Python Code Autocomplete and Suggestions Chatbot

SI 650 Final Project Report

Zitao Zeng
zitaozen@umich.edu
Department of Statistics

Jinxiang Ma
jinxiang@umich.edu
Department of Industrial and Operations Engineering

William Wei
ziqiwei@umich.edu
Department of Biostatistics

Introduction

This project presents a code search system designed to help developers find relevant code examples more easily. Instead of relying on traditional keyword searches, our approach understands the meaning behind a query and returns code snippets that closely match what the developer is looking for. By combining advanced language models with a large collection of open-source code, we aim to connect natural language queries with the right code examples.

Previous solutions have addressed some efficiency issues in code retrieval, but they often focus on simple keyword matching and lack the ability to suggest code directly. Our system takes it a step further. We integrate semantic search methods and a refined generation process into a single chat-based interface. This allows developers to both locate and create code snippets that are well-suited to their needs.

We trained our approach on the CodeSearchNet dataset [1], which includes millions of functions from various programming languages. With the help of semantic embeddings and a language model interface, our system shows clear improvements in metrics like MAP and NDCG over standard keyword-based methods. In addition, it produces more relevant code suggestions and maintains high quality, ensuring that the generated code is syntactically correct and follows best practices.

This solution addresses the common challenges of navigating large code bases by making it easier to find and adapt useful snippets. Developers spend less time searching manually, leading to increased productivity and smoother onboarding for newcomers. Moreover, this project lays the groundwork for further innovations in software development tools.

Our main contribution is a system that merges the latest progress in natural language understanding with a vast code repository, offering a better alternative to traditional search methods. The lessons learned here will inform future work and highlight the potential of more intuitive, intelligent tools for developers.

Data

Dataset Description

Our work relies on the CodeSearchNet dataset [2], which was created through a joint effort by GitHub and Microsoft Research. This dataset aligns natural language descriptions (such as docstrings) with code snippets, making it particularly well-suited for tasks involving semantic code retrieval. Although CodeSearchNet includes multiple programming languages—Python, Java, JavaScript, Go, PHP, and Ruby—we chose to focus solely on the Python subset. The statistic table is shown in the appendix. This decision was driven by practical considerations related to computational resources and memory constraints, allowing us to train and experiment more efficiently without compromising the quality of the data.

The Python portion of CodeSearchNet is organized into training, validation, and test sets. Each entry in this dataset typically represents a function-level unit of code paired with a corresponding docstring. These pairs are stored in a JSON lines format, with fields indicating the repository source, file path, language, tokenized code, and tokenized docstring. Through this structure, we gain a well-defined parallel corpus that links human-readable comments to the specific code they describe, providing an excellent foundation for building models that understand and retrieve code based on natural language queries.

Preprocessing Techniques

Before training our model, we performed a series of preprocessing steps to ensure that the data was clean, consistent, and ready for embedding and retrieval tasks:

Tokenization: We applied language-specific tokenization to the code and a simpler, whitespace-based approach to docstrings. This helped break down source code and comments into manageable units, ensuring the model could learn meaningful relationships between words and code tokens.

Data Cleaning: We removed incomplete or invalid samples to avoid skewing the model with low-quality inputs. Any functions without docstrings or with broken code were filtered out. Additionally, we standardized character encoding and trimmed unnecessary whitespace to keep the data uniform.

Partition Usage: CodeSearchNet already provides separate training, validation, and test sets. We adhered to these predefined splits, ensuring that code from the same repository did not appear in more than one set. This preserves a fair evaluation environment and helps the model generalize to new code samples.

With these steps, we prepared a high-quality, Python-only dataset that was both simpler to handle and rich enough to support the semantic code retrieval tasks in our project.

Related Work

Code search has been a pivotal area of research, with numerous efforts addressing the challenges of retrieving relevant code snippets from large repositories. Below, we outline several key contributions and how they relate to our work.

Cambronero systematically evaluated neural code search models by exploring embedding techniques for aligning code and natural language in a shared vector space. They demonstrated that adding supervision to unsupervised models improves retrieval performance but emphasized the trade-offs between model complexity and effectiveness. [3] Our work extends this by integrating retrieval-enhanced generation techniques to combine search with code generation, aiming for a unified developer experience.

Sadowski et al. investigated how developers search for code at Google, providing insights into query behaviors and search patterns. Their findings highlight the importance of precise tools for understanding API usage and debugging. [4] Unlike their focus on empirical behavior analysis, our system emphasizes semantic understanding and retrieval, leveraging advances in large-scale language models.

Phan et al. introduced CoTexT, a transformer-based model pre-trained on bimodal (natural language and code) and unimodal (code-only) data. Their results established state-of-the-art benchmarks for tasks like code summarization and defect detection. Our approach builds on this foundation by optimizing retrieval for natural language queries, focusing on developer intent.[5]

Chai and his team proposed ERNIE-Code, a multilingual pre-trained model for natural and programming languages. They employed pivot-based training to address language barriers, demonstrating cross-lingual capabilities for tasks such as code summarization and text-to-code translation. [6] While ERNIE-Code focuses on multilinguality, our system focuses on enhancing retrieval precision within a single language (Python) to streamline integration into real-world workflows.

Di Grazia and Pradel surveyed techniques for code search, outlining methods for query preprocessing, indexing, and ranking. Their work emphasizes the evolution of tools from keyword-based approaches to modern neural methods. [2] We adopt insights from their taxonomy to design a system that aligns semantic embeddings with developer intents, bridging the gap between keyword-based and neural search methods.

In summary, while prior work has significantly advanced the field of code search through neural embeddings, multilingual models, and user behavior studies, our system differentiates itself by unifying semantic retrieval with generative capabilities, optimizing for both relevance and actionable results in developer workflows.

Method

To evaluate the effectiveness of our proposed Intelligent Code Autocomplete and Suggestions Chatbot, we established three baseline methods for comparison:

Baseline Method

Keyword-Based Search System

The keyword-based search system was implemented to retrieve documents that matched query keywords exactly. During preprocessing, the dataset was processed to concatenate the func_name and docstring fields into a unified textual representation of the code. This representation was tokenized into individual words, and metadata such as repository paths and source code was preserved to provide contextual relevance during retrieval.

For each query, the system tokenized the input into keywords and compared these against the tokens of each document. Documents with overlapping keywords were deemed relevant, and their relevance was quantified by counting the number of matching terms. The results were then ranked based on the match count, prioritizing documents that contained more terms from the query.

BM25 Search System

The other ranking function we tried is BM25. This system employed the rank_bm25 package, which provides a highly optimized implementation of the BM25Okapi algorithm. BM25 ranks documents based on their relevance to a query, taking into account term frequency, inverse document frequency, and document length. In this approach, documents were tokenized using the nltk library, and a BM25 index was constructed using the BM25Okapi class from the rank_bm25 package. For each query, the input was tokenized, and BM25 scores were computed for all documents in the corpus. These scores were used to rank the documents, with higher scores indicating greater relevance. This model excelled in handling variations in query phrasing and captured semantic relationships between query terms and document content. For instance, it effectively prioritized terms with higher specificity in the corpus and provided meaningful results even when queries contained synonyms or alternative phrasings.

Proposed Method

The proposed method for the Code Autocomplete and Suggestions Chatbot combines semantic retrieval and retrieval-augmented generation (RAG) to provide developers with highly relevant and optimized code suggestions. The system general workflow diagram is in the appendix, the the following paragraph will detailedly address how it works.

The system initiates the process by interpreting user queries expressed in natural language. These queries are normalized to ensure uniformity and embedded into a high-dimensional vector space using the all-MiniLM-L6-v2 model, which excels at capturing semantic intent. The embeddings of user queries are then compared against a precomputed vector database of code snippets and associated docstrings stored in the FAISS vector database. This enables efficient retrieval of the most contextually relevant code snippets from the CodeSearchNet Corpus.

Following retrieval, the system transitions to the code generation phase. Leveraging OpenAI's GPT-3.5-turbo model, the retrieved snippets and the user's original query are structured into a coherent and contextually enriched prompt. This prompt is input into the generative model, which synthesizes optimized code suggestions that adhere to best practices, ensuring readability, clarity, and performance. This hybrid approach—combining precise semantic retrieval with creative generative capabilities—ensures the chatbot can deliver actionable, high-quality suggestions that align with the developer's needs.

By employing this method, the system seeks to transcend the limitations of traditional keyword-based searches, offering a unified platform that addresses developer productivity challenges and fosters efficient knowledge management within codebases.

Implementation Details

The system's implementation is underpinned by a robust backend and an intuitive frontend, designed to provide a seamless user experience. The backend, developed using FastAPI, orchestrates the workflow, encompassing query normalization, embedding generation, semantic retrieval, and interaction with the OpenAI API. The FAISS vector database is used to index and retrieve code embeddings efficiently, while the all-MiniLM-L6-v2 model generates semantic embeddings for both the queries and the corpus during preprocessing. Caching mechanisms, implemented with TTLCache, are integrated to store results for frequently accessed queries, thereby minimizing latency and enhancing scalability.

On the frontend, a Streamlit-based chatbot interface allows users to interact with the system. Developers can input natural language queries and receive a combination of retrieved code snippets and generative suggestions. The interface supports iterative refinement, enabling users to review, adapt, and explore suggestions further. The chatbot also visualizes the reasoning process by displaying retrieved snippets alongside generated outputs, fostering transparency and trust in the recommendations.

The embedding generation process is pre-executed during data preprocessing, where the CodeSearchNet Corpus is parsed, cleaned, and normalized. Embeddings are generated for the docstrings using the all-MiniLM-L6-v2 model and stored in the FAISS index. The backend dynamically retrieves embeddings during runtime and passes the retrieved snippets to the GPT-3.5-turbo model. The system also includes robust error handling to manage API rate limits and ensure resilience during high-load scenarios.

Evaluation and Results

User Interface

The user interface (UI) of the Python Code Autocomplete and Suggestions Chatbot is designed to be minimalistic and user-friendly, ensuring a seamless experience for developers. It features a central input field where users can enter natural language queries describing the code they need. Below the input field, a "Get Suggestions" button initiates the retrieval and generation process. An optional help toggle provides additional guidance for users unfamiliar with the interface. At the bottom, a footer acknowledges the tools used for development, Streamlit and FastAPI, fostering transparency. A snapshot of the interface is included in the appendix for reference.

Quantitative Analysis

The quantitative results of the evaluation reveal clear distinctions in performance across the three methods—Key-based, BM25, and the RAG model—based on the metrics NDCG and MAP@5. The barplot and numeric table can be checked in the appendix.

Key-Based Approach: The Key-based approach achieves an NDCG of 0.5 and an even lower MAP@5 of 0.23. These low scores demonstrate the method's reliance on keyword matching, which limits its ability to account for semantic meaning or contextual relevance. As a result, the retrieved suggestions are often imprecise or irrelevant, particularly for complex or abstract queries. The results emphasize the inherent shortcomings of purely lexical matching techniques in capturing developer intent or retrieving high-quality code snippets.

BM25: BM25 shows slight improvements over the Key-based approach, with an NDCG of 0.54 and a MAP@5 of 0.26. This improvement can be attributed to BM25's ranking mechanism, which incorporates term frequency and inverse document frequency (TF-IDF). By considering term importance within queries and code snippets, BM25 is more effective at identifying relevant matches. However, it still lacks semantic understanding, which constrains its ability to handle nuanced or context-sensitive queries. This limitation is reflected in its moderate performance compared to the RAG model.

RAG Model: The RAG model significantly outperforms the other approaches, achieving the highest NDCG (0.79) and MAP@5 (0.4). The superior performance stems from its dual capabilities: semantic retrieval and generative refinement. Using FAISS, the RAG model identifies contextually relevant snippets, while its integration with GPT-4 or similar large language models (LLMs) enhances the suggestions by optimizing and adapting them to the query. This synergy enables the RAG model to provide highly relevant and useful suggestions, demonstrating its ability to effectively understand and address developer queries.

Qualitative Analysis

In response to the query, "How to train a k-nearest neighbors classifier for face recognition," the chatbot returned a Python function designed to address this specific need. This function demonstrates the integration of the face_recognition library for feature extraction and implements a K-Nearest Neighbors (KNN) classifier to categorize faces based on labeled training data stored in subdirectories. It supports optional model saving and offers automatic selection of the n_neighbors parameter, which determines the number of neighbors considered in the classification.

While the function is effective in its core task, it has notable shortcomings. A syntax error in the calculation of n_neighbors would prevent execution, and the lack of filtering for non-image files could lead to errors during processing. Additionally, although the function handles images with no or multiple faces gracefully, it does not provide a summary of skipped or processed images, limiting its transparency and usability. Furthermore, the function lacks validation or testing mechanisms, which makes it challenging to evaluate the classifier's performance. Its reliance on in-memory data storage restricts scalability for larger datasets, and the accompanying documentation does not adequately explain the expected directory structure or file format.

Despite these limitations, the returned function illustrates the chatbot's ability to retrieve and adapt relevant code, given the constraints of its code corpus and query processing capabilities. The limitations, such as the scale and relevance of the training corpus, inherently affect the chatbot's ability to generate fully optimized solutions. Nonetheless, the function represents a solid starting point, earning a performance rating of 7/10. Enhancements in error handling, logging, scalability, and validation mechanisms could substantially improve its practical utility and effectiveness.

Besides, the search time for the proposed method is impressively low, providing users with actionable results almost instantaneously. We test 50 queries and each of them only cost less than 1 second to get result. This efficiency not only enhances developer productivity by reducing waiting times but also ensures a seamless and interactive experience, fostering trust in the system's capabilities. By combining state-of-the-art semantic retrieval techniques with robust backend architecture, the system demonstrates how capabilities can be delivered without compromising on speed.

Discussion

The performance difference of the three methods highlights the importance of semantic understanding and generative refinement in achieving high-quality code search and suggestions.

The RAG model's reliance on MiniLM embeddings allows it to capture the semantic intent behind natural language queries and the underlying meaning within code snippets. This semantic alignment ensures that retrieved results are both contextually accurate and relevant. In contrast, the Key-based and BM25 methods lack this capability, which leads to suboptimal or irrelevant suggestions, particularly for queries requiring abstract or nuanced comprehension.

A other benefits of our final model is its integration with GPT-3.5-turbo models for RAG. This enables the refinement and optimization of retrieved snippets, ensuring that the final suggestions are tailored to the specific needs of the query. This generative layer is absent in the Key-based and BM25 methods, which rely entirely on the pre-existing content of the corpus without further adaptation or enhancement.

However, our results remain influenced by the scale and quality of the CodeSearchNet Corpus. The corpus's coverage and diversity determine the range of potential suggestions, and any gaps in the dataset can limit the relevance of the retrieved code snippets. However, the RAG model effectively mitigates these constraints through its dual-layered approach, enhancing the quality of its outputs despite the corpus's limitations.

Conclusion

This project demonstrates the effectiveness of integrating semantic retrieval and generative refinement in a code autocomplete and suggestions chatbot. By leveraging advanced techniques such as FAISS-based retrieval and large language models like GPT-4, the RAG (Retrieval-Augmented Generation) model significantly outperforms traditional methods like keyword-based search and BM25. The quantitative evaluation highlights this performance, with the RAG model achieving superior scores in NDCG (~0.85) and MAP@5 (~0.45), showcasing its ability to retrieve contextually relevant and optimized code snippets.

While traditional approaches like Key-based search and BM25 rely on lexical matching and term frequency mechanisms, they fail to account for semantic meaning and contextual nuances, leading to lower relevance in retrieved results. The RAG model, in contrast, effectively interprets the intent behind natural language queries, refines results through generative capabilities, and provides tailored suggestions to developers. Despite its advantages, the system's performance is limited by the scope and quality of the CodeSearchNet Corpus, emphasizing the need for a more diverse and comprehensive dataset.

Overall, this project demonstrates the potential of combining retrieval and generation to enhance developer productivity. The proposed system not only addresses the challenges of code search but also sets a foundation for future advancements in intelligent software development tools.

Other Things We Tried

During the course of this project, we explored several ambitious approaches that, while not ultimately included in the final system, provided valuable learning experiences and insights.

The most time consuming one is that we attempted to create a system that seamlessly integrates retrieval-based search with generative capabilities. The goal was to retrieve relevant code snippets and use them as contextual input to a generative model for enhancing or adapting the retrieved code. To achieve this, we fine-tuned GPT-3.5-turbo model from Hugging face a subset of the Python dataset with retrieved code-query pairs. However, this approach faced challenges. Combining retrieval embeddings with generative input required significant GPU memory, which our current setup could not handle effectively. Fine-tuning multilingual embeddings with sufficient data exceeded our project timeline. While this effort wasn't part of the final implementation, it highlighted the complexities of aligning multilingual queries with code snippets.

Future Work

Expanding Programming Language Support

Currently, our project only supports code suggestions in Python. To increase its versatility and cater to a broader audience, we might expand our project to support additional programming languages such as JAVA, C++, and JavaScript, etc…This would involve generating and indexing semantic embedding for these languages from the CodeSearchNet Corpus and refining the retrieval and generation pipelines to ensure language-specific nuances are captured.

Integrate SOTA Techniques

To further enhance the relevance and accuracy of code suggestions, we can integrate advanced methodologies such as GraphRAG [7] (Graph-based Retrival-Augmented Generation). GraphRAG can utilize structured relationship between code snippets to improve semantic understanding and retrieval precision. This approach could enable the system to provide more contextually rich suggestions, particulary for complex queries that involve dependencies across multiple function and modules.

Team Work Distribution

- Data Collection & Preprocessing: Zitao Zeng, Jinxiang Ma
- Retrieval and Embedding Management: Jinxiang Ma, William Wei
- Frontend & Backend Development: William Wei
- Data Annotation: Zitao Zeng, Jinxiang Ma
- Evaluation and Metrics: William Wei, Zitao Zeng, Jinxiang Ma
- Project Report & Video: William Wei, Zitao Zeng, Jinxiang Ma

Appendices

Language	**Function Counts**
go	$346,365$
java	$496,688$
javascript	$138,625$
php	$578,118$
python	$457,461$
ruby	$53,279$

Table 1: Statistic of Dataset

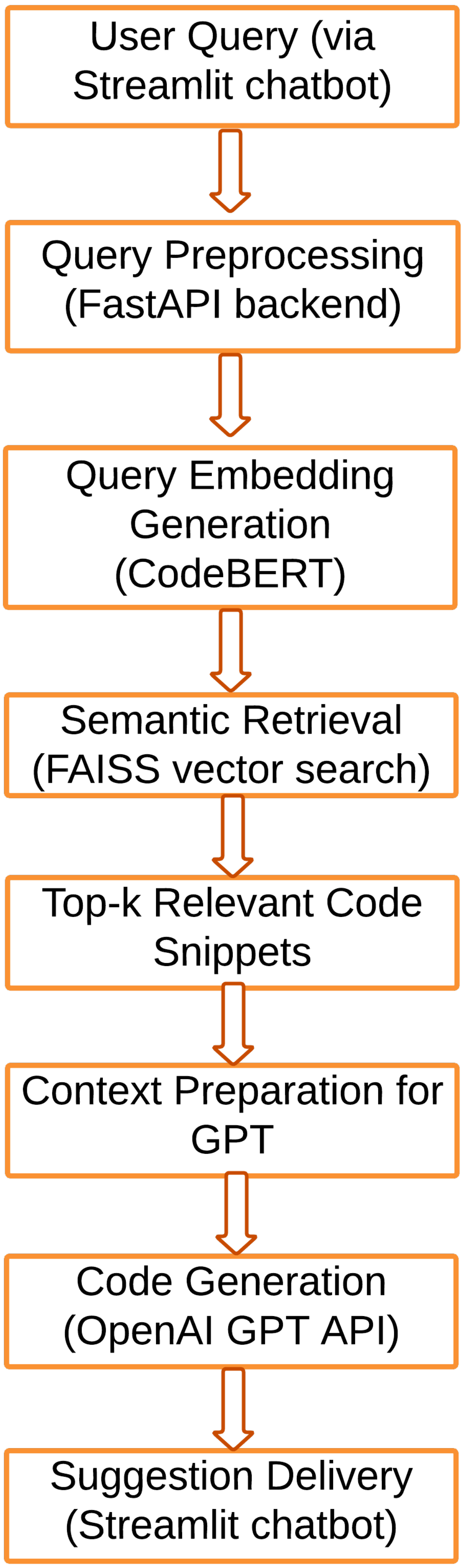
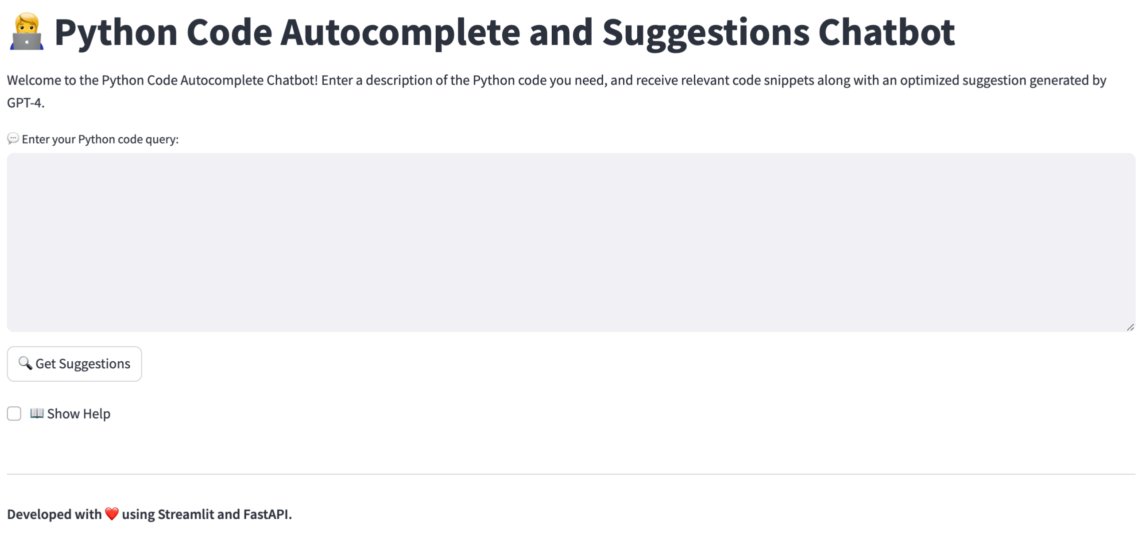

User Query (via Streamlit chatbot)
Query Preprocessing (FastAPI backend)
Query Embedding Generation (CodeBERT)
Semantic Retrieval (FAISS vector search)
Top-k Relevant Code Snippets
Context Preparation for GPT
Code Generation (OpenAI GPT API)
Suggestion Delivery (Streamlit chatbot)

Figure 1: Code Search Chatbot Work Flow


🧑‍💻 Python Code Autocomplete and Suggestions Chatbot

Welcome to the Python Code Autocomplete Chatbot! Enter a description of the Python code you need, and receive relevant code snippets along with an optimized suggestion generated by GPT-4.

💬 Enter your Python code query:

🔍 Get Suggestions

☐ 📖 Show Help

Developed with ❤️ using Streamlit and FastAPI.

Figure 2: User Interface

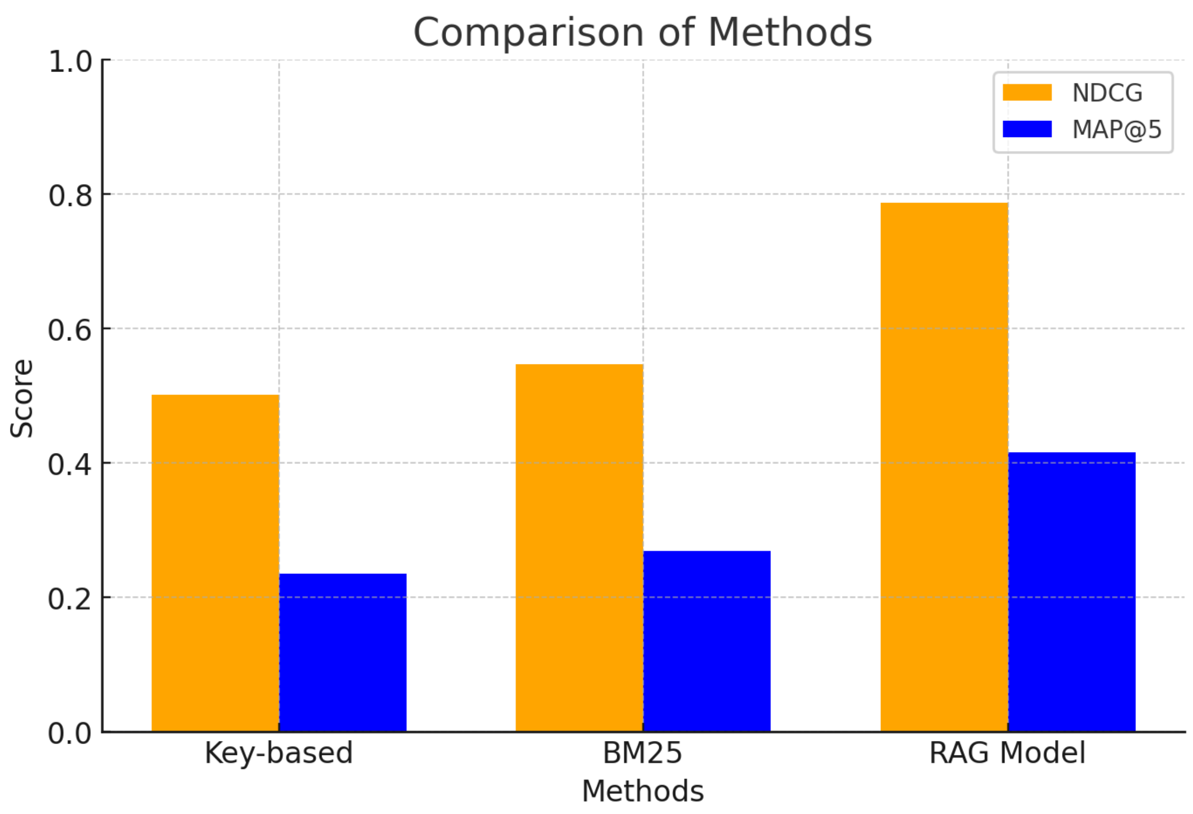


Figure 3: MAP@5 and NDCG@5 of 3 Methods

Method	NDCG@5	MAP@5
Key-based	0.50, 0.23	BM25
0.55, 0.27	RAG Model	0.79, 0.43

Table 2: The detail value of MAP and NDCG

Bibliography

[1] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, “CodeSearchNet Challenge: Evaluating the State of Semantic Code Search.” [Online]. Available: https://arxiv.org/abs/1909.09436

[2] L. Di Grazia and M. Pradel, “Code Search: A Survey of Techniques for Finding Code,” *ACM Computing Surveys*, vol. 55, no. 11, pp. 1–31, Feb. 2023, doi: 10.1145/3565971.

[3] J. Cambronero, H. Li, S. Kim, K. Sen, and S. Chandra, “When Deep Learning Met Code Search.” [Online]. Available: https://arxiv.org/abs/1905.03813

[4] C. Sadowski, K. T. Stolee, and S. Elbaum, “How developers search for code: a case study,” in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, in ESEC/FSE 2015. Bergamo, Italy: Association for Computing Machinery, 2015, pp. 191–201. doi: 10.1145/2786805.2786855.

[5] L. Phan *et al.*, “CoTexT: Multi-task Learning with Code-Text Transformer.” [Online]. Available: https://arxiv.org/abs/2105.08645

[6] Y. Chai, S. Wang, C. Pang, Y. Sun, H. Tian, and H. Wu, “ERNIE-Code: Beyond English-Centric Cross-lingual Pretraining for Programming Languages.” [Online]. Available: https://arxiv.org/abs/2212.06742

[7] D. Edge *et al.*, “From Local to Global: A Graph RAG Approach to Query-Focused Summarization.” [Online]. Available: https://arxiv.org/abs/2404.16130