SCOTCH: A SEMANTIC CODE SEARCH ENGINE FOR IDEs

Anonymous authors
Paper under double-blind review

ABSTRACT

Code search is the task of finding relevant code snippets given a natural language query. In order to facilitate real time code search, we introduce Scotch, a semantic code search tool that runs within an IDE. The semantic nature of code search in Scotch allows us to leverage the semantic meaning of code via learned vector representations, while the in-IDE nature helps to improve developers' productivity by eliminating the need to navigate to web-browsers to search for code. The query used for code search is oftentimes ambiguous without the surrounding context of the search. In direct contrast to traditional search engines tailored to take a single line of input, the in-IDE nature of Scotch allows it to automatically infer code context during search and utilize it for search results. Hence, we propose the task 'contextual code search' and present an analysis of how this code context can help improve the relevance of search results. Since no existing dataset could fit our task of contextual code search, we collect and contribute a dataset of about 19M functions from GitHub repositories with permissive licenses, which is the first large-scale dataset openly available for the task of contextual code search. We also present a small, manually-curated test set to assess the code ranking quality for code search. We finetune the CodeBERT model (Feng et al., 2020) to perform code search given a natural language query with and without surrounding code context. Results from automated as well as human evaluation suggest that the inclusion of code context in search significantly improves the retrieval of the correct code snippet and slightly hinders the ranking quality among annotated code snippets. Our work provides motivation and resources for future research into contextual code search.

1 INTRODUCTION

Programming has become increasingly prevalent with the rise of code utility in our lives. For example, GitHub, the largest code repository, recorded more than 16M new users and 61M new repositories in 2021.[1] With the rise in code volume, code search becomes an important problem and many tools have been proposed to solve this problem. Most code search engines depend on keyword-based or pattern matching between the query, which is usually a short natural language intent or regular expressions, and the database of code snippets to find relevant search results. Several *semantic code search* approaches have also been proposed to perform code search, which leverage a deeper understanding of code via learned vector representations (Husain et al., 2019; Heyman & Van Cutsem, 2020). However, in practice, there has been relatively less work on leveraging semantic code search methods to improve and facilitate real-time code search for multiple programming languages. In this work, we present Scotch, an in-IDE semantic code search tool that currently supports four programming languages. Since Scotch is an in-IDE tool and thereby has access to user's current codebase, it offers additional flexibility in terms of search input compared to traditional search engines tailored to take only a single line of query. Hence, we also explore this flexibility to improve code search quality.

A comprehensive survey on code search tools by Liu et al. (2021) suggests that existing code search tools likely return many irrelevant code snippets in response to the queries which are missing complete and precise semantics. Husain et al. (2019) also discuss the presence of ambiguity in the interpretations of search queries without appropriate context. The ambiguity/incompleteness in query

[1] https://octoverse.github.com/

Code Context:

```
import json
import sys
from collections import OrderedDict
from pygments import highlight
from pygments.formatters import TerminalFormatter
from xml.etree import ElementTree as ET
import xmlformatter
```

Query: Parses data and formats it.

Our Model:

```
def format_code(data, is_xml=False, sort_keys=False):
    if is_xml:
        ET.fromstring(data)
        formatter = xmlformatter.Formatter(indent=2, indent_char=' ',
                    encoding_output='UTF-8', preserve=['literal'])
        return formatter.format_string(data)
    else:
        obj = json.loads(data, object_pairs_hook=OrderedDict)
        return json.dumps(obj, sort_keys=sort_keys, indent=2,
                    ensure_ascii=False).encode('UTF-8')
```

Baseline:

```
def get_gogdata(self, url, *args, **kwargs):
    resp = self.get(url, *args, **kwargs)
    gogdata = {}
    for script in find_scripts(resp.text):
        matches = GOGDATA_RE.finditer(resp.text)
        for match in matches:
            subkey = match.group(1)
            value = match.group(2)
            value_parsed = json.loads(value)
            if subkey:
                data = {subkey: value_parsed}
            else:
                data = value_parsed
            gogdata.update(data)
    return gogdata
```

Figure 1: Example of an incomplete natural language query for code search (left). The query does not specify what type of file to parse and format. However, code context provides hints about the types of files to parse and APIs to use, thereby improving the search result from our contextual code search model (center), as compared to the result from the baseline model (right) which only takes the natural language query as input.

is not surprising given the highly contextual nature of code in terms of APIs and libraries being used, data structures being referred to, etc. Multiple works have adopted *query reformulation* i.e., augmenting the query with related words from Stack Overflow (Sirres et al., 2018), relevant APIs class-names (Zhang et al., 2017), or synonyms (Lemos et al., 2014) to improve search results. Li et al. (2016) expand the query with API usage patterns represented using method call relationship graphs and demonstrate improved code retrieval for jQuery framework. In search of easily extensible and scalable framework to augment search queries, we study a simpler, language-agnostic method for query augmentation i.e., providing the surrounding context of the natural language (NL) query, which can be automatically inferred from an IDE, as input to the search algorithm and thus enabling *contextual code search*. Here, code context refers to code from the same file that appears before the NL query that contains the intent for the subsequent lines of code. We hypothesize that code context contains rich supplementary information about user's intent during search as shown in Figure 1 and study how incorporating such code context affects search results using multiple evaluation settings.

We collect a massive dataset of about 19M functions with code context from open-source code repositories, with permissive licenses from GitHub, spanning four programming languages i.e., Python, Java, Javascript and Go. Out of those 19M functions, nearly 4M have corresponding docstrings. Following Husain et al. (2019) and Miceli-Barone & Sennrich (2017), we assume that the docstrings represent natural language queries. Next, we finetune CodeBERT (Feng et al., 2020), a pretrained model trained on bimodal data of natural language and programming languages to maximize similarity between the concatenated context and query encoding and target code representation. We assume the dot product to represent similarity between two representations, and maximize the dot product between them using cross entropy loss.

We study how code context affects code search in two settings. First, we evaluate the search models' ability to rank the correct snippet among a set of distractors. Second, we provide a small, manually-curated test set to evaluate our models' ability to rank annotated code candidates (Husain et al., 2019). We find that the inclusion of code context to the query significantly improves the quality of topmost retrieved code snippet and the model's ability to select the correct candidate from a set of distractors. We investigate the scenarios where code context complements natural language queries to improve code search and find that the code context often helps by removing ambiguities primarily involving libraries to be used, data structures or objects being referred to, etc. We also find that the inclusion of code context slightly hinders the model's ability to rank annotated few code snippets.

Finally, we deploy Scotch as an extension for Visual Studio Code IDE.[2] To the best of our knowledge, Scotch is the first in-IDE tool that performs semantic code search in multiple programming languages and additionally leverages code context to complement search query. Currently, Scotch can perform semantic code search for four programming languages (Python, Javascript, Java, and Go) from within the Visual Studio Code IDE. For each search query, Scotch uses ScaNN (Guo et al., 2020b)

[2]https://visualstudio.microsoft.com/vs/

Features	Scotch	CSN	PyTorrent
Total Functions	19.5M	6.4M	2.8M
Functions with docs	4.7M	2.3M	2.8M
Code Context	Yes	No	No
No. of Programming Languages	4	6	1
Source	GitHub	GitHub	Python libraries

Table 1: Comparison of the Scotch dataset with CSN (Husain et al., 2019) and PyTorrent (Bahrami et al., 2021). Numbers are rounded-off.

to calculate dot products between query and target vector representations efficiently in real-time. We implement separate ScaNN searchers for each programming languages, thereby giving rise to modularity that makes it easy to add more programming languages to Scotch going forward.

2 Related Work

Natural Language Code Search. Natural language code search is the task of finding relevant code from a collection of code based on a natural language query. Although code search engines that exploit keyword or pattern matching exist, multiple recent attempts have explored deep learning methods to perform code search (Gu et al., 2018; Sachdev et al., 2018; Cambronero et al., 2019; Heyman & Van Cutsem, 2020). Shuai et al. (2020) employ a LSTM network and co-attention mechanisms to embed code and query for search. Husain et al. (2019) introduce the CodeSearchNet corpus which contains nearly 2M pairs of NL query and target code and an additional 4M code snippets for pretraining. Feng et al. (2020) propose the bi-modal CodeBERT model which is pretrained on a hybrid objective function of Masked Language Modelling (MLM) and Replaced Token Detection (RTD), to support downstream applications involving code and NL queries. Gu et al. (2021) improve the vector representation of code by introducing tree-serialization on Abstract Syntax Tree (AST) of the code. Salza et al. (2021) leverage pretrained BERT (Devlin et al., 2019) to finetune on source code for improved code search. Guo et al. (2020a) and Ling et al. (2021) perform structure-aware encoding of code snippets with the use of graphs and ASTs. We finetune CodeBERT with context, query and target code triples for the task of code search.

Code-to-Code Search. Mukherjee et al. (2020) proposed the task of code search without any explicit query. They used code context as the input and the query is inferred from the context. To the best of our knowledge, there have not been any further works exploring this direction.

Code Search User Interfaces. Most existing code search tools are browser-based. Prominent examples include Sourcegraph[3] and GitHub Code Search[4]. In contrast, Xu et al. (2021) built a code retrieval plugin for PyCharm, which utilizes the Bing[5] Search Engine to search for relevant code in Stack Overflow in response to a query. Stack Overflow also provides an internal search engine. We present the first contextual semantic code search tool that functions from within an IDE.

3 Scotch Dataset

Existing code corpora, notably Husain et al. (2019), do not contain code context that can be leveraged to improve code search. Hence, we collect the Scotch dataset, which is a large dataset of functions along with their code context. Scotch contains about 19.5 million functions spanning four programming languages: Python, Javascript, Java, and Go. To the best of our knowledge, it is the largest openly available corpus of functions code. Table 1 presents detailed comparison of the Scotch dataset with existing datasets. The functions are collected from open-source repositories from GitHub. We use the SEART GitHub search engine (Dabic et al., 2021) to obtain a list of open-source repositories fulfilling our selection criteria. To ensure quality, we only list repositories with a license,

[3]https://sourcegraph.com/search
[4]https://cs.github.com/
[5]https://www.bing.com/

Programming Language	Train	Validation	Test
Python	1,438,036	179,754	179,755
Javascript	609,814	76,227	76,227
Java	604,892	75,611	75,612
Go	547,884	68,485	68,486
Total	3,200,626	400,077	400,080

Table 2: Number of natural language query and code pairs for various programming languages in each split of the Scotch dataset.

Language	Total	R=0	R=1	R=2	R=3
Python	230	31	57	53	89
Javascript	248	136	44	19	49
Java	242	39	104	64	35
Go	237	20	82	101	34
Total	957	226	287	237	207

Table 3: The distribution of annotations in the manually curated test set of Scotch for various programming languages and each rating in the relevance scale 0-3 (R).

5 or more stars, and exclude forks. We clone those repositories using helper scripts from Coooper et al. (2021) and extract functions, along with their detailed information including identifier name, URL, docstring and code context, from raw code files using our lightweight parser built on top of `function_parser` (Husain et al., 2019). The parser allows us to extract functions from code files. We add the capability to extract appropriate code context for each function. Additionally, our parser extracts both named functions, anonymous functions, and also method definitions inside classes for Javascript. On top of this, we use the following filtering criteria:

- Exclude functions without explicitly permissive licenses i.e. Includes licenses: MIT License, Apache License 2.0, BSD 3-Clause "New" or "Revised" License, BSD 2-Clause "Simplified" License, and ISC License.
- Exclude single-lined functions.
- Exclude functions whose docstrings contain non-Engligh characters.
- Files containing multiple same functions are excluded.

We perform de-duplication on the resulting functions to avoid multiple copies of the same function. Following Coooper et al. (2021), we remove duplicate functions by obtaining a list of alphanumeric characters in a function and filtering out functions with the same sequence of alphanumeric characters. By doing so, we retain 19.5M functions in the final dataset. About 4.7M of these functions contain corresponding docstrings. To obtain a NL-Code dataset, we use following filtering criteria:

- Following Husain et al. (2019), functions with 'test' keyword in function identifier are excluded.
- Functions with no docstring or docstrings less than 3 tokens separated by white-space are excluded.
- Comments are removed from function code and corresponding code context.

Finally, we obtain a dataset of 4M functions with corresponding docstrings and code context. We use 80/10/10 split to get the train, validation and test sets respectively. Detailed statistics of the resulting dataset for various programming languages and by each split is presented in Table 2.

Additionally, we curate a small test set to assess the ranking quality of our search models. For each of the four programming languages, we choose 50 random queries from the automatically curated test set. For each query, we collect the top-5 search results from an ensemble of our models with and without access to code context (see Section 4) and remove duplicate code candidates for all search results. We then ask programmers proficient in the respective languages to rate the relevance of each search result given the natural language query and code context. We follow the same relevance scale as Husain et al. (2019) i.e. a scale of 0-3 where higher score represents more relevance to the

query and context. We collect 957 annotations in total. Table 3 provides a summary of the collected annotations and relevance scores for each programming language.

4 OUR MODELS

In this section, we describe the two models used in our experiment i.e. CodeBERT for code search with context and without context (baseline).

4.1 CODEBERT (BASELINE)

CodeBERT (Feng et al., 2020), a bimodal pretrained model trained on code and natural language, achieves superior performance compared to neural baselines reported in Husain et al. (2019). Hence, we use CodeBERT model as encoder and use the representation of the [CLS] token as the high-dimensional representations of code and query. We prepend each NL query with a token to indicate the programming language it deals with. The target code snippet is also concatenated with the identifier of the function that includes the name of the parent class whenever applicable. We use separate encoders for NL query and code, although same encoders are used for all programming languages. For each query q_i and corresponding code c_i, we use cross entropy loss to maximize the inner product of correct NL-code pair i.e. (q_i, c_i) and minimize the inner product of distractor NL-code pairs i.e. (q_i, c_j) $\forall\, j \neq i$. Formally, for each batch with batch size of N, the loss is given by

$$1/N \sum_{i=1}^{N} -\log \frac{\exp(I(q_i, c_i))}{\sum_j \exp(I(q_i, c_j))}$$

where $I(q_i, c_j)$ is the inner product of vector representations of q_i and c_j obtained from the encoders.

4.2 CODEBERT WITH CODE CONTEXT

In order to adapt CodeBERT for contextual code search, we finetune the CodeBERT model as explained in Sec. 4.1, however, code context is prepended to the query and passed into the CodeBERT encoder to get the representation [CLS] token. For a triple of context, query and code (x_i, q_i, c_i), $[x; q]$ is the code context concatenated with natural language query, which is passed into the encoder as input. Code is encoded in an identical fashion as Sec. 4.1. The loss is given by,

$$1/N \sum_{i=1}^{N} -\log \frac{\exp(I([x_i; q_i], c_i))}{\sum_j \exp(I([x_i; q_i], c_j))},$$

where $I([x_i; q_i], c_j)$ is the inner product of vector representations of $[x_i; q_i]$ and c_j from the encoders.

5 SYSTEM DESCRIPTION

In this section, we describe the overall Scotch tool, the model and search algorithm deployed for code search, and the features of the tool.

5.1 CODE SEARCH MODEL

We use the CodeBERT model as described in Sec. 4.2 for contextual code search in Scotch. The model is finetuned on 4M samples from the Scotch dataset containing context, query and target code. However, the search is conducted over the entire Scotch dataset i.e. nearly 19M functions. The vector representations for these functions are precomputed using the finetuned CodeBERT model.

5.2 DEPLOYMENT

Every incoming query (along with the context) is encoded in real-time using the CodeBERT model and used for a similarity-based search from the database of functions. In practice, it is not feasible to calculate inner product between query vector and each of the function vectors due to the large

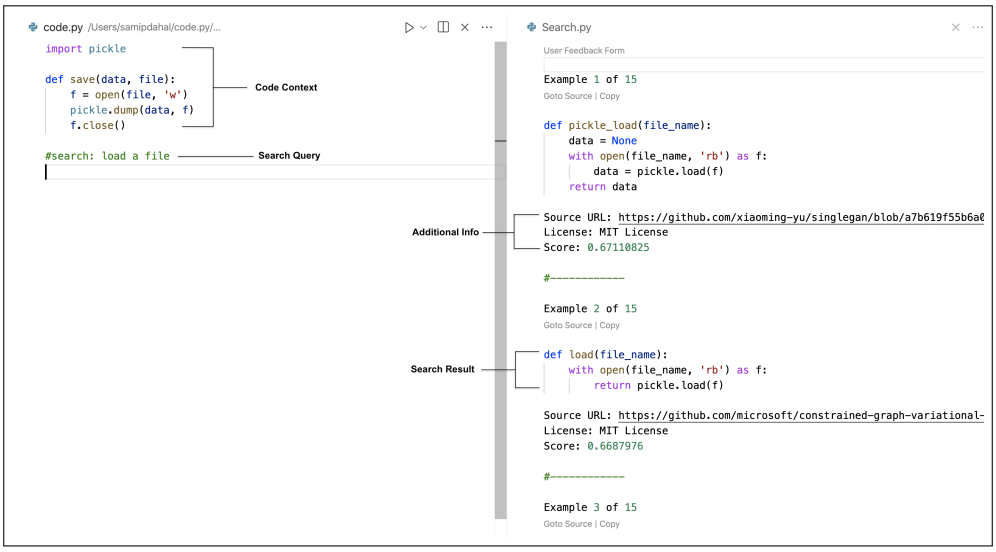

Figure 2: Annotated user interface of the Scotch Visual Studio Code extension. Scotch identifies a query when the user beings a comment with the token 'search' in their current working file (left) and automatically grabs the preceding code context. In response, Scotch displays a file (right) containing the search results. In this example, the user's query *load a file* is ambiguous but Scotch resolves the ambiguity using contextual code search.

number of functions in our database. Hence, we use ScaNN (Guo et al., 2020b), an Approximate Nearest Neighbor (ANN) search algorithm. With ScaNN, we can query approximate neighbors in sub-linear time. We build separate ScaNN searchers for each programming languages. We normalize all function vectors and query vector to keep the inner product between 0 and 1. The functions with highest inner product scores (similarity) are returned as ranked search result.

5.3 EXTENSION

We build a Visual Studio Code extension named Scotch to facilitate in-built contextual code search for IDE users. To the best of our knowledge, this is the first in-IDE semantic code search tool supporting multiple languages and contextual search. Due to its in-IDE nature, a developer can search for code without leaving the IDE. Moreover, it demonstrates that we can utilize code context easily to improve code search from inside an IDE, in a similar fashion to code auto-completion. This feature is not available in traditional search engines tailored to take only query as input. Figure 2 shows a screenshot of the user interface of our extension. The major features are highlighted below:

Query: Users can write query in any commented line in the current working file (left panel in Figure 2). Each query should be prepended with the prompt 'search:'.

Code Snippets: Given a query and the programming language of current user's document, Scotch uses ScaNN searcher of the corresponding language to retrieve relevant code snippets. Resulting code snippets are displayed in a separate document in right hand side. For each query, Scotch returns the top 15 relevant code snippets.

Source Code URL: For each code snippet, we also provide the URL where the source code is located. This feature can help developers understand the context of the source code and further explore the source if useful.

License: We also provide the license of the repository the source code belongs to. This is crucial because many licenses imposes certain restrictions in use cases. However, all the code snippets in the Scotch dataset belong to GitHub repositories with explicitly permissive licenses as mentioned in Section 3.

Search Quality: We provide the search score for each code snippet to help users assess the quality and relevance of search result. Our search score is the inner product between the normalized query vector and code vectors, and ranges between 0 and 1.

Model	Python	Javascript	Java	Go	MRR_{Avg}	MRR_{All}
CodeBERT (baseline)	0.8828	0.7673	0.8872	0.9079	0.8613	0.9168
CodeBERT + code context (ours)	**0.9535**	**0.9069**	**0.9726**	**0.9716**	**0.9511**	**0.9739**

Table 4: Results on the test set of the Scotch dataset using MRR metric.

Model	Python	Javascript	Java	Go	NDCG_{Avg}
CodeBERT (baseline)	**0.8537**	**0.6898**	0.7542	**0.8977**	**0.7988**
CodeBERT + code context (ours)	0.8374	0.6283	**0.8107**	0.8463	0.7807

Table 5: Comparative results using NDCG metric on manually annotated test set.

User Feedback: A user feedback is linked at the top of each search document. We hope to improve Scotch in future based on user feedbacks.

Installation: The extension can be installed free of cost either from VS Code Marketplace or VS Code extension itself. It is compatible with VS Code 1.59.0 and up.

6 EVALUATION

Following (Husain et al., 2019), we use Mean Reciprocal Rank (MRR), computed as $\frac{1}{N}\sum_{i=1}^{N}\frac{1}{rank_i}$ where N is the total number of queries, as our automatic evaluation metric. During evaluation, we use a batch size of 5000 for each query i.e. N=5000 in our implementation of MRR. The ranking is done based on the inner product of vector representation of query and target code snippets. We calculate MRR scores for all languages independently, where all the distractor snippets belong to the same programming language. Following (Feng et al., 2020), we use macro-average MRR of all languages, represented by MRR_{Avg}, as an overall evaluation metric. We also calculate MRR_{All}, where the distractors are sampled from all programming languages.

We also perform a comparative human evaluation of the predictions from the two models described in Sec. 4.[6] For each sample, annotators are presented with the NL query, code context and the top predictions from the two models in a randomized order. They are asked to indicate the prediction that is more relevant to the given query and context.

In addition, we use normalized discounted cumulative gain (NDCG) score as the metric to evaluate models on the manually-curated test set described in Sec. 3. In contrast to MRR, NDCG deals with multiple relevant snippets and is position-sensitive, which makes it an appropriate metric to evaluate ranking quality. We use the NDCG metric to evaluate the ability of our models to rank annotated code snippets in the test set.

7 RESULTS

Quantitative Results. Table 4 shows results from CodeBERT models with and without code context on the test set of Scotch dataset, using MRR metric as evaluation. The model with access to code context (row 2 in Table 4) significantly outperforms the baseline model (row 1 in Table 4) without access to code context across all programming languages. The largest improvements come for Javascript, followed by Java, Python, and Go programming languages. The macro-average score MRR_{Avg} is nearly 9% higher for contextual code search. We also see 6% improvement with the MRR_{All} metric. This suggests the code context can provide the code search model important clues about the task in hand.

Results on the manually collected test set using NDCG metric as evaluation are shown in Table 5. The inclusion of code context results in a slight degradation in the contextual code search model's ability to rank annotated code snippets from our manually-curated test set (see row 2 in Table 5). On

[6]Our annotators are computer science students and software developers with a sound knowledge of the respective programming languages.

Programming Language	Win%	Lose%	Tie%
Java	20.45	9.09	70.45
Javascript	57.49	10.0	32.5
Go	22.44	8.16	69.38
Python	13.04	17.39	69.56

Table 6: Human Evaluation. Win% represents the % times predictions from contextual CodeBERT were chosen over the baseline model without code context and Lose% represents vice-versa. Tie% represents % times both model predictions were found to be correct.

Model	Rank > 1	Rank > 15
CodeBERT	46475	8593
CodeBERT with code context	16011	1514

Table 7: Analysis of ranks of correct code snippets in the validation set.

average, the NDCG score for the model without code context is 1.8% higher than the model with code context. We hypothesize that selectively including only the important aspects of code context might improve the ranking ability and leave it to future works to study this in detail.

Human Evaluation. We select 50 random queries for each programming languages from the validation set and build ScaNN searchers with all the functions in our validation set for both of our models. Then, we query respective ScaNN searchers to obtain top prediction for each model. These samples are presented to our annotators as outlined in Sec. 6. Annotators are also allowed to tag queries as incoherent wherever applicable, since these samples are drawn from the automatically collected dataset. Approximately 10% of the samples were tagged as incoherent and discarded; results on remaining samples are presented in Table 6. We see that the model with code context significantly outperforms our baseline model in three out of four programming languages, further strengthening the case for contextual code search.

Overall, we find that our contextual code search model improves the quality of search results in two out of three evaluation settings and shows promise as a worthwhile research direction.

8 Analysis of Search Quality

We further studied predictions from our models on the validation set of the Scotch dataset to assess their search quality. We calculate the rank of the correct code snippet as predicted by both of the CodeBERT models within a set of distractors ($N = 5000$). Table 7 shows the number of instances where the rank of correct code snippet was more than 1 (i.e. correct code snippet not in top-1 result) and more than 15 (i.e. correct code snippet not in top-15 result. This is to simulate real-world deployment of Scotch where we return the top 15 search results for each query). We can see that the search result of the model with code context is significantly better both in terms of top-1 result and top-15 results. We find that one obvious way code context helps search is when keywords and function names present in the code context also occur in the target code. In many such cases, the model mirrors keyword matching. More interestingly, we find that the code context also helps the model when the query is ambiguous or incomplete. Figure 1 shows an example of an incomplete and ambiguous query. When provided with code context, the contextual code search model is able to find the code snippet that makes correct library calls and implements the functionality.

9 Conclusion

In this paper, we present Scotch, a semantic code search tool that operates within an IDE. We experiment with the use of code context to supplement natural language query. We collect a dataset of about 19M functions with code context and experiment with models that do and do not utilize code context for code search. We found that code context supplements natural language query with additional information about the search and improves search results. Hence, we add the functionality to conduct real-time contextual and semantic code search automatically using Scotch.

10 ETHICS

Dataset. In the course of preparing the Scotch dataset, we have made sure to understand the legal implications before using and/or redistributing open-source code. Code under certain restrictive licenses have limited use cases. Hence, as mentioned in Section 3, we collected code from repositories with permissive licenses. Moreover, for each search result, we also display the permissive license the code is under and include a link to the URL of original source code to ensure attribution to the original authors.

Models. We do not anticipate any unintended effects of deploying our code retrieval model through a freely available IDE extension.

REFERENCES

Mehdi Bahrami, NC Shrikanth, Shade Ruangwan, Lei Liu, Yuji Mizobuchi, Masahiro Fukuyori, Wei-Peng Chen, Kazuki Munakata, and Tim Menzies. Pytorrent: A python library corpus for large-scale language models. *arXiv preprint arXiv:2110.01710*, 2021.

Jose Cambronero, Hongyu Li, Seohyun Kim, Koushik Sen, and Satish 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*, pp. 964–974, 2019.

Nathan Coooper, Artashes Arutiunian, Santiago Hincapié-Potes, Ben Trevett, Arun Raja, Erfan Hossami, Mrinal Mathur, and contributors. Code Clippy Data: A large dataset of code data from Github for research into code language models, July 2021. URL `https://github.com/ncoop57/gpt-code-clippy`.

Ozren Dabic, Emad Aghajani, and Gabriele Bavota. Sampling projects in github for MSR studies. In *18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021*, pp. 560–564. IEEE, 2021.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pp. 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics. doi: 10.18653/v1/N19-1423. URL `https://www.aclweb.org/anthology/N19-1423`.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pp. 1536–1547, Online, November 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.findings-emnlp.139. URL `https://aclanthology.org/2020.findings-emnlp.139`.

Jian Gu, Zimin Chen, and Martin Monperrus. Multimodal representation for neural code search. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 483–494. IEEE, 2021.

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

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*, 2020a.

Ruiqi Guo, Philip Sun, Erik Lindgren, Quan Geng, David Simcha, Felix Chern, and Sanjiv Kumar. Accelerating large-scale inference with anisotropic vector quantization. In *International Conference on Machine Learning*, 2020b. URL `https://arxiv.org/abs/1908.10396`.

Geert Heyman and Tom Van Cutsem. Neural code search revisited: Enhancing code snippet retrieval through natural language intent. *arXiv preprint arXiv:2008.12193*, 2020.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*, 2019.

Otávio AL Lemos, Adriano C de Paula, Felipe C Zanichelli, and Cristina V Lopes. Thesaurus-based automatic query expansion for interface-driven code search. In *Proceedings of the 11th working conference on mining software repositories*, pp. 212–221, 2014.

Xuan Li, Zerui Wang, Qianxiang Wang, Shoumeng Yan, Tao Xie, and Hong Mei. Relationship-aware code search for javascript frameworks. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 690–701, 2016.

Xiang Ling, Lingfei Wu, Saizhuo Wang, Gaoning Pan, Tengfei Ma, Fangli Xu, Alex X Liu, Chunming Wu, and Shouling Ji. Deep graph matching and searching for semantic code retrieval. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 15(5):1–21, 2021.

Chao Liu, Xin Xia, David Lo, Cuiyun Gao, Xiaohu Yang, and John Grundy. Opportunities and challenges in code search tools. *ACM Computing Surveys (CSUR)*, 54(9):1–40, 2021.

Antonio Valerio Miceli-Barone and Rico Sennrich. A parallel corpus of python functions and documentation strings for automated code documentation and code generation. In *Proceedings of the Eighth International Joint Conference on Natural Language Processing (Volume 2: Short Papers)*, pp. 314–319, 2017.

Rohan Mukherjee, Swarat Chaudhuri, and Chris Jermaine. Searching a database of source codes using contextualized code search. *arXiv preprint arXiv:2001.03277*, 2020.

Saksham Sachdev, Hongyu Li, Sifei Luan, Seohyun Kim, Koushik Sen, and Satish Chandra. Retrieval on source code: a neural code search. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pp. 31–41, 2018.

Pasquale Salza, Christoph Schwizer, Jian Gu, and Harald C Gall. On the effectiveness of transfer learning for code search. *arXiv preprint arXiv:2108.05890*, 2021.

Jianhang Shuai, Ling Xu, Chao Liu, Meng Yan, Xin Xia, and Yan Lei. Improving code search with co-attentive representation learning. In *Proceedings of the 28th International Conference on Program Comprehension*, pp. 196–207, 2020.

Raphael Sirres, Tegawendé F Bissyandé, Dongsun Kim, David Lo, Jacques Klein, Kisub Kim, and Yves Le Traon. Augmenting and structuring user queries to support efficient free-form code search. *Empirical Software Engineering*, 23(5):2622–2654, 2018.

Frank F Xu, Bogdan Vasilescu, and Graham Neubig. In-ide code generation from natural language: Promise and challenges. *arXiv preprint arXiv:2101.11149*, 2021.

Feng Zhang, Haoran Niu, Iman Keivanloo, and Ying Zou. Expanding queries for code search using semantically related api class-names. *IEEE Transactions on Software Engineering*, 44(11): 1070–1082, 2017.