Missouri State University

BearWorks

MSU Graduate Theses

Spring 2020

A Software Source Code Recommendation System for Code Reuse From Private Repositories

Md Mazharul Islam
Missouri State University, MdMazharul068@live.missouristate.edu

As with any intellectual project, the content and views expressed in this thesis may be considered objectionable by some readers. However, this student-scholar's work has been judged to have academic value by the student's thesis committee members trained in the discipline. The content and views expressed in this thesis are those of the student-scholar and are not endorsed by Missouri State University, its Graduate College, or its employees.

Follow this and additional works at: https://bearworks.missouristate.edu/theses

 Part of the Software Engineering Commons

Recommended Citation

Islam, Md Mazharul, "A Software Source Code Recommendation System for Code Reuse From Private Repositories" (2020). *MSU Graduate Theses*. 3493.
https://bearworks.missouristate.edu/theses/3493

This article or document was made available through BearWorks, the institutional repository of Missouri State University. The work contained in it may be protected by copyright and require permission of the copyright holder for reuse or redistribution.
For more information, please contact BearWorks@library.missouristate.edu.

A SOFTWARE SOURCE CODE RECOMMENDATION SYSTEM FOR CODE REUSE FROM PRIVATE REPOSITORIES

A Master's Thesis

Presented to

The Graduate College of

Missouri State University

In Partial Fulfillment

Of the requirements for the Degree

Master of Science, Computer Science

By

Md Mazharul Islam

May 2020

A SOFTWARE SOURCE CODE RECOMMENDATION SYSTEM FOR CODE REUSE FROM PRIVATE REPOSITORIES

Computer Science

Missouri State University, May 2020

Master of Science

Md Mazharul Islam

ABSTRACT

Motivated by the idea of reusing existing source code from previous projects within a software company, in this thesis, I present a new source code recommendation technique to help programmers find relevant implementations or sample code based on software requirement specifications. My proposed technique assists programmers to search existing code repositories using natural language query. My approach summarizes the uploaded code into sentences or phrases to match them against user queries. This version of my proposed technique extracts and analyzes the content of Python code (such as variables, functions, docstrings, and comments) to generate code summary for each function which is then mapped to the respective functions. My approach also provides the user query suggestions so that after issuing a generic query the user can reformulate their query based on the suggestions. A web-based deployment of the tool (available at http://socer.razib.info/search) allows a user to enter a textual search query and returns the relevant code search results that were most relevant to the query. It also allows users to upload new code to enrich the codebase with tested code. If adopted, my proposed technique will benefit a software company to build a trusted codebase enabling large-scale software code reuse.

Keywords: Code recommendation, code reuse, code search, software code, query reformulation.

A SOFTWARE SOURCE CODE RECOMMENDATION SYSTEM FOR CODE REUSE FROM PRIVATE REPOSITORIES

By
Md Mazharul Islam

A Master's Thesis
Submitted to the Graduate College
Of Missouri State University
In Partial Fulfillment of the Requirements
For the Degree of Master of Science, Computer Science

May 2020

Approved:

Razib Iqbal, Ph.D., Thesis Committee Chair

Jamil M. Saquer, Ph.D., Committee Member

Siming Liu, Ph.D., Committee Member

Julie Masterson, Ph.D., Dean of the Graduate College

In the interest of academic freedom and the principle of free speech, approval of this thesis indicates the format is acceptable and meets the academic criteria for the discipline as determined by the faculty that constitute the thesis committee. The content and views expressed in this thesis are those of the student-scholar and are not endorsed by Missouri State University, its Graduate College, or its employees.

ACKNOWLEDGEMENTS

First, I would like to thank Dr. Razib Iqbal for his continuous mentoring throughout the research. His guidance helped me a lot to make good progress in my research. I enjoyed and learned a lot working with him. He also helped me a lot outside the academic field.

I would like to thank Dr. Lloyd A. Smith for helping me with all the academic works and Graduate Assistantship duties.

I dedicate this thesis to my parents.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

INTRODUCTION

Large amount of source code is available in every software company's private repository and most of them are not analyzed and reused properly. A software company can reduce development time and cost by reusing source code from previous projects. Whenever a software company starts a new project, they usually build new a module of the software from scratch even though some of the features of that module were already implemented in another project. The main reason behind writing the code from scratch is sometimes it is difficult for a developer to find source code snippets from another project developed by another developer. It is also time-consuming if a developer wants to find the source code of those features manually from files.

To reuse source code in private repositories, I have developed a system to collect, analyze, and search Python source code. To improve the means by which reliable code is reused, my proposed method is motivated by finding similar source code based on specific software requirements or natural language even before the actual development phase starts. My technique is proposed focusing on the private repositories of source code where the user can also enrich the private codebase with tested, well-documented reusable source code. Even though programming by example was found to be intuitive to many developers as per [1] and [2] where the programmers are recommended with example code. There are existing techniques, such as [3][4] that rely on the developers' coding activities to search or recommend the relevant source code. Compared to these techniques, our goal is three-fold:

a) Search relevant code using textual software requirements specifications and/or natural language,

b) Present a lightweight code search tool to reuse source code that have been tested and delivered earlier, and

c) Suggest queries to the user so that they can reformulate their queries based on the suggestions to get better code search results.

Even though the primary focus of this thesis is source code recommendation techniques, during the preliminary evaluation of our proposed work, it was discovered that query suggestion plays a vital role for my proposed method to be more effective since it assists the users to enter a specific or structured query. This scenario happens because the user might not have a clear picture of the entire codebase. As a result, the user might be looking for a source code performing certain functionality entering inappropriate queries.

The rest of the thesis is organized as follows: an overview of the existing approaches for source code recommendation, query reformulation and information retrieval from source code is given in the *Related Works* section. In the *Proposed Code Search Method* section, I present my proposed technique for source code recommendation followed by a reformulated search query suggestion technique and an overview of our developed code search tool and its main components. The effectiveness and accuracy of my proposed method are presented in the *Evaluation* section. Finally, in the *Conclusion* section, I give my concluding remarks and directions for my future work.

RELATED WORKS

Code Search Tools and Plug-ins

There are plug-ins and search tools, such as Strathcona [3] and PARSEWeb [4], for standard integrated development environments that recommend code fragments based on the code context and the structural details of the developer's activities. Both Strathcona and PARSEWeb rely on code repositories and the code quality depends on the overall quality of the repositories they use. Moreover, Strathcona uses structural context to recommend source code which is helpful for the developers who are using framework for object-oriented programming. When developers use a framework, they often become confused and unsure about which subclass to call, which object to instantiate, and which methods to call. Even though there are documents and relevant examples for using a large framework, the developers find it hard to locate and incorporate the appropriate example when they are not familiar with the framework. Their approach uses the structural context to form a query that is extracted automatically from the code a developer is writing. Strathcona locates relevant code in an example repository by heuristically matching the structure of the code which is under development to the example code. Therefore, this approach is helpful for the developers who are using framework for object-oriented programming.

Similar to Strathcona, PARSEWeb, which works with Java object-oriented source code, interacts with Google Code Search engine to gather source code and analyzes partial code samples using Abstract Syntax Trees and Directed Acyclic Graphs that can handle control-flow information and method inlining. However, it might not be intuitive for the developers to search

for the related frameworks or code in PARSEWeb if they are unsure of the class structure and dependencies.

Code search engines, such as Koder [5] and Krugle [6], search in a large set of open source repositories collected from the internet. These search engines work like other search engines, such as Google and Bing, but the only difference is that they show source code as the results. Both Krugle and Koders return results which link to the actual project files, without guiding the developer to the location of the required code snippet in the project file. This requires the developers to read and understand the search results and then look for the possible match for their query. However, these open-source results might lead to vulnerable code in the final product.

Another tool SWIM [7] suggests C# code snippets by translating user queries into existing APIs of interest using the clickthrough data from the Bing search engine. While the SWIM user query does not need to contain specific type names or methods of interest, it heavily depends on the standard libraries and frameworks to find and match the appropriate APIs. Some examples of queries to to API translation: (append strings: StringBuilder.Append, ToString), (generate md5 hash code: MD5.ComputeHash), (download le from url : WebClient.DownloadFile) etc.

Similar to SWIM, other tools like MICA [1] and Exampler [2] also use standard libraries, such as Java Development Kit, to search for API examples for recommending sample code. But they are restricted to providing a limited set of examples based on the API only. For example, if the user enters keywords secure and send, and the corresponding API calls encrypt, and email are connected via some dataflow, then an application with these connected API calls are more relevant to the query than ones where these calls are not connected. To get benefitted from tools

like MICA and Exampler, the developer must have good knowledge of the whole software workflow such as how API has been used to get the desired outcome.

Sourcerer [8] is an infrastructure for downloading, parsing, storing, and analyzing Internet-scale software corpora, as well as a search engine supporting keyword-based and structure-based querying. In Sourcerer, the authors applied topic modelling to mine and search Internet-scale software repositories. The code search feature in Sourcerer is built using indexed keys and ranked entities. Query keywords entered by a user are matched against the set of keywords where each key is mapped to a list of entities, and each entity has a rank associated with it. Thus, the list of entities that are matched against the sets of matching keys are returned as search results to the user.

CodeGenie [9] is a tool that uses test cases as an interface for code search. For example, JUnit test classes must be created to define the desired features and then the testcases are used to find the relevant code in the internet repositories. Therefore, the developers first have to translate the software features based on the requirements specification to code representing the testcases before the CodeGenie could be used.

There is another recommendation tool, Example Overflow [10] which brings together social media and code recommendation system. In Example Overflow, the authors relied on Stack Overflow for code snippets. They save the code snippet along with related metadata such as the question title, the user rating, etc. in order to find code snippets that may not contain the search query keyword, but the keyword appears in the contextual data. But it is still unknown if crowdsourced software development would be able to scale well. For searching, it uses keyword search based on the Apache Lucene [11] library, which internally uses the term frequency-

inverse document frequency (tf-idf) weight [12]. However, in order for Apache Lucene to search properly, we need to define the parameters that need to be analyzed and indexed.

Authors in [13] proposed a system that can generate pseudocode in natural language from source code. It might not be helpful for us because it generates pseudocode from every information present in the source code. For example, if the source has a conditional expression like "if x % 5 == 0:" then it is translated into "if x is divisible by 5". But we are looking for summary which can describe the main functionality of a function.

Authors in [14] proposed a semantic based code search and their main approach is to take a set of candidate solutions, attempt to transform that set into a more appropriate set, check the resultant set against the user's specifications, and then output all solutions that meet these specifications. This system requires the user to augment or change the specifications based on the output that the original descriptions/query produced.

Another paper published in 2018 [15] proposed a system named GITSEARCH, a code search engine, on top of GitHub and Stack Overflow Q&A data. Their approach leverages common developer questions and the associated expert answers to augment user queries with the relevant source code entities in order to improve the performance of matching relevant code examples within large code repositories. In contrast, we are relying on private repositories without any metadata, such as user Q&A or comment available in Github/Stack Overflow.

Authors in paper [16] proposed a technique that automatically identifies relevant and specific API classes from Stack Overflow Q&A site for a programming task written as a natural language query, and then reformulates the query for improved code search. Similar to [15], the authors of this paper also used crowdsourced data from Stack Overflow for code searching.

Authors in [17] proposed a technique which searches clone of source code instead of natural language search query. To apply their technique any user must have strong background on the code structure to find particular code snippets as the user has to give source code as input rather than user query.

Code Summarizer and Information Retrieval from Source Code

For source code recommendation we also need to know what kind of information a source code provides. There are several works on retrieving information from source codes. Authors in [18] proposes a code summarizer which generates documents from source code. This system uses method context which focuses on the methods or functions of source codes to extract information. They use three supporting technologies for their work: The Software Word Usage Model (SWUM) [19], the design of Natural Language Generation (NLG) systems [20] and the algorithm PageRank [21]. The Software Word Usage Model (SWUM) is a technique for representing program statements as sets of nouns, verbs, and prepositional phrases. Consider a method which has the signature static String scanPublicId(StringBuffer, XMLReader, char XMLEntityResolver). SWUM first splits the identifier names using the typical Java convention of camel case. Next, it reads verbs from the method as the starting word from the method identifier (e.g., “scan”). SWUM also extracts noun phrases, such as “public id”, and deduces a relationship of the nouns to the verbs. For example, “public id” is assumed to be the direct object of “scan” because it follows “scan” in the method identifier. Other nouns, such as “string” or “xml reader”, are read from the return types and arguments, and are interpreted under different assumptions. The NLG system translates the output of SWUM into readable natural language sentences. Then with the help of PageRank algorithm sorts the functions based on their

importance. Methods that are called many times or that are called by other important methods are ranked as more important than methods which are called rarely.

Authors in [22] proposes a method which automatically generates summary comments for Java Methods. The main contributions of this paper are: An algorithm is presented to automatically extract important code statements for a method's summary comment. Then, a text generation technique is presented that takes a Javacode statement as input and outputs a natural language phrase which represents the code. Then, a human evaluation of the accuracy, content adequacy, and conciseness of the automatically generated leading descriptive summary comments was stated.

Authors in [23] proposed a model, CODE-NN which uses Long Short-Term Memory (LSTM) networks with attention to produce sentences that describe C# code snippets and SQL queries. Their model was trained on a dataset of code snippets with short description. They collected data from StackOverflow, which contains programming-related posts where each post comprises a short title, a detailed question, and one or more responses, of which one can be marked as accepted. In our approach this type of technique is not applicable as we are relying on source code private repositories which are not labeled or has any feedback from the user.

Query Reformulation

Developers use code search engines to collect code snippets using generic natural language queries. But such queries often do not lead to relevant results. Users might not get the desired code search results based on their natural language queries due to vocabulary mismatch issues, which is also evident in the literature, such as [24, 25]. Vocabulary mismatch is a common phenomenon in natural language, and it occurs when different people name the same

thing differently. For this reason, developers frequently reformulate their query by removing the irrelevant keywords and adding more relevant keywords. This manual reformulation takes lots of trials and errors as well as significant development time and efforts. Automatic Query reformulation can help the developers to find their desired code snippet in less amount of time.

Authors in [26] proposed a Query Reformulation framework that works by enriching a user's search with additional terms. These additional terms are selected from highest-ranked artifacts retrieved in response to initial query.

Authors in [27] proposed a query reformulation technique named QECK (Query Expansion based on Crowd Knowledge) where they expand Query based on Crowd Knowledge. QECK retrieves relevant Question & Answer (Q&A) pairs in a collection extracted from Stack Overflow for a given free-form query. Then identifies the software-specific words from these documents and generates an expansion query by adding words to the original query.

Authors in [28] proposed an approach to reformulate the raw query by leveraging crowd knowledge form millions of developers to improve the retrieval results. They build a software - specific domain lexical database which is used to expand and optimize the input queries.

Motivated by Example Overflow to find code snippets that may not contain the search query keyword, my proposed method uses the summary (a.k.a. descriptor) generated from the source code (such as, variables, comments, function name, function body and docstrings) rather than using crowdsourced metadata ensuring that there is no redundant information in the summary. Unlike Krugle and Koder, my goal is to reuse the local code repository which can be reused for future developments because my proposed method does not use the open-source results which might lack the quality for reusing them in another project.

PROPOSED CODE SEARCH METHOD

In this thesis, a code search technique is proposed where the user can search source code with their natural language query and also gets query suggestions to reformulate their queries. In my proposed approach, the summary from source code is generated first. The summary is generated by retrieving information from the functions of the source code. These summaries are called “function descriptor”. After that when the user issues a search query for source code, the query is matched with the function descriptors. Then based on the similarity, my proposed technique returns source code to the user and also suggests queries so that the user can reformulate query to get the desired results. In the rest of this section, I discuss my theoretical contribution and then give an overview of a tool that was built implementing my proposed code search and recommendation techniques.

Technical Contribution

I developed a function descriptor generator which generates a summary from the source code. The descriptors consist of natural language words, phrases, and sentences. I used Abstract Syntax Tree (AST) for extracting important attributes like function name and variable names from the source code. An AST is a representation of the abstract syntactic structure of source code where each node denotes a construct (i.e. variable names, function identifiers, etc.) occurring in the source code. Abstract syntax tree does not represent every detail of real syntax, but it represents the structural and content-related details. For example, in AST grouping parentheses are not represented in the separate nodes in the tree structure. Likewise, an if-condition-then expression is denoted by a single node with three branches. Instead of using raw

source code, we use AST because it does not include unnecessary punctuation and delimiters (braces, semicolons, parentheses, etc.).

Figure 1 shows the steps associated with the function descriptor generation process. All the functions from the AST are extracted first, and then the corresponding function names, docstring, variable names, and comments are extracted to generate a meaningful summary for each function.

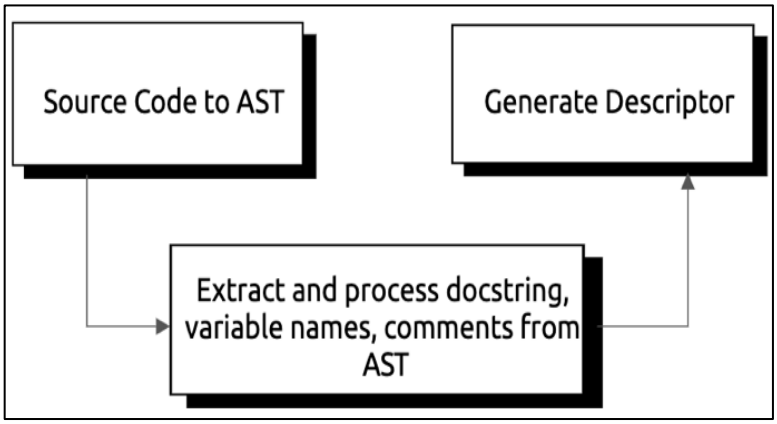

Figure 1. Workflow of function descriptor generator

Most of the time, readable variable and function names contain important information about the workflow and functionality of the function. Therefore, meaningful words are processed and considered which can enrich the function descriptor. For example, some variables are named using the camel casing convention, e.g. saveTemperatureValue, based on which my proposed technique attempt to produce the output “save temperature value”. However, if there is a variable “saveTemperatureAsdf”, then the output “save temperature” will be produced and the part “Asdf” will be ignored as it does not have any meaning. Similarly, the variable names having underscore (“_”) in them are processed, e.g. a variable named get_temperature_value would produce the output “get temperature value”. Finally, the comments from functions were extracted

by traversing the function's source code using simple text processing. In Python, comments begin with a hash mark (#) and for extracting the comments my proposed technique looks for hash mark and then parse the associated comments. After extracting the function names, docstring, variable names, and comments, my proposed method combines all the information and generates a summary, which is the output of the function descriptor generator as shown in Table 1. In the table we have a function named writeOutputHsv() which outputs HSV value. In that function, we have 2 single-line comments. It has some variables like bead, radius etc. which are meaningful. The method name is writeOutputHsv which has three words added by camel casing naming convention. We separate these three to "Write", "output" and "HSV" then add to our function descriptor along with the comments. There is another example where we observe another important aspect of the function descriptor generator. In some functions, several variable and method calls will occur more than once, and the record of that occurrence is kept in the function descriptor. In the function "detect gesture" in Table 1, the method cv2.circle is called twice, and two instances of the word "circle" are stored in the function descriptor which denotes the importance of the word. The algorithm of generating function descriptor is given in Algorithm 1.

For a user search query consisting of natural English language, my system matches it with the stored function descriptors based on similarity. A database is maintained for storing source code along with their descriptor which we also call the codebase.

Table 1. Sample function body with its descriptor

Function	Descriptor

`def writeOutputHsv(writer, colorBeads):`
`  i = 1`
`  for bead in colorBeads:`
`  # here, the colorsys conversion function expects values between 0-1 for rgb`
`  hsv = colorsys.rgb_to_hsv(`
`  bead[0][0]/255, bead[0][1]/255, bead[0][2]/255)`
`  h = hsv[0]`
`  s = hsv[1]`
`  # the returned values are placed in an array in the order h, s, v`
`  v = hsv[2]`
	write Output Hsv
	bead
	radius
	rgb to hsv
	here, the colorsys conversion function expects values between 0-1 for rgb
	the returned values are placed in an array in the order h, s, v

`def detectGesture(self, contours, defects, img):`
`  """Function for detecting gesture from image"""`
`  if defects is None:`
`  return ['0', img]`
`  if len(defects) <= 2:`
`  return ['0', img]`
`  num_fingers = 1`
`  for i in range(defects.shape[0]):`
`  start_idx, end_idx, farthest_idx, _ = defects[i, 0]`
`  start = tuple(contours[start_idx][0])`
`  end = tuple(contours[end_idx][0])`
`  far = tuple(contours[farthest_idx][0])`
`  cv2.line(img, start, end, [0, 255, 0], 2)`
`  if angleRad(np.subtract(start, far),`
`  np.subtract(end, far)) < deg2Rad(self.angle_cuttoff):`
`  num_fingers += 1`
`  cv2.circle(img, far, 5, [0, 255, 0], -1)`
`  else:`
`  cv2.circle(img, far, 5, [0, 0, 255], -1)`
`  return (min(5, num_fingers), img)`
`  cv2.circle(img, far, 5, [0, 0, 255], -1)`
 	
`  return (min(5, num_fingers), img)`
	Function for detecting gesture from image
	detect Gesture
	num fingers
	start
	end
	far
	shape
	start idx
	end idx
	farthest idx
	line
	num fingers
	angle cuttoff
	circle
	circle
	subtract
	subtraxct

Algorithm 1 Algorithm for Function Descriptor Generator

```
Input: Source Code
Output: Function Descriptor
module=ast.parse(source_code)
function_list=[ ]
for each node in module.body do
    if isinstance(node, ast.FunctionDef) then
        function_list.append(node)
    end if
end for
for each function in function_list do
    function_ast=ast.parse(function)
    for each function_node in ast.walk(function_ast) do
        processed_attribute=process_attr(func_node.attr)
        // attribute consists of variable and function names
        attribute_list.add(processed_attribute)
    end for
    comments= get_comments(function)
    docstring = ast.get_docstring(function)
    function_descriptor=generate_descriptor(attribute_list,comments,docstring)
end for

return function_descriptor
```

Before doing any text processing we need to convert the texts into their vectorized forms. To explain this let us consider a sentence “A quick brown fox jumps over a lazy dog”. We can understand the sentence as we know the semantics of the words and sentences. But the computer will not understand this sentence because it can only understand data in numerical form. For this reason, we need to vectorize the text data so that the computer can understand it better.

For calculating the similarity among the function descriptors and user query, we use a text mining technique called term frequency–inverse document frequency (tf-idf). This tf-idf is a statistical measure to evaluate the importance of a word to a document (i.e. set of words) in a

collection or corpus. The importance increases proportionally to the number of times a word appears in the document but is offset by the frequency of the word in the corpus. As mentioned earlier, we need to vectorize the documents and when we are vectorizing the document we cannot just consider the words that are present in that particular document. If we do that then the vector length of two documents will be different. It is not feasible to measure the similarity between two vectors having different lengths. To properly vectorize the documents, my proposed method vectorizes the list of all possible words in the corpus. In this way, we will get the vectors of same size for documents with different length.

Typically, the tf-idf weight is composed of two terms - normalized Term Frequency (TF) and Inverse Document Frequency (IDF) as described below.

● TF: Term Frequency measures how frequently a term (i.e. word) appears in a document. Since every document is different in length, it is possible that a term would appear many more times in long documents than shorter ones. Thus, the term frequency is divided by the document length (i.e. the total number of terms in the document) as a way of normalization. In our case, we calculated the TF using the following formula:

TF(t) = (Number of times term t appears in phrases and sentences of descriptor) / (Total number of terms in the phrases and sentences of descriptor)

● IDF: Inverse Document Frequency measures how important a term is. While computing TF, all terms are considered equally important. However, it is known that certain terms, such as "is", "of", and "that", may appear a lot of times but have little importance. Thus, we need to weigh down the frequent terms while scaling up the rare ones. For IDF calculation, we apply the following formula:

IDF(t) = log_e(Total number of phrases and sentences of descriptor / Number of phrases and sentences of descriptor with term t in it)

Then by taking the multiplicative value of TF and IDF we get the TF-IDF weight for every document.

I selected TF-IDF weight for several reasons. Other existing approaches like bag of words approach has some limitations because they do not account for noise. In other words, certain words are used to formulate sentences but do not add any semantic meaning to the text. For example, the most commonly used word in the English language is "the" which represents 7% of all words written or spoken. So, we have to consider the occurrences of these types of words and weigh down the importance of them. To handle scenarios like this TF-IDF approach is the best choice as discussed earlier.

The Cosine similarity method is applied to measure how similar the documents are irrespective of their sizes. Cosine similarity is a metric used to measure how similar the documents are irrespective of their size. Mathematically, it measures the cosine of the angle between two vectors projected in a multi-dimensional space. The cosine similarity is advantageous because even if the two similar documents are far apart by the Euclidean distance (due to the size of the document), chances are they may still be oriented closer together. Because the similarity depends on the angle, for a smaller angle the similarity will be higher. From each phrase or sentence, we derived a vector. Then the phrases or sentences are viewed as a set of vectors in a vector space. The following formula is used to calculate the similarity between the user query and function descriptor:

$$cosine\ similarity(d1, d2) = \frac{Dot\ Product(d_1, d_2)}{||d_1|| * ||d_2||}$$

$$Dot\ Product(d_1, d_2) = d_1[0] * d_2[0] + d_1[1] * d_2[2] + \quad \ldots \quad + \ d_1[n] * d_2[n]$$

$$||d_1|| = \sqrt{d_1[0]^2 + d_1[1]^2 + \cdots + d_1[n]^2}$$

$$||d_2|| = \sqrt{d_2[0]^2 + d_2[1]^2 + \cdots + d_2[n]^2}$$

Finally, the relevant functions are returned according to their similarity scores in response to the user query as output. The outline of my approach for code search is shown in Algorithm 2. Now, the traditional code search tools, such as Krugle, allow searching for code, where the search results consisting of the source file is returned. In contrast, our method returns the results with the actual code with a similarity score.

Algorithm 2 Algorithm for Code Search

Input: User Query
Output: Source Code

```
for each document in function Summary do
    descriptor_vector =TfidfVectorizer.fit_transform(document.descriptor)
    query_vector =TfidfVectorizer.fit_transform(userQuery)
    similarity=cosine_similarity(document.descriptor, userQuery)
    if similarity > 0 then
        code_with_similarity.append(document.functionBody,similarity)
    end if
end for
code_for_recommendation=sort(code_with_similarity)

return code_for_recommendation
```

My method also handles multiple versions of source code. When a user uploads a new version of a project, the source code is stored if the current version is different from the previous version and also the source code of the previous version is stored. The advantage of keeping all the versions is that if a user performs a search, then my method will recommend the version which will meet user query the most. Because the latest version of a source code snippet might

have additional functionality which may be redundant to the user if the user is looking for some basic functionalities.

For example, there are two solutions to the “two sum problem” shown in Table 2 where version 1 solves the problem in nonlinear time and version 2 solves the problem in linear time using a Hash table. For the query “two sum solution in nonlinear time”, my technique will recommend the code of version 1 before version 2 which has no additional information about using a Hash table to solve the problem. However, if the query was “two sum solution using hash table” it would recommend version 2 first because it fulfills user query better than version 1.

For nested functions, the function names are added into the caller function’s descriptor, which contributes to the similarity score. Therefore, for a given query, my method might also include the caller function in the search result when the search in fact referred to the nested function. In Table 3 an example of a function with nested calls is provided. The function named “SumofSquares” calls another function named “multiply_square”. This Calle function’s name will be added to the Caller function’s descriptor.

My proposed method also takes into consideration of recursive function calls and populates the function descriptors with recursive calls of the function by including the function names in the descriptor. For example, a recursive function is given in Table 3 where the function name is calculate_factorial and it is recursively called inside the function. The descriptor “calculate factorial” is generated for the function name and when it finds a recursive call inside the function it adds the occurrence of that recursive call in the descriptor.

Table 2. Handling multiple versions of Code

Version	Code

1	`def two_sum(nums,target):`
	`  # implements two sum in nonlinear time`
	`  for i in range(len(nums)):`
	`  left = nums[i+1:]`
	`  for j in range(len(left)):`
	`  if (nums[i] + left[j]) == target:`
	`  return i, j+i+1`
	
	Similarity scores for the query "two sum solution in nonlinear time" is 0.568, and for the query "two sum solution using hash table" is 0.193

2	`def two_sum(nums,target):`
	`  # implements using hash table in linear time`
	`  hash_table={}`
	`  for i in range(len(nums)):`
	`  hash_table[nums[i]]=i`
	`  for i in range(len(nums)):`
	`  if target-nums[i] in hash_table:`
	`  if hash_table[target-nums[i]] != i:`
	`  return [i, hash_table[target-nums[i]]]`
	`  return []`
	Similarity scores for the query "two sum solution in nonlinear time" is 0.168, and for the query "two sum solution using hash table" is 0.615

Multiple branches of decision statements are also kept in our method. For multiple branches, there will be multiple calls of variables and all the occurrences of the variable names are stored if they are readable. In Table 3, an example of a function with multiple branches is given. In the example, there are three variables red, green, blue which are used to create multiple branches. Therefore, they are used multiple times in the function. Our method stores these multiple occurrences of the variables in the function descriptor. For the variable named "red", it has 7 occurrences in the function and these 7 occurrences are stored in the function descriptor.

Table 3. Examples of various function types

Function Type	Function Body

With nested Function call	`def SumofSquares(Array, n):`
	`  # Total sum of squares of the array of elements`
	`  Sum = 0`
	`  for i in range(n):`
	`  # Square of Array[i] element is stored in SquaredValue`
	`  SquaredValue = multiply_square(Array[i])`
	`  # Cummulative sum is stored in Sum variable`
	`  Sum += SquaredValue`
	`  return Sum`

Recursive	`def calculate_factorial(x):`
	`  """This is a recursive function`
	`  to find the factorial of an integer"""`
	`  if x == 1:`
	`  return 1`
	`  else:`
	`  return (x * calculate_factorial(x-1))`

With multiple branches	`def getIdentitiyCode(self, img):`
	`  avg_color_per_row = np.average(img, axis=0)`
	`  avg_color = np.average(avg_color_per_row, axis=0)`
	`  blue, green, red = avg_color`
	`  print('%s %s %s' % (red, green, blue))`
	`  if red < 128 and blue < 128 and green < 128:`
	`  return 1`
	`  elif red > 200 and blue > 200 and green > 200:`
	`  return 2`
	`  elif red > blue and red > green:`
	`  return 3`
	`  elif blue > green and blue > red:`
	`  return 4`
	`  elif green > blue and green > red:`
	`  return 5`
	`  else`
	`  return 6`

Authors in [24] did a quantitative survey on this vocabulary mismatch problem and their results show that 80% of the time different people from the same background will name the same thing differently. To address this issue, my proposed technique offers a feature that suggests queries to the users based on their generic query and relevant code in the repository so that the user can reformulate their query based on the suggestions if necessary.

As discussed earlier, my approach generates descriptors for functions that have natural language phrases and sentences. These descriptors are exploited for suggesting queries to the users. For generating query suggestions, at first, the descriptors are retrieved from the code search results with top similarity scores based on the query issued by the user. Then, the phrases and sentences from those descriptors are considered for suggesting them as new queries. To select the phrases or sentences as the candidates for the suggested query list, the semantic similarity method is applied to the user issued query. The words and phrases from the descriptor which are semantically similar to user issued query are suggested. For calculating this semantic similarity between the user query and the phrases or sentences from the descriptors, tf-idf technique is used which was mentioned earlier in the Proposed Method section. The pseudocode of this query suggestion generation process is given in Algorithm 3.

In the literature, there are existing works, such as [29, 30, 31], where the authors use tf-idf to select the appropriate expansion terms for query reformulation. However, compared to those works, my method relied on the descriptors of the source code only because we do not use user feedbacks and metadata of the source code from StackOverflow or Github. Once the tf-idf is calculated for the user issued query and extracted phrases or sentences, the similarity between them is measured using cosine similarity. From each phrase or sentence, a vector is derived.

Then the phrases or sentences are viewed as a set of vectors in a vector space. For example, for the queries “sort items” and “how to do merge sort” the output vector will be:

[[0.81480247 0. 0.57973867] [0. 0.81480247 0.57973867]

In the vector, there are six numbers because there are six unique words in those two queries which are: sort, items, how, to, do, merge. Also, the zero values which are present in the vector will account for the similarity score calculation for dissimilar words.

Then using the formula of cosine similarity on this tf-idf weights, the similarity between two phrases or sentences is calculated. Based on this similarity score, the phrases or sentences is suggested to the user so that s/he can reformulate the original query.

Algorithm 3 Algorithm for Query Suggestion

Input: User Query, Function Descriptors of Recommended Source Code
Output: Query Suggestions

```
for each sentence_or_phrase in function Descriptor do
    sentence_or_phrase_vector =TfidfVectorizer.fit_transform(sentence)
    query_vector =TfidfVectorizer.fit_transform(userQuery)
    similarity=cosine_similarity(sentence_or_phrase, userQuery)
    if similarity > 0 then
        suggested_queries.append(sentence,similarity)
    end if
end for
suggested_queries.sort(similarity)
suggested_queries=suggested_queries[:10] {takes top 10 queries}

return suggested_queries
```

Code Search Tool

Based on my proposed technique I implemented a code search tool. In Figure 2 the component diagram is shown representing different modules of the code search tool. It consists of four main modules: Source code preprocessing and validation, Function descriptor generator, Source code search, and Query suggestion. In the figure, query suggestion module was not illustrated as it operates inside the Source Code Search Module.

Source Code Preprocessing and Validation. In my proposed technique, Python source code is considered for the codebase. As my method recommends source code from the private repository, the codebase can be enriched by adding source code files. To maintain the quality of the codebase my implemented tool preprocesses and validates source code so that the recommended source code can be reused easily into another project without any major modification. As mentioned in the proposed technique source code is converted into Abstract syntax tree.

In order to make ASTs error-free, the new source code is automatically preprocessed primarily for indentation which otherwise would lead to errors while constructing the AST. For example, in Table 4, two functions are given, which are almost identical with the same indentation and function name and body. However, in the for loop, the range function [Definition of range function: range(stop), range (start, stop [, step])] was not implemented correctly in Function 2. If we upload both of these functions, then my implemented tool will ignore Function 2 due to the syntax error.

To correct the indentation, the pycodestyle [32] tool is used to determine which parts of the code need to be formatted first. This pycodestyle is used by autopep8 [33] Python module which formats the Python code to comply with PEP8 [34] style guide.

Finally, the source code is converted into AST using the Python's ast [35] module so that if there is an error in generating the AST then the code will be ignored. The ast module is used to process the trees of Python abstract syntax grammar.

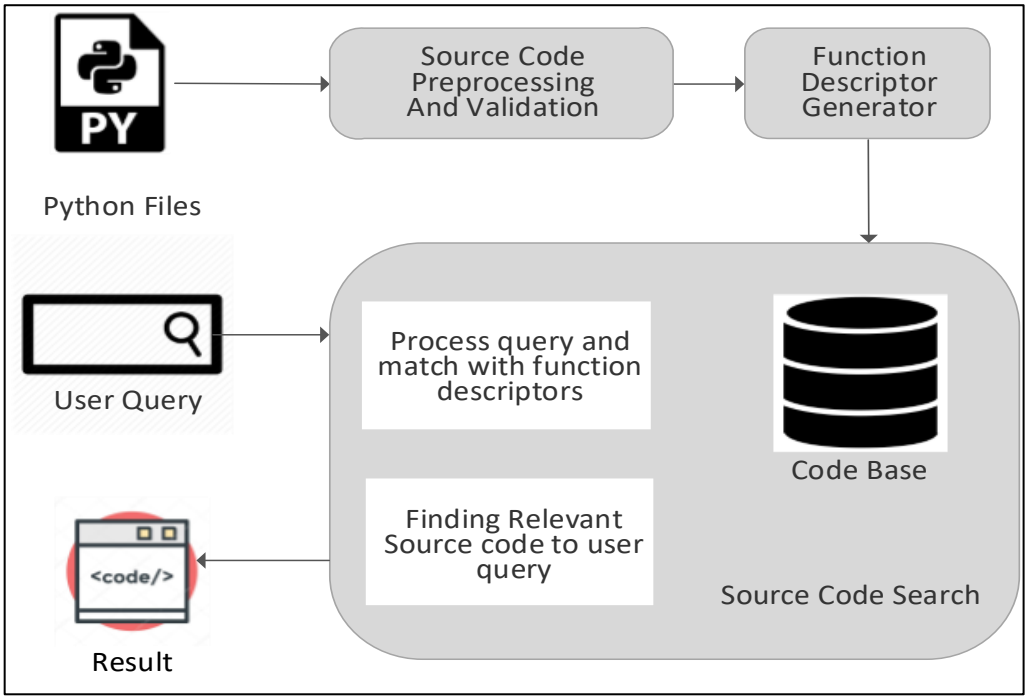

Figure 2. The complete workflow of the proposed system

Function Descriptor Generator. The function descriptor generator module retrieves information from source code. This retrieved information are called the function descriptor. These function descriptors contain information extracted from abstract syntax tree and comments of source code. Python's ast [35] module converts the source code into abstract syntax tree. Python ast module has a helper function named ast.parse() which parses the source code into an AST node. After parsing the source code ast.walk(*node*) function is used which recursively yields all descendant nodes in the tree starting at *node*. Then the type of each node is checked so

that the system can identify which nodes will be used for our Function Descriptor Generator. My system considers the nodes having information about the variables and function names. To select the nodes containing information about the variables the system checks if the node is an instance of ast.Attribute() type and for the function names the system checks if the node is an instance of ast.FunctionDef() type. For extracting the docstring of a function the system uses ast.get_docstring() function of the ast module.

Table 4. Example of valid and invalid Function

Serial	Valid/ Invalid	Function Body

1	Valid	`def reverse_string(str):`
		`  new_strings = []`
		`  index = len(str)`
		`  for i in range (len(str)):`
		`  index -= 1`
		`  new_strings.append(str[index])`
		`  return ''.join(new_strings)`

2	Invalid	`def reverse_string(str):`
		`  new_strings = []`
		`  index = len(str)`
		`  for i in range len(str):`
		`  index -= 1`
		`  new_strings.append(str[index])`
		`  return ''.join(new_strings)`

WordNet [36] and synset [37] is used to lookup extracted word from variable names within a dictionary. WordNet is the lexical database i.e. dictionary for the English language having the semantic relations between words, which is designed for natural language processing.

Synset is an interface that is used to look up in the wordnet. For example, we will check if the word “temperature” has meaning and then we will call the function wordnet.synset(“temperature”). Wordnet will return true if it finds the word “temperature” in their lexical database. By following this process, we can select the meaningful words which are later used in enriching our function descriptor.

Now, for faster query search, instead of generating function descriptors for each search query, all the uploaded source code’s functions along with their descriptors are stored in the database. When a user uploads a source code file using our web interface, my tool uses its function descriptor generator to generate the summary of each function of the source code and stores them in the database along with the corresponding function name and body.

Source Code Search. For every function, the descriptor is generated first and then stored in the database along with the source code of the function. Then the user issues a query for searching their source code and my implemented tool matches the query with the function descriptor by semantic similarity technique and then recommends relevant source code to the user.

Python’s TfidfVectorizer [38] module is used for converting function descriptors and user query into a matrix of tf-idf features. After calculating the tf-idf weights, the similarity score between the user query and function descriptor is calculated to rank the functions based on the cosine similarity.

SQLite is used as our database for initial deployment. Any site that gets fewer than 100k hits/days works fine with SQLite [39]. This 100k hits/day figure is a conservative estimate, not a hard upper bound that means if necessary, SQLite can handle more than that. SQLite has been demonstrated to work with 10 times that amount of traffic.

Query Suggestion. For implementing the Query Suggestion module, phrases and sentences generated from the Function Descriptor Generator module are used. These phrases and sentences are vectorized using Python's TfidfVectorizer [38] and converted into tf-idf features. Then the user issued query is also converted into tf-idf feature using the same technique. Then the sentences and phrases are ranked based on the similarity with user issued query and suggested as queries to the user. This similarity is calculated on the tf-idf features of the phrases, sentences of the function descriptor and user issued query.

For example, if a user is looking for a code snippet that implements a specific sorting algorithm: "merge sort", but entered a generic query, such as "sort items", then the tool will recommend relevant code snippets as per proposed method for the query "sort items" along with some query suggestions (see Table 5). Now, based on the current code repository, if we use the query "sort items" then the tool will return code snippets of the bubble sort algorithm and selection sort algorithm as the top two results as shown in Table 6. Code snippet for merge sort is ranked in the third position. Therefore, in this example, the user might not get the merge sort algorithm as the top result initially because the query was generic, and the user did not mention the merge keyword in the query. To alleviate this issue, along with the code search results, the system will suggest queries to the users for them to reformulate the original query to get modified code search results. For example, let us assume that based on suggested queries the user reformulates his query to "How to do merge sort". Then the user will get results where merge sort is ranked 1st as shown in Table 7.

Table 5. Query Suggestions for the query "Sort items"

Serial	Query Suggestions
1	selection sort
2	sorted by blue
3	sorted by red
4	sorted by value
5	merge sort
6	bubble sort

Table 6. Code search result for the query “Sort Items” with similarity scores

Functions	Similarity Score

`def bubbleSort(nlist):`
`  for passnum in range(len(nlist)-1,0,-1):`
`  for i in range(passnum):`
`  if nlist[i]>nlist[i+1]:`
`  temp = nlist[i]`
`  nlist[i] = nlist[i+1]`
`  nlist[i+1] = temp`
	0.260

`def selection_sort(arr):`
`  for i in range(len(arr)):`
`  minimum = i`
`  for j in range(i + 1, len(arr)):`
`  if arr[j] < arr[minimum]:`
`  minimum = j`
`  arr[minimum], arr[i] = arr[i], arr[minimum]`
`  return arr`
	0.175

`def merge_sort(arr):`
`  if len(arr) <= 1:`
`  return arr`
`  mid = len(arr) // 2`
`  left, right = merge_sort(arr[:mid]), merge_sort(arr[mid:])`
`  return merge(left, right, arr.copy())`
	0.175

Table 7. Code search result for the query “How to do merge sort” with similarity scores

Functions	Similarity Score

`def merge_sort(arr):`
`  if len(arr) <= 1:`
`  return arr`
`  mid = len(arr) // 2`
`  left, right = merge_sort(arr[:mid]), merge_sort(arr[mid:])`
`  return merge(left, right, arr.copy())`
	0.449

`def bubbleSort(nlist):`
`  for passnum in range(len(nlist)-1,0,-1):`
`  for i in range(passnum):`
`  if nlist[i]>nlist[i+1]:`
`  temp = nlist[i]`
`  nlist[i] = nlist[i+1]`
`  nlist[i+1] = temp`
	0.261

`def selection_sort(arr):`
`  for i in range(len(arr)):`
`  minimum = i`
`  for j in range(i + 1, len(arr)):`
`  if arr[j] < arr[minimum]:`
`  minimum = j`
`  arr[minimum], arr[i] = arr[i], arr[minimum]`
`  return arr`
	0.175

Let us consider another example, where the user provides issues generic query “image operation” which does not indicate any specific operation. After issuing this query my implemented tool will suggest the user some queries given in Table 8, which will provide the user with an overview of the available related source code of the codebase. After going through the suggestions, the user will be able to specify their query to get the source code which performs that specific operation like cropping the image, read images, list images, etc.

Table 8. Query Suggestions for the query "image operation"

Serial	Query Suggestions
1	get stitched image
2	the pixels from the image
3.	read image
4	list images
5	blob from image
6	crops the image

Web-Based Deployment. I have a web-based deployment of my system where the user can search source code by issuing a query in natural language. After issuing the query the user will get related source code along with query suggestions. The user can also upload their source code to enrich the codebase. This web-based system has two main interfaces: search and upload. In the search interface, users will search source code by natural language query. The screen capture of the search interface is given in Figure 3 and the upload interface is given in Figure 4. We share the screen capture of the suggested queries in Figure 5 and recommended source code returned by the system in Figure 6 for a sample query “track person activity”.

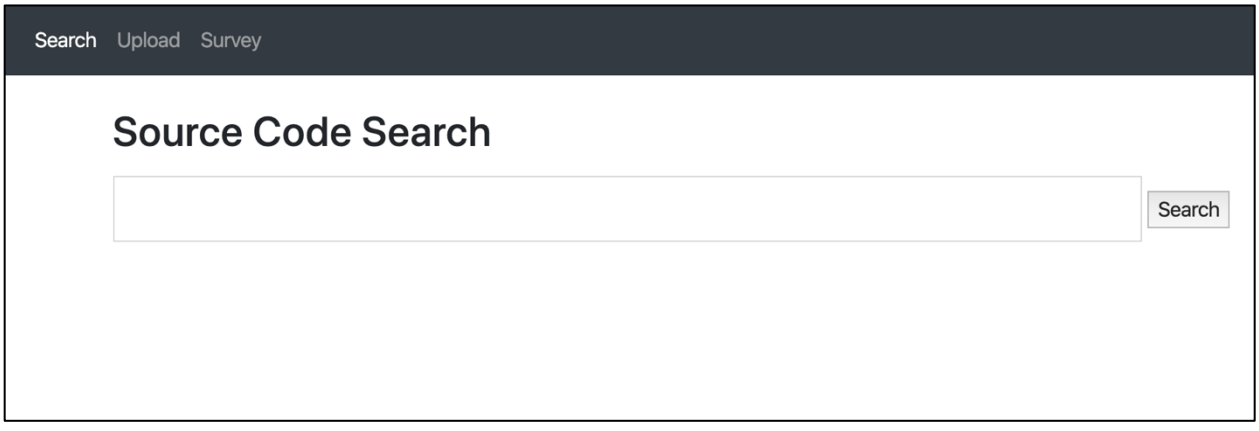

Figure 3. Screen capture of the web interface’s Search module

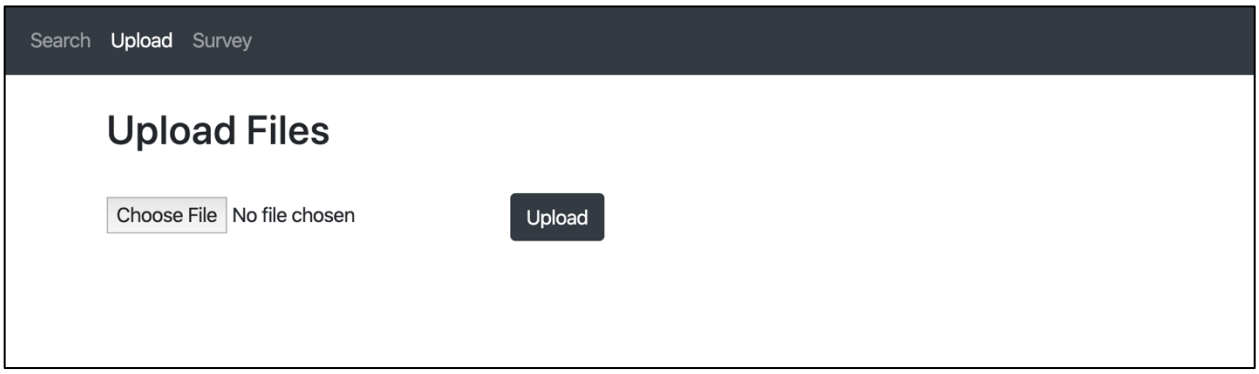

Figure 4. Screen capture of the web interface's Upload module

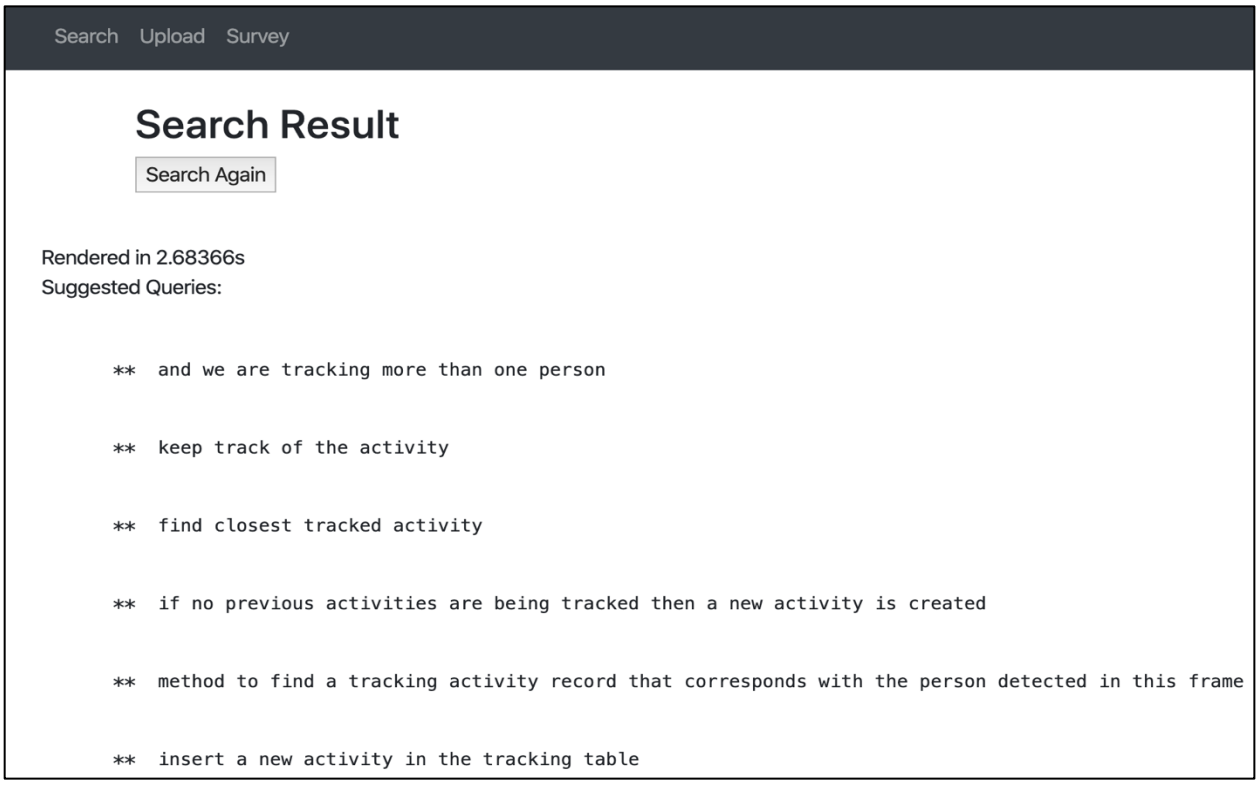

Figure 5. Screen capture of suggested queries for the query "track person activity"

```
####        Input String Was: track person activity

##Function               :

def find_closest_tracked_activity(self, rect_start, newLabel,

    all_detected_points):

    detected_person_count = len(all_detected_points)

    all_detected_points_except_this_one = list([x for x in

        all_detected_points if x[0] != rect_start[0] or x[1] != rect_start[1]])

    self.unused_tracked_list = list(set(self.tracked_list) - set(self.

        used_activity))

    if not self.tracked_list:

        return self.begin_new_tracking(rect_start)

    else:

        closest_t = None

        for t in self.unused_tracked_list:

            if closest_t:

                closest_t = t if distance(t.getRect_start(), rect_start
```

Figure 6. Screen capture of recommended source code for the query “track person activity”

EVALUATION

My proposed technique recommends source code from private source code repository and most of the existing source code recommendation systems use crowdsourced data or API documentation to recommend source codes to the user. So, it is not feasible to compare my proposed method's effectiveness in terms of accuracy with other systems.

To evaluate the effectiveness of my proposed approach, the codebase was initially populated with Python projects from a Software Engineering Capstone class. Around 583 functions were retrieved from these projects.

A survey was conducted to gain feedback from the users about the accuracy of my proposed method. A total of 28 senior undergraduate and graduate Computer Science students (with internship and real-world software development experiences) participated in this survey where they tested my implemented tool by uploading Python source code from their own software projects. They also searched for related functions at different stages of uploading their code using natural languages and specific requirements from their own projects. The three questions used for the survey cover the following aspects:

1. *If they had any prior experience with any source code recommendation system*,

2. *Did they get the expected outputs from the system based on their queries*, and

3. *If they would be able to use any of these search results (i.e. source code) on a different project*.

Details of the survey questions are given in Appendix A. For the first question, the participants answered either "yes" or "no". For the last two questions, the participants gave their

feedback on a scale of 0 to 5. where 0 represents strongly disagree and 5 represents strongly agree. The detailed representation of the scores are given in Table 9.

Table 9. Representation of Survey Rating

Rating	Representation
0	Strongly Disagree
1	Disagree
2	Partially Disagree
3	Partially Agree
4	Agree
5	Strongly Agree

After compiling the survey results, we found that the majority of the participants (23 out 28) did not use a code recommendation system before and a graphical presentation of question 1's answer is given in Figure 7. For the second question, 53.6% of the participants gave 4 out of 5 and 25% of participants gave 5 out of 5. For the third question, 35.7% of participants gave 4 out of 5 and 32.1% participants gave 5 out of 5. The rest of the participants responded to somewhat agree, i.e., a score of 3, for the last two questions. Figure 8 and Figure 9 graphically shows the response of survey question 2 and 3 respectively. The survey result demonstrates the potential of my tool to search private code repositories using natural language queries. Also, some outputs of my implemented system with some generic queries are illustrated in Appendix B.

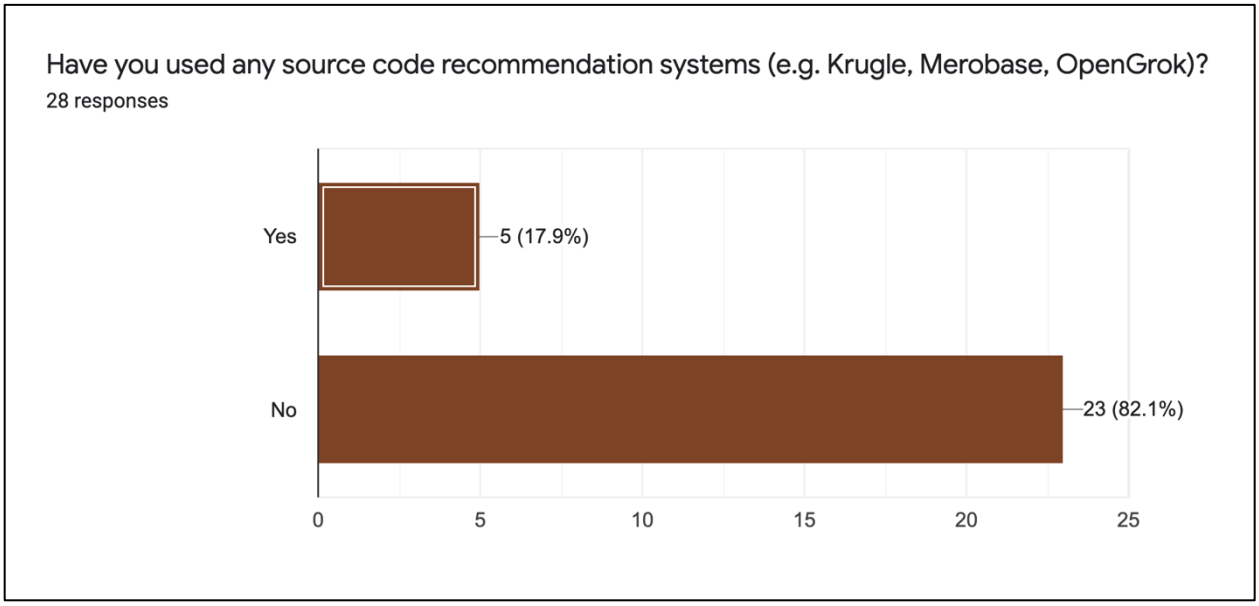

Figure 7. Survey Result for Question 1

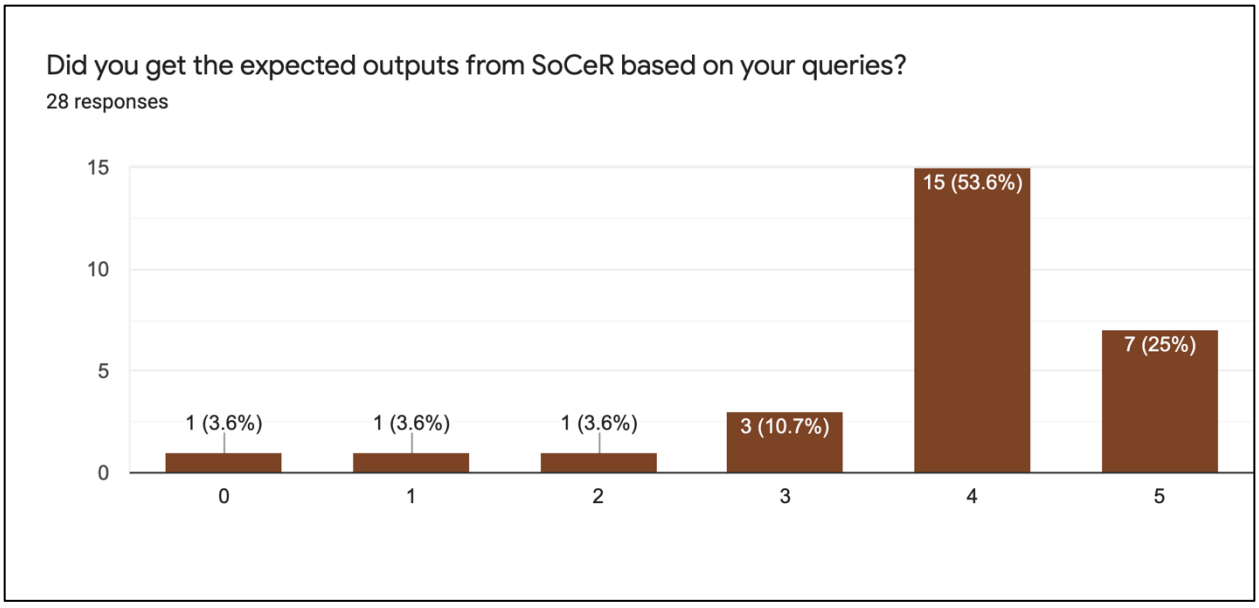

Figure 8. Survey Result for Question 2

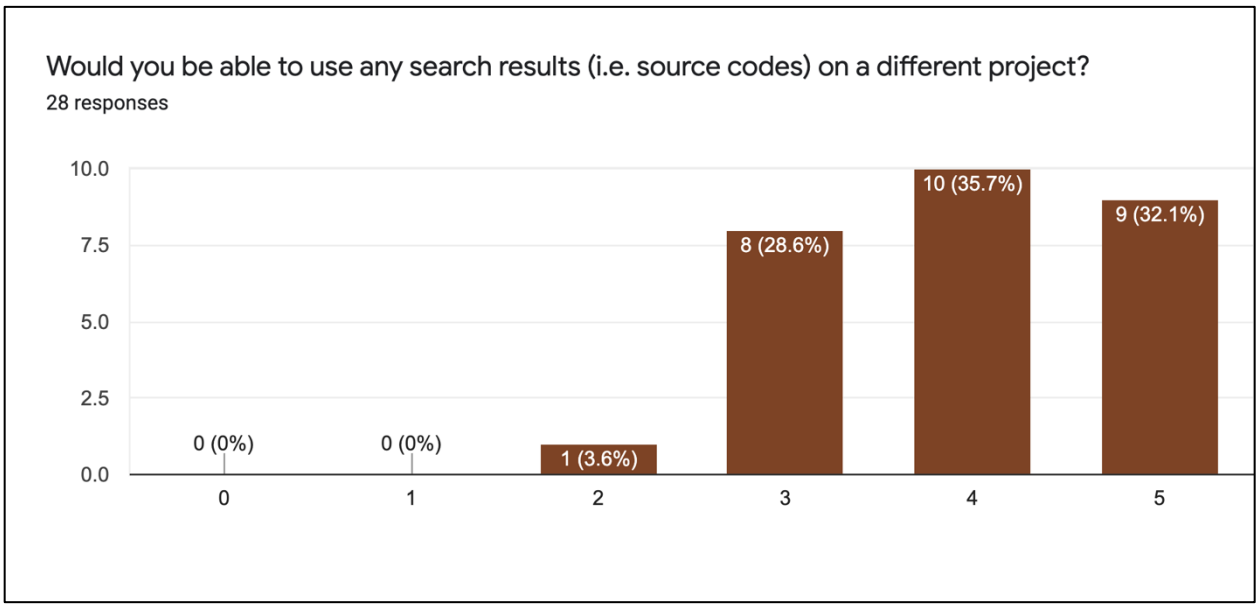

Figure 9. Survey Result for Question 3

In terms of accuracy, my system suggests source code for some queries which are not expected sometimes. For example, let us consider a query “google map location tracking”. In the codebase we do not have any source code related to “google map location tracking”. Still, my system recommends source code snippets which are not relevant to user issued query. In Table 10 I have listed two functions recommended by my system for the query “google map location tracking”. The function “decode_phrase_GS” does not deal with google speech API but it was recommended for this above-mentioned query because it has “google” keyword in itself. Even though this function does not track anything using google map location. The second function “two_sum” was also recommended by the system because it uses the HashMap data structure that has the keyword “map” which matches the input query. That is why this function was recommended to the user.

Table 10. Code search result for the query “google map location tracking” with similarity scores

Functions	Similarity Score

`def decode_phrase_GS(self, wav_file):`
`"""`
`wav_file: .wav file created from save_speech function.`
`decode using Google Speech API module.`
`returns the formmated string.`
`"""`
`r = sr.Recognizer()`
`with sr.AudioFile(wav_file) as source:`
`audio = r.record(source)`
`try:`
`return r.recognize_google(audio)`
`except sr.UnknownValueError:`
`print("Google Speech Recognition could not understand audio")`
`return None`
`except sr.RequestError as e:`
`print(`
`"Could not request results from Google Speech Recognition service; {0}".format(e))`
`return None`
	0.090

`def two_sum(nums,target):`
 	
`# implements using hash map in linear time`
 	
`hash_table={}`
`for i in range(len(nums)):`
`hash_table[nums[i]]=i`
 	
`for i in range(len(nums)):`
`if target-nums[i] in hash_table:`
`if hash_table[target-nums[i]] != i:`
`return [i, hash_table[target-nums[i]]]`
 	
`return []`
	0.0834

In Tables 11,12 the behavior of similarity score change is shown for different combination of parameters i.e. function name, variable names, docstring, and comments. In the experimental setup, I issued a query and picked up the top-ranked function returned by my system. Then, the function was manipulated to observe the impact of function descriptor parameters on the output result. For the Query “how to implement bubble sort algorithm” the top-ranked result has a similarity score of 0.189 where all four parameters are considered for generating the descriptor. When the function name is considered as the only parameter to populate function descriptor, the similarity score changes to 0.579. But when comments or variable names are considered as the parameters, the similarity score becomes 0 which means comments or variable names in that particular code does not have any match with the input query. Then the combinations of two parameters are considered and the change of similarity score is checked. For taking combination of two parameters we kept the function name parameter fixed as it accounted for the highest similarity score and added other parameters like docstring or variable names to make the combination. For the combination of docstring and function name, the similarity score is 0.368. For this particular example function name plays the most important role in similarity score and if we consider other combinations with function names the similarity score decreases.

Let’s consider another example given in Table 12, where the input query is “Send email with attachment”. Here every parameter of function descriptor contributes to the similarity score. If we consider the comments as the only parameter, the similarity score is 0.314. Unlike the first example, the similarity score doesn’t decrease if take more combination of parameters because every parameter is contributing to the similarity score. If we take a combination of comments and function name, the similarity score changes to 0.431. Finally, when we consider all the parameters the similarity score becomes the which is 0.529 which is a better score than other

combination's scores. The same output behavior of similarity score is observed for the output of query "unified hand gesture detection" and "find shortest path in graph" which are given in Tables 11 and 12.

Table 11. Similarity Scores for different combinations of Parameters-A

Parameter Combination for Input query: "how to implement bubble sort algorithm"	Similarity Score	Parameter combination for Input Query: "unified hand gesture detection"	Similarity Score
Function Name	0.579	Docstring	0.245
Docstring	0.206	Function Name	0.056
Comments	0	Comments	0.267
Variable Names	0	Variable Names	0.091
Function name, Docstring	0.368	Comments, Function Name	0.263
Function Name, Variable Names	0.579	Comment, Docstring	0.364
Function Name, Comments	0.101	Comments, Variable Names	0.260
Function name, Docstring, Comment	0.189	Comment, Docstring, Variable Names	0.352
Function name, Docstring, variable	0.368	Comment, Docstring, Function name	0.360
All	0.189	All	0.350

Table 12. Similarity Scores for different combinations of Parameters-B

Parameter Combination for Input Query: "Send email with attachment".	Similarity Score	Parameter Combination for Input Query: "find shortest path in graph"	Similarity Score
Function Name	0.249	Function Name	0.173
Docstring	0.234	Docstring	0.132
Comments	0.314	Variable	0.174
Variable Names	0.114	Comments	0.188
Comments, Function name	0.431	Comments, Function name	0.291
Comments, docstring	0.417	Comments, Docstring	0.243
Comments, Variable Names	0.344	Comments, Variable Names	0.230
Comment, docstring Function name	0.514	Comment, Function name, Docstring	0.279
Comments, Docstring, Variable Names	0.438	Comment, Function name, Variable Names	0.278
All	0.529	All	0.338

To get an overview of how my proposed technique works in comparison to other publicly available recommendation tools in Figure 10, a screen capture of popular code search engine krugle is shared where a query "image operation" is issued. We can also observe the output from my system for the same query in Figure 11. The detailed result of the output is given in Appendix B. In Krugle, it only matches the search keywords in the source code and highlights that part of source code and the user has to navigate through the source code file to get the

complete source code snippets. But my system returns full functions where the users do not need to navigate through the project files, and they can directly reuse the recommended source code to their own projects.

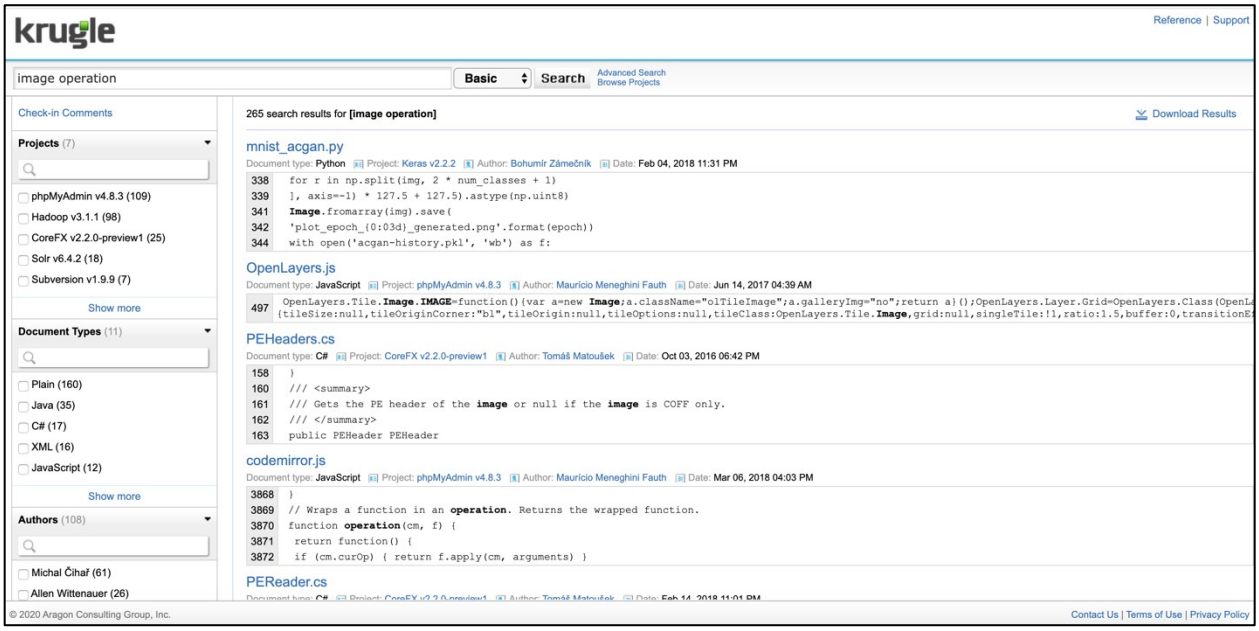

Figure 10. Screen Capture from code recommendation tool Krugle for the query “image operation”

```
####        Input String Was: image operation

##Function               :

    def __frame_to_image__(self, frame):

        return PIL.ImageTk.PhotoImage(image=PIL.Image.fromarray(frame))


Similarity With Input:0.4952364388460705

--------------------------------------------------------------------------

##Function               :

    def __display_image__(self, image):

        self.image = image

        self.debug_canvas.create_image(

            0, 0, image=self.image, anchor=tkinter.NW)


Similarity With Input:0.4558669899766566

--------------------------------------------------------------------------

##Function               :

def readImagesFromDirectory(path):

    imagePaths = sorted(list(imutils.paths.list_images(path)))

    images = []

    for imagePath in imagePaths:

        image = cv2.imread(imagePath)

        images.append(image)

    return images
```

Figure 11. Screen Capture from the system for the query “image operation”

CONCLUSION

In this thesis, I presented a novel technique for searching source code in private code repositories using natural language queries. The main contributions of this thesis include a code descriptor that represents a summary of the source code, a code recommendation technique that takes natural language as input and returns relevant source code from the local repository. I have also incorporated a query suggestion feature where the users will be suggested with related queries based on their initial textual query and relevant code descriptors.

Instead of searching the source code directly, this approach finds the semantic similarity between the user query and the descriptor of the source code and returns the relevant results with high similarity scores. Since my goal is to promote reliable code reuse based on a verified codebase, my proposed technique does not rely on a crowdsourced software repository as observed in the existing code recommendation tools and techniques discussed in the "Related Works" section. Moreover, my proposed technique can be used to find relevant source code even before the actual coding takes place by means of allowing the software requirement specifications to be used as the search queries. As my technique recommends source code from private source code, software companies will be greatly benefitted from it. Also, the users do not need to have knowledge of the software structure or workflow, instead, they can use the software requirements as queries to get relevant implementation or code snippets.

The Accuracy of our proposed method depends on some factors. For example, the quality of the descriptors depends on the quality of documentation of source code. By documentation, I refer to the naming convention of the function and variables, good structured docstrings and comments.

The quality of query suggestions also depends on the documentation of the source code. As mentioned earlier in the "Proposed Code Search Method" section, suggested queries are generated from the descriptors of the functions. The codebase of proof of concept tool was populated with the source code of student projects and some of the projects had poorly documented source code. Therefore, it might fail to suggest relevant source codes given its limited codebase size.

Last but not least, our method does not recommend source code that has syntax errors which makes the recommended source code reusable. Additionally, it can handle multiple versions of the source code during recommendation. At present my proposed technique only works with Python source code. My goal is to extend the functionalities of my proposed technique for other structured programming languages in the future.

REFERENCES

[1] J. Stylos and B. A. Myers, “Mica: A Web-Search Tool for Finding API Components and Examples,” Visual Languages and Human-Centric Computing (VL/HCC’06), 2006, pp. 195–202.

[2] M. Grechanik, C. Fu, Q. Xie, C. McMillan, D. Poshyvanyk, and C. Cumby, “A Search Engine for Finding Highly Relevant Applications,” in Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering – Vol. 1, New York, NY, USA, 2010, pp. 475–484.

[3] R. Holmes and G. C. Murphy, “Using structural context to recommend source code examples,” in Proceedings. 27th International Conference on Software Engineering (ICSE), 2005, pp. 117–125.

[4] S. Thummalapenta and T. Xie, “Parseweb: A Programmer Assistant for Reusing Open Source Code on the Web,” in Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering, New York, NY, USA, 2007, pp. 204–213.

[5] The Koders source code search engine, 2005. http://www.koders.com

[6] The Krugle source code search engine. http://www.krugle.com

[7] M. Raghothaman, Y. Wei, and Y. Hamadi, “SWIM: Synthesizing What I Mean: Code Search and Idiomatic Snippet Synthesis,” in Proceedings of the 38th Intl. Conference on Software Engineering, 2016, pp. 357–367.

[8] E. Linstead, S. Bajracharya, T. Ngo, P. Rigor, C. Lopes, and P. Baldi, “Sourcerer: mining and searching internet-scale software repositories,” Data Min Knowl Disc, vol. 18, no. 2, Apr. 2009, pp. 300–336.

[9] O. A. L. Lemos et al., “CodeGenie: Using Test-cases to Search and Reuse Source Code,” in Proceedings of the 22nd Intl. Conference on Automated Software Engineering (ASE), New York, NY, USA, 2007, pp. 525–526.

[10] A. Zagalsky, O. Barzilay, and A. Yehudai, “Example Overflow: Using social media for code recommendation,” in International Workshop on Recommendation Systems for Software Engineering, 2012, pp. 38–42.

[11] M. McCandless, E. Hatcher, and O. Gospodnetic, Lucene in Action, Second Edition: Covers Apache Lucene 3.0. Greenwich, CT, USA: Manning Publications Co., 2010.

[12] H. C. Wu, R. W. P. Luk, K. F. Wong, and K. L. Kwok, “Interpreting TF-IDF Term Weights As Making Relevance Decisions,” ACM Trans. Inf. Syst., vol. 26, no. 3, Jun. 2008, pp. 13:1–13:37.

[13] Y. Oda et al., “Learning to Generate Pseudo-Code from Source Code Using Statistical Machine Translation (T),” in 30th IEEE/ACM Intl. Conference on Automated Software Engineering, 2015, pp. 574–584.

[14] S. P. Reiss, “Semantics-based code search,” in Proceedings of 31st Intl. Conference on Software Engineering, USA, 2009, pp. 243–253.

[15] Raphael Sirres, Tegawendé F. Bissyandé, Dongsun Kim, David Lo, Jacques Klein, and Yves Le Traon, “Augmenting and Structuring User Queries to Support Efficient Free-Form Code Search”, Empirical Software Engineering Journal, Vol. 23, No. 5, 2018, pp. 2622–2654.

[16] M. M. Rahman and C. Roy, "Effective Reformulation of Query for Code Search Using Crowdsourced Knowledge and Extra-Large Data Analytics," 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), Madrid, 2018, pp. 473-484

[17] C. Ragkhitwetsagul and J. Krinke, “Siamese: scalable and incremental code clone search via multiple code representations,” Empir Software Eng, vol. 24, no. 4, Aug. 2019, pp. 2236–2284.

[18] P. W. McBurney and C. McMillan, “Automatic documentation generation via source code summarization of method context,” in Proceedings of the 22nd International Conference on Program Comprehension, Hyderabad, India, Jun. 2014, pp. 279–290.

[19] E. Hill, L. Pollock, and K. Vijay-Shanker, “Automatically capturing source code context of NL-queries for software maintenance and reuse,” in 2009 IEEE 31st International Conference on Software Engineering, May 2009, pp. 232–242.

[20] E. Reiter and R. Dale, “Building applied natural language generation systems,” Natural Language Engineering, vol. 3, no. 1, Mar. 1997, pp. 57–87.

[21] A. N. Langville and C. D. Meyer. Google's PageRank and Beyond: The Science of Search Engine Rankings. Princeton University Press, Princeton, NJ, USA, 2006.

[22] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker, “Towards automatically generating summary comments for Java methods,” in Proceedings of the IEEE/ACM

international conference on Automated software engineering, Antwerp, Belgium, Sep. 2010, pp. 43–52.

[23] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Summarizing Source Code using a Neural Attention Model," in Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), Berlin, Germany, Aug. 2016, pp. 2073–2083.

[24] G. W. Furnas, T. K. Landauer, L. M. Gomez, and S. T. Dumais, "The vocabulary problem in human-system communication," Commun. ACM, vol. 30, no. 11, Nov. 1987, pp. 964–971.

[25] D. Liu, A. Marcus, D. Poshyvanyk, and V. Rajlich, "Feature location via information retrieval based filtering of a single scenario execution trace", in Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering, 2007, pp. 234-243.

[26] B. Sisman and A. C. Kak, "Assisting code search with automatic Query Reformulation for bug localization," in 2013 10th Working Conference on Mining Software Repositories (MSR), May 2013, pp. 309–318.

[27] L. Nie, H. Jiang, Z. Ren, Z. Sun, and X. Li, "Query Expansion Based on Crowd Knowledge for Code Search," IEEE Trans. Serv. Comput., vol. 9, no. 5, Sep. 2016, pp. 771–783.

[28] Z. Li, T. Wang, Y. Zhang, Y. Zhan, and G. Yin, "Query reformulation by leveraging crowd wisdom for scenario-based software search," in Proceedings of the 8th Asia-Pacific Symposium on Internetware, Beijing, China, Sep. 2016, pp. 36–44.

[29] S. Haiduc, G. Bavota, A. Marcus, R. Oliveto, A. De Lucia, and T. Menzies, "Automatic query reformulations for text retrieval in software engineering," in Proceedings of the International Conference on Software Engineering, San Francisco, CA, USA, 2013, pp. 842–851.

[30] L. Nie, H. Jiang, Z. Ren, Z. Sun, and X. Li, "Query Expansion Based on Crowd Knowledge for Code Search," IEEE Trans. Serv. Comput., vol. 9, no. 5, Sep. 2016, pp. 771–783.

[31] M. M. Rahman, "Supporting code search with context-aware, analytics-driven, effective query reformulation," in Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings, Montreal, Quebec, Canada, 2019, pp. 226–229.

[32] https://pypi.org/project/pycodestyle/

[33] https://pypi.org/project/autopep8/0.8/

[34] https://www.python.org/dev/peps/pep-0008/

[35] https://docs.python.org/3/library/ast.html

[36] Christiane Fellbaum (1998, ed.) WordNet: An Electronic Lexical Database. Cambridge, MA: MIT Press.

[37] https://www.nltk.org/_modules/nltk/corpus/reader/wordnet.html

[38] https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html

[39] https://www.sqlite.org/index.html

APPENDICES

Appendix A. Survey Questionnaire

1. Have you used any source code recommendation systems (e.g. Krugle, Merobase, OpenGrok)?
    - Yes
    - No
2. Did you get the expected outputs from the system based on your queries?
    - 0 (Strongly Disagree)
    - 1
    - 2
    - 3
    - 4
    - 5 (Strongly Agree)
3. Would you be able to use any search results (i.e. source code) on a different project?
    - 0 (Strongly Disagree)
    - 1
    - 2
    - 3
    - 4
    - 5 (Strongly Agree)

Appendix B. Output for Sample Queries

Input Query: how to populate database entry

Query suggestions:

```
* as well as updating the database.
* database manager
* populate db
* database import
* if there are device actions, populate them.
* inserts each word into the database.
* delete everything from the database. nuke
* save compound data into the database
* loops through each file and adds each word to the database.
```

Recommended Source Code:

```
def populateDB():

    for name in fileNames:  # Loops through each file and adds each word to the database.

        print("Populating " + wordTypes[name] + " words...")

        with open(directory + name, "r") as fp:

            dbCursor.execute("SELECT WORD_TYPES.ID FROM WORD_TYPES WHERE WORD_TYPE=?;", [

                             wordTypes[name]])  # Grabs ID for current word type.

            wordTypeId = dbCursor.fetchone()[0]

            fileContents = fp.readlines()

            for word in fileContents:  # Inserts each word into the database.

                dbCursor.execute(

                    "INSERT INTO WORDS (WORD, WORD_TYPE_ID) VALUES (?, ?)", (word.rstrip(),
wordTypeId))

        # If there are device actions, populate them.

        if deviceActions.get(wordTypes[name]):

            print("Adding Device Actions for: ", wordTypes[name])

            dbCursor.executemany(

                "INSERT INTO STATES (STATE, WORD_TYPE_ID, EXTREMITY_LEVEL) VALUES (?, ?, ?)",
deviceActions[wordTypes[name]])

    dbConnection.commit()
```

```
Similarity with Input:0.19238656477558375

--------------------------------------------------------------------------

##Function               :

def __set_up_configuration__(self):

    self.open_eye_threshold = self.database_manager.get_open_eye_threshold()

    self.minimum_time_increment = self.database_manager.get_minimum_time_increment()

    self.maximum_time_increment = self.database_manager.get_maximum_time_increment()


Similarity with Input:0.11671773546032795

--------------------------------------------------------------------------

##Function               :

def getSelectStatement(self):

    return (

        'select id, label, start_time, end_time, camera_id, next_camera_id, has_arrived from
tracking where id = %s'

         % self.id)


Similarity with Input:0.11576041312872845

--------------------------------------------------------------------------

##Function               :

def getSelectStatement(self):

    return (

        'select id, camera_IP, left_cam_id, right_cam_id, is_online from camera where id = %s'

         % self.id)


Similarity with Input:0.11576041312872845

--------------------------------------------------------------------------
```

```
##Function                :

def getUpdateStatement(self):

    return (

        "update tracking set end_time = current_timestamp, next_camera_id = %s, has_arrived =
'%s' where id = %s"

         % (self.next_camera_id if self.next_camera_id else 'null', 'T' if

        self.has_arrived else 'F', self.id))



Similarity with Input:0.10637919674747379

----------------------------------------------------------------------

##Function                :

def set_value(self, value):

    for option in self.database_manager.get_commands():

        if value == option["command_text"] and value != "None":

            self.display_error_message("Smart Home Device Linked",

                                       "This smart home device has already been linked to "

                                       "another command. Please chose a different device "

                                       "for this command.")

            return



Similarity with Input:0.10494943774156197
```

Input query: track person activity

Query suggestions:

* and we are tracking more than one person
* keep track of the activity

```
* find closest tracked activity
* if no previous activities are being tracked then a new activity is created
* method to find a tracking activity record that corresponds with the person detected in this
frame represented by rect_start and newlabel
* insert a new activity in the tracking table
* mark it as being detected so we know it's an active tracking
* update a preexisting activity in the tracking table
* initialize detected value of the activities we are tracking to false up front and those that
are
* find all the traced activities not yet paired up with a person in this frame
```

Recommended Source Code:

```
####

##Function               :

def find_closest_tracked_activity(self, rect_start, newLabel,

    all_detected_points):

    detected_person_count = len(all_detected_points)

    all_detected_points_except_this_one = list([x for x in

        all_detected_points if x[0] != rect_start[0] or x[1] != rect_start[1]])

    self.unused_tracked_list = list(set(self.tracked_list) - set(self.

        used_activity))

    if not self.tracked_list:

        return self.begin_new_tracking(rect_start)

    else:

        closest_t = None

        for t in self.unused_tracked_list:

            if closest_t:

                closest_t = t if distance(t.getRect_start(), rect_start

                    ) < distance(closest_t.getRect_start(), rect_start

                    ) else closest_t

                if newLabel != None and closest_t.getLabel() == newLabel:

                    self.used_activity.append(closest_t)

                    return closest_t
```

```
            else:
                closest_t = t
        more_people_than_activities = detected_person_count > len(self.
            tracked_list)
        if (not closest_t or more_people_than_activities and self.
            is_this_activity_closer_to_someone_else(closest_t,
            all_detected_points_except_this_one, rect_start)):
            print(more_people_than_activities)
            print(closest_t)
            closest_t = self.begin_new_tracking(rect_start)
        self.used_activity.append(closest_t)
        return closest_t


Similarity With Input:0.3978061784997695

--------------------------------------------------------------------------

##Function               :

def saveActivityLabel(self, t):
    conn = self.mysql.connect()
    cursor = conn.cursor()
    print(('saving %s', t.getLabel()))
    cursor.execute("update tracking set label = '%s' where id = %s" % (t.
        getLabel(), t.getID()))
    conn.commit()


Similarity with Input:0.3384250280330595

--------------------------------------------------------------------------

##Function               :
```

```
def getIdentitiyCode(self, img):

    avg_color_per_row = np.average(img, axis=0)

    avg_color = np.average(avg_color_per_row, axis=0)

    b, g, r = avg_color

    print('%s %s %s' % (r, g, b))

    if r < 128 and b < 128 and g < 128:

        return 1

    elif r > 200 and b > 200 and g > 200:

        return 2

    elif r > b and r > g:

        return 3

    elif b > g and b > r:

        return 4

    elif g > b and g > r:

        return 5

    else:

        return 6


Similarity with Input:0.33821653926182
```

```
Input Query: image operation
```

Query suggestions:

* get stitched image
* the pixels from the image
* read image
* list images
* blob from image
* crops the image
* image file1
* upload images
* completed images

Recommended Source Code:

```
##Function               :

    def __frame_to_image__(self, frame):

        return PIL.ImageTk.PhotoImage(image=PIL.Image.fromarray(frame))


Similarity With Input:0.4952364388460705

--------------------------------------------------------------------------

##Function               :

def __display_image__(self, image):

    self.image = image

    self.debug_canvas.create_image(

        0, 0, image=self.image, anchor=tkinter.NW)


Similarity With Input:0.4558669899766566

--------------------------------------------------------------------------

##Function               :

def readImagesFromDirectory(path):

    imagePaths = sorted(list(imutils.paths.list_images(path)))

    images = []

    for imagePath in imagePaths:

        image = cv2.imread(imagePath)

        images.append(image)

    return images

Similarity With Input:0.44290821862299184

--------------------------------------------------------------------------

##Function               :

def writeImage(path, image):
```

```
    cv2.imwrite(path, image)


Similarity With Input:0.3360969272762574

----------------------------------------------------------------------

##Function               :

def PullImages(filename):

    """""""

    file = PyPDF2.PdfFileReader(open(filename, 'rb'))

    xObject = file.getPage(0)

    xObject = xObject['/Resources']['/XObject'].getObject()

    images = []

    for obj in xObject:

        if xObject[obj]['/Subtype'] == '/Image':

            size = xObject[obj]['/Width'], xObject[obj]['/Height']

            data = xObject[obj]._data

            if xObject[obj]['/ColorSpace'] == '/DeviceRGB':

                mode = 'RGB'

            else:

                mode = 'P'

            if xObject[obj]['/Filter'] == '/FlateDecode':

                img = Image.frombytes(mode, size, data)

                img.save(filename + '.png')

                images += [filename + '.png']

            elif xObject[obj]['/Filter'] == '/DCTDecode':

                img = open(filename + '.jpg', 'wb')

                img.write(data)

                img.close()

                images += [filename + '.jpg']
```

```
            elif xObject[obj]['/Filter'] == '/JPXDecode':
                img = open(filename + '.jp2', 'wb')
                img.write(data)
                img.close()
                images += [filename + '.jp2']
    return images

Similarity With Input:0.3018552604934785
--------------------------------------------------------------------------
##Function               :
    def stitchOrderedImages(self):
        # Create stitcher and stitch images
        stitcher = cv2.createStitcher(True)
        status, image = stitcher.stitch(self.images)

Similarity With Input:0.2764558756128082
--------------------------------------------------------------------------
##Function               :
def test_directory_setup():
    createUploadDir()
    filename1 = 'test/coverage_test/file_upload/test/test_image.jpg'
    image_file1 = Image.open(filename1)
    checkImagesAndSaveToDirectory(
        [image_file1], 'test/coverage_test/file_upload')
    filename2 = 'test/coverage_test/file_upload/test/invalid_test_image.png'
    image_file2 = Image.open(filename2)
    checkImagesAndSaveToDirectory(
        [image_file2], 'test/coverage_test/file_upload')
```

```
Similarity With Input:0.24845822273736765

--------------------------------------------------------------------------

##Function               :

    def set_debug_frame(self, frame):

        self.__display_image__(

            self.__frame_to_image__(self.__bgr_to_rgb__(frame)))


Similarity With Input:0.24845822273736765

--------------------------------------------------------------------------

##Function               :

def uploadImagesAndConfigure():

    newDir = file_util.createUploadDir()

    timestamp = newDir.split("/")[3]

    paramDict[timestamp] = Parameters()

    # convert js 'bool' to python Bool

    paramDict[timestamp].wantsCrushedBeads = True if request.args['wantsCrushed'] == 'true'
else False

    paramDict[timestamp].wantsWaterBubbles = True if request.args['wantsBubbles'] == 'true'
else False

    paramDict[timestamp].detectionAlgorithm = request.args['colorAlgorithm']

    paramDict[timestamp].minRadius = int(request.args['minBead'])

    paramDict[timestamp].maxRadius = int(request.args['maxBead'])

    if 'maglevel' in request.args:

        paramDict[timestamp].magnificationLevel = request.args['maglevel']

    images = request.files.getlist("images")

    if not file_util.checkImagesAndSaveToDirectory(images, newDir):

        return jsonify({"status": 1, "msg": "One or more of the images that were uploaded are
in the incorrect format. Accepted formats: "

                        + (", ".join(file_util.ALLOWED_IMAGE_EXTENSIONS))})
```

```
    # redirect to homepage

    return jsonify({"status": 0, "msg": "Success", "location":
newDir.replace("Server/resources/uploads", "")})

Similarity With Input:0.24041534570913445

--------------------------------------------------------------------

##Function               :

def deepHand(image):

    d = 227

    image = cv2.resize(image, (d, d))

    blob = cv2.dnn.blobFromImage(image, 1.0, (d, d), (104.0, 177.0, 123.0))

    net.setInput(blob)

    results = net.forward()

    return results[0]


Similarity With Input:0.2285881613795591

--------------------------------------------------------------------

##Function               :

    def twoRoundStitch(self, sourceDirectory, resultsDirectory):

        output = mp.Queue()

        # first we run the two rounds with WTA_K set to 4

        self.resultsDirectory = resultsDirectory

        self.setDirectory(sourceDirectory)

Similarity With Input:0.1862248497286786

--------------------------------------------------------------------
```