2015 30th IEEE/ACM International Conference on Automated Software Engineering

CodeHow: Effective Code Search based on API Understanding and Extended Boolean Model

Fei Lv*, Hongyu Zhang†, Jian-guang Lou†, Shaowei Wang‡, Dongmei Zhang†, and Jianjun Zhao§
*School of Software, Shanghai Jiao Tong University, China
lvfei-sjtu@sjtu.edu.cn
†Microsoft Research, Beijing, China
{honzhang, jlou, dongmeiz}@microsoft.com
‡School of Information Systems, Singapore Management University, Singapore
shaoweiwang.2010@phdis.smu.edu.sg
§Department of Computer Science and Engineering, Shanghai Jiao Tong University, China
zhao-jj@sjtu.edu.cn

***Abstract*—Over the years of software development, a vast amount of source code has been accumulated. Many code search tools were proposed to help programmers reuse previously-written code by performing free-text queries over a large-scale codebase. Our experience shows that the accuracy of these code search tools are often unsatisfactory. One major reason is that existing tools lack of query understanding ability. In this paper, we propose CodeHow, a code search technique that can recognize potential APIs a user query refers to. Having understood the potentially relevant APIs, CodeHow expands the query with the APIs and performs code retrieval by applying the Extended Boolean model, which considers the impact of both text similarity and potential APIs on code search. We deploy the backend of CodeHow as a Microsoft Azure service and implement the front-end as a Visual Studio extension. We evaluate CodeHow on a large-scale codebase consisting of 26K C# projects downloaded from GitHub. The experimental results show that when the top 1 results are inspected, CodeHow achieves a precision score of 0.794 (i.e., 79.4% of the first returned results are relevant code snippets). The results also show that CodeHow outperforms conventional code search tools. Furthermore, we perform a controlled experiment and a survey of Microsoft developers. The results confirm the usefulness and effectiveness of CodeHow in programming practices.**

I. Introduction

Programming is sometimes hard. One of the challenges a programmer faces when writing new code is to find out how to implement a certain functionality (e.g., how to implement quick sort). The other challenge a programmer faces is the reuse of an Application Programming Interface (API, e.g., File.AppendText in .NET framework). A large-scale software framework, such as the .NET framework, could contain hundreds or even thousands of APIs. Programmers often do not remember exactly how a certain API method should be reused. In a survey conducted at Microsoft in 2009, 67.6% respondents mentioned that there are obstacles caused by inadequate or absent resources for learning APIs [1].

Over years a huge number of open source and industrial software systems have been developed. The source code of these systems is typically stored in source code repositories and can be treated as important reusable assets for developers. Previously written programs can help developers understand how others addressed the similar problems and can serve as a basis for writing new programs. Thus, there is a great demand for automated tools that can help developers search through a large codebase to find relevant code for a specific programming task.

Today's code search engines, e.g., Krugle [2], Ohloh [3], and Sourcerer [4], treat source code as plain texts and perform code search based on the text similarity between code snippets and a query, utilizing information retrieval models such as the standard Boolean model [5], the vector space model (VSM) [5], or the Apache Lucene model (which is essentially a variant of VSM)[1]. We evaluated the existing code search tools and found that the accuracy of these tools are often unsatisfactory. The desired code snippets are often not found in the returned results. As shown in our user study (Section VI), developers consider only 25.7% to 38.4% of the top 10 results returned by Ohloh useful.

Recently a number of techniques have been proposed to address the weakness of existing tools [6], [7], [8]. These techniques tackle the problem from the angle of query refinement. They expand a user query using semantic similar words [6], [7], or using a reformulation strategy [8]. It was observed that automatically expanding a query with inappropriate synonyms may produce even worse results than not expanding the query [9].

We believe that one major limitation of existing code search tools is the lack of query understanding. These tools often adopt conventional text similarity matching techniques (such as Boolean model or vector space model) to retrieve relevant code snippets. They do not consider query understanding and could therefore lead to inaccurate return results. For example, given a query "read file" and two code snippets A and B. Suppose the code snippet A contains many irrelevant APIs such as "Console.read()", "File.Exist()" and "File.AppendText()" but does not contain the relevant API "File.ReadLines()". Code snippet B contains only one occurrence of the relevant API "File.ReadLines()". Among the search results, A could be returned by the standard Boolean model because it contains all the query terms. A could even be ranked higher than B by the vector space model because of its higher term frequency. If we could know that the API "File.ReadLines()" is associated with

[1]http://lucene.apache.org/

978-1-5090-0025-8/15 $31.00 © 2015 IEEE
DOI 10.1109/ASE.2015.42

IEEE computer society

the query "read file", we could use this information to increase the ranking of snippet B and improve the query results. This example shows that through API understanding, more accurate code search could be achieved.

In this paper, we propose CodeHow, a code search approach that considers both API understanding and text similarity matching. We understand a query by identifying the APIs that the query may refer to. To do so, we enrich each API with its online documentation (e.g., the documentation at MSDN) and identify the APIs whose documentations match the query. Having identified the potential APIs for a query, we need to incorporate the API information into the code retrieval process. For this, we propose to apply the Extended Boolean model [10], which integrates the benefit of standard Boolean model and vector space model. We expand the user query with the identified APIs and apply the Extended Boolean model to retrieve the code snippets that match the expanded query.

We have implemented the backend of CodeHow as a Microsoft Azure cloud service. We evaluate CodeHow using real-world queries over a large-scale codebase consisting of 26K C# projects downloaded from GitHub. The evaluation results show that when the top 1 results are inspected, CodeHow achieves a precision score of 0.794 (i.e., 79.4% of the first returned results are relevant code snippets). These results are better than the results achieved by a conventional Lucene-based code search tool. We also perform a controlled experiment and a survey of Microsoft developers. The results confirm the usefulness and effectiveness of CodeHow in programming practices.

The contributions of this paper are as follows:

- We propose a code search technique that could understand the APIs a user query refers to and considers both text similarity and potential APIs.
- We propose to apply Extended Boolean model to incorporate the impact of both text similarity and potential APIs on code search.
- We have implemented CodeHow and deployed it as a scalable cloud service. Our experimental results show that CodeHow outperforms conventional code search tools.

The rest of this paper is organized as follows. We present the overall structure of CodeHow in Section II. We describe the API understanding component in Section III and the code retrieval component in Section IV. Our in-house experiment and user study are described in Section V and Section VI, respectively. We discuss our work and present threats to validity in Section VII. The related work is introduced in Section VIII. We conclude the paper in Section IX.

II. The Overall Structure of CodeHow

Figure 1 presents the overall structure of CodeHow. CodeHow constructs a codebase by collecting projects from open source repositories (e.g., Codeplex, Github) and an organization's local repositories. After collecting the projects, CodeHow performs preprocessing (tokenization, stemming, etc) on the source code and indexes the code at the method level using Elastic Search[2].

[2]https://www.elastic.co/

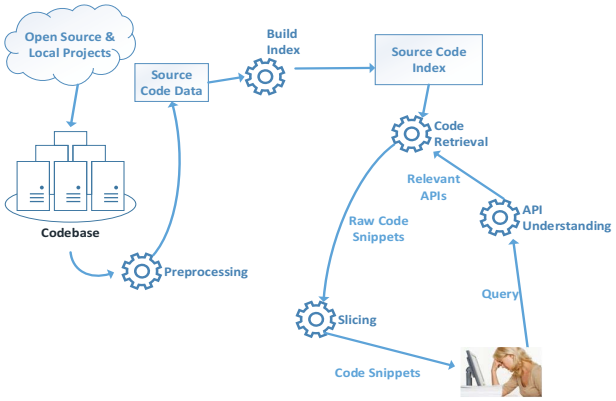

Fig. 1. The Overall Structure of CodeHow

Given a user query, CodeHow first feeds it to the API Understanding component (Section III) to figure out the potential APIs that are relevant to the query. The Retrieval component (Section IV) then expands the user query with the potential APIs and retrieves relevant code snippets from the codebase. A raw code snippet contains source code of a class method (function), which could be verbose and mixed with irrelevant code. In order to make the returned code snippets compact and easy to understand, CodeHow applies static slicing techniques to remove the irrelevant code, and presents to users the sliced code snippets.

The backend of CodeHow is deployed as a Microsoft Azure cloud service[3]. The Elastic Search engine is running on five Azure virtual machines (including 1 master node and 4 workers). The front-end of CodeHow is implemented as a Microsoft Visual Studio extension, which can help Visual Studio users search code during programming. Figure 2 gives a screenshot of CodeHow user interface.

The distinctive features of CodeHow are its API Understanding and Code Retrieval components, which are described in details in Section III and Section IV, respectively.

III. API Understanding

CodeHow tries to understand the potential APIs that the users would like to query. Figure 3 shows the outline of the proposed API understanding method. For an API library (such as Microsoft .NET framework), CodeHow first collects the description of each API from its online documentation. After obtaining the description of an API, CodeHow computes the similarity between the textual description and the query as well as the similarity between the API name and the query. It then combines the two similarity values for each API and returns the potentially relevant APIs that match the query. We elaborate the API understanding process in this section.

A. API Enrichment and Preprocessing

We enrich APIs by obtaining API descriptions from their online documents, including the full qualified API name, summary, and the remarks (full descriptions). As an example,

[3]http://azure.microsoft.com/en-us/

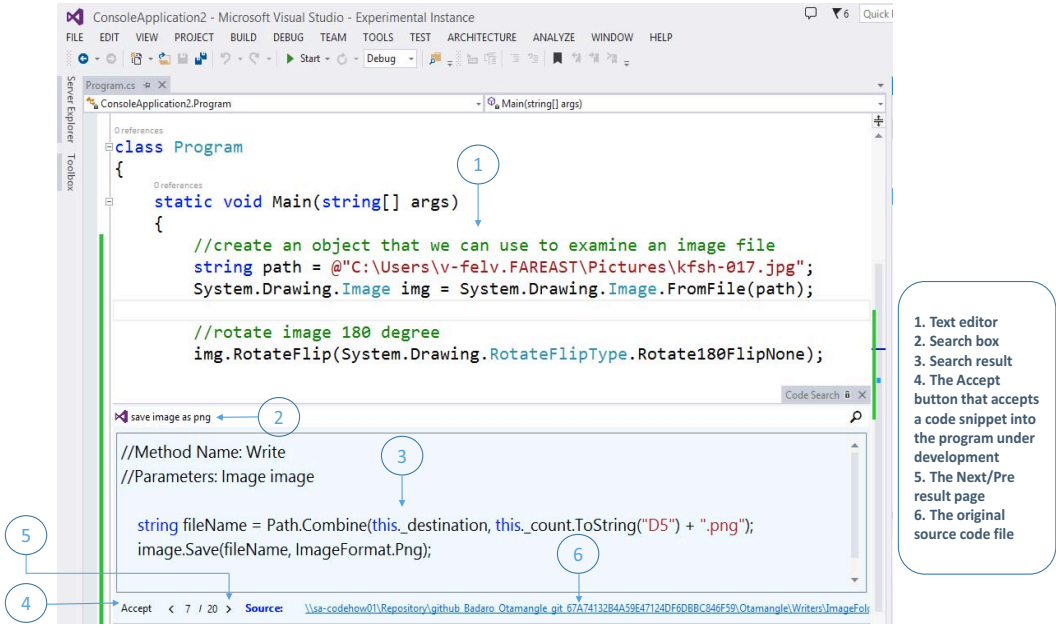

Fig. 2. A Screenshot of CodeHow User Interface

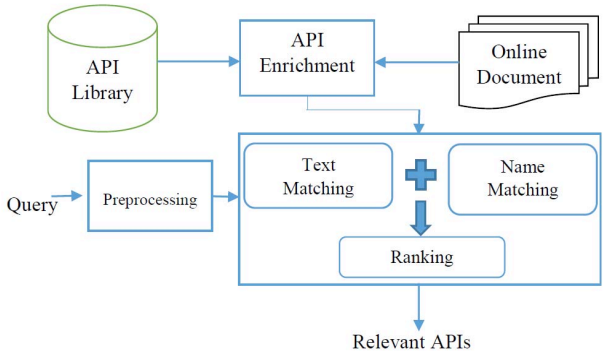

Fig. 3. The Framework of the API Understanding Component

TABLE I. AN EXAMPLE OF .NET API

Field	Text
Full Qualified Name	System.IO.File.ReadLines
Summary	Reads the lines of a file.
Remarks	The ReadLines and ReadAllLines methods differ as follows: When you use ReadLines, you can start enumerating the collection of strings before the whole collection is returned...

Table I shows the description of the .NET API File.ReadLines obtained from the online MSDN document[4]. We treat each API description as a document and use it for matching a user query.

For a free-text query and a description of an API, we perform three preprocessing steps: text normalization, stop word removal, and stemming. The goal is to break the text into terms that can be analyzed by an information retrieval technique. First, we perform text normalization, which involves the removal of punctuation marks and tokenization. Second, we remove stop words such as "on", "the", "are" and so on. Finally, we perform stemming, which reduces inflected or derived words into a common root-form. For example the word "reading" and "reads" are reduced to the root form "read". We use the standard Porter Stemmer to perform this stemming step [11].

[4]http://msdn.microsoft.com/en-us/library/dd383503(v=vs.110).aspx

B. Identifying Relevant APIs

We now describe how we identify relevant APIs for a user query. The process consists of two components: Text Matching component and Name Matching component.

In the Text Matching component, for each API api_i in a library, we compute the text similarity score $api_i^t.score$ between the query and the API's description using the standard Vector Space Model (VSM). Each document d is treated as a vector. Each value in the vector corresponds to the weight of a term t in d. The weight is calculated based on term frequency and inverse document frequency. The similarity is computed as cosine similarity. The top k APIs that have the highest textual similarity are returned as candidate results. We define the resultant APIs obtained from the calculation of the Text Matching component as API_{text}:

$$API_{text} = \{api_1^t, api_2^t, ..., api_k^t\}$$

In the Name Matching component, we compute the similarity between the user query and the APIs' Full Qualified Name (FQN). For each API api_i in a library, we calculate the similarity scores $api_i^n.score$ between the query and the API's FQN using VSM, and return the top k candidate APIs. We define the resultant APIs obtained from the calculation of the Name Matching component as API_{name}:

$$API_{name} = \{api_1^n, api_2^n, ..., api_k^n\}$$

We define the APIs appearing in both API_{text} and API_{name} as $API_{overlap}$, and define the APIs appearing

only in API_{text} or API_{name} as $API_{notOverlap}$. We rank the APIs appearing in $API_{overlap}$ higher than the APIs in $API_{notOverlap}$, because we assume the APIs appearing in both candidate lists are more relevant than the ones only appearing in one candidate list. We calculate the combined scores of the APIs as follows:

$$api_i.score = \begin{cases} api_i^t.score + api_i^n.score \\ (\text{if } api_i \in API_{overlap}) \\ \\ \frac{MinOverlapScore \times api_i^{t/n}.score}{maxNotOverlapScore + a} \\ (\text{if } api_i \notin API_{overlap}) \end{cases} \quad (1)$$

where $MinOverlapScore$ is the minimum score of all APIs in $API_{overlap}$, and $maxNotOverlapScore$ is the maximum score of all APIs in $API_{notOverlap}$. If api_i only appears in API_{text}, then $api_i^{t/n}.score$ equals to $api_i^t.score$. If api_i only appears in API_{name}, then $api_i^{t/n}.score$ equals to $api_i^n.score$. The parameter a is an adjustment factor to make sure that the score of $API_{overlap}$ is larger than that of $API_{notOverlap}$. In this study, we empirically set a to 0.1.

Equation (1) says that, if api_i appears in both API_{name} and API_{text} candidate lists, its score is the sum of the two scores. Otherwise, its score is calculated based the similarity score in corresponding candidate list. We compute the scores in above way to make sure that all APIs are ranked according to the following criteria:

1) $API_{overlap}$ ranks higher than $API_{notOverlap}$.
2) The API with higher similarity score ranks higher than the one with lower similarity score.

Finally, we rank all APIs according to their scores and obtain the top k potentially relevant APIs:

$$API_{relevant} = \{api_1, api_2, ..., api_k\}$$

Example: suppose for the query *how to read file line by line*, the Name Matching component identifies the following potentially relevant APIs (API_{text}) with similarity scores: {File.ReadLines=0.5, File.ReadAllText=0.4, File.ReadAllLines=0.4}. The Text Matching component identifies the following APIs (API_{name}) with similarity scores: {FileIO.TextFieldParser.ReadLine= 0.9, File.ReadLines=0.6, File.ReadAllLines=0.5}. The overlapping APIs ($API_{Overlap}$) are File.ReadLines and File.ReadAllLines. We compute their score as 0.5 + 0.6 = 1.1, and 0.4 + 0.5 = 0.9, respectively. The non-overlapping APIs ($API_{notOverlap}$) are ReadLinesFromFile.File and File.ReadAllText. Thus, we get $MinOverlapScore$ value 0.9 and $maxNotOverlapScore$ value 0.9. Thus the scores for FileIO.TextFieldParser.ReadLine and File.ReadAllText are 0.81 and 0.36, respectively. Finally, the rank of potentially relevant APIs ($API_{relevant}$) are as follows: 1) File.ReadLines (score=1.1); 2) File.ReadAllLines (score=0.9); 3) FileIO.TextFieldParser.ReadLine (score=0.81); 4) File.ReadAllText (score=0.36).

IV. Code Retrieval

Source code can be treated as texts. Conventional information retrieval techniques can be applied to search for code snippets that are relevant to a given user query based on the text similarity between the query and code snippets. Through API understanding, the APIs that are potentially relevant to a user query are identified. The API information can complement the conventional text-based code retrieval.

In our work, we propose an integrated retrieval method that considers both text similarity and API information. In this section, we describe the construction of queries and the retrieval of code snippets given the queries.

A. Query Expansion

A query Q_t containing *n* terms is defined as:

$$Q_t = (t_1, t_2, ..., t_n)$$

For a code snippet, we consider the following three features:

- Relevant APIs: The APIs that the code snippet contains.
- Method Body: the method body, which contains source code that implements a functionality.
- Method Name: the method's Full Qualified Name (FQN), which gives a brief summarization of the method's functionality.

We define the three code snippet features as follows:

$$F = (f_1, f_2, f_3)$$

where f_1 stands for "API", f_2 stands for "Method Body", f_3 stands for "Method Name". A query can thus be expressed in terms of $f_i : t_i$ where $t_i \in Q_t$ and $f_i \in F_t$. It means to search a field f_i that contains the term t_i.

We construct a Boolean query expression for retrieving code snippets that matches to the query in terms of text similarity:

$$q_{text} = (f_2 : t_1 \vee f_3 : t_1) \wedge (f_2 : t_2 \vee f_3 : t_2)... \\ \wedge (f_2 : t_n \vee f_3 : t_n)$$

This query expression searches for code snippets that contain the terms t_1, t_2, ... , t_n in fields f_2 (Method Body) and f_3 (Method Name).

After the API understanding phrase (Section III), we get k potentially relevant APIs $API_{relevant}$. In our work, we set k to 10. For each API api_i in $API_{relevant}$, we tokenize its name and get a keyword list A_i. We then construct Boolean query expressions for each API as follows:

$$q_{api_i} = f_1 : api_i \wedge (f_2 : t_1 \vee f_3 : t_1) \wedge (f_2 : t_2 \vee f_3 : t_2)... \\ \wedge (f_2 : t_k \vee f_3 : t_k)$$

where $api_i \in API_{relevant}$ and $t_k \in (Q_t - A_i)$. This query expression searches for code snippets that contain the potentially relevant API api_i in fields f_1 (API) as well as other query terms in fields f_2 (Method Body) and f_3 (Method Name). Note that we remove the terms that appear in A_i from the query Q_t. This is because the impact of these terms have already been considered in the api_i item.

Each query expression defined above can be used to search for the code snippets. A code snippet may be thus retrieved by

more than one query expressions. We consider a code snippet that can be retrieved by multiple query expressions more important. In our approach we combine the query expressions and obtain an expanded query for retrieving code snippets:

$$q_{expand} = (q_{api_1}, q_{api_2}, ..., q_{api_k}, q_{text})$$

Example: For the following query: *how to save an image in png format?*

The query terms $Q_t = (save, image, png, format)$. The potentially relevant APIs $API_{relevant}$ we obtain from the API Understanding component (Section III) are $System.Drawing.Image.Save$ and $System.Drawing.Imaging.ImageFormat.Png$. The associated query expressions are as follows:

$$\begin{aligned} q_{api_1} =& (f_1 : System.Drawing.Image.Save) \wedge \\ & (f_2 : png \vee f_3 : png) \wedge (f_2 : format \vee f_3 : format) \\ q_{api_2} =& (f_1 : System.Drawing.Imaging.ImageFormat.Png) \wedge \\ & (f_2 : save \vee f_3 : save) \\ q_{text} =& (f_2 : save \vee f_3 : save) \wedge (f_2 : image \vee f_3 : image) \wedge \\ & (f_2 : png \vee f_3 : png) \wedge (f_2 : format \vee f_3 : format) \end{aligned}$$

The resulting expanded query is therefore: $q_{expand} = (q_{api_1}, q_{api_2}, q_{text})$.

B. Applying Extended Boolean Model to Code Search

1) Extended Boolean Model: To retrieve relevant code snippets based on the queries, we adopt the Extended Boolean model [10], which is an intermediate between the standard Boolean model and the vector space model. In a standard Boolean model, when two query terms are related by an AND connective, both terms must be present in order to retrieve a document. When an OR connective is used, at least one of the query term must be present. Therefore, a standard Boolean model is often too strict (no results returned), or too general (too many results returned). Furthermore it does not consider the term weight and the ranking of the results. A vector space model eliminates many disadvantages of the standard Boolean model by weighting both query and document terms and by computing the similarity between query and documents. VSM has been implemented by many text search engines such as Apache Lucene. However, VSM has limitation too. It does not support structural queries consisting of complex AND and OR relations. Furthermore, VSM may lead to inaccurate results in the context of code search due to the differences in term frequencies. For example, given a two-term query "A B", VSM may prefer a code snippet containing A frequently and B less frequently, over a code snippet that contains both A and B, which both appear less frequently. However, the term A could be irrelevant to the code search.

An Extended Boolean model [10] combines the characteristics of the vector space model and Boolean model, and ranks the similarity between queries and documents. In the Extended Boolean model, a document d is represented as a vector. Given query expressions $Q = (q_1, q_2, ...q_t)$, a generalized disjunctive and a generalized conjunctive query are defined as follows:

$$q_{or} = q_1 \vee^p q_2 \vee^p ... \vee^p q_t$$

$$q_{and} = q_1 \wedge^p q_2 \wedge^p ... \wedge^p q_t$$

In an Extended Boolean model, the similarity between a Boolean query expression and d is computed using p-norm (a general form of normalized Euclidean distance):

$$sim(q_{or}, d) = \left(\frac{w_{t_1,q}^p w_{t_1,d}^p + w_{t_2,q}^p w_{t_2,d}^p + ... + w_{t_n,q}^p w_{t_n,d}^p}{w_{t_1,q}^p + w_{t_2,q}^p + ... + w_{t_n,q}^p}\right)^{\frac{1}{p}} \quad (2)$$

$$sim(q_{and}, d) = 1 - \left(\frac{w_{t_1,q}^p (1 - w_{t_1,d})^p + ... + w_{t_n,q}^p (1 - w_{t_n,d})^p}{w_{t_1,q}^p + w_{t_2,q}^p + ... + w_{t_n,q}^p}\right)^{\frac{1}{p}} \quad (3)$$

where $w_{t_i,q}$ represents the weight of query term q_i, $w_{t_i,d}$ represents the weight of query term q_i in document d. The parameter p is the parameter used in p-norm. We set the value of p empirically (with a default value of 3). More details on these equations can be obtained from [10].

2) Retrieval of Code Snippets based on Expanded Query: In our approach, the weight of a term t in an expanded query with respect to a code snippet d is defined as follows:

$$w_{t,d} = \begin{cases} api.score(\text{if term } t \text{ is an API}) \\ 0.5 + (1 - 0.5) \times f_{t,d} \times \frac{Idf_t}{max_i Idf_i} \\ (\text{if term } t \text{ is not an API}) \end{cases} \quad (4)$$

In Equation (4), if the term is an API (such as System.Drawing.Image.Save), its weight is the API score obtained from Equation (1). If the term is not an API (such as png), its weight is measured by its normalized term frequency. $f_{t,d}$ is the frequency of the term t in code snippet d, and Idf_t is the inverse document frequency of the term t, The constant 0.5 is used for balancing the term weight.

In our approach, matches in the fields f_1 (API) or f_3 (Method Name) are considered more important than matches in f_2 (Method Body). Therefore, we set the weight of term $w_{t,q}$ in an expanded query q as follows: for the query term related to $(f_2 : t)$, its weight is set to 1. For the term related to $(f_1 : API)$ or $(f_3 : t)$, its weight is set to 1.5.

The similarity between the expanded query q_{expand} and a document d is defined as follows:

$$sim(q_{expand}, d) = \sum_{i=1}^{k} sim(q_{api_i}, d) + sim(q_{text}, d) \quad (5)$$

where $sim(q_{api_i}, d)$ is the similarity between an API query expression and the code snippet d. $sim(q_{text}, d)$ is the similarity between the text query expression and the code snippet d. These similarity values are computed according to Equations (2) and (3) defined by the Extended Boolean model.

V. IN-HOUSE EXPERIMENT

To evaluate the effectiveness of CodeHow in retrieving relevant code snippets, we perform an in-house evaluation. In this section, we present our experimental setting, evaluation metrics, and experimental results.

A. Experimental Setting

The codebase we use in our experiments consists of 26K C# projects downloaded from Github[5] (an open source project repository). The total size of these projects is around 629

[5]https://github.com/explore

TABLE II. THE LIST OF QUERIES USED IN IN-HOUSE EXPERIMENT

ID	Query
1	copy paste data from clipboard
2	open url in html browser
3	track mouse hover
4	highlight text range in editor
5	convert utc time to local time
6	converting String to DateTime
7	get current date and time
8	get file name without extension
9	how can I decode HTML characters
10	how can I download HTML source
11	how do I round a decimal value to 2 decimal places
12	how to change RGB color to HSV
13	how to convert an IPv4 address into a integer
14	how to delete all files and folders in a directory
15	how to execute a sql select
16	how to generate random int number
17	how to get Color from Hexadecimal color code
18	how to get temporary folder for current user
19	if a folder does not exist create it
20	ping a hostname on the network
21	Process.start: how to get the output
22	sending email through Gmail
23	append string to file
24	calculate md5 checksum
25	how to Deserialize XML document
26	how to get mac address
27	how to play a sound
28	how to save image in png format
29	read file line by line
30	remove cookie
31	verify folder exists
32	how to reverse a string
33	quick sort
34	how to split string into words

GB, containing about 8.3 million C# source code files, and 11.4 million methods. The code search workload (including indexing and retrieving) is performed by the Elastic Search engine running on Microsoft Azure. The client side is an extension of Microsoft Visual Studio Ultimate 2013.

We have used 34 real-world queries in the in-house experiment, as shown in Table II. Among them, 4 queries (1-4) come from previous study[6] [12], 18 queries (5-22) are widely viewed queries collected from the StackOverflow website[7]. The rest of the queries (23-34) are collected from the actual Microsoft Bing search logs[8].

B. Research Questions

The objective of the in-house experiment is to investigate the overall effectiveness of our approach. We also want to investigate the benefit of distinctive features of our approach. We have identified the following research questions:

RQ1: How effective is CodeHow?

This RQ evaluates the effectiveness of CodeHow in retrieving relevant code snippets based on user queries. To answer this question, we run CodeHow using the queries specified in Table II. Two authors of this paper manually inspect the top 20 results returned from each query to judge whether they are relevant or not. Only the results receiving relevant feedback from both authors are labeled as relevant.

[6]Most of queries in [12] come from Eclipse FAQ website and they are Java programming language specific. We filter out those Java specific queries and obtain 4 language insensitive queries.

[7]http://stackoverflow.com/

[8]http://www.bing.com/

In our experiments, we also compare our approach with a conventional Lucene-based code search approach. Apache Lucene implements a variant of VSM and is behind many existing code search tools such as Sourcerer [4]. Sourcerer also incorporates several heuristics to rank the code snippets, including code-as-text (text similarity), FQN of entities, and code popularity. In our implementation of the Lucene-based code search tool, we consider both text similarity and FQN of the methods. We do not include code popularity (computed using PageRank) in our implementation as adding PageRank does not significantly improve the code search accuracy [4]. No API understanding nor Extended Boolean model is used in the Lucene implementation of code search. Therefore, the accuracy of our Lucene-based code search tool should be similar to what Sourcerer could achieve.

RQ2: Is the proposed API understanding method effective?

One distinctive feature of our approach versus existing code search approaches is the API Understanding component that is described in Section III. It tries to understand the potential APIs that are related to the query before performing code retrieval. Answers to this research question help us evaluate whether this feature is useful for code search or not. To answer this question, we compare two versions of CodeHow, one with query understanding and the other without query understanding (referred to as $CodeHow_{noQU}$). In the implementation of $CodeHow_{noQU}$, we omit the API understanding component and feed the user query to the Retrieval component (described in Section IV) directly. That is, we do not identify any potentially relevant APIs. We consider only the impact of text similarity on code search.

RQ3: Is the proposed Extended Boolean model effective?

Another distinctive feature of our approach versus existing code search approaches is the Extended Boolean model used in code retrieval (Section IV). Answers to this research question will shed light on whether this feature is useful for code search or not. To answer this question, we implement a variant of CodeHow (referred to as CodeHow$_{noEB}$), which uses Apache Lucene to retrieve code snippets based on an expanded query. We compare CodeHow (using Extended Boolean model) and CodeHow$_{noEB}$ (using Lucene). The API Understanding component remains the same for both implementations.

C. Evaluation Metrics

To evaluate the effectiveness of CodeHow, we make use of the $Precision@k$ metric:

$$Precision@k = \frac{1}{|Q|}\sum_{i=1}^{|Q|}\frac{|relevant_{i,k}|}{k}$$

where $relevant_{i,k}$ represents the relevant code snippets for query i in the top k returned results, Q is a set of queries. $Precision@k$ takes an average on all queries whose relevant answers could be found by inspecting the top k (k = 1, 5, 10, 20) of the returned code snippets. A better code search tool should allow developers to discover the needed code by examining fewer returned results. Thus, the higher the metric values, the better the code search performance.

We also make use of Mean Reciprocal Rank (MRR), which is a statistic for evaluating a process that produces a list of

TABLE III. THE COMPARISON BETWEEN CODEHOW AND LUCENE-BASED CODE SEARCH

	CodeHow	Lucene-based
Precision@1	0.794	0.618
Precision@5	0.823	0.476
Precision@10	0.776	0.435
Precision@20	0.706	0.372
MRR	0.867	0.704

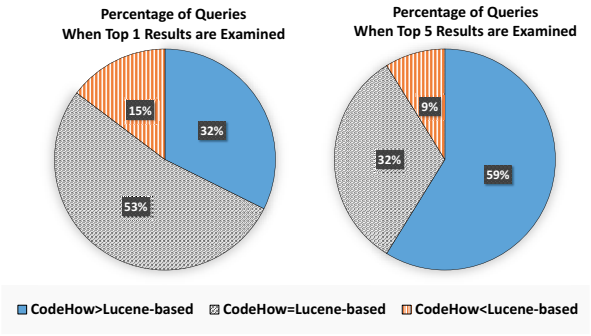

Fig. 4. The Percentage of Queries that CodeHow Performs Better/Worse than Lucene-based Code Search

possible responses to a query [13]. The reciprocal rank of a query is the multiplicative inverse of the rank of the first relevant answer. The mean reciprocal rank is the average of the reciprocal ranks of results of a set of queries Q and could be calculated as follows:

$$MRR = \frac{1}{|Q|}\sum_{i=1}^{|Q|}\frac{1}{rank_i}$$

The value of MRR is between 0 and 1. The higher the MRR value, the better the code search performance.

D. Experimental Results

RQ1: The Overall Effectiveness of CodeHow

We evaluate CodeHow using the queries specified in Table II. We also compare our approach with a conventional Lucene-based code search tool. Table III presents the overall results. When the top 1 results are inspected, CodeHow achieves a precision score of 0.794, which means that 79.4% of the first returned results are relevant code snippets. When the top 5 results are inspected, CodeHow achieves a precision score of 0.823. These results are considered satisfactory. Note that some queries in Table II are associated with explicit .NET APIs (e.g., the query *how to save image in png format*), while some queries are not (e.g., the query *quick sort*). CodeHow can effectively handle both of these queries.

Table III shows that the Lucene-based code search tool achieves a score of 0.618 when the top 1 results are inspected. CodeHow achieves 28.5%, 72.8%, 78.3%, and 89.7% improvements in terms of Precision@1, Precision@5, Precision@10, and Precision@20, respectively. In terms of MRR, CodeHow obtains a score of 0.867, which also outperforms the Lucene-based code search tool (0.704) by 23.2%.

Figure 4 shows the percentage of queries that CodeHow performs better/worse than the Lucene-based code search tool.

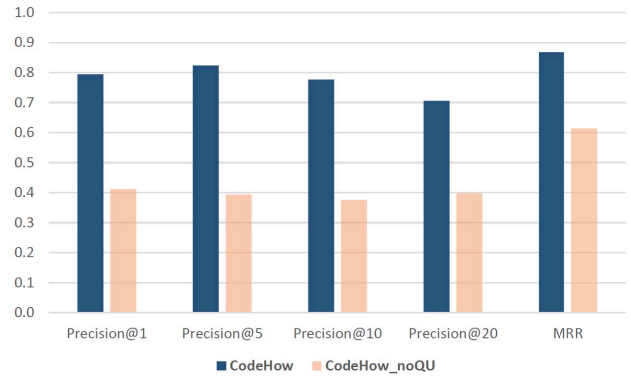

Fig. 5. The Comparison between CodeHow and CodeHow$_{noQU}$

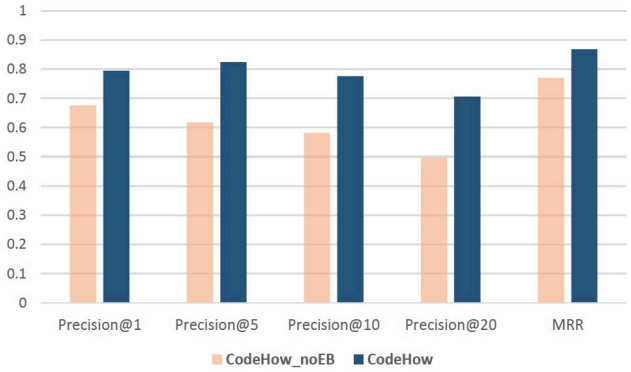

Fig. 6. The Comparison between CodeHow and CodeHow$_{noEB}$

We can see that when the top 1 returned results are examined, CodeHow wins in 32% of the queries and loses in 15% of the queries. In terms of the top 5 results, CodeHow outperforms the Lucene-based tool in 59% queries and loses in only 9% of the queries. We also perform a Wilcoxon signed-rank test [14]. The p-values are all less than 0.05, when the top 5, 10, and 20 returned results are examined. The results confirm that the improvement achieved by CodeHow is statistically significant.

RQ2: The Effectiveness of API Understanding

To answer this question, we compare two versions of CodeHow: CodeHow and CodeHow$_{noQU}$. We present the overall comparison results in Figure 5. CodeHow$_{noQU}$ achieves precision scores of 0.412, 0.394, 0.376, and 0.397 when the top 1, 5, 10, 20 results are inspected, respectively. CodeHow achieves 79.4%, 82.3%, 77.6%, and 70.6% improvement in terms of Precision@1, Precision@5, Precision@10, and Precision@20, respectively. In terms of MRR, CodeHow achieves a 41.3% improvement compared with CodeHow$_{noQU}$. We also perform a Wilcoxon signed-rank test and the result indicates that the improvement achieved by API understanding is statistically significant.

RQ3: The Effectiveness of the Extended Boolean Model

To answer this question, we compare the complete implementation of CodeHow and CodeHow$_{noEB}$. We present the results in Figure 6. CodeHow$_{noEB}$ achieves precision scores of 0.676, 0.618, 0.582, and 0.496 when the top 1, 5,

10, 20 returned results are inspected, respectively. CodeHow outperforms CodeHow$_{noEB}$ in terms of these metrics. In terms of MRR, CodeHow achieves a 12.7% improvement compared with CodeHow$_{noEB}$. A Wilcoxon signed-rank test confirms that the improvement achieved by the Extended Boolean model is statistically significant.

VI. User Study

A. A Controlled Experiment

To evaluate the effectiveness of code search tools in practice, we design a user study involving 20 Microsoft developers and interns. The developers are software development engineers (SDEs), whose programming experiences ranging from 3 to 17 years. The interns are undergraduate and postgraduate students, who are majored in computer science or information technology.

We first design three programming tasks and each participant is required to complete these three tasks using two code search tools: CodeHow and Ohloh [9]. Ohloh is chosen as it is one of the most popular commercial code search engines. The three tasks are as follows:

- Task 1: Sending emails - write a program to read a list of email addresses from a text file, and then send an email with an attachment file to all the email addresses.
- Task 2: Converting text files to XML documents - write a program to transform structured customer information recorded in a text file to an xml document.
- Task 3: Image format conversion - write a program to read an image in JPEG format, rotate it 180, and then convert it to PNG format.

Our goal is to evaluate the accuracy of two code search tools in retrieving code snippets based on user queries. During the experiment, participants entered a series of queries that described the given tasks, and searched for the code snippets that are needed to complete the tasks, using both code search tools. We recorded the number of queries the participants used, as well as the code snippets they examined for completing the tasks. We also asked them to examine the top 10 returned results for each query, and mark which of them are helpful for completing the tasks.

Our results show that the participants entered in total 95 queries to each code search tool during the experiment. Table IV shows the percentage of relevant code snippets the participants marked for each code search tool. CodeHow returns around 10%-20% more relevant results than Ohloh for all the three tasks. CodeHow also outperforms Ohloh in terms of MRR, which measures the rank of the first relevant returned result.

TABLE IV. The user study results

	Percentage of Useful Results		MRR	
	Ohloh	**CodeHow**	**Ohloh**	**CodeHow**
Task 1	37.9%	57.9%	0.734	0.834
Task 2	25.7%	36.5%	0.401	0.592
Task 3	38.4%	54.5%	0.577	0.782

B. A User Survey

In March 2015, we demonstrated CodeHow in a company-wide technology exhibition held in Microsoft Redmond campus. We conducted a survey of 45 Microsoft developers who tried our tool in the event. In the survey, we asked them if CodeHow is helpful for their programming tasks, by giving a score on a five-point Likert scale (strongly agree is 5 and strongly disagree is 1). For the 45 participants, the average Likert score was 4.15 (with standard deviation of 0.83). 40 (88.89%) participants stated that the search results returned by CodeHow are mostly or partially correct.

In the survey, we asked the participants in which scenarios they think CodeHow is more effective (helping figure out how to use an API, or helping figure out how a functionality can be implemented, or none of them). 26 out 45 (57.78%) participants marked that CodeHow is more effective in helping with API usage, and the rest of the participants considered that CodeHow is more effective in helping with a functionality. All except 2 participants (95.56%) expressed that they would like to search and reuse open source code such as those in GitHub (when licenses are in compliance).

Overall, the user study confirms the usefulness of CodeHow in programming practice.

VII. Discussions

A. The Incorrect Return Results

Although CodeHow is effective, it still cannot locate relevant code snippets for all the queries. One unsuccessful query example is *Convert utc time to local time*. We find that some returned results are actually about *Convert local time to utc time*. Similar examples include *How to change RGB color to HSV*. This is because our approach cannot distinguish semantic meanings of different orders of words.

Another query example that leads to incorrect results is *How to get Color from Hexadecimal color code*. Some code snippets contain color code strings such as "#FFDFD991". However, our approach cannot recognize these strings. If there are no identifiers or comments describing the code, CodeHow would miss these code snippets. The same problem also appears when searching for SQL, LINQ and regular expressions.

The above examples show the importance of understanding the semantic meanings of query and code. Currently CodeHow can only understand the relevant APIs a query implies. In the future, we plan to perform deeper natural language analysis of query and code, aiming for achieving better understanding of query and code and more accurate search results.

B. The Usefulness of Code Search Tools

Through our experiments, especially the user study, we have obtained the following insights. Although these observations are obtained from our experience on CodeHow, they may be applicable to other code search tools and can be used to guide the future development of these tools.

[9] http://code.ohloh.net/

- Targeted problems: Code search can be considered as an opportunistic problem solving, where developers need to find missing information for completing software development tasks [15]. Gallardo-Valencia and Sim [16] further proposed three types of opportunistic problems that code search could help resolve: $Remembering$ (developers knew exactly what they are looking for and only wanted to remember syntax details or find facts), $Clarification$ (developers had a high-level understanding of what they want to implement, but did not know precisely how to do it), and $Learning$ (developers wanted to acquire new concepts). Through our experiments, we find that code search tools such as CodeHow is effective in solving these three problems. We also find that experienced developers can better use our tool as they could enter queries that are directly related to their problems, while inexperienced developers may not realize some relevant code snippets even they contain APIs that are useful for their tasks. One user study participant also pointed out that CodeHow is more useful for querying a local codebase, where a general search engine cannot reach.
- Code comprehension: we observe that developers, especially inexperienced developers, often have difficulties in comprehending the returned code snippets. This becomes an obstacle for using code search tools such as CodeHow. Developers may turn to a general search engine such as Bing or Google to study the API usage from online materials. This problem could be partially mitigated by a better code summarization method [17], [18], which can automatically summarize necessary code fragments associated with the query. Designing a better User Interface (e.g, highlighting matching code, linking API to online documentation/discussion forum, etc) could also increase the usability of the code search tools in practice.
- Reuse granularity: we observe that the code search tools are more effective when used at a fine-granularity level. Large code snippets that contain many lines of code may hinder program comprehension and are more likely to cause the "architecture mismatch" problem, which occurs when the code snippet to be reused requires a different software architecture or a different object-oriented framework.

C. Threats to Validity

We have identified the following threats to validity:

Threats to internal validity: our empirical study involves human subjects. The limited number and the programming capabilities of the human subjects may bias the results. The process of determining the relevance of a code snippet could be also subjective. In the future, we plan to conduct experiments and user studies involving more subjects, API methods, and programming tasks to further reduce this threat.

Threats to external validity: we have used 34 queries in our in-house experiment. Although these queries are real-world queries collected from the StackOverflow website, Bing search logs, and the related work, admittedly they do not cover all types of queries that a developer may ask. Also, although our codebase consists of 26K projects and 11.4 million methods, it is just a tiny sample of all available source code. In the future, we plan to reduce the threats to external threats by investigating more queries over a much larger codebase.

VIII. Related Work

A. Code Search

Currently, many code search approaches have been proposed to help users find relevant code. Ohloh [3], Krugle [2], and Sando [19] are code search engines that can return code snippets containing the keywords (or Regular Expressions) specified in a query. Strathcona [20] is a code snippet recommender, which locates a set of relevant code snippets by matching the structure of the code under development with the code snippets in codebase. Sourcerer [4] is a Lucene-based infrastructure for large-scale code search. Bajracharya et al. [12] performed code search using Structural Semantic Indexing, which associates words to source code entities based on similarities of API usage. S6 [21] is a test-driven code search engine that can map high-level queries into relevant code fragments through a set of program transformations. SNIFF [22] annotates a code snippet with the documentation of Java APIs the code snippet contains, and then performs free-text queries over the annotated source code. Unlike the above work, our approach performs API understanding before retrieving relevant code.

Web search engines, such as Bing or Google, could also help developers reuse APIs. However, Stylos and Myers [23] observed many problems and inefficiencies in using general Web search engines for code reuse, because these search engines are not designed to specifically support programming tasks. Tools such as Mica [23], Blueprint [24] and Assisme [25] use general search engines to search for results and automatically extract relevant code snippets from the returned results. eXoaDocs [26] facilitates API reuse by embedding API documents with code examples mined from the Web.

Our approach supports free-text queries. There are also many approaches that treat an object, a function, or a partial program as a query, and search for matching code snippets [20], [27], [28], [29], [30]. Furthermore, our work focuses on the selection of code snippets that answer user's queries. There are many tools that focus on recommending a group of connected functions. For example, Portfolio visualizes relevant functions and their usages and uses PageRank to mine the relationship of functions [31].

Exemplar [32] is a tool that finds relevant applications from a large archive of applications. Similar to our work, Exemplar also performs search over a large-scale codebase based on user queries and utilizes API document information. However Exemplar returns executable projects while our tool returns code snippets.

Our work is also related to concept location, which is the process of linking textual descriptions to corresponding source code files. Concept location may also be referred to as feature location [33], [34], bug localization [35], or trace recovery [36] in different contexts. For example, Antoniol et al. [37] applied both a probabilistic and a vector space model to recover links

between source code units and free text documents (such as manual pages or requirements). Marcus and Maletic [38] used Latent Semantic Indexing (LSI) for the similar purpose. The underlying assumption is that programmers use meaningful words for code units, and these words capture application-specific knowledge. Therefore, text retrieval techniques can be used to link concepts expressed in natural language with code units.

B. Query Refinement

Recently, several methods have been proposed to improve the effectiveness of code search or concept location via query refinement. For example, Gay et al. [39] improved the effectiveness of concept location by incorporating user feedback using the Rocchio algorithm. Haiduc et al. [8] proposed Refoqus, a tool that can predict the quality of a query and automatically recommend a query reformulation strategy. Wang et al. [40] proposed an active code search approach, which incorporates user feedback to refine the query. Dietrich et al. [41] proposed an approach that utilizes feedback captured from a validated set of queries and traceability links to improve the efficacy of future queries. Many methods have also been proposed to expand a user query by retrieving the semantic similar words from websites [42] or software [9], [6], [7]. Hill et al. [43] proposed to automatically extract natural language phrases from source code identifiers and categorize them into a hierarchy, which helps developers recognize alternative words for query reformulation. It was observed that automatically expanding a query with inappropriate synonyms may produce even worse results than not expanding the query [9]. Different from the above work, our work identifies potentially relevant APIs that a user query refers to, expands the query with the APIs, and utilizes the Extended Boolean model to handle the expanded query. It is also interesting to explore the synergy between our work and the existing query refinement work.

IX. Conclusion

In this paper, we propose CodeHow, a code search technique that applies Extended Boolean model to retrieve code snippets that match users' free-text queries. CodeHow could recognize the potential APIs a query refers to and incorporates the API information to improve the accuracy of the search results. We have implemented the backend of CodeHow as a Azure cloud service and the front-end as a Visual Studio extension. We have conducted experiments over a codebase containing 26K C# projects. Our evaluation results show that CodeHow is effective and outperforms conventional Lucene-based code search. We have also performed a controlled experiment and a user survey involving Microsoft developers and interns. The results confirm the usefulness of the tool.

In the future, we plan to address the issues discussed in Section VII. Currently, the CodeHow tool only supports code search for C# programs. We are planning to support more programming languages such as Java. We will also work on methods for synthesizing sample usage code from the code search results.

Acknowledgment

We thank all Microsoft developers and interns who participated in our user study and provided us helpful comments. We thank Yi Wei at Microsoft Research Cambridge for providing us the query logs of Bing code search and valuable suggestions. We thank our colleagues at Microsoft product teams, especially Chandrashekhar Kaushik, Sumit Saluja, Anuj Jain, Shabbar Husain, and Jialiang Ge (Scott) for the amazing effort on technology transfer. This work was performed when the first and fourth authors were interns at Microsoft Research. We also thank other intern students who helped with the implementation and maintenance of CodeHow, especially Wenhao Song, Sheng Tian, Peiyong Lin, Senlan Yao, and Qing Ren.

References

[1] M. P. Robillard, "What makes APIs hard to learn? answers from developers," *IEEE Software*, vol. 26, no. 6, pp. 27–34, 2009.

[2] Krugle code search. [Online]. Available: http://www.krugle.com/

[3] Ohloh code search. [Online]. Available: https://code.ohloh.net/

[4] E. Linstead, S. Bajracharya, T. Ngo, P. Rigor, C. Lopes, and P. Baldi, "Sourcerer: mining and searching internet-scale software repositories," *Data Mining and Knowledge Discovery*, vol. 18, pp. 300–336, 2009.

[5] R. Baeza-Yates and B. Ribeiro-Neto, *Modern Information Retrieval: The Concepts and Technology behind Search*. Addison-Wesley, 2011.

[6] E. Hill, L. L. Pollock, and K. Vijay-Shanker, "Improving source code search with natural language phrasal representations of method signatures," in *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE '11)*, 2011, pp. 524–527.

[7] J. Yang and L. Tan, "Inferring semantically related words from software context," in *9th IEEE Working Conference of Mining Software Repositories (MSR'12)*, 2012, pp. 161–170.

[8] 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 2013 International Conference on Software Engineering (ICSE '13)*, 2013, pp. 842–851.

[9] G. Sridhara, E. Hill, L. L. Pollock, and K. Vijay-Shanker, "Identifying word relations in software: A comparative study of semantic similarity tools," in *Proceedings of the 16th IEEE International Conference on Program Comprehension (ICPC '08)*, 2008, pp. 123–132.

[10] G. Salton, E. A. Fox, and H. Wu, "Extended boolean information retrieval," *Commun. ACM*, vol. 26, pp. 1022–1036, 1983.

[11] M. F. Porter, "An algorithm for suffix stripping," *Program*, vol. 14, pp. 130–137, 1980.

[12] S. K. Bajracharya, J. Ossher, and C. V. Lopes, "Leveraging usage similarity for effective retrieval of examples in code repositories," in *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'10)*, 2010, pp. 157–166.

[13] 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 (ICSE '10)*, 2010, pp. 475–484.

[14] F. Wilcoxon, "Individual comparisons by ranking methods," *Biometrics Bulletin*, vol. 1, no. 6, pp. 80–83, 1945.

[15] P. N. Robillard, "Opportunistic problem solving in software engineering," *IEEE Software*, vol. 22, no. 6, pp. 60–67, Nov. 2005.

[16] S. E. Sim and R. Gallardo-Valencia, *Finding Source Code on the Web for Remix and Reuse*. Springer, 2013.

[17] A. T. T. Ying and M. P. Robillard, "Code fragment summarization." in *Proc. ESEC/FSE'13*, 2013, pp. 655–658.

[18] S. Haiduc, J. Aponte, and A. Marcus, "Supporting program comprehension with source code summarization." in *Proc. 32nd International Conference on Software Engineering (ICSE'10)*, 2010, pp. 223–226.

[19] D. Shepherd, K. Damevski, B. Ropski, and T. Fritz, "Sando: An extensible local code search framework," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE '12)*, 2012, p. 15.

[20] R. Holmes, R. J. Walker, and G. C. Murphy, "Strathcona example recommendation tool," in *Proceedings of the 10th European Software*

Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-13), 2005, pp. 237–240.

[21] S. P. Reiss, "Semantics-based code search," in *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*, 2009, pp. 243–253.

[22] S. Chatterjee, S. Juvekar, and K. Sen, "Sniff: A search engine for Java using free-form queries," in *Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering (FASE '09)*, 2009, pp. 385–400.

[23] J. Stylos and B. A. Myers, "Mica: A web-search tool for finding API components and examples," in *Proceedings of the Visual Languages and Human-Centric Computing (VLHCC '06)*, 2006, pp. 195–202.

[24] J. Brandt, M. Dontcheva, M. Weskamp, and S. R. Klemmer, "Example-centric programming: Integrating web search into the development environment," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '10)*, 2010, pp. 513–522.

[25] R. Hoffmann, J. Fogarty, and D. S. Weld, "Assieme: Finding and leveraging implicit references in a web search interface for programmers," in *Proceedings of the 20th Annual ACM Symposium on User Interface Software and Technology (UIST '07)*, 2007, pp. 13–22.

[26] J. Kim, S. Lee, S. Hwang, and S. Kim, "Towards an intelligent code search engine," in *Proc. of the Twenty-Fourth AAAI Conference on Artificial Intelligence (AAAI-10)*, 2010, pp. 1358–1363.

[27] N. Sahavechaphan and K. Claypool, "Xsnippet: Mining for sample code," in *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA'06)*, 2006, pp. 413–430.

[28] S. Thummalapenta and T. Xie, "Parseweb: A programmer assistant for reusing open source code on the web," in *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE '07)*, 2007, pp. 204–213.

[29] M. Bruch, M. Monperrus, and M. Mezini, "Learning from examples to improve code completion systems," in *Proceedings of the Joint Meeting of the 7th European Software Engineering Conference and 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, August 2009, pp. 213–222.

[30] A. T. Nguyen, H. A. Nguyen, T. T. Nguyen, and T. N. Nguyen, "Grapacc: A graph-based pattern-oriented, context-sensitive code completion tool," in *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*, 2012, pp. 1407–1410.

[31] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu, "Portfolio: Finding relevant functions and their usage," in *Proceedings of the 33rd International Conference on Software Engineering (ICSE '11)*, 2011, pp. 111–120.

[32] C. McMillan, M. Grechanik, D. Poshyvanyk, C. Fu, and Q. Xie, "Exemplar: A source code search engine for finding highly relevant applications," *IEEE Transactions on Software Engineering*, vol. 38, no. 5, pp. 1069–1087, 2012.

[33] T. Biggerstaff, B. Mitbander, and D. Webster, "The concept assignment problem in program understanding," in *Proceedings of the 15th international conference on Software Engineering (ICSE '93)*, 1993, pp. 482–498.

[34] N. Wilde and M. C. Scully, "Software reconnaissance: Mapping program features to code," *Journal of Software Maintenance: Research and Practice*, vol. 7, pp. 49–62, 1995.

[35] J. Zhou, H. Zhang, and D. Lo, "Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports," in *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*, 2012, pp. 14–24.

[36] E. Keenan, A. Czauderna, G. Leach, J. Cleland-Huang, Y. Shin, E. Moritz, M. Gethers, D. Poshyvanyk, J. I. Maletic, J. H. Hayes, A. Dekhtyar, D. Manukian, S. Hossein, and D. Hearn, "Tracelab: An experimental workbench for equipping researchers to innovate, synthesize, and comparatively evaluate traceability solutions," in *Proc. ICSE'12*, 2012, pp. 1375–1378.

[37] G. Antoniol, G. Canfora, G. Casazza, A. D. Lucia, and E. Merlo, "Recovering traceability links between code and documentation," *IEEE Transactions on Software Engineering (TSE '02)*, pp. 970–983, 2002.

[38] A. Marcus and J. I. Maletic, "Recovering documentation-to-source-code traceability links using latent semantic indexing," in *Proceedings of the 25th International Conference on Software Engineering (ICSE '03)*, 2003, pp. 125–135.

[39] G. Gay, S. Haiduc, A. Marcus, and T. Menzies, "On the use of relevance feedback in IR-based concept location," in *Proceedings of the 25th IEEE International Conference on Software Maintenance (ICSM '09)*, 2009, pp. 351–360.

[40] S. Wang, D. Lo, and L. Jiang, "Active code search: Incorporating user feedback to improve code search relevance," in *Proceedings of the 29th IEEE/ACM International Conference on Automated Software Engineering (ASE '14)*, 2014.

[41] T. Dietrich, J. Cleland-Huang, and Y. Shin, "Learning effective query transformations for enhanced requirements trace retrieval," in *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE '13)*, 2013, pp. 586–591.

[42] Y. Tian, D. Lo, and J. L. Lawall, "Automated construction of a software-specific word similarity database," in *Proc. of CSMR-WCRE*, 2014, pp. 44–53.

[43] E. Hill, L. L. Pollock, and K. Vijay-Shanker, "Automatically capturing source code context of NL-queries for software maintenance and reuse," in *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*, 2009, pp. 232–242.