Learning to search for software artifacts

Breandan Considine, Xujie Si, Jin Guo

June 14, 2021

1 Introduction

Humans are adept information foraging agents. We can quickly find relevant information in a large corpus by recognizing and following textual landmarks. Software projects are composed of a variety of semi-structured documents containing many clues where relevant information may be found. In this work, we train a model to navigate and read software artifacts like source code and documentation, in order to facilitate common programming tasks such as code search, completion, or defect prediction.

Early work in program learning realized the importance of graph-based representations (?), however explicit graph construction requires extensive feature-engineering. More recent work in program synthesis has explored incorporating a terminal (?), graphical (?) or other user interface to explore the space of valid programs, however do not consider the scope or variety of artifacts in a software project. Others have shown the feasibility of learning a local graph (?) from source code, but still require an explicit parser to form the initial graph and adaption in settings where style, terminology and document structure vary remains a challenge.

Imagine a newly-hired developer with no prior knowledge about a closed-source project. She receives access to the team's Git repository and is assigned her first ticket: "Fix test broken by `0fb98be`". After locating the commit and familiarizing herself with the code, she writes a query to StackOverflow, finds a relevant solution, copies the code into the project, makes a few changes, presses run, and the test passes.

In order to achieve a similar task, an information-seeking agent first must explore a local database for information, construct a query, and search a remote database. Similar to a human developer, it might traverse the project to gather information, taking note of various keywords, APIs and design patterns. While traversing the project, the model constructs a projectspecific knowledge graph, which it uses to locate other relevant artifacts written by authors who have solved the same problem.

2 Prior literature

Prior work in the code search literature explores the text-to-code (**?**) setting, in which queries are typically plaintext strings composed by the user, or code-to-code (**?**) setting where the query might be a synthetic string or vector. Searches are typically compared using using, e.g., mean reciprocal rank (MRR), mean average precision (MAP), normalized discounted cumulative gain (NDCG), precision and recall at top N-results (P/R@N), or similar metrics. Although some (**?**) do incorporate other features from the local document, prior work does not consider the query in the context of a broader project. The ability to align contextual features from the surrounding project, we argue, is essential to delivering semantically relevant search results.

Name	Type	Key contribution	Evaluation method
Our Work	Either-to-Either	Learning to search	Prediction accuracy
FaCoY (**?**)	Code-to-code	Similarity search	MRR
Yogo **?**	Code-to-code	Equational reasoning	Case study
CSNet (**?**)	Text-to-code	Big code benchmark	MRR
Aroma (**?**)	Code-to-code	Structural Search	Recall@N
AUSearch (**?**)	Code-to-code	Type resolver	Highlight accuracy
Cambronero (**?**)	Text-to-code	Semantics	Answered@N
DeepCS (**?**)	Text-to-code	Deep learning	FRank, *@N, MRR
Lancer (**?**)	Code-to-code	BERT embedding	HitRate@k, MRR

Table 1: Comparison of existing code search tools.

Our work seeks to help users discover software artifacts sharing common features with the surrounding context by automatically constructing a search query and ranking results by contextual similarity. Unlike prior literature which requires a language-specific parser (**?**), by aligning learned features from the surrounding document, we can more easily adapt to unconstrained settings such as work-in-progress code and freeform Q&A text, which may contain incomplete or syntactically invalid text at the time of query construction. While partly inspired by research on duplicate detection, we make no assumptions about provenance or user intent.

3 Method

We fetch a dataset of repositories sampled repositories on GitHub, containing a mixture of filetypes representing source code and natural language artifacts. From each repository, we index all substrings of every line in every file using a variable-height radix tree producing a multimap of `kwdIndex: String -> List<Location<F, O>>` of `String` queries to file-offset pairs. We also encode CodeBERT (**?**) sequence embeddings to substrings `knnIndex: Vector -> Location<F, O>` using a Hierarchial Navigavble Small World Graph (**?**) (HNSWG).

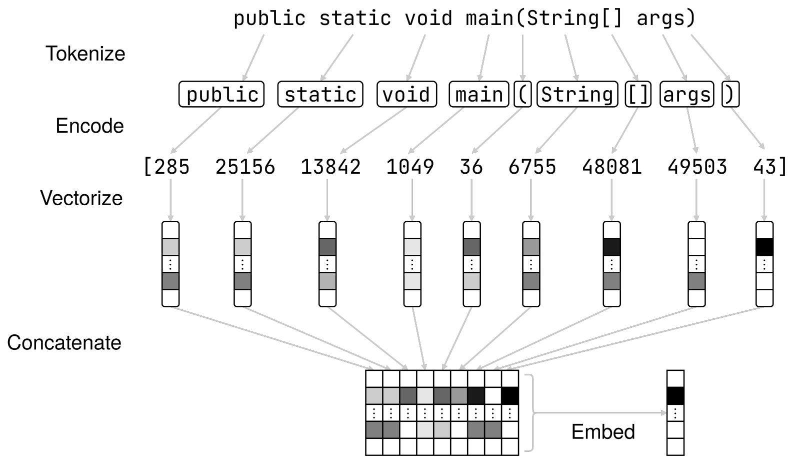


Figure 1: CodeBERT takes a unicode sequence and emits a vector sequence, which we accumulate into a single vector.

For each token in a string, CodeBERT emits a length-768 vector, so a line with n tokens produces a matrix of shape $\mathbb{R}^{768\times(n+1)}$, the first column of which contains the final hidden layer activations. We concatenate the CodeBERT sequence '`<s>`', encode the source tokens and vectorize the encoded sequence using the vocabulary, then take the first row as our sequence embedding as depicted in Fig. **??**. In § **??**, we compare various distance metrics to fetch the nearest sequence embeddings in our database and compare precision and recall across various metrics.

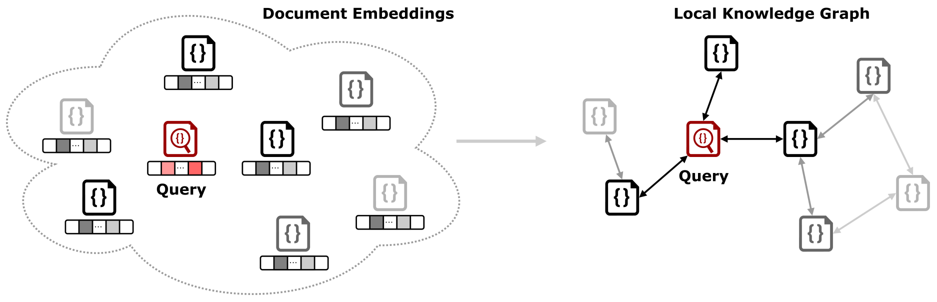


Figure 2: To compute our query neighborhood, we traverse the HNSWG up to max depth d, i.e. $d = 1$ fetches the neighbors, $d = 2$ fetches the neighbors-of-neighbors, and $d = 3$ fetches the neighbors-of-neighbors-of-neighbors.

For a given query and context, we first compute the context embedding and using a distance metric, fetch the k-nearest neighboring documents in latent space, forming the depth-1 nodes in our graph, then repeat this procedure recursively, pruning with a beam search heuristic based on the total path length in latent space. This procedure is depicted in Fig. **??**.

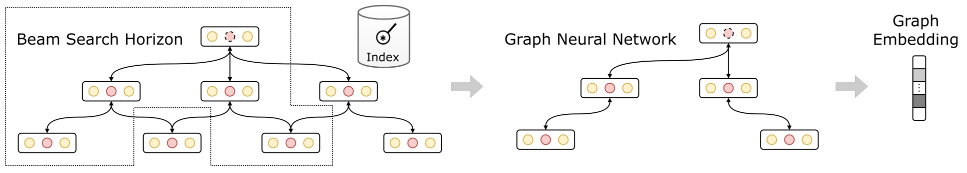


Figure 3: Unlike language models which directly learn the data distribution, our model is designed to query an unseen database only available at test time. The model scores and selectively expands a subset of promising results within the database using a beam search heuristic, then runs message passing over the resulting graph to obtain the final task prediction.

Consider the depth-1 beam search procedure (Fig. **??**): We can use either vector search or keyword search to perform the context expansion, although vector search has the disadvantage of needing to rebuild the vector embedding index at each step during training. Keyword search is comparatively cheaper, as the index only needs to be built once for each repo on MiniGithub.

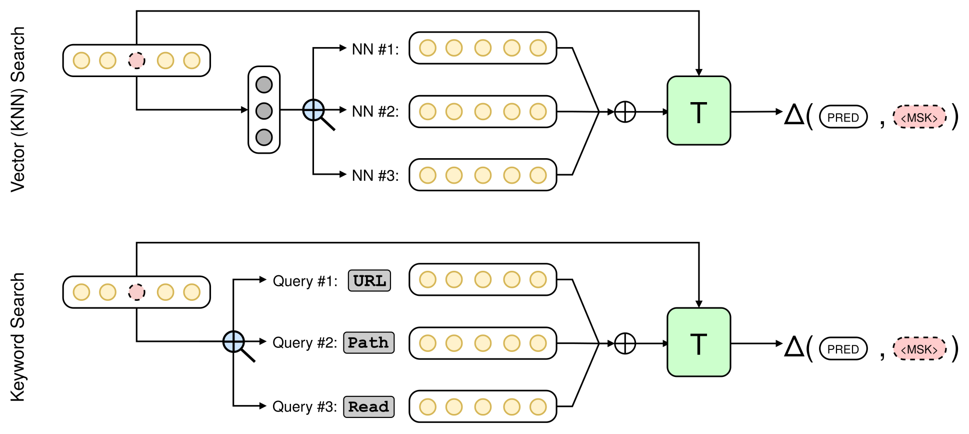


Figure 4: In the vector search procedure, we compute an embedding for the masked source context, then retrieve the k-nearest sequence embeddings in our database. In the keyword search, we compute an n-hot vector to select the keywords, and retrieve the nearest matching sequences in our index. In both cases, we embed the results, feed them into a transformer, compute a loss between the predicted output and the masked token, and backpropagate.

Once the beam search procedure is complete, we have a graph representing the neighborhood of the nearest neighboring code fragments in the parent project. This forms a so-called *virtual knowledge graph*, with nodes decorated by the CodeBERT sequence embeddings and edges decorated with the direction vector between documents. We then run p rounds of message passing to obtain our graph embedding or *neighborhood summary vector* (Fig. **??**).

4 Experiments

In this work, we attempt to understand the relationship between entities in a software project. Our research seeks to answer the following questions:

1. Which contexts in a software project share mutual information?
2. To what degree can we claim the model has learned to:
   (a) Locate contextually relevant artifacts within a software project?
   (b) Comprehend the semantic content of the artifacts traversed?

(c) Apply the knowledge gathered to perform the assigned task?

In contrast with classical code completion models which only require a file-local context, our method is designed to navigate an entire project. In the following experiments, we compare completion accuracy with a vanilla sequence prediction model, as well as an AST-structured sequence prediction model trained from a corpus of Java projects on the same task.

We hypothesize that by jointly learning to choose locations in a project over which to attend while solving a downstream task, such as masked sequence completion, our model will produce a feature representation capable of locating and extracting information from semantically relevant contexts. We evaluate our hypothesis both qualitatively and quantitatively.

In our first set of experiments, we try to understand what is shared between sequences mapped to similar latent vectors. Do similar sequences share salient keywords? Are those keywords relevant to the task?

In our second experiment, we try to measure the information gain from including and excluding various filetypes through ablation. For graphs containing filetypes which include Markdown or Java, what kinds of information do these resources provide and which categories are most salient?

In our third and final set of experiments, we compare performance across hyperparameters. Does contextual expansion lead to better task performence for a given sequence prediction task? By relaxing edge-construction criteria and increasing hyperparamers such as beam search budget, we would expect corresponding task performance to increase.

If our hypothesis is correct, the virtual knowledge graph will span both natural language and source code artifacts. If so, this would provide evidence to support the broader hypothesis (**?**) that documentation is a useful source of information. In addition to being useful for the prediction task itself, we anticipate our model could also be used for knowledge graph extraction and suggest semantically relevant code snippets to developers.

5 Evaluation

We evaluate on the following loss function, adapted from (**?**). Given a set of code and context pairs, $\mathbf{F} = (\mathbf{c}_i, \mathbf{d}_i)_{i=1}^{N}$, where $\mathbf{S} \sim_{i.i.d.} \mathbf{R}_d$:

$$\mathcal{L} = \frac{1}{N} \sum_{i}^{N} \log \left(\frac{\langle \varphi(\mathbf{c}_i), \varphi(\mathbf{d}_i) \rangle}{\sum_{j \neq i}^{N} \langle \varphi(\mathbf{c}_j), \varphi(\mathbf{d}_i) \rangle} \right) \tag{1}$$

In other words, we minimize the distance between true code and context pairs, while maximizing the distance between distractors $(\mathbf{c}_j, \mathbf{d}_i)_{i \neq j}$.

6 Results

Our preliminary results compare distance metrics (Fig. **??**), explore embedding quality (Fig. **??**) and visualize the synthetic knowledge graphs (Fig. **??**).

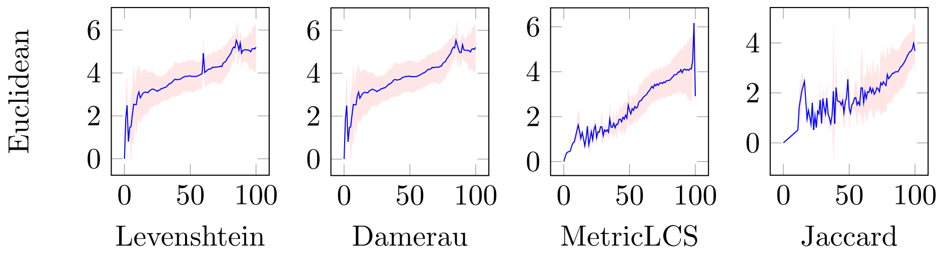


Figure 5: CodeBERT latent space distance correlates with string distance.

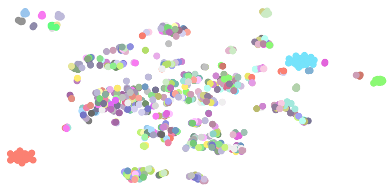
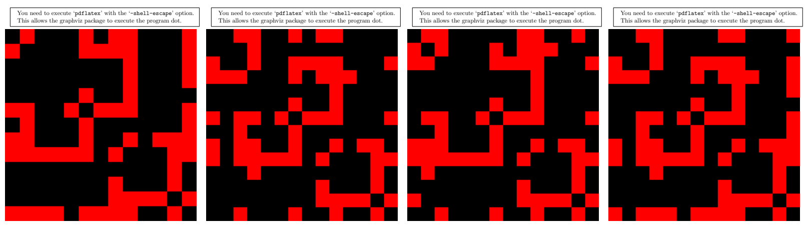


Figure 6: Reduced dimensionality TNSE embeddings colored by line length.



You need to execute 'pdflatex' with the '-shell-escape' option. This allows the graphviz package to execute the program dot.

You need to execute 'pdflatex' with the '-shell-escape' option. This allows the graphviz package to execute the program dot.

You need to execute 'pdflatex' with the '-shell-escape' option. This allows the graphviz package to execute the program dot.

You need to execute 'pdflatex' with the '-shell-escape' option. This allows the graphviz package to execute the program dot.

Figure 7: Example graphs sampled by beam search with $d = 10, w = 5$.