CodeScholar: Growing Idiomatic Code Examples

ANONYMOUS AUTHOR(S)

Programmers often search for usage examples for API methods. A tool that could generate realistic, idiomatic, and contextual usage examples for one or more APIs would be immensely beneficial to developers. Such a tool would relieve the need for a deep understanding of the API landscape, augment existing documentation, and help discover interactions among APIs. We present CodeScholar, a tool that generates idiomatic code examples demonstrating the common usage of API methods. It includes a novel neural-guided search technique over graphs that grows the query APIs into idiomatic code examples. Our user study demonstrates that in ≈70% of cases, developers prefer CodeScholar generated examples over state-of-the-art large language models (LLM) like `GPT3.5`. We quantitatively evaluate 60 single and 25 multi-API queries from 6 popular Python libraries and show that across-the-board CodeScholar generates more realistic, diverse, and concise examples. In addition, we show that CodeScholar not only helps developers but also LLM-powered programming assistants generate correct code in a program synthesis setting.

ACM Reference Format:
Anonymous Author(s). 2023. CodeScholar: Growing Idiomatic Code Examples. 1, 1 (October 2023), 24 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Suppose a programmer is learning to build an image classifier. The programmer is aware of the `torch` library in Python and some relevant Application Programming Interface (API) methods like `nn.Conv2D` and `nn.ReLU`, but is unsure how to use them. The programmer looks up the official documentation of these APIs and finds a vast list of parameters and a few usage examples:

```
>>> m = nn.Conv2d(16, 33, 3, stride=2)
>>> input = torch.randn(20, 16, 50, 100)
>>> output = m(input)
```

```
>>> m = nn.ReLU()
>>> input = torch.randn(2)
>>> output = m(input)
```

However, these examples don't describe the *realistic* and *common* usage of these APIs. Specifically, the programmer wants to know the real-world usage of these API methods. Ideally, a tool that returns a few code snippets (shown below) demonstrating some ways in which these APIs are used together frequently would be highly beneficial. We call them *idiomatic code examples*.

```
# Example 1
self.feat = nn.Sequential(
    nn.Conv2d(27, 16, kernel_size=3), nn.MaxPool2d(2), nn.ReLU(),
    nn.Conv2d(16, 32, kernel_size=3), nn.MaxPool2d(2), nn.ReLU())

# Example 2
self.feat = nn.Sequential(nn.Conv2d(64, 4, 3, padding=1), nn.ReLU())
```

Research in industry, such as at Facebook [Barnaby et al. 2020; Luan et al. 2019], has found that developers frequently query for such API usage examples in their internal code-to-code search tools, although they were deployed to get recommendations to modify or improve code. The

XXXX-XXXX/2023/10-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

widespread adoption of APIs has benefited developers. But to use them effectively, one requires a deep understanding of the API or access to high-quality documentation, which helps understand the API's behavior. While the former is entirely infeasible, the latter is often limited in practice. In addition, [Saied et al. 2015] find that these problems are exacerbated when developers use multiple APIs. They find that while API methods are consistently used along with other API methods, their co-usage relationships are seldom documented. [Casalnuovo et al. 2020] identify that developers prefer predictable and natural expressions in code. Developers use the term "*idiomatic*" to refer to meaningful code that other developers find natural [Allamanis and Sutton 2014; Hindle et al. 2016]. We believe these properties also apply to API usage examples; they are preferable when realistic and idiomatic. Overall, this indicates a need for *developer tools that augment existing documentation with searchable idiomatic usage examples.*

1.1 Existing Tools

A few existing techniques could potentially be used to search for idiomatic usage examples. First, are searchable online forums and websites like `StackOverflow`[1], `ProgramCreek`[2] providing code examples upvoted by other users. However, these tools are limited by popular queries from other users and also return an extensive list of results with no guarantees of idiomatic examples in them. Recently, automated techniques have been proposed to provide programmers with good API usage examples through code search [Glassman et al. 2018; Katirtzis et al. 2018]. However, they (1) struggle to rank the myriad of search results [Starke et al. 2009] and (2) lack idiomaticity that emphasizes frequent over niche usage. E.G. [Barnaby et al. 2020] attempts to solve this by mining multiple common usage examples. However, it focuses on single API queries and hence falls short in retrieving examples involving API co-usage. It also uses simple heuristics like minimum usage frequency that can be restrictive to newer APIs and more expressive usage examples. On another front, Large Language Models (LLM) like GPT-X [Brown et al. 2020; OpenAI 2023], Codex [Chen et al. 2021a], GitHub Copilot [3], and CodeLLaMa [Roziere et al. 2023] have demonstrated ability to generate code from natural language prompts [Agrawal et al. 2023; Su et al. 2023; Vikram et al. 2023]. However, code-related tasks are quite challenging for LLMs [Bubeck et al. 2023; OpenAI 2023; Touvron et al. 2023]. A particularly concerning failure mode has been termed *hallucination*, where the model generates incorrect or non-existent code. [Patil et al. 2023] find that hallucination is especially true with tasks involving APIs due to the vast space of arguments and other complexities.

1.2 CodeScholar

This paper introduces **CodeScholar**, a novel solution to the aforementioned challenges. CodeScholar takes a set of query API methods as input, which it then grows into full-blown code examples using a neural-guided search algorithm. The code examples generated are such that they contain all API methods in the query and are *idiomatic*, i.e., non-trivial and frequently occurring in a large corpus of real-world programs. Figure 1 illustrates this for a query API `json.load`.

CodeScholar capitalizes on a novel insight that idiomatic code examples are fundamentally a set of frequent subgraphs in a large corpus of programs represented as graphs. At a high level, CodeScholar builds on a new graph abstraction of programs called Simplified Abstract-Syntax Trees (SAST). Given a set of query API methods, the search starts from a set of candidate programs containing these APIs, translated to SASTs. CodeScholar iteratively adds nodes to these SASTs while optimizing ideal properties of code examples, such as reusability, diversity, and expressivity. Also, CodeScholar monitors these properties and stops the search at a unique convergence

[1] https://stackoverflow.com/
[2] https://www.programcreek.com/
[3] https://github.com/features/copilot/

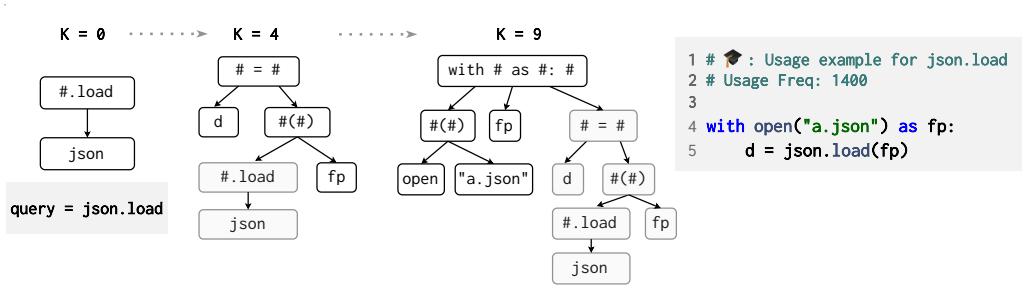


Fig. 1. Overview of CodeScholar: The user's query is transformed into a simplified AST (SAST). Then, an iterative neural-guided search algorithm (K = iteration) adds nodes to the SAST to maximize the "idiomaticity" of the program. An in-order traversal of the resulting SAST retrieves an idiomatic code example.

condition, allowing for a dynamic approach free from heuristics like minimum frequency or maximum program size constraints. CodeScholar then maps SASTs back to programs and returns idiomatic code examples along with their usage frequency (number of times it was found in the corpus). Below, we describe some advantages of CodeScholar through a motivating example comparing CodeScholar to some existing solutions.

1.2.1 ***A Motivating Example.*** Suppose Alice is a novice Python developer who needs a flat list of files from a directory with nested sub-directories. She is aware of a `listdir` function in the `os` library and wants to learn how to use it. Alice searches and finds a top-rated `StackOverflow` post:

StackOverflow

```
import os
path = os.getcwd()
dir_list = os.listdir(path)
```

While correct, she cannot use this on her directory. She then prompts `ChatGPT` [OpenAI 2023], a large-language model, for some usage examples which returns the following 3 examples:

GPT

```
file_list = os.listdir('.')

file_list = os.listdir('/path/to/directory')

file_list = [file for file in os.listdir('.') if file.endswith('.txt')]
```

These snippets are more helpful; the second one also runs on her directory, but she notices that some files are missing from the output. She's also unsure if this code is used in practice. She then turns to CodeScholar to find a more idiomatic code example. CodeScholar returns a variety of examples and their usage frequency, one of which is with frequency 130:

CodeScholar

```
def getListOfFiles(dirName):
    listOfFile = os.listdir(dirName)
    allFiles = list()
    for entry in listOfFile:
        fullPath = os.path.join(dirName, entry)
        if os.path.isdir(fullPath):
            allFiles = allFiles + getListOfFiles(fullPath)
        else:
            allFiles.append(fullPath)
    return allFiles
```

The CodeScholar generated example is more *idiomatic* and solves Alice's problem. It not only shows what `os.listdir` does, but also how it fails by showing an explicit handling of nested directories. CodeScholar generated examples are by construction grounded in real-world code, in stark contrast to language models that hallucinate [Patil et al. 2023]. Consequently, the examples generated are realistic and contain context, improving the understanding of the API. Unlike prior search-based solutions to mining idiomatic code [Allamanis and Sutton 2014; Barnaby et al. 2020], CodeScholar is language-agnostic, does not require offline mining, and supports multi-API queries. Overall, CodeScholar usage examples are designed to be more realistic, expressive, comprehensive, and meaningful–properties of ideal usage examples.

We have implemented CodeScholar in Python and shared it as an open-source tool [4]. Since providing users with good quality API usage examples is the ultimate goal of CodeScholar, we will show in Section 4.1 that developers strongly prefer CodeScholar generated examples over those from state-of-the-art language models like `GPT3.5`. In Section 4.2, we identify an information-theoretic inspired metric to quantify how closely generated code examples represent real-world usage. With that, we will show that for several single and multi-API queries, CodeScholar generates examples that are *realistic*, *diverse*, and *concise*. Lastly, in Section 4.4 we show that CodeScholar not only helps developers, but also AI programming assistants in a retrieval-augmented program synthesis setting.

In summary, this paper makes the following contributions:

- It reformulates idiomatic code examples as frequent subgraphs in a large corpus of real-world programs represented as graphs.
- It presents a novel neural-guided search algorithm for scalable mining of representative and idiomatic code examples for query APIs.
- It proposes a unique reusability-expressivity-diversity convergence condition for early termination of idiomatic code search.
- It identifies a simple information-theoretic distance metric to quantify how close generated code examples are to real-world usage.
- It realizes the above techniques in CodeScholar, a tool that supports single and multi-API queries and returns multiple usage examples with idiomaticity and provenance information.

2 APPROACH

CodeScholar finds idiomatic code examples by dynamically growing graphs representing a query of API methods, as shown in Figure 1. CodeScholar's new graph abstraction of programs called `SAST` is concise, concrete and maps directly to source code (Section 2.1). With this graph abstraction, CodeScholar capitalizes on a key insight that reduces idiomatic code examples to frequent subgraphs (Section 2.2). Rather than mining frequent patterns or generating code given a prompt, CodeScholar uses a search algorithm to grow query graphs into diverse and contextually rich code examples while optimizing several ideal properties (Sections 2.3-2.4).

2.1 SAST: Simplified Abstract-Syntax Tree

CodeScholar works on a graph representation of a program. A program's abstract syntax tree (AST) can be represented as a graph that CodeScholar could use. However, AST can have many abstract nodes, which could result in poor learning. Aroma [Luan et al. 2019] and [Barnaby et al. 2020] use simplified parse trees to address the complexity of an AST. Simplified parse trees, created from parse trees, do not have information about node types. Moreover, simplified parse trees are

[4] we do not provide the URL to respect anonymity

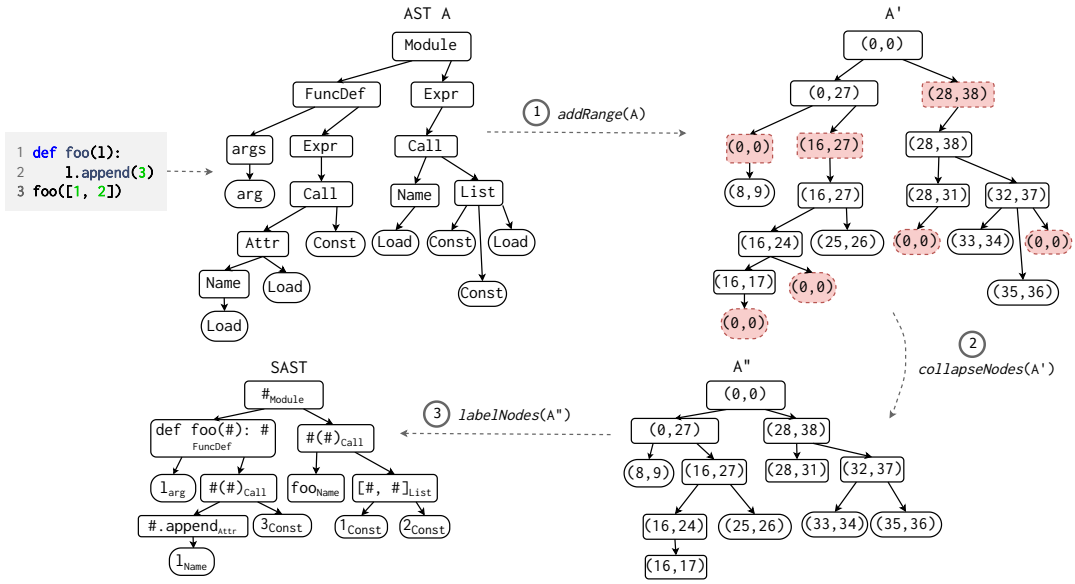


Fig. 2. Transformation of an AST to a SAST. First, the *addRange* transformation adds start and end code locations for each node of the AST. Then *collapseNodes* simplifies the tree by removing redundant nodes (marked red). Lastly, *labelNodes* augments nodes with the substrings mapped to the subtree rooted at that node, resulting in the SAST. The SAST nodes retain the original AST node types along with their new labels.

created using a third-party library, which is not as robust and maintainable as Python's own `ast` module. We propose to use simplified ASTs (called SAST), which associate node type information with each node in a simplified parse tree.

We formally define a Simplified Abstract-Syntax Tree (i.e., SAST) of a program as a data structure to represent programs. It is recursively defined as a non-empty list whose elements could be:

- a token representing keywords (such as `while` and `if`) and symbols (such as `+`, `*`, `:`, `}`),
- a token representing non-keywords (such as variable, field, and method names) or
- a simplified abstract-syntax tree.

Figure 2 shows an example SAST. Each node in the SAST has an associated label representing the subtree rooted at that node. Once CodeScholar creates the SAST for a code snippet, the subsequent steps in CodeScholar are language-agnostic. We now describe how CodeScholar transforms ASTs into a SASTs. In the following, we assume the AST to be a graph $A = (V, E)$ where V represents a set of nodes and E is a set of edges. We use $n \in A$ to denote a node in the set of nodes V of an AST $A = (V, E)$.

First, CodeScholar adds concrete syntax information to the nodes in the AST using the function *addRange*. Specifically, CodeScholar adds the start and end locations of the span of code that maps to each node, which CodeScholar calculates using line number and column offset information available in the parser-generated AST. Formally, $addRange(A)$ transforms AST A of program p by augmenting each node $n \in A$ with a tuple $range(n) = (l, r)$, i.e., the start and end locations of the span of code corresponding to n in p.

Next, CodeScholar collapses the AST by removing redundant nodes using the function *collapseNodes*. Specifically, *collapseNodes* collapses nodes that are too abstract (e.g., `withitem`, `arg`, etc.) to represent any relevant corresponding concrete syntax into their parent nodes. Similarly, nodes

pointing to redundant concrete syntax (relative to the parent node) are also collapsed, resulting in a simplified graph structure. *collapseNodes* is implemented as follows. Let A be the AST of a program p transformed by *addRange*. Then, collapseNodes(A) is a transformation on A that performs the following modification to each node $n \in A$ with a parent node n'. Let C_n and $C_{n'}$ be the set child of nodes of n and n', respectively. If $range(n) == (0, 0)$ (i.e., no corresponding code span), or n is a node with a single child c and $range(c) = range(n)$, then $C_{n'} = C_{n'} \cup C_n$ and remove n from A.

Finally, CodeScholar uses the function *labelNodes* to augment each node in the collapsed AST with a *label* representing a substring of the program mapped to the subtree rooted at that node. The formal definition follows. Formally, for each node $n \in A$ with children C, the *label* for n is obtained by taking the code span δ corresponding to $range(n)$ and replacing substrings corresponding to the spans of all child nodes in C with the special token '#'.

As illustrated in Figure 2, given the three AST transformation functions *addRange*, *collapseNodes*, and *labelNodes*, if p is a program with AST A, the simplified abstract-syntax tree of p is then:

$$\texttt{SAST}(p) = labelNodes(collapseNodes(addRange(A)))$$

Next, we define some properties of a SAST. The size of a SAST S, defined as $size(S)$, is the total number of nodes in S. Number of holes in a SAST, denoted by $holes(S)$ is the total count of incomplete subtrees in SAST S. It is computed as follows:

$$\begin{aligned} h(n) &= |\text{'\#' tokens in the } label \text{ of node } n| \\ c(n) &= |\text{number of outgoing edges for node } n| \\ holes(S) &= \bigotimes_{n \in S} h(n) - c(n) \end{aligned}$$

SAST directly maps nodes to code tokens, facilitating the retrieval of the original program through *in-order traversal*. While the above definitions describe SAST as a transformed AST, it is worth noting that extensions can introduce additional edge types, such as control flow and data flow. As a result, in the following sections, we consider SAST as a more general graph representation of code.

2.2 Problem Statement

Given a set of query API methods, such as {nn.Conv2d, nn.ReLU}, CodeScholar aims to find idiomatic code examples from a large corpus of code, demonstrating the simultaneous usage of all the API methods in the query. As in prior work [Allamanis and Sutton 2014], we view a *code idiom as a syntactic fragment of code frequently occurring across software projects*. For example,

```
self.feat = nn.Sequential(
    nn.Conv2d(27, 16, kernel_size=3), nn.MaxPool2d(2), nn.ReLU(),
    nn.Conv2d(16, 32, kernel_size=3), nn.MaxPool2d(2), nn.ReLU())
```

is a common idiom to build convolutional neural networks demonstrating API methods in the query {nn.Conv2d, nn.ReLU}. We define this task more formally below.

Let $Q = \{a_1, \ldots, a_m\}$ be a set of query API methods, and $P = \{p_1, \ldots, p_n\}$ be the corpus of programs from which we want to mine idioms. Let $G \subseteq G'$ denote that a graph G is a subgraph of a graph G'. A code snippet p is called a *code idiom* if it is a non-trivial code snippet and the SAST of p, which is a graph, is the subgraph of the SASTs of a large subset of programs in P. Formally, p is a code idiom if the set $P_p = \{p_i \mid \texttt{SAST}(p) \subseteq \texttt{SAST}(p_i)$ and $p_i \in P\}$ is "reasonably large" and p is "a non-trivial code snippet". Note that an idiom is better if the size of P_p is larger, which implies that p frequently appears in programs. Similarly, a code snippet is non-trivial if it has a substantial size. For example, the program pass could be a frequent program in P, but its size is so small that it is a trivial program. We discuss how CodeScholar determines the "reasonably large" size of P_p

and the non-triviality of p in a novel search technique in Section 2.4, a contribution of this paper. An *idiomatic code example*, given a query $Q = \{a_1, \ldots, a_m\}$ (a set of API methods), is a program p such that p is a code idiom in P and each query API $a_i \in Q$ appears in p. Each API method in Q could be treated as a standalone program. Therefore, formally, p is an idiomatic example if SAST of each $a_i \in Q$ is a subgraph of the SAST(p). A query Q can have multiple idiomatic code examples. CodeScholar's goal is to discover a set of such examples given a query of API methods.

Relations to Subgraph Matching. The problem of mining idiomatic code examples can be reduced to the well-studied subgraph matching problem [Corneil and Gotlieb 1970; Ullmann 1976]. Given two graphs G and G', we say G is *isomorphic* to the graph G', that is $G \equiv G'$, if there exists a bijection $f : V_G \rightarrow V_{G'}$ such that $(f(v), f(u)) \in E_G$ if an only if $(v, u) \in E_{G'}$. In the above definition, we use V_G and E_G to denote the nodes and edges of G. Note that in the case where nodes and edges are labeled (such as labels in the SAST representation of programs), the bijection f must also match the labels of nodes and edges. The goal of *subgraph matching*, given a query graph G_q and a target graph G_t, is to count the number of subgraphs in G_t that are isomorphic to G_q. Specifically, we want to compute the cardinality of the set $\{H | H \subseteq G_t \text{ and } G_q \equiv H\}$. If the cardinality of the set is 0, then G_q is not a subgraph of G_t.

Several scalability challenges to the subgraph matching (or isomorphism) problem are due to its NP-completeness [Cook 1971]. Traditional approaches use combinatorial search algorithms [Cordella et al. 2004; Gallagher 2006] but do not scale to large problem sizes. Existing scalable solutions for subgraph isomorphism [Sun et al. 2012] involve expensive pre-processing to store locations of many small 2–4 node components and decomposing the queries into these components. Although this allows scaling to large target graphs, the query size cannot scale to more than a few nodes before query decomposition becomes complex. Therefore, we need a scalable technique to answer subgraph queries fast.

Scalable Subgraph Matching. Here, we can use Graph Neural Networks (GNNs) to answer such queries approximately and quickly. Recent advancements in GNNs and neural subgraph matching [Lou et al. 2020] propose techniques to predict whether a query graph is a subgraph of a target graph. NeuroMatch [Lou et al. 2020] decomposes graphs into small overlapping neighborhoods and trains a GNN to learn an ordered embedding that preserves subgraph relations. They have demonstrated effectiveness in practical tasks such as mining frequent motifs in biology, social networks, and knowledge graphs.

One might wonder how approximation would impact the quality of the mined idiomatic examples. Fortunately, we can tolerate minor mismatches in code idioms because two semantically equivalent code snippets may differ in exact syntax due to differences in variable names and control structures. In the next section (2.3), we briefly describe the NeuroMatch approach and focus on how we adapt it for code.

2.3 Neural Subgraph Matching for Code

CodeScholar builds on NeuroMatch [Lou et al. 2020] to perform approximate subgraph matching for code. Consider arbitrary query and target programs whose SASTs are G_q and G_t, respectively. Then, the subgraph matching problem involves predicting if G_q is isomorphic to a subgraph of G_t.

CodeScholar works in three stages: (1) a *featurization stage*, where CodeScholar featurizes the graph representations of code for learning, (2) an *embedding stage*, where CodeScholar computes the embedding of each subgraph centered at each node of G_t, and (3) a *query stage* where the query graph G_q is compared against the target graph in the embedding space for subgraph isomorphism.

2.3.1 ***Featurization Stage.*** SAST nodes are labeled with strings representing the corresponding code spans. Our learning algorithm uses these labels as primary node features. We convert each label

into a real-valued embedding vector by passing it through pre-trained large-language models for code (code-LLM) like `CodeBert` [Feng et al. 2020]. Code-LLMs provide general-purpose embeddings useful for downstream NL-PL tasks such as natural language code search and documentation generation. Using code-LLMs to featurize `SAST` nodes, we capture code semantics in embeddings, supplementing the structural graph features. For instance, nodes labeled `fp` and `file` will map to similar embeddings as they are interchangeably used to refer to a file pointer in code. We further enhance each node with structural features like node degree, clustering coefficient, and page rank, known to improve GNNs [Lou et al. 2020].

In the following sections, we use the notation G_i^j to denote a radial k-hop *neighborhood* anchored at node j in graph G_i.

2.3.2 ***Embedding Stage.*** First, we decompose the target graph G_t into small overlapping radial k-hop *neighborhoods* G_t^u anchored at a node u, for each node u in G_t. The GNN then generates an embedding z_t^u for each G_t^u. Given the target graph node embeddings z_t^u and an anchor node $v \in G_q$, the subgraph prediction function determines if G_t^u is a subgraph isomorphic to G_q^v; i.e., v's k-hop neighborhood in G_q. Notably, the prediction solely relies on the node embeddings z_q^v and z_t^u of nodes v and u, respectively. We enable this during training by enforcing the embedding geometry to encode the subgraph relation between graphs and learn *order embeddings* [McFee and Lanckriet 2009]. Order embeddings ensure that the subgraph relations are reflected in the embeddings: **if G_q^v is a subgraph of G_t^u, then the embedding z_q^v of node v is element-wise less than or equal to u's embedding z_t^u**:

$$z_q^v[i] \leq z_t^u[i]\ \forall_{i=1}^{D} \iff G_q^v \subseteq G_t^u \tag{1}$$

where D is the embedding dimension. We learn *order embeddings* using a max-margin loss function that encodes the above order constraint. Formally, for arbitrary query and target embeddings z_q and z_t, we define the loss as:

$$\mathcal{L}(z_q, z_t) = \tilde{\sum_{(z_q,z_t)\in P}} E(z_q, z_t) + \tilde{\sum_{(z_q,z_t)\in N}} \max\{0, \alpha - E(z_q, z_t)\} \tag{2}$$

$$\text{where } E(z_q, z_t) = \|\max\{0, z_q - z_t\}\|_2^2 \tag{3}$$

Here, P denotes the set of positive examples where z_q's corresponding neighborhood is a subgraph of z_t's, and N denotes the set of negative examples. For positive examples, the loss is proportional to $E(z_q, z_t)$, which denotes the magnitude of the violation of the order constraint. For negative examples, the amount of violation E should be at least α to have zero loss.

Training Data To achieve high generalization, we train the GNN with randomly generated query and target graphs. Let G_t be a graph in our training set. Positive examples P are constructed by randomly sampling a target neighborhood $G_t^u \subseteq G_t$ anchored at node u, and a query $G_t^v \subseteq G_t^u$ anchored at node v using random breadth-first (BFS) walks. When sampling G_t^v, we walk G_t^u starting at u, ensuring the existence of a subgraph isomorphism mapping from v to u. Negative examples N are created by randomly selecting *different* nodes u and v in G_t and performing random traversals. We use a k-layer GNN to embed node u, which captures the k-hop neighborhood G_t^u. The choice of the number of layers, k, depends on the query graph size. For the idiom search task, a k in the range $[7, 10]$ works well.

Overall, the GNN's message passing ensures that embedding node u is equivalent to embedding the neighborhood G_t^u centered at u. Consequently, comparing two node embeddings z_t^u and z_t^v essentially compares the structure of subgraphs G_t^u and G_t^v. This property is crucial for the query stage, where predictions are made using a simple element-wise less-than comparison between two embeddings at negligible cost.

2.3.3 ***Query Stage.*** At the query stage, we determine if G_q is a subgraph of G_t. We construct a prediction function $f(z_q^v, z_t^u)$ that predicts whether the k-hop neighborhood G_q^v anchored at node v is a subgraph of the k-hop neighborhood G_t^u anchored at node u. This prediction implies that node v maps to node u in the subgraph isomorphism mapping. We define $f(z_q^v, z_t^u)$ as a function of $E(z_q^v, z_t^u)$, representing the magnitude of violation of the order constraint and the subgraph relation. A small violation (under a threshold t) indicates that the query is likely a subgraph of the target, as the embeddings align with the order constraint:

$$f(z_q^v, z_t^u) = \begin{cases} 1 & \text{iff } E(z_q^v, z_t^u) \ \ddot{Y} \ t \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

Note, while `NeuroMatch` [Lou et al. 2020] uses a fixed user-defined threshold t, we update this with a learnable threshold. We do this by training a multi-layer perceptron (MLP) classifier on top of our GNN during training that makes a binary prediction. We train this classifier with a traditional negative log-likelihood loss function.

2.4 Neural-guided Idiom Search

We describe a novel search technique that uses neural-subgraph matching to solve the idiomatic example search problem described in Section 2.2. We start by formalizing some extended properties of `SASTs` (Section 2.1) and order embeddings (Section 2.3). First, we observe that while neural-subgraph matching predicts if G_q is a subgraph of G_t, it also enables estimating the frequency of a G_q in a set of target graphs. We define the frequency of a query graph in a set of graphs as follows:

2.4.1 ***Estimating Frequency of Graphs.*** Assume g and f to be the GNN encoder and the classifier of the pre-trained neural-subgraph matching model (Section 2.3), respectively. Let G_q^v denote the query graph anchored at node v and $\mathcal{T}$ be a set of target graphs. Note that the query graph could be disconnected. For instance, a `SAST` with multiple independent API methods. Consequently, the frequency estimate should ensure all connected components of the query are present in the target.

Let $decompose(G)$ be a function that decomposes a graph G into radial k-hop *neighborhoods* G^u anchored at node u, $\forall u \in G$. Then, the frequency of the query graph G_q^v in target graphs $\mathcal{T}$ is the number of neighborhoods that contain *all connected components* of the query graph:

$$freq(G_q^v, \mathcal{T}) = \tilde{O}_{\forall G_t \in \mathcal{T}} \ \tilde{O}_{\forall G_t^u \in decompose(G_t)} \gamma(G_q^v, G_t^u) \tag{5}$$

$$\text{where } \gamma(G_q^v, G_t^u) = \mathbb{I}\ \ \forall G_c \in CC(G_q^v), \quad f(g(G_c), g(G_t^u)) = 1$$

Here, CC returns connected components of a graph, and $\mathbb{I}$ is an indicator function that returns 1 if all elements in the argument set are 1, 0 otherwise. Note that frequency is estimated here at the neighborhood (subgraphs anchored at some node) level, proportional to the frequency at the graph level. To efficiently compute $freq$ for a large set of target graphs $\mathcal{T}$, we implement it using the following optimizations:

(1) *Offline Pre-processed Embeddings*: First, we decompose all graphs $G_t \in \mathcal{T}$ (our search space) into radial k-hop neighborhoods G_t^u anchored at each node $u \in G_t$. Additionally, we can randomly sample nodes u from G_t to compute the neighborhoods. Then, each neighborhood is embedded using the GNN encoder g and stored offline.
(2) *Online Batched Inference*: Second, during inference, the embedding for the query graph $g(G_q^v)$ is computed and compared against all pre-processed target embeddings using traditional batched processing on a GPU—making it efficient and scalable.

Next, we observe that SASTs for programs can be grown idiomatically by adding nodes to the graph that optimize a user-defined metric for the "idiomaticity" of a program.

2.4.2 ***Growing SASTs Idiomatically.*** Suppose we want to search for idiomatic code examples for an API method a in a corpus of programs P. The search can be reformulated as generating a program starting from a by "idiomatically" adding tokens to the program. A token can be idiomatically added to a program if the token makes the resulting program more frequent in P than by adding any other token. If a program is represented as a sequence of tokens, then such additions would be restricted to the end of the token sequence. However, if we use the SAST representation of a program, we can equivalently add a node to the SAST "idiomatically". The advantage of doing so is that a node can be added at any location in the program. We describe how CodeScholar grows a SAST by idiomatically adding nodes to a program:

Let p be an arbitrary program with $\texttt{SAST}(p) = S$. Let S_k be a random subgraph of S with $size(S_k) = k$ and $frontier(S_k)$ be the set of nodes $u \in S \setminus S_k$ with an edge to nodes $v \in S_k$. Let $\mu(S) = {freq(S,\mathcal{T})}/{holes(S)}$ be a metric of "*idiomaticity*" that CodeScholar maximizes while growing SASTs. We use this metric as maximizing μ involves maximizing the frequency while minimizing the holes (incomplete subtrees) in the resulting SAST. As a result, CodeScholar optimizes for the *reusability* and *completeness* of programs. Then, CodeScholar grows S_k idiomatically by adding a node from its *frontier* that maximizes μ:

$$S_{k+1} \leftarrow S_k \cup u^* \mid u^* = \underset{u \in frontier(S_k)}{\arg\max} \ \mu(S_{k+1}) \tag{6}$$

Note that the node added is in the frontier of S_k, which means it has an edge to some node in S_k. Therefore, the operation of including a node adds the corresponding edge as well.

2.4.3 ***Search Algorithm.*** CodeScholar utilizes the insights above in a search algorithm for idiomatic code examples. The search starts with some skeleton graphs, called *seed* graphs, where each seed graph is from the program corpus and contains all query API graphs. We cannot use the set of query graphs as a seed graph because such a graph will be disconnected with no frontier and cannot be directly expanded into code examples. The seed graphs are their equivalents, enabling CodeScholar to grow and map them to idiomatic code examples. We next describe the algorithm in detail.

Let $Q = \{a_1, \ldots, a_m\}$ be a set of query API methods and $P = \{p_1, \ldots, p_n\}$ be a corpus of programs from which we want to mine code examples. Let $\mathcal{Q}$ and $\mathcal{T}$ be the set of SASTs corresponding to Q and P, respectively. CodeScholar begins the search by finding seed graphs, denoted as $\mathcal{S}$, drawn from the set $\mathcal{T}$. Each seed graph is a candidate graph that contains all query API graphs in $\mathcal{Q}$. Since API methods (and their SASTs) are small, traditional search methods such as regex, text-based search, or combinatorial methods for subgraph matching suffice. In our implementation, CodeScholar uses a efficient 2-phase approach to construct $\mathcal{S}$:

(1) ElasticSearch[5] is an open-source full-text search engine based on Apache Lucene [Białecki et al. 2012]. CodeScholar uses ElasticSearch to build an offline index of the large program corpus P. CodeScholar then queries the index for a set of candidate programs $P_e \subseteq P$ that contain all API methods in Q.
(2) CodeScholar then finds the graphs $\mathcal{T}_e \subseteq \mathcal{T}$ that correspond to the candidate programs P_e. For each candidate graph $G_t \in \mathcal{T}_e$, CodeScholar uses combinatorial subgraph matching [Cordella et al. 2004] to locate a subgraph in G_t that contains all query API graphs G_q in $\mathcal{Q}$. These subgraphs are the set of seed graphs $\mathcal{S}$.

[5] https://github.com/elastic/elasticsearch

CodeScholar's search then grows the seed graphs $G_s \in \mathcal{S}$ towards idiomatic code examples. As described, CodeScholar adds a node to the seed graph from its *frontier* that maximizes an "*idiomaticity*" metric $\mu(G_s) = {}^{freq(G_s,\mathcal{T})}/_{holes(G_s)}$. Iteratively performing this growing procedure an arbitrary k number of times for each seed graph in $\mathcal{S}$ generates graphs (SASTs) of size $size(S) + k$. CodeScholar maps these graphs to idiomatic code examples using an in-order traversal.

2.4.4 ***Search Optimizations*** While simple and complete, this search algorithm can be inefficient if implemented naïvely. To make it efficient and scalable, CodeScholar employs a *beam search* strategy to prune the search space and avoid growing all seed graphs. Then, CodeScholar identifies a unique *convergence condition* that decides when to stop growing seed graphs.

Beam Search Beam search, a heuristic algorithm [Bisiani 1987], selectively expands the most promising nodes in graph exploration. It is a version of best-first search, organizing partial solutions based on a heuristic. CodeScholar employs beam search to limit the search space by retaining the top b seed graphs with the highest μ score after each iteration, where b represents the beam width.

Convergence Condition While beam search helps prune the search space, we must decide when to stop growing graphs. To identify a stopping criteria, we observe some properties of an ideal set of idiomatic code examples:

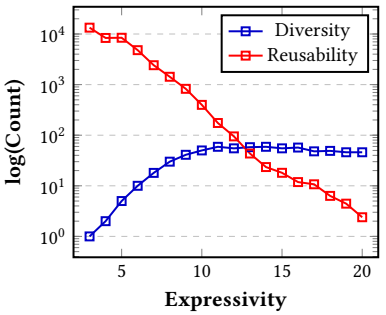


Fig. 3. CodeScholar tracking *Diversity* and *Reusability* of generated usage examples and reaching the convergence condition for `np.mean` at *Expressivity* (size) 12.

(1) Reusability: Idiomaticity is inherently related to the notion of *reusability*. An ideal set of usage examples should be *reusable* as building blocks in many scenarios. Our approach provides a natural way to estimate reusability as the average frequency of the SASTs returned after each iteration of the search algorithm.

(2) Expressivity: Secondly, we want usage examples that are *expressive* enough to show a complete usage of the APIs of interest. The *size* of the SASTs returned after each iteration of the search algorithm is a good proxy for this.

(3) Diversity: Lastly, an ideal set of examples should be diverse. Usage examples exhibit clusters, such as standalone usages like `np.mean(x, axis=0)`, and co-usage with other APIs like `round(np.mean(t), 5)`. We measure *diversity* by the number of distinct clusters of usage examples returned. Our implementation uses Weisfeiler Lehman (WL) graph hash [Shervashidze et al. 2011] to find clusters in SASTs returned after each iteration. The hash function iteratively aggregates and hashes the neighborhoods of each node. WL hashes are identical for isomorphic graphs and guarantee that non-isomorphic graphs will get different hashes.

CodeScholar measures these properties during the search and identifies a convergence condition. The search converges at an ideal *expressivity* (size) when the *reusability* (frequency) and *diversity* (number of distinct clusters) of the usage examples returned after each iteration reach an *equilibrium*. Figure 3 illustrates this for `np.mean`. Note that our goal is to maximize expressivity while also keeping reusability and diversity high. The diversity of usage examples increases with the size of the returned SASTs because larger SASTs have more nodes to explore, leading to greater diversity. Conversely, reusability is inversely proportional to the size of the returned SASTs, as larger SASTs are less frequent in the corpus. Once the reusability and diversity converge, continuing the search

any further will result in more distinct clusters (diversity) than their frequencies (reusability). Hence, CodeScholar stops the search at this convergence condition.

3 IMPLEMENTATION

We implement our approach in a tool called CodeScholar in `Python` and release it as open-source.[6]

Data To build CodeScholar, we first mine a large corpus of `Python` code snippets from public GitHub repositories where CodeScholar can search for idiomatic usage examples. In our mining process, we exclude forked repositories, repositories with less than ten stars, and repositories with the last commit date before 2020. That leaves us with ≈85K repositories that amount to ≈4M python files. We then filter on a set of top-6 most popular libraries used in the mined repositories: `pandas`, `numpy`, `os`, `sklearn`, `matplotlib`, and `torch`. Next, we filter out files larger than 1MB in size or have $\dot{Y}$ 10 tokens. This filtering leaves us with ≈1.5M python files. Finally, we extract all the methods/functions from the filtered files and filter out methods/functions that contain the top-100 most popular APIs from each of the top-6 libraries. We find these APIs by searching for unigrams matching API calls and sorting them by frequency. This procedure results in a large corpus of ≈1.4M methods/functions that CodeScholar searches.

Con gs While our training procedure and model architecture is described in Section 2.3.2, here, we point out some additional details. We train our graph neural network using the Adam optimizer with a learning rate of 1×10^{-4}. We train our model for 5 iterations on a single NVIDIA GeForce RTX 3080 Ti GPU. Each iteration involves training on the entire dataset in a batched fashion. We use a batch size of 64. We use a beam width of 10 for our search algorithm and employ the proposed convergence condition to stop the search automatically.

4 EVALUATION

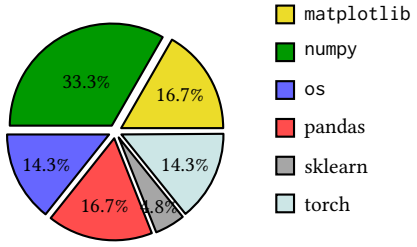
This section aims to assess how good CodeScholar generated code examples are. To do so, in Section 4.1, we evaluate if developers prefer CodeScholar examples over results from state-of-the-art language models like `GPT3.5`. In Section 4.2, we then analyse CodeScholar results and quantify how closely the generated examples represent real-world usage. Then, in Section 4.3, we perform some ablations showcasing CodeScholar's contributions to neural-subgraph matching for code. Lastly, in Section 4.4, we show that CodeScholar not only helps developers, but also AI programming assistants in a retrieval-augmented program synthesis setting.

4.1 Survey with Developers

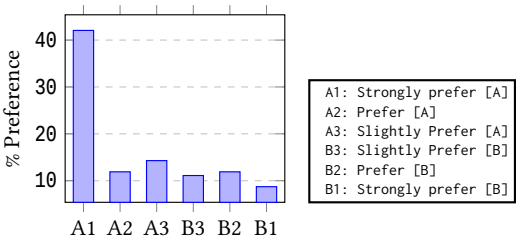
We conducted a survey to measure the quality of code examples generated by CodeScholar, compared to state-of-the-art code generation language models such as `GPT3.5` [Brown et al. 2020; OpenAI 2023]. We follow a similar survey structure as in prior work such as [Barnaby et al. 2020]. We invited 30 developers working in research labs and software companies to be interviewed for the survey. With a response rate of 70%, 21 developers completed the survey. We conducted ≈10-minute semi-structured interviews with these participants.

The survey displayed six popular Python libraries and asked participants to select two they were most familiar with. Figure 4a shows the distribution of the libraries chosen by participants. The survey then showed three API methods in the two selected libraries. For each, two code examples were listed: a randomly selected example generated by CodeScholar (Option A) and a randomly selected example generated by `GPT3.5` (Option B). We use random sampling to avoid any selection biases. Participants were asked three questions about these examples, which we describe next.

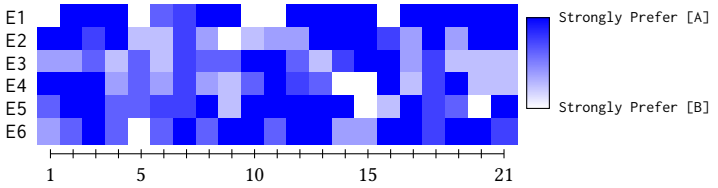
[6] we do not provide the URL to respect anonymity.



(a) Dist. of participant selected libraries



(b) Percentage for each degree of preference



(c) Heatmap of preferences for 6 examples (Y-Axis) and 21 participants (X-axis)

Fig. 4. Results of the survey on preferences between CodeScholar and GPT3.5. (a) shows the distribution of libraries selected by the survey participants. (b) shows the breakdown of the degree of preferences chosen by participants (higher is better). (c) shows a heatmap of the individual responses of 21 participants for each of the 6 examples they were asked to evaluate. In (c), darker cells are better for CodeScholar. In (b) and (c) `[A]` and `[B]` represent CodeScholar and GPT3.5, respectively.

Q1. Suppose you were learning to use this library. Which example would you prefer to understand the API's complete usage? Participants were asked to respond with either Option A or B and a degree of preference: Strongly prefer, Prefer, and Slightly Prefer. Figure 4b shows the distribution of participant preferences. Overall, in 68.3% of the cases, CodeScholar was preferred (either of the 3 degrees of preference) over GPT3.5. Notably, a significant 42.1% of the time CodeScholar examples were strongly preferred over GPT3.5. On the other hand, only 8.73% of the time GPT3.5 examples were strongly preferred. Figure 4c shows a heatmap of the individual preferences of all 21 participants over the 6 examples (3 for each library chosen) they viewed. Notably, the darker the cells, the stronger the indication that participants preferred CodeScholar's examples. It is thus evident from the heatmap that participants consistently favored CodeScholar over GPT3.5. In fact, 15 participants preferred more CodeScholar examples than GTP3.5's, and 6 had an equal preference for both; i.e., none favored a majority of GPT3.5 examples. Participants described while choosing that CodeScholar examples were clear, grounded, and covered more meaningful behaviors of the API.

> P_{15}: *"[A] example for* `os.mkdir` *is so cool! It explains not just what the API does but also what it does not (fails when path exists)"*
>
> P_{17}: *"Both [A] and [B] have information beyond the API. But [A] seems to have the right context, whereas [B] is vague or irrelevant and could be confusing"*

Q2. Which category of examples is more representative of real-world usage? Notably, 100% of the participants said that CodeScholar examples (Option A) were more realistic and representative of real-world scenarios. Some independently remarked that the *realism* of CodeScholar generated examples made them much more useful for understanding the API:

> P_5: *"[A] feels more realistic, which allows me to abstract away the key concepts of the API, whereas [B] has hardcoded inputs that are just difficult to process."*
>
> P_8: *"[A] having real and familiar variables and functions around the API's usage makes it more meaningful."*

Q3. What do you like or dislike about the examples shown? Most participants reiterated that they like the realism of CodeScholar generated examples. Providing a familiar or realistic usage context over simple API inputs helps them understand the API better. Some even remarked that having random inputs in examples is noisy and adds additional cognitive load:

> P_4: *"I actually don't need the dummy dataframe that [B] has; it is a distraction because my data will be different anyway."*

Some disliked that unlike CodeScholar, `GPT3.5` examples, while simple to read and execute, focused a lot on input and output values without any context:

> P_{21}: *"Example [B] for* `np.dot` *having two dummy* `numpy` *arrays as inputs tells me very little about the API. A lot of* `numpy` *methods fit the same inputs."*
>
> P_{18}: *"For* `np.dot`*, [B] is definitely more runnable. But, the* `np.dot(W, x) + b` *in [A] is such a familiar expression that I understand what the API does instantly."*

Participants also pointed out that CodeScholar examples frequently showed integration or inter-usage with other APIs, which added to their usefulness:

> P_7: *"I like that I can search for* `plt.ylim`*, but get examples where it is used with* `plt.xlim`*—much like how I would plot data in the real world!"*

Lastly, a few participants found that `GPT3.5` generated examples, while too simple to explain the API's behavior statically, could be executed more easily for further inspection.

4.2 Analysis of Generated Examples

In Section 4.1, our user study highlights various favorable qualities of code examples generated by CodeScholar. Many participants were particularly interested in the *realism* of the examples, reflecting usage patterns in real-world scenarios.

Here, we propose a novel and meaningful metric to quantitatively evaluate the realism or representativeness of tool-generated code examples. We capture the notion of realism as the distance between the distribution of *real-world* usage examples and the distribution of *tool-generated* usage examples. We can measure such a distance using the *Earth Mover's Distance* (EMD) [Rubner et al. 2000] metric. Intuitively, EMD measures the minimum cost of transforming one distribution into another, and a lower EMD implies the distributions are closer. Intuitively, this measures *"How close are the tool-generated examples to real-world scenarios?"*

Setup We first define the two distributions to measure the EMD between the distribution of *tool-generated* and *real-world* usage examples. For the *real-world* distribution, we pick *all* code snippets in our large mined corpus (Section 3) that contain the query APIs. We then convert these snippets into vectors using OpenAI's GPT-3.5 [Brown et al. 2020] Embedding API [7] and aggregate them as a uniform distribution. Similarly, we embed *tool-generated* examples into a distribution of vectors. Finally, we compute the EMD between the two distributions.

We evaluate EMD in multiple usage settings of CodeScholar. First, we evaluate single API queries. This models a setting where developers initially explore and learn individual APIs before using them in conjunction with others. It also asses how CodeScholar generated examples can augment existing API documentation. Here, we evaluate on a set of 60 API methods from 6 popular

[7] `text-embedding-ada-002`

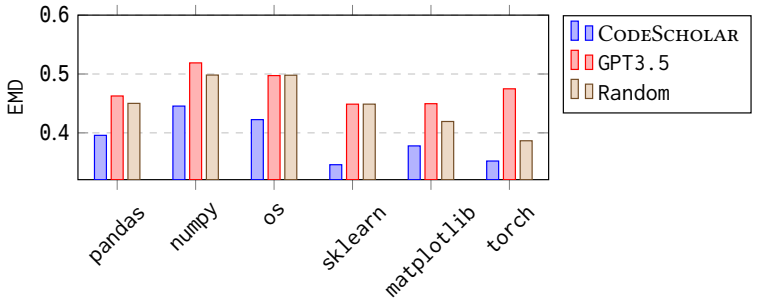


Fig. 5. Average Earth Mover's Distance (EMD) between *tool generated* API usage examples and *real-world* usage examples for single API queries for various libraries. Lower EMD is better, implying the generated examples are closer to real-world usage. Y-axis: EMD, X-axis: Python Library

python libraries: `pandas`, `numpy`, `os`, `sklearn`, `matplotlib`, and `torch`. We select the top-10 API methods for each library by inspecting unigram frequency in the mined corpus.

Second, we evaluate multi-API queries, a common scenario where developers use multiple APIs in conjunction with code for orchestration or integration. We evaluate on a set of 25 multi-API queries, categorized as follows. `single-library-pairs` are 15 queries with API pairs from the same library. `mixed-library-pairs` are 5 queries with API pairs from different libraries. Lastly, `triplets` are 5 queries that contain three APIs. We identify these queries by inspecting frequencies of API bigrams (pairs) and trigrams (triplets) for `pandas`, `numpy`, `os`, `matplotlib`, and `torch` in our corpus. We exclude `sklearn` because its APIs were mostly used in isolation. We consider the top-3 bigrams per library for `single-library-pairs`, and an overall top-5 for `mixed-library-pairs` and `triplets`.

Baselines We compare CodeScholar with two baselines: `Rand` and `GPT3.5`. In `Rand`, we randomly select examples from the *real-world* distribution. By comparing against `Rand`, we aim to show that EMD captures the non-triviality of generating idiomatic and realistic code examples, particularly that CodeScholar goes beyond simple random sampling of code snippets from the real world. Second, we compare with `GPT3.5` (`gpt-3.5-turbo`), a state-of-the-art large-language model (LLM) for code generation tasks. LLMs like `GPT3.5` excel in producing idiomatic (commonly used) code. Note, we compare against `GPT3.5` over `GPT4` because it is cheaper and powers the popular user-facing tool ChatGPT. We carefully engineered prompts for `GPT3.5` to return real-world idiomatic examples given the APIs of interest. This prompt engineering process, which CodeScholar doesn't require, is essential due to LLMs' sensitivity to formatting and instructions. For `Rand` we choose as many examples as CodeScholar generates for a query, and the `GPT3.5` prompt imposes no constraints on the number of examples to generate, ensuring a fair comparison. Note that we don't compare with E.G. [Barnaby et al. 2020] because it's closed-source, even though it supports single API queries.

4.2.1 ***Single API Queries*** Figure 5 shows the average EMD for APIs from the 6 libraries. Note again that a distribution with a lower EMD implies that the examples generated are more representative of the actual usage than one with a higher EMD. Overall, we observe that CodeScholar has a low average EMD of 0.39 compared to a 0.47 for `GPT3.5` and 0.45 for `Rand`. CodeScholar examples have a lower EMD than both `GPT3.5` and `Rand` across all libraries, showcasing that they are more realistic. Notably, a higher EMD for `Rand` than CodeScholar indicates that generating diverse and representative usage examples is more non-trivial than randomly picking some examples from *real-world*. Below, we discuss a few usage examples generated by CodeScholar vs `GPT3.5`.

Example 1. Suppose we have a trained deep-learning classifier that returns a tensor of predicted probabilities for each class during inference. Here, we would use the `torch.argmax` API to get the class with the highest probability. `GPT3.5` returns the following example for `torch.argmax`:

GPT

```
tensor1 = torch.tensor([1, 5, 3, 9, 2])
max_index1 = torch.argmax(tensor1)
```

On the other hand, here are 2 CodeScholar examples that are observably more *representative* of the usage scenarios for `torch.argmax`; in fact, they are directly applicable to our task.

CodeScholar

```
torch.argmax(logits, dim=(logits.dim() - 1), keepdim=True)
preds = torch.argmax(y_hat.detach(), dim=(- 1))
```

Example 2. Suppose we have a directory containing many text files and want to process each. The processing function `foo` takes a file path as input. Here, we would use the `os.listdir` API to get the list of files in the directory and run `foo` on it. `GPT3.5` returns the following 2 examples:

GPT

```
l = [file for file in os.listdir('.') if os.path.isdir(file)]
file_list = os.listdir('/path/to/directory')
```

While correct, it does not showcase the API's *complete behavior*. However, one of CodeScholar's examples captures that `os.listdir` returns a relative path—more useful for our task.

CodeScholar

```
for f in os.listdir('./training_dataset/orange'):
    color_hist_of_image(('./training_dataset/orange/' + f))
```

Example 3. We also observe that compared to `GPT3.5`, CodeScholar generates a very *diverse* set of examples for each API. For instance, here are 3 examples for `df.groupby` in `pandas`:

CodeScholar

```
df.groupby('Period').Choice.value_counts(normalize=True).unstack
df.groupby(['quarter']).apply(do_one_merger_breakup)
df.groupby(['id']).aggregate(take_first_annotation)
```

Similarly, here are 3 examples for `np.matmul` in `numpy` showing various popular use cases such as computing squared-euclidean distance, forward pass of neural nets and pseudo-inverses.

CodeScholar

```
np.matmul((X_mean - Y_mean), (X_mean - Y_mean).T)
np.matmul(self.x, self.w1) + self.b1
np.matmul(np.matmul(np.transpose(X), V_inv), X)
```

4.2.2 ***Multi API Queries***

Figure 6 shows the EMD for the three categories of multi-API queries. Overall, similar to Section 4.2.1, we observe that on average CodeScholar has a lower average EMD 0.45 compared to a 0.52 for `GPT3.5` and 0.48 for `Rand`. For 96% (all but 1) of the queries, CodeScholar has lower EMD compared to `GPT3.5`, implying CodeScholar's usage examples are more representative for multi-API queries too. We discuss some interesting examples below.

Example 4. `(np.mean, np.sqrt)` is a query from `single-library-pairs`, for which `GPT3.5` returns a good usage example, with details about the input:

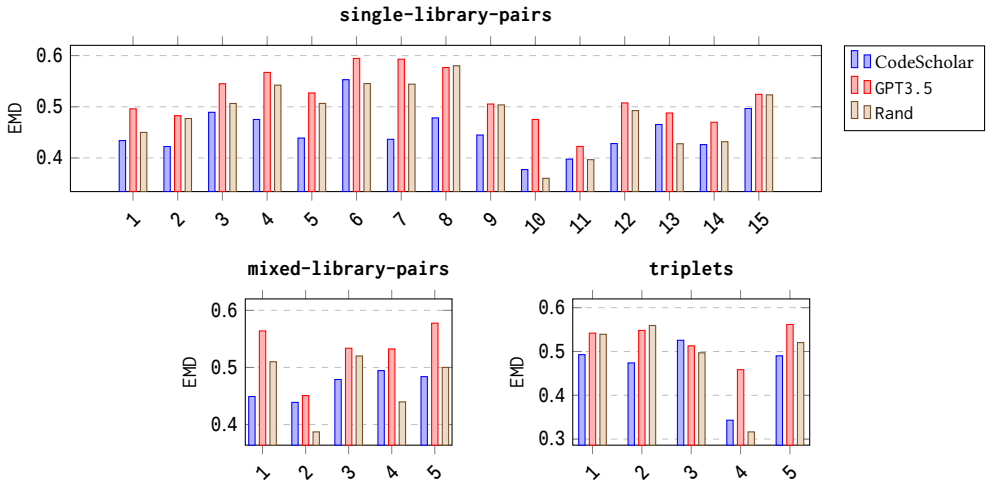


Fig. 6. Earth Mover's Distance (EMD) between *tool generated* API usage examples and *real-world* usage examples for multi-API queries. `single` and `mixed` `library-pairs` are queries with a pair of APIs from single and multiple libraries, respectively. `triplets` are queries with three APIs. Lower EMD is better, implying the generated examples are closer to real-world usage. Y-axis: EMD, X-axis: API ID

GPT

```
array = np.array([[1, 2, 3], [4, 5, 6]])
mean_axis_0 = np.mean(array, axis=0)
sqrt_mean_axis_0 = np.sqrt(mean_axis_0)
```

Whereas CodeScholar returns a more representative, concrete, and familiar real-world example related to calculating root-mean-squared error:

CodeScholar

```
def calc_rmse(true, pred):
    return np.sqrt(np.mean(np.square((true - pred))))
```

Example 5. Another `single-library-pair` is `(torch.load, torch.save)`. `GPT3.5` returns a long program (>20 LOC) that is correct but is too involved.

GPT

```
model = MyModel()
↕ 4 lines
for epoch in range(num_epochs):
      ↕ 7 lines
torch.save(model.state_dict(), 'model.pth')
loaded_model = MyModel()
loaded_model.load_state_dict(torch.load('model.pth'))
↕ 6 lines
```

However, CodeScholar finds a *concise* but expressive example:

CodeScholar

```
try:
    ext_tokenizer = torch.load('mbart-large-50.tokenizer.pt')
except FileNotFoundError as e:
    torch.save(ext_tokenizer, 'mbart-large-50.tokenizer.pt')
```

Example 6. Finally, we explore a challenging `triplet` query: `(os.path.join, os.makedirs, os.path.exists)`. CodeScholar generates examples that demonstrate a prevalent idiom—checking if a directory exists before creating one:

CodeScholar

```
def get_config_path(self):
    config_dir = os.path.join(self.parent_dir, self.config_dir_name)
    if (not os.path.exists(config_dir)):
        os.makedirs(config_dir)
```

GPT3.5 tends to generate examples that only satisfy a subset of the APIs, go beyond the query APIs, or use non-existing ones, which is referred to as *hallucination* [Patil et al. 2023].

GPT

```
project_root = "/path/to/project"
directories = ["src", "tests", "docs"]
os.makedirs(project_root)
for directory in directories:
    os.makedirs(os.path.join(project_root, directory))

file_path = "/path/to/file.txt"
if not os.path.exists(file_path):
    with open(file_path, "w") as file:
        file.write("Hello, world!")
```

4.3 Evaluating Neural Subgraph Matching for Code

Next, we evaluate the performance of neural subgraph matching to predict if a query SAST is a subgraph of a target SAST. While a comprehensive evaluation of general neural subgraph matching is presented in Luo et al. [Lou et al. 2020], we primarily focus on its performance for programs. Particularly, we are interested in ablations identifying how various components of our adaptation affect the performance of the GNNs predictions: (1) the choice of graph representation and (2) the choice of node features.

Setup To evaluate our models, we first sample 10,000 programs from the large corpus of code we mined from GitHub (described in Section 3). Then, we perform the random sampling procedure described in the *Training Data* section of Section 2.3. To summarize, we construct synthetic positive and negative examples for subgraph prediction performing random BFS walks on the SASTs of the sampled programs. We then split the resulting data into 640000 training (80%) and 160000 test (20%) examples. In Table 1, we report the accuracy (% of correct predictions), recall (% positive examples correctly predicted), and precision (% positive predictions that were correct) on the test set.

Results The NM-AST model uses an AST representation and AST node types as One-Hot Encoded node features. This model achieves high accuracy and recall but a poor precision of 0.7. We conjecture this results from the AST being quite large and containing many unnecessary node types that do not carry much information. Next, we evaluate the NM-SAST-Abstract model by replacing the AST with a SAST. However, we continue to use the AST node types as node features (hence the Abstract suffix)

Table 1. Ablations for Neural Subgraph Matching for Code. Acc., Prec., and Rec. are accuracy, precision, and recall, respectively.

Model	Acc.	Prec.	Rec.
NM-AST	0.80	0.70	1.00
NM-SAST-Abstract	0.85	0.77	0.99
NM-SAST	0.90	0.82	1.00
NM-SAST-CodeBERT	**0.96**	**0.93**	**1.00**

to evaluate the effect of simplifying graph structure. We observe that this leads to a significant improvement in precision of 7%. Then we evaluate the `NM-SAST` model, which uses the `SAST` graph and the `SAST` node labels as node features encoded using Byte-Pair Encoding (BPE) [Sennrich et al. 2016]. This leads to a further improvement in precision and accuracy of 5%. While BPE effectively encodes tokens of the source code span, it lacks a semantic understanding of what the tokens represent. Consequently, we evaluate the `NM-SAST-CodeBERT` model, which uses `SAST` graphs and embeds the `SAST` node labels using a pre-trained CodeBERT model [Feng et al. 2020]. This improves our model performance to 96% accuracy and 93% precision.

4.4 CodeScholar for Retrieval-Augmented Generation

We showed that CodeScholar is effective at generating more representative and diverse API usage examples compared to LLMs such as `GPT-3.5`. Although LLMs are more general-purpose and can generate code for arbitrary tasks given a natural language description, even state-of-the-art LLMs find reasoning and code-related tasks more challenging than others [Brown et al. 2020; OpenAI 2023; Touvron et al. 2023]. This is especially true with tasks involving APIs, primarily due to their inability to generate accurate input arguments and their tendency to hallucinate. Recent work has shown that augmenting LLMs with retrievers can improve their performance on such tasks [Patil et al. 2023]. Here, we aim to evaluate CodeScholar's ability to act as a *retriever* for such retrieval-augmented generation (RAG) models.

Setup For evaluation, we use ODEX [Wang et al. 2022]—an open-domain execution-based natural language (NL) to code generation dataset. Note, we do not use other popular benchmarks such as HumanEval [Chen et al. 2021b], APPS [Hendrycks et al. 2021], or MBPP [Austin et al. 2021] because these tasks do not involve the use of APIs. However, ODEX covers various library APIs and methods in `Python`. It contains 439 examples with NL descriptions, corresponding code snippets, and test cases. We sample from this 85 examples that deal with APIs in the top-6 libraries mentioned in Section 3. We follow Chen et al. [Chen et al. 2021b] and measure the performance as the execution accuracy using the *pass@k* metric—the fraction of problems having at least one correct prediction within k samples. Correct here means that the generated code, when executed, passes all test cases.

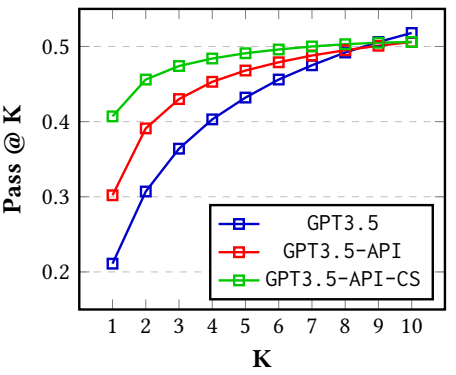


Fig. 7. Pass@K for the ODEX natural language to code (NL2Code) task (higher is better). `GPT3.5` is a vanilla LLM-based program synthesizer. `GPT3.5-API` uses chain-of-thought to find relevant APIs before synthesis. `GPT3.5-API-CS` additionally adds CodeScholar examples for the relevant APIs to the synthesis prompt.

Results Figure 7 shows the results of our evaluation. We start by evaluating `GPT3.5` with no retrieval. Here, we prompt the model with the NL description and 3 few-shot examples of the task and ask it to generate the code. We observe that the model scores quite low (0.2) on *pass*@1, indicating that the model struggles to identify the correct intent. The model, however, expectedly improves to 0.5 (*pass*@10) when given more tries and variability. The next baseline `GPT3.5-API` first asks `GPT3.5` to generate a relevant API method for the given task. Then, the model is asked to generate the code snippet using the generated API method. This prompting style is called *chain-of-thought* [Wei et al. 2022]. Adding the API method to the prompt improves *pass*@1 to 0.3.

Lastly, we evaluate `GPT3.5-API-CodeScholar`, which is similar to `GPT3.5-API` but in addition to the API method, we also add 10 CodeScholar generated API usage examples to the prompt; i.e., retrieval-augmented generation. This improves the performance of `GPT3.5-API` to 0.4 (*pass*@1), a 50% relative improvement overall. Note: we do not intend this to be a comprehensive evaluation of `GPT3.5` or RAG models. We leave that for future work. However, our evaluation shows the potential of tools like CodeScholar to improve the performance of LLMs on code-related tasks.

5 Discussion

Extensibility. CodeScholar searches for idiomatic usage examples by treating programs as `SAST` graphs. However, the general framework proposed in this work is independent of the graph representation used. For example, one could find idiomatic patterns in control-flow graphs or computation graphs of machine-learning models. Also, CodeScholar is independent of the datasets used in this work. One can index and search over any dataset (public codebases or private repositories within organizations) at any granularity using a pre-trained CodeScholar. Further, the `SAST` representation introduced in this work can be constructed for languages other than Python. One could potentially use `SAST` for cross-language code search tasks.

Generalizing with LLMs. From the examples in Section 4.2.1 and 4.2.2, we observe that while CodeScholar examples are more *representative* of real-world usage, `GPT3.5` generates code that is *generic*. Generic code requires minimal edits from the user to be *runnable*. Future work can explore prompting LLMs to polish CodeScholar generated examples or create inputs to execute them.

Search vs Generation. CodeScholar is a search-based approach that grows program graphs to find accurate idiomatic usage examples limited to examples that occur in the real world. However, generative approaches (LLMs) can synthesize examples for newer APIs and their interactions with existing APIs. Also, the generative nature of LLMs allows them to be faster than graph-search-based tools while sacrificing correctness. Future work can explore developing generative approaches for program graphs.

6 Related Work

Code Search. Prior work has shown that developers often search for code to learn new APIs and find code snippets that solve specific tasks [Brandt et al. 2009; Montandon et al. 2013; Sadowski et al. 2015]. Developers use several code search tools and engines to find code snippets. However, most of these tools focus on enriching the query with additional information, such as type information [Mandelin et al. 2005] and test cases [Lemos et al. 2007]. More relevant to our work are code-to-code search engines such as `FaCoY` [Kim et al. 2018], which take a code snippet as a query and retrieve relevant code snippets [Gu et al. 2018; Kim et al. 2018; Wang et al. 2010]. However, unlike our work, these tools are limited to searching code snippets that are similar to the query and are not designed to help developers learn APIs or find API usage examples. A recent study on code search [Di Grazia and Pradel 2023] surveyed 30 years of research, giving a comprehensive overview of challenges and techniques that address them. Interestingly, they find that despite different goals, code search relates to other problems such as code completion [Chen et al. 2021a] and clone detection [Roy and Cordy 2007]. CodeScholar supports this by reformulating search as a growing/completion task while optimizing properties like frequency of matches.

Efforts to improve keyword-based code search [McMillan et al. 2011a; Sachdev et al. 2018] are adjacent to this work. Portfolio [McMillan et al. 2011b] retrieves functions and visualizes their usage chains. CodeHow [Lv et al. 2015] augments the query with API calls which are retrieved from documentation to improve search results. CoCabu [Sirres et al. 2018] augments queries with

structural code entities. While these search techniques focus on improving the query, they do not directly support generating idiomatic usage examples for APIs, especially multi-API queries.

Idiom and Pattern Mining. Tools that mine idioms or idiomatic examples from code repositories are relevant to our work. Allamanis et al. [Allamanis and Sutton 2014] propose HAGGIS—a Bayesian approach for mining idioms from code. GraPacc [Nguyen et al. 2012] achieves pattern-oriented code completion by first mining graph-represented coding patterns using GrouMiner [Nguyen et al. 2009], then searching for input code to produce completions. More recent work [Nguyen et al. 2016a, 2018, 2016b] has improved this by predicting the next API call given a code change. All these approaches are limited to offline mining of idioms and usage patterns ahead of time and are not interfaces for searching idiomatic code. In contrast, CodeScholar provides an interface for idiomatic code search and can be extended for offline idiom mining by growing random graphs instead of seed graphs matching the query. As discussed in Section 1.1 tools like E.G. [Barnaby et al. 2020] are similar to our work. Unlike CodeScholar, E.G. does not support multi-API queries and uses simple stopping criteria, such as constant values for max nodes and min support. CodeScholar, on the other hand, uses dynamic criteria that identify ideal properties of idiomatic code. Additionally, E.G. is a closed-source tool, which restricted its use in our single-API evaluation.

LLMS for APIs. More recently, there has been a growing interest in using language models for API-related tasks [Liang et al. 2023; Patil et al. 2023; Schick et al. 2023; Shen et al. 2023]. For example, Patil et al. [Patil et al. 2023] finetune a LLaMA [Touvron et al. 2023] model on a large corpus of API documentation of ML APIs. These models are limited to specific API domains (e.g., hugging-face APIs), highly dependent on curating good-quality training data, and require significant resources. CodeScholar, on the other hand, is a generic tool that can search any code corpus, uses auto-generated training data, and is significantly more resource-efficient than LLMs. Further, as shown in Section 4.4, CodeScholar can potentially be used to improve the code generated by LLMs.

7 Conclusion

We presented CodeScholar, a tool that generates idiomatic code examples demonstrating the common usage of API methods. CodeScholar is built on the novel insight that treating programs as graphs reduces idiomatic code search to frequent subgraph matching. Based on this insight, CodeScholar employs a neural-guided search algorithm over graphs that grows query APIs into idiomatic code snippets. We showed that developers strongly prefer CodeScholar generated examples over those from state-of-the-art language models like `GPT3.5`. Our quantitative evaluation suggested that, for several single and multi-API queries, CodeScholar generates realistic, diverse, and concise examples. Lastly, we demonstrated that CodeScholar can also improve the correctness of LLM-powered assistants for general program synthesis tasks.

References

Lakshya A Agrawal, Aditya Kanade, Navin Goyal, Shuvendu K Lahiri, and Sriram K Rajamani. 2023. Guiding Language Models of Code with Global Context using Monitors. *arXiv preprint arXiv:2306.10763* (2023).

Miltiadis Allamanis and Charles Sutton. 2014. Mining idioms from source code. In *Proceedings of the 22nd acm sigsoft international symposium on foundations of software engineering*. 472–483.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732* (2021).

Celeste Barnaby, Koushik Sen, Tianyi Zhang, Elena Glassman, and Satish Chandra. 2020. Exempla Gratis (E.G.): Code Examples for Free. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 1353–1364. https://doi.org/10.1145/3368089.3417052

Andrzej Białecki, Rob Muir, and Grant Ingersoll. 2012. Apache Lucene 4. In *OSIR@SIGIR*. https://api.semanticscholar.org/CorpusID:17420900

R Bisiani. 1987. Beam Search: Encyclopedia of Artificial Intelligence, SC Shapiro (ed.): 56-58.

Joel Brandt, Philip J Guo, Joel Lewenstein, Mira Dontcheva, and Scott R Klemmer. 2009. Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 1589–1598.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, et al. 2023. Sparks of artificial general intelligence: Early experiments with gpt-4. *arXiv preprint arXiv:2303.12712* (2023).

Casey Casalnuovo, Kevin Lee, Hulin Wang, Prem Devanbu, and Emily Morgan. 2020. Do programmers prefer predictable expressions in code? *Cognitive science* 44, 12 (2020), e12921.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021a. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021b. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

Stephen A Cook. 1971. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*. 151–158.

Luigi P Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. 2004. A (sub) graph isomorphism algorithm for matching large graphs. *IEEE transactions on pattern analysis and machine intelligence* 26, 10 (2004), 1367–1372.

Derek G Corneil and Calvin C Gotlieb. 1970. An efficient algorithm for graph isomorphism. *Journal of the ACM (JACM)* 17, 1 (1970), 51–64.

Luca Di Grazia and Michael Pradel. 2023. Code search: A survey of techniques for finding code. *Comput. Surveys* 55, 11 (2023), 1–31.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. Association for Computational Linguistics, Online, 1536–1547. https://doi.org/10.18653/v1/2020.findings-emnlp.139

Brian Gallagher. 2006. Matching Structure and Semantics: A Survey on Graph-Based Pattern Matching.. In *AAAI Fall Symposium: Capturing and Using Patterns for Evidence Detection*, Vol. 45.

Elena L Glassman, Tianyi Zhang, Björn Hartmann, and Miryung Kim. 2018. Visualizing API usage examples at scale. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. 1–12.

Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. In *Proceedings of the 40th International Conference on Software Engineering*. 933–944.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, et al. 2021. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938* (2021).

Abram Hindle, Earl T Barr, Mark Gabel, Zhendong Su, and Premkumar Devanbu. 2016. On the naturalness of software. *Commun. ACM* 59, 5 (2016), 122–131.

Nikolaos Katirtzis, Themistoklis Diamantopoulos, and Charles Sutton. 2018. Summarizing Software API Usage Examples Using Clustering Techniques.. In *FASE*. 189–206.

Kisub Kim, Dongsun Kim, Tegawendé F Bissyandé, Eunjong Choi, Li Li, Jacques Klein, and Yves Le Traon. 2018. FaCoY: a code-to-code search engine. In *Proceedings of the 40th International Conference on Software Engineering*. 946–957.

Otávio Augusto Lazzarini Lemos, Sushil Krishna Bajracharya, Joel Ossher, Ricardo Santos Morla, Paulo Cesar Masiero, Pierre Baldi, and Cristina Videira Lopes. 2007. Codegenie: using test-cases to search and reuse source code. In *Proceedings of the 22nd IEEE/ACM international conference on Automated software engineering*. 525–526.

Yaobo Liang, Chenfei Wu, Ting Song, Wenshan Wu, Yan Xia, Yu Liu, Yang Ou, Shuai Lu, Lei Ji, Shaoguang Mao, et al. 2023. Taskmatrix. ai: Completing tasks by connecting foundation models with millions of apis. *arXiv preprint arXiv:2303.16434* (2023).

Zhaoyu Lou, Jiaxuan You, Chengtao Wen, Arquimedes Canedo, Jure Leskovec, et al. 2020. Neural subgraph matching. *arXiv preprint arXiv:2007.03092* (2020).

Sifei Luan, Di Yang, Celeste Barnaby, Koushik Sen, and Satish Chandra. 2019. Aroma: Code recommendation via structural code search. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–28.

Fei Lv, Hongyu Zhang, Jian-guang Lou, Shaowei Wang, Dongmei Zhang, and Jianjun Zhao. 2015. Codehow: Effective code search based on api understanding and extended boolean model (e). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 260–270.

David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. 2005. Jungloid mining: helping to navigate the API jungle. *ACM Sigplan Notices* 40, 6 (2005), 48–61.

Brian McFee and Gert Lanckriet. 2009. Partial order embedding with multiple kernels. In *Proceedings of the 26th Annual International Conference on Machine Learning*. 721–728.

Collin McMillan, Mark Grechanik, Denys Poshyvanyk, Chen Fu, and Qing Xie. 2011a. Exemplar: A source code search engine for finding highly relevant applications. *IEEE Transactions on Software Engineering* 38, 5 (2011), 1069–1087.

Collin McMillan, Mark Grechanik, Denys Poshyvanyk, Qing Xie, and Chen Fu. 2011b. Portfolio: finding relevant functions and their usage. In *Proceedings of the 33rd International Conference on Software Engineering*. 111–120.

João Eduardo Montandon, Hudson Borges, Daniel Felix, and Marco Tulio Valente. 2013. Documenting apis with examples: Lessons learned with the apiminer platform. In *2013 20th working conference on reverse engineering (WCRE)*. IEEE, 401–408.

Anh Tuan Nguyen, Michael Hilton, Mihai Codoban, Hoan Anh Nguyen, Lily Mast, Eli Rademacher, Tien N Nguyen, and Danny Dig. 2016a. API code recommendation using statistical learning from fine-grained changes. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 511–522.

Anh Tuan Nguyen, Tung Thanh Nguyen, Hoan Anh Nguyen, Ahmed Tamrawi, Hung Viet Nguyen, Jafar Al-Kofahi, and Tien N Nguyen. 2012. Graph-based pattern-oriented, context-sensitive source code completion. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 69–79.

Thanh Nguyen, Ngoc Tran, Hung Phan, Trong Nguyen, Linh Truong, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N Nguyen. 2018. Complementing global and local contexts in representing API descriptions to improve API retrieval tasks. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 551–562.

Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H Pham, Jafar M Al-Kofahi, and Tien N Nguyen. 2009. Graph-based mining of multiple object usage patterns. In *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT symposium on the Foundations of Software Engineering*. 383–392.

Tam The Nguyen, Hung Viet Pham, Phong Minh Vu, and Tung Thanh Nguyen. 2016b. Learning API usages from bytecode: A statistical approach. In *Proceedings of the 38th International Conference on Software Engineering*. 416–427.

OpenAI. 2023. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL]

Shishir G Patil, Tianjun Zhang, Xin Wang, and Joseph E Gonzalez. 2023. Gorilla: Large language model connected with massive apis. *arXiv preprint arXiv:2305.15334* (2023).

Chanchal Kumar Roy and James R Cordy. 2007. A survey on software clone detection research. *Queen's School of computing TR* 541, 115 (2007), 64–68.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950* (2023).

Yossi Rubner, Carlo Tomasi, and Leonidas J Guibas. 2000. The earth mover's distance as a metric for image retrieval. *International journal of computer vision* 40 (2000), 99–121.

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

Caitlin Sadowski, Kathryn T Stolee, and Sebastian Elbaum. 2015. How developers search for code: a case study. In *Proceedings of the 2015 10th joint meeting on foundations of software engineering*. 191–201.

Mohamed Aymen Saied, Omar Benomar, Hani Abdeen, and Houari Sahraoui. 2015. Mining multi-level API usage patterns. In *2015 IEEE 22nd international conference on software analysis, evolution, and reengineering (SANER)*. IEEE, 23–32.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools. *arXiv preprint arXiv:2302.04761* (2023).

Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Neural Machine Translation of Rare Words with Subword Units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Berlin, Germany, 1715–1725. https://doi.org/10.18653/v1/P16-1162

Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. 2023. Hugginggpt: Solving ai tasks with chatgpt and its friends in huggingface. *arXiv preprint arXiv:2303.17580* (2023).

Nino Shervashidze, Pascal Schweitzer, Erik Jan Van Leeuwen, Kurt Mehlhorn, and Karsten M Borgwardt. 2011. Weisfeiler-lehman graph kernels. *Journal of Machine Learning Research* 12, 9 (2011).

Raphael Sirres, Tegawendé F Bissyandé, Dongsun Kim, David Lo, Jacques Klein, Kisub Kim, and Yves Le Traon. 2018. Augmenting and structuring user queries to support efficient free-form code search. In *Proceedings of the 40th international*

conference on software engineering. 945–945.

Jamie Starke, Chris Luce, and Jonathan Sillito. 2009. Working with search results. In *2009 ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation*. IEEE, 53–56.

Yiming Su, Chengcheng Wan, Utsav Sethi, Shan Lu, Madan Musuvathi, and Suman Nath. 2023. HotGPT: How to Make Software Documentation More Useful with a Large Language Model?. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*. 87–93.

Zhao Sun, Hongzhi Wang, Haixun Wang, Bin Shao, and Jianzhong Li. 2012. Efficient subgraph matching on billion node graphs. *arXiv preprint arXiv:1205.6691* (2012).

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. *arXiv preprint arXiv:2307.09288* (2023).

Julian R Ullmann. 1976. An algorithm for subgraph isomorphism. *Journal of the ACM (JACM)* 23, 1 (1976), 31–42.

Vasudev Vikram, Caroline Lemieux, and Rohan Padhye. 2023. Can Large Language Models Write Good Property-Based Tests? *arXiv preprint arXiv:2307.04346* (2023).

Xiaoyin Wang, David Lo, Jiefeng Cheng, Lu Zhang, Hong Mei, and Jeffrey Xu Yu. 2010. Matching dependence-related queries in the system dependence graph. In *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering*. 457–466.

Zhiruo Wang, Shuyan Zhou, Daniel Fried, and Graham Neubig. 2022. Execution-based evaluation for open-domain code generation. *arXiv preprint arXiv:2212.10481* (2022).

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems* 35 (2022), 24824–24837.