# Benchmarking Readability-Aware Boosting and Re-Scoring Techniques for Hybrid Lexical–Dense Code Search

1st Budi Susanto
*Electrical Engineering*
*and Information Technology*
*Universitas Gadjah Mada*
Yogyakarta, Indonesia
budisusanto490229@mail.ugm.ac.id

2nd Ridi Ferdiana*
*Electrical Engineering*
*and Information Technology*
*Universitas Gadjah Mada*
Yogyakarta, Indonesia
ridi@ugm.ac.id

3rd Teguh Bharata Adji
*Electrical Engineering*
*and Information Technology*
*Universitas Gadjah Mada*
Yogyakarta, Indonesia
adji@ugm.ac.id

***Abstract*—Code search is a critical capability in modern software development, supporting efficient reuse, comprehension, and maintenance. While lexical retrieval methods such as BM25 remain strong baselines, recent neural approaches have introduced dense vector representations that facilitate semantic matching. However, hybrid retrieval pipelines rarely incorporate code readability, even though readability strongly influences how developers interpret and assess code relevance. This study provides a systematic benchmark of readability-aware ranking strategies for both lexical and hybrid code search.**

**Three ranking strategies are examined on the CodeSearchNet Java dataset: (1) a BM25 baseline, (2) readability-aware boosting that integrates a supervised readability score into the lexical scoring function, and (3) re-scoring techniques, i.e. additive, multiplicative, and $\alpha$-fusion, that combine lexical scores with readability signals. The evaluation is further extended to hybrid lexical and dense retrieval by fusing BM25 with dense code embeddings and applying readability-aware re-scoring to the fused rankings.**

**Across 93 evaluated queries, readability-aware boosting yields a consistent improvement of approximately 5.15% over the lexical BM25 baseline, while multiplicative re-scoring offers an additional gain of about 7.44%. Moreover, readability-aware $\alpha$-fusion provides the most stable improvements in hybrid retrieval, achieving an MRR@10 of 0.51 (an increase of roughly 10.87% over the BM25 baseline) and an nDCG@10 of 0.28 (an improvement of approximately 29.54%). Collectively, these results demonstrate that readability signals substantially enhance both lexical and hybrid code search, and they establish the first comprehensive benchmark of readability-aware ranking strategies.**

***Index Terms*—code search, BM25, dense retrieval, re-scoring, boosting, readability, hybrid retrieval**

## I. Introduction

Semantic code retrieval has become an essential part of supporting modern software development. With the ever-growing code repositories and developers need to quickly find relevant code snippets, the effectiveness of ranking models in code retrieval is a highly determining factor. Information retrieval models based on lexical matching, such as BM25 (Best Match 25) [1], have proven to be robust in various code search tasks. In their experiments, Sachdev et al. [2] demonstrated that utilizing the BM25 method delivers better performance than applying basic vector similarity search (Neural code search). Zhang et al. [3] also reinforced that BM25 with tokenization configuration outperforms neural bag-of-words and self-attention methods. He et al. [4] showed that BM25 can effectively retrieve highly relevant data in Retrieval-Augmented Generation (RAG) systems for coding tasks. It can be said that BM25's performance can often compete with dense vector neural bag-of-words-based methods.

However, code snippets are not just ordinary textual documents; they have structures, naming patterns, and readability levels that can influence developers' perceptions of relevance in the search results. In this regard, the use of neural bag-of-words is considered insufficient for the semantic needs of a code snippet. Efforts to further enhance code search performance by strengthening the semantics of code snippets have been made through the development of code-specific embedding vector models. Gu et al. [5] proposed leveraging a deep learning-based model (DeepCS) that combines dense vectors of code and descriptions into a single vector space. The complex DeepCS model was simplified by Liu et al. [6] by applying it within an information retrieval (IR) model to strengthen semantic recognition of queries and code snippets (CodeMatcher). This model implements reranking between the method name weight and the code snippet. Reranking successfully delivered better performance than CodeHow and DeepCS. Heyman et al. [7] developed an improved vector embedding model from the initial neural code search [2] by applying augmentation to the training corpus and utilizing all identifiers in the code. The resulting embedding vector models were applied in code search as an ensemble (code and description). This model outperformed BM25 and NCS. Gandhi et al. [8] applied a reranking model to the BM25 ranking results, using it as an initial filter, with a neural CodeBERT model trained on bug reports and commit messages from git repositories. This reranking model boosted performance by up to 80%.

On the other hand, code reading is one of the most frequently performed activities by application developers. The easier code is to read, the more it supports the quality of the developed software, including its maintenance. Sorour et al. [9] emphasized that readability is a crucial objective in the software development process, as it affects overall software quality. Code readability is influenced not only by its syntactic formulation but also by aspects such as visual clarity, structural simplicity, and documentation support. With this perception, it is considered necessary to incorporate readability factors into code search systems. The background for this need is that code search involves selecting reusable code solution patterns [10]. A survey by Sadowski et al. [11] showed that 26% of programmers use code search services in local project repositories for reusable code practices. Therefore, readability factors can serve as additional signals in code search ranking. The hypothesis regarding this argument is that readability weights have the potential to improve ranking quality.

In the IR context, such signals can be exploited through boosting, readability-aware weighting, or client-side re-scoring, resulting in hybrid models that combine textual signals (BM25) and intrinsic code quality. The hybrid model for search systems has been proven to improve recall compared to using dense vector-based models alone [12]. The combination of sparse and dense vector models (hybrid approach) in search systems is more effective than using either sparse or vector models individually [13].

This study is aimed at understanding to what extent the rankings from traditional models based on lexical matching, dense vectors, and additional code readability weighting signals can contribute to the effectiveness of code search. First, this study evaluates to what extent pure BM25 can deliver effective retrieval results on the code search dataset used, given that BM25 remains a strong baseline in many previous studies. Next, it examines whether integration of readability scores through boosting and re-scoring techniques—using add, mul, or $\alpha$-fusion approaches—can improve ranking effectiveness compared to the BM25 baseline, especially since the readability signal provides an intrinsic code quality perspective not captured by lexical signals. Finally, this research experiments with hybrid models to find out whether there is improved search performance from applying weighted-sum fusion between sparse and dense vector models. Accordingly, this study compares these four approaches to identify which method delivers the most significant and stable improvements in MRR@10 and nDCG@10, so that the most effective readability signal integration strategy for code search can be determined.

## II. Methodology

To develop and test the code search model, this study uses a collection of Java code snippets and test queries from the `CodeSearchNet (CSN) Challenge Corpus` [14] provided on Hugging Face via the `ir-datasets` package[1] [15]. This package provides the `CodeSearchNet` (CSN) and `CodeSearchNet Challenge` datasets that can be used directly. The code snippet corpus for each language, including Java, is provided in the training, validation, and testing sections. For the experiments in this study, all Java code snippets across these three sections are combined as the search corpus.

Next, the study explores a code search architecture based on a hybrid approach that combines lexical BM25 with dense vectors [16]. To generate dense vectors for each query and code snippet, this research uses `CodeSearch-ModernBERT-Owl-Plus`[2] as the LLM (Large Language Model), which has been fine-tuned from `ModernBERT-Owl-Plus` for the code search task. The dimensionality of the resulting dense vectors is 768. This model has been trained using a dual-tower (docstring, code) approach, enabling it to produce dense vectors for both code and natural language text. Additionally, `CodeSearch-ModernBERT-Owl-Plus` has been trained using a hard negative mining approach to better distinguish between relevant and less or non-relevant results. `ModernBERT-Owl-Plus` is a domain-adapted derivative of `ModernBERT` [17], designed to overcome the limitations of the classic CodeBERT model in understanding the syntactic and semantic structure of program code.

Fig. 1 illustrates the relationships among the components involved in this code search exploration. The code search functions use the Apache SOLR system version 9.10[3], which enables hyperparameter tuning for the lexical BM25 method. Furthermore, version 9.10 has introduced dense vector search with HNSW indexing. Apache SOLR provides a boosting mechanism using the code readability weight on top of BM25. Re-scoring, on the other hand, is performed on the client side and involves the code readability weight via Additive Re-Scoring, Multiplicative Re-Scoring, and $\alpha$-weighted fusion. The best performance result among the BM25 lexical baseline, boosting, and re-scoring with readability weights will be used to test the hybrid model based on Weighted-Sum fusion.

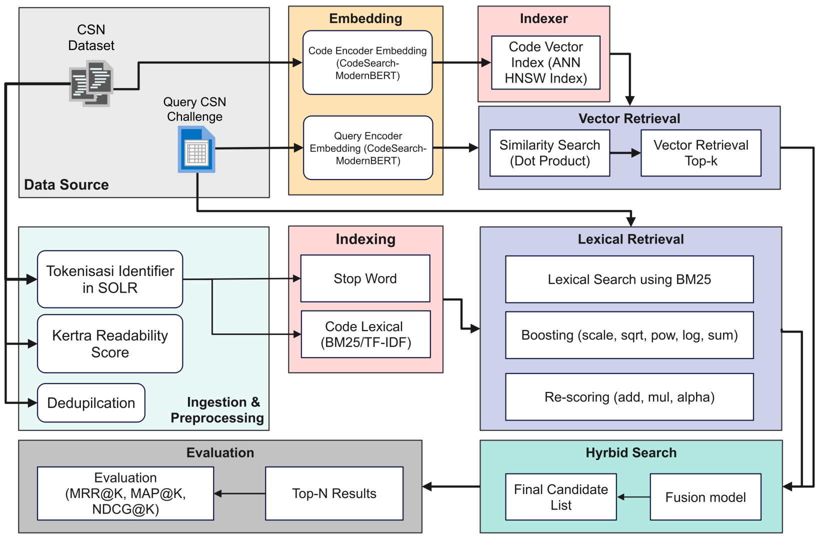
Performance evaluation results are measured using two common metrics in information retrieval systems: Mean Reciprocal Rank (MRR@10) to measure how quickly the system finds the most relevant result, and Normalized Discounted Cumulative Gain (NDCG@10) to assess the overall ranking quality among the top ten results. These metrics are chosen because they represent a balance between early precision and graded relevance—both of which are important in code search, where users usually seek the most relevant snippet within a limited set of results.

At the ingestion and preprocessing stages, one crucial step is deduplication. Deduplication is the process of cleaning redundancy without sacrificing diversity among relevant snippets. In this context, every code snippet that forms part of the ground-truth relevance for the query dataset is retained. This study implements deduplication for Java code snippets that

[1] https://pypi.org/project/ir-datasets/

[2] https://huggingface.co/Shuu12121/CodeSearch-ModernBERT-Owl-Plus

[3] https://solr.apache.org/



CSN Dataset
Query CSN Challenge
Data Source
Embedding
Code Encoder Embedding (CodeSearch-ModernBERT)
Query Encoder Embedding (CodeSearch-ModernBERT)
Indexer
Code Vector Index (ANN HNSW Index)
Vector Retrieval
Similarity Search (Dot Product)
Vector Retrieval Top-k
Tokenisasi Identifier in SOLR
Kertra Readability Score
Dedupilcation
Ingestion & Preprocessing
Indexing
Stop Word
Code Lexical (BM25/TF-IDF)
Lexical Retrieval
Lexical Search using BM25
Boosting (scale, sqrt, pow, log, sum)
Re-scoring (add, mul, alpha)
Evaluation
Evaluation (MRR@K, MAP@K, NDCG@K)
Top-N Results
Hyrbid Search
Final Candidate List
Fusion model

Fig. 1. Code Search Architecture Test Plan

are exact duplicates as well as those that are not relevant but are duplicates of ground-truth entries. No duplicate Java code snippets were found in the `ir-dataset` package's dataset.

The next process is to validate whether all relevant URLs in this relevance definition list are available within the CodeSearchNet corpus. The results showed that 26 URLs no longer provided their code snippets. In these cases, similar code was searched for via `archive.net` or by searching for the snippet's package and method names. Through this search, one snippet URL could not be found. This code snippet had a relevance value of 0, so it was excluded from the evaluation.

Another step taken was to review, among the 99 query definitions, whether the relevance list in the qrels.csv file contains queries where all query IDs are assigned a relevance value of 0. If all are 0, the study excludes those queries from testing. Through this validation process, 6 queries were found where all relevance values were 0. This step is necessary to ensure that the computation of MRR and NDCG performance metrics is truly based on queries with at least one result scoring $> 0$.

Additionally, since the code search ranking system in this study requires a complete Java method declaration snippet, every $code\_text$ entry was validated to ensure it represents a valid, complete Java method declaration. The validation process revealed 463 code snippets that needed to be corrected into a complete method declaration. The method for selecting whole method declaration snippets is based on the $func\_name$ value. After the entire validation process, a total of 497,144 Java-language CSN documents were ready for use.

The code readability weights for Java snippets used in this study are based on a readability metric model previously developed by the researcher, called KERTA. This metric model can produce 56 metric features from a Java code snippet, grouped into 5 conceptual dimensions of code readability. The dimensions defined in KERTA are Visual Clarity, Structural Simplicity, Documentation Support, Naming Transparency, and Cognitive Load. The KERTA readability weight model is based on a corpus of Java code snippets that have been classified into three readability categories: unreadable, neutral, and readable. The global readability weight produced by KERTA for a code snippet is in the range [0,1], so no further normalization is needed. For every Java code snippet in the CSN, its global readability weight will be calculated using the KERTA weight model.

BM25-based experiments were conducted without involving dense vector retrieval in order to purely observe the influence of each lexical component. Testing involved various combinations of query fields (**qf**) and phrase fields (**pf**), as well as incorporating a boost function (**bf**) based on the readability score to assess the contribution of code quality factors to the relevance of search results. The parameters $b$ and $k1$ in BM25 were set to 0.1 and 0.7 to accommodate the varying lengths of code snippets, while the $\alpha$ value was used to control the influence level of the BM25 score on the final results.

This study applies hyperparameter tuning to the weights of each defined search basis field, specifically the parameters **qf**, ($\lambda$) lambda, and ($\alpha$) alpha. As a result, each test configuration will involve several iterations of combination to determine its best performance.

## III. Results and Discussion

### A. *Boosting and Re-scoring with Code Readability Weighting*

Testing related to code search with Solr has been specifically configured to handle code snippets as individual documents. The metadata provided by CodeSearchNet for each code snippet and used in this code search test includes $code\_text$, $code\_tokens$, and $docstring\_text$. The $code\_tokens$ field is not preprocessed because it already contains the result of tokenization for each code snippet. The $docstring\_text$ field is preprocessed by converting each token to lowercase, removing words defined as stop words except for Java language keywords, and applying Porter stemming. Meanwhile,

TABLE I
PERFORMANCE RESULTS OF BM25-BASED CODE SEARCH WITH BOOSTING, READABILITY WEIGHT RE-SCORING

| Config | Boosting | qf | | | $\lambda$ | $\alpha$ | MRR@10 | NDCG@10 |
|---|---|---|---|---|---|---|---|---|
| | | *code_text* | *docstring_text* | *code_tokens* | | | | |
| BM25 Baseline | - | 7 | - | - | - | - | 0.343544 | 0.163907 |
| | - | - | - | 7 | - | - | 0.404416 | 0.153278 |
| | - | 5 | - | 7 | - | - | 0.456763 | 0.200736 |
| | - | 5 | 2 | - | - | - | 0.354297 | 0.175364 |
| | **-** | **5** | **1** | **6** | **-** | **-** | **0.457497** | **0.217139** |
| BM25 + Kerta Boosting | **Scale** | **6** | **1** | **7** | **-** | **-** | **0.481059** | **0.22405** |
| | SQRT Scale | 6 | 1 | 7 | - | - | 0.462186 | 0.220881 |
| | Log Sum | 5 | 1 | 6 | - | - | 0.457497 | 0.216951 |
| | POW | 6 | 1 | 7 | - | - | 0.462186 | 0.220881 |
| BM25 + Kerta Re Scoring | Additive | 6 | 1 | 8 | 0.6 | - | 0.471625 | 0.218356 |
| | **Multiplicative** | **6** | **1** | **8** | **0.15** | **-** | **0.491547** | **0.227134** |
| | Alpha | 6 | 1 | 8 | - | 0.8 | 0.485079 | 0.224493 |

$code\_text$ undergoes more complex preprocessing steps, including: URL/email-based tokenization to avoid excessive dot splitting, separating words from camelCase and snake_case, and removing Java operators.

Unlike general text search, code search requires a mechanism capable of balancing lexical matching and syntactic suitability, as search terms often contain identifiers, technical terms, or naming patterns unique to programming languages. Therefore, the initial configuration focuses on testing the combination of several main fields, namely $code\_text$, $code\_tokens$, and $docstring\_text$, with adjustments to BM25 parameters and the application of readability-based boosting ($kerta\_score$).

As a comparison, the initial configuration using only the $code\_text$ field served as the baseline, followed by a series of tests involving combinations of fields including identifier tokenization ($code\_tokens$) and semantic descriptions ($docstring\_text$). The next stage involved adding a boosting component based on $kerta\_score$, which represents code readability levels, using various functions (linear, root, logarithmic, and power) to observe to what extent readability impacts search performance. Finally, the $alpha$ parameter was adjusted to assess the balance between strengthening the BM25 score and the additional contribution of boosting to the search results.

Table I presents the results of several code search configuration tests based on the BM25 method for CodeSearchNet data and KERTA readability weights. The test configurations are defined into three groups. The **first** group involves basic BM25 function testing with five configurations. The **second** group is defined based on the best-performing configuration from the first group, applying a boosting function by incorporating the KERTA readability weight into the search weighting. The tested boosting functions include scale (normalizing the field values to a new scale [0,1]), square root (smoothing the differences between documents with high values to prevent dominance by large scores), log (using the logarithm to reduce large differences), sum (to ensure the log is not zero), and pow (a power function to strengthen high scores). All tests for each boosting function were carried out using grid search to find the best performance among parameter value choices.

The **third** test group applies additive fusion re-scoring, multiplicative fusion, and alpha ($\alpha$). The re-scoring function with linear addition (*additive*) is used for each ranking weight of the retrieved documents, applying equation (1).

$$\hat{s}(d_i) = s_{\text{bm25}}(d_i) + \lambda \cdot s_{\text{kerta}}(d_i) \tag{1}$$

In the add function, the readability weight serves as an additive bonus to the BM25 score.

- If ( $\lambda > 0$ ), documents with high readability will be pushed to the top.
- If ( $\lambda = 0$ ), same as pure BM25.

*Rescoring* with proportional reinforcement (multiplication) serves to proportionally multiply the search ranking weight by the KERTA weight (equation (2)).

$$\hat{s}(d_i) = s_{\text{bm25}}(d_i) \cdot (1 + \lambda \cdot s_{\text{kerta}}(d_i)) \tag{2}$$

Meanwhile, for normalization-based *weighted fusion* rescoring, it is applied in the following order of calculation:

1) Normalization of BM25 and KERTA scores by min-max:

$$s' * \text{bm25}(d_i) = \frac{s * \text{bm25}(d_i) - \min(s_{\text{bm25}})}{\max(s_{\text{bm25}}) - \min(s_{\text{bm25}}) + \varepsilon} \tag{3}$$

$$s' * \text{kerta}(d_i) = \frac{s * \text{kerta}(d_i) - \min(s_{\text{kerta}})}{\max(s_{\text{kerta}}) - \min(s_{\text{kerta}}) + \varepsilon} \tag{4}$$

2) Combine the two linearly:

$$\hat{s}(d_i) = \alpha \cdot s' * \text{bm25}(d_i) + (1 - \alpha) \cdot s' * \text{kerta}(d_i) \tag{5}$$

where:

- ( $\alpha$ ) adjust a balance between BM25 contribution and readability.
- ( $\alpha = 1.0$ ) only for BM25 relevance score ranking.
- ( $\alpha = 0.0$ ) only for readability score based ranking.

The test results show significant performance variation across configurations, both in MRR@10 and NDCG@10 metrics. The comparison between configurations illustrates how the combination of query fields (**qf**) and BM25 parameters affects the system's ability to find and rank relevant code snippets. In general, performance consistently improves when the $code\_tokens$ field is included along with $code\_text$, indicating that identifier token representation plays a significant role in semantic matching between queries and code snippets.

Configurations that rely solely on $code\_text$ yield the lowest MRR and NDCG scores, indicating that raw text-based search is insufficient to capture syntactic variations in source code. Conversely, adding $code\_tokens$ provides a substantial improvement in both the precision of the top results (MRR) and ranking stability (NDCG). Meanwhile, the inclusion of $docstring\_text$ shows a limited influence, as the docstrings in the dataset do not always align with the functional content of the code being described.

More significant improvements arise when the system is enhanced with a readability score-based boosting function ($kerta\_score$). The results show that highly readable snippets tend to be more relevant to user queries, so integrating readability as a factor contributes positively to ranking. Among the various forms of boosting functions tested, the linear transformation of $kerta\_score$ delivers the most consistent results, whereas non-linear functions (such as root, logarithmic, or exponentiation) do not provide significant improvements.

Additionally, variations in the $multiplication$ parameter show a consistent pattern: increasing the scaling factor for the BM25 score improves performance up to an optimal point at $lambda = 0.15$. This finding demonstrates that the prominence of the BM25 score remains important in lexical search-based systems, but it can be effectively augmented by additional signals such as code readability. Overall, the combination of $code\_text$^6 $docstring\_str$^1 $code\_tokens$^8 with *boosting* $scale(kerta\_score, 0, 1) * 5$ and $alpha = 0.8$ delivers the best performance, with MRR@10 of 0.491547 and NDCG@10 of 0.227134.

Thus, the configuration "qf=$code\_text$^6.0 $docstring\_str$^1.0 $code\_tokens$^8.0" and "pf=$code\_tokens$^1.0 $code\_text$^1.0" can be interpreted as an effective "semantically dominant, phrase-aware retrieval" strategy. This approach demonstrates that synergy between a high semantic weight on tokens, contextual support from code content, and moderate phrase boosting can consistently enhance code search performance—both in terms of top-rank relevance (MRR) and graded relevance ranking (NDCG).

This optimal configuration represents a search approach that balances the semantic representation of identifiers and the textual context provided by both code content and documentation. For the **qf** parameter, the highest weight is assigned to $code\_tokens$ (^8.0), followed by $code\_text$ (^6.0) and $docstring\_str$ (^1.0). This arrangement of weights indicates that the system emphasizes token-based semantic matching, such as function, variable, or class names, as these elements best represent the functional meaning of a code snippet. Meanwhile, $code\_text$ provides syntactic context and comments within the function body, while the $docstring$ still contributes, albeit on a smaller scale, enriching descriptive semantic understanding.

The **pf** parameter is used to provide a phrase boost to search results that contain token sequences that match the query, whether in the $code\_tokens$ or $code\_text$ fields, each with a moderate weight (^1.0). With this setup, the system gives additional scores to results that maintain phrase structure aligned with the query's word order, without overpowering the main influence of semantic matching. This phrase-aware retrieval approach helps balance precision and recall, while enhancing the system's ability to identify results that are relevant and naturally structured.

## B. Hybrid Search (Dense vector and BM25 rescoring)

Testing of the hybrid retrieval model was conducted by combining two search approaches, namely dense vector-based search (KNN) and lexical BM25 search, which was rescored using readability scores ($kerta\_score$). The evaluation process used 93 queries (out of a total of 99 queries, with 5 queries having no documents of relevance $> 0$) from the Java dataset, resulting in three comparison scenarios, as shown in Table II.

TABLE II
HYBRID MODEL PERFORMANCE RESULTS

| Config | MRR@10 | nDCG@10 |
|---|---|---|
| dense vector only | 0.403772 | 0.210577 |
| hybrid | **0.507207** | **0.281275** |

The integration of readability scores ($kerta\_score$) into BM25 through a multiplicative rescoring approach (using the formula (2)) results in an increase in MRR@10 of approximately 7.1% compared to BM25 using the query model "$code\_text$^5.0 $docstring\_str$^1.0 $code\_tokens$^6.0". When the results of dense retrieval (embedding-based) are fused with $BM25 \cdot (1 + \lambda \cdot kerta)$ using the weighted sum fusion method ($\alpha = 0.35$, min–max normalization), performance increases further, reaching MRR@10 = 0.5072 and nDCG@10 = 0.2813. This improvement indicates that the two types of representation, dense and lexical, are complementary.

Although the margin between the hybrid model and the BM25 multiplication rescoring approach is only 3%, it can be said that the hybrid approach is effective because documents with higher readability receive proportional boosts to their lexical scores. As a result, outputs that are easier to read (e.g., those with clear code structure, descriptive naming, and concise comments) are more likely to appear at the top ranks. This shows that $kerta\_score$ serves as a semantic-quality prior that helps balance textual similarity and syntactic/semantic quality of the retrieved code snippets.

The dense vector component captures semantic similarities between queries and code snippets even without direct term matches, such as the relationship between "serialize JSON" and a function named '`toJsonString`'. Meanwhile, BM25 maintains its advantage in capturing explicit terminological matches, particularly in very specific variable names and comments. With $\alpha = 0.35$, the contribution of dense retrieval is kept from overwhelming the lexical results, which aligns with the nature of code search requiring a balance between semantic understanding and syntactic matching.

Score normalization using the min–max approach plays a crucial role in making the dense and BM25 scores comparable before fusion. Without normalization, the score distribution from BM25 (usually in a small range with high variance across queries) can suppress the contribution of the dense vector. The choice of weighted sum fusion (wsum) is also proven to be more stable compared to rank-based fusion methods (such as Reciprocal Rank Fusion) because it maintains the proportionality among already normalized scores.

These results show that the hybrid model with readability-based client-rescore and weighted fusion is an effective configuration for code search systems that consider result quality. The architecture can be interpreted as three layers:

1) Lexical retrieval (BM25), which functions to capture direct term matches.
2) Quality-aware rescoring (KERTA), which functions to prioritize documents that are easy to read and well-structured.
3) Semantic retrieval (Dense KNN), which adds semantic context from embeddings trained with code–text alignment.

This combination enhances the system's ability to present code snippets that are not only semantically relevant but also high in readability quality, thereby being more useful to users.

## IV. Conclusion

This study demonstrates that BM25, with optimized field configuration and parameters, serves as a strong lexical baseline for code retrieval, especially when queries directly match syntactic structures or identifiers. BM25's stable performance makes it a primary reference point for other approaches.

The integration of KERTA readability scores, whether through boosting or re-scoring, has been proven to improve ranking effectiveness. Boosting provides moderate gains, while re-scoring strategies (ADD, MUL, and $\alpha$-fusion) offer more precise score combinations and result in more consistent increases in MRR@10 and nDCG@10. This confirms that readability is an important relevance signal for prioritizing code snippets that are easier to understand.

Meanwhile, dense vector-based models have not individually outperformed BM25, but they serve a complementary role. When combined with BM25 and Kerta scores in a hybrid re-scoring scheme, this approach delivers the best performance among all tested configurations. Thus, the combination of sparse and dense relevance, combined with readability signals, is the most effective strategy for improving the quality of code retrieval.

## Acknowledgment

The authors would like to thank the Faculty of Information Technology, Universitas Kristen Duta Wacana, for the financial support provided for the publication of this paper.

## References

[1] S. Robertson and H. Zaragoza, "The probabilistic relevance framework: BM25 and beyond," *Foundations and Trends in Information Retrieval*, vol. 3, pp. 333–389, 2009.

[2] S. S. Sachdev, H. H. Li, S. S. Luan, S. S. Kim, K. K. Sen, and S. S. Chandra, "Retrieval on source code: A neural code search," *MAPL 2018 - Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages, co-located with PLDI 2018*, pp. 31–41, 2018.

[3] X. Zhang, J. Xin, A. Yates, and J. Lin, "Bag-of-words baselines for semantic code search," in *Proceedings of the 1st Workshop on Natural Language Processing for Programming*, 2021, pp. 88–94.

[4] P. He, S. Wang, S. Chowdhury, and T.-H. Chen, "Evaluating the effectiveness and efficiency of demonstration retrievers in RAG for coding tasks," *arXiv [cs.SE]*, Oct. 2024.

[5] X. Gu, H. Zhang, and S. Kim, "Deep code search," *Proceedings - International Conference on Software Engineering*, pp. 933–944, 2018.

[6] C. Liu, X. Xia, D. Lo, Z. Liu, A. E. Hassan, and S. Li, "CodeMatcher: Searching code based on sequential semantics of important query words," *arXiv [cs.SE]*, May 2020.

[7] G. Heyman and T. Van Cutsem, "Neural code search revisited: Enhancing code snippet retrieval through natural language intent," *arXiv:2008. 12193 Search. . .*, 2020.

[8] S. Gandhi, L. Gao, and J. Callan, "Repository-level code search with neural retrieval methods," *arXiv [cs.IR]*, Feb. 2025.

[9] S. E. Sorour, H. E. Abdelkader, K. M. Sallam, R. K. Chakrabortty, M. J. Ryan, and A. Abohany, "An analytical code quality methodology using latent dirichlet allocation and convolutional neural networks," *Journal of King Saud University - Computer and Information Sciences*, vol. 34, no. 8, Part B, pp. 5979–5997, September 2022.

[10] V. Bauer, J. Eckhardt, B. Hauptmann, and M. Klimek, "An exploratory study on reuse at google," in *Proceedings of the 1st International Workshop on Software Engineering Research and Industrial Practices*, ser. SER&IPs 2014. New York, NY, USA: Association for Computing Machinery, June 2014, pp. 14–23.

[11] C. Sadowski, K. T. Stolee, and S. Elbaum, "How developers search for code: A case study," *2015 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE 2015 - Proceedings*, pp. 191–201, 2015.

[12] N. Arabzadeh, X. Yan, and C. L. A. Clarke, "Predicting efficiency/effectiveness trade-offs for dense vs. sparse retrieval strategy selection," in *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*, ser. CIKM '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 2862–2866. [Online]. Available: https://doi.org/10.1145/3459637.3482159

[13] J. Lin, X. Ma, S.-C. Lin, J.-H. Yang, R. Pradeep, and R. Nogueira, "Pyserini: A python toolkit for reproducible information retrieval research with sparse and dense representations," in *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*, ser. SIGIR '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 2356–2362. [Online]. Available: https://doi.org/10.1145/3404835.3463238

[14] H. Husain, H.-H. Wu, T. Gazit, G. Miltiadis, and A. M. Brockschmidt, "CodeSearchNet challenge: Evaluating the state of semantic code search," *arXiv:1909. 09436*, 2019.

[15] S. MacAvaney, A. Yates, S. Feldman, D. Downey, A. Cohan, and N. Goharian, "Simplified data wrangling with ir_datasets," in *SIGIR*, 2021.

[16] H. Husain, "How to create natural language semantic search for arbitrary objects with deep learning," https://towardsdatascience.com/semantic-code-search-3cd6d244a39c, May 2018, accessed: 2024-3-21.

[17] B. Warner, A. Chaffin, B. Clavié, O. Weller, O. Hallström, S. Taghadouini, A. Gallagher, R. Biswas, F. Ladhak, T. Aarsen, N. Cooper, G. Adams, J. Howard, and I. Poli, "Smarter, better, faster, longer: A modern bidirectional encoder for fast, memory efficient, and long context finetuning and inference," *arXiv [cs.CL]*, Dec. 2024.