using codet5 and bi-lstm, Appl. Sci. 14 (13) (2024) 5795.
[64] E. Okewu, P. Adewole, O. Sennaike, Experimental comparison of stochastic optimizers in deep learning, in: Computational Science and Its Applications–ICCSA 2019: 19th International Conference, Proceedings, Part V, Saint Petersburg, Russia, July 1–4, 2019, vol. 19, Springer, 2019, pp. 704–715.
[65] P. Salza, C. Schwizer, J. Gu, H.C. Gall, On the effectiveness of transfer learning for code search, IEEE Trans. Softw. Eng. 49 (2022) 1804–1822.