2015 12th Working Conference on Mining Software Repositories

Sameness: An Experiment in Code Search

Lee Martie and André van der Hoek
University of California, Irvine
Department of Informatics
Irvine, CA 92697-3440 U.S.A.
{lmartie,andre}@ics.uci.edu

***Abstract*—To date, most dedicated code search engines use ranking algorithms that focus only on the relevancy between the query and the results. In practice, this means that a developer may receive search results that are all drawn from the same project, all implement the same algorithm using the same external library, or all exhibit the same complexity or size, among other possibilities that are less than ideal. In this paper, we propose that code search engines should also locate both diverse and concise (brief but complete) sets of code results. We present four novel algorithms that use relevance, diversity, and conciseness in ranking code search results. To evaluate these algorithms and the value of diversity and conciseness in code search, twenty-one professional programmers were asked to compare pairs of top ten results produced by competing algorithms. We found that two of our new algorithms produce top ten results that are strongly preferred by the programmers.**

***Index Terms*—Code, search, diversity, similarity, results, top ten, concise, sameness**

I. Introduction

Programmers search for source code on the web, seeking algorithm implementations and API usage examples for direct copy-paste reuse, to learn from, or to recall how to program them [25, 33, 38, 36]. Programmers often find code written by other programmers, from a variety of projects, in huge quantities (100s of terabytes), with different licenses, and as classes or incomplete snippets [3, 9, 10, 12, 13, 43]. This is very different from searching code locally. Here, programmers find code in smaller quantities, written by themselves or colleagues, as complete projects, in IDEs that organize code for navigation, and where often search is for debugging, refactoring and other maintenance tasks [42, 29, 14].

To better support code search on the web, code search engines have been developed in both research and practice. In particular, these search engines have focused on finding *relevant* source code, searching over large repositories and utilizing the structure and semantics of code to rank results. Open Hub [9] (a popular public code search engine and formerly named Koders [46]) supports search for classes across GitHub [5] and Open Hub repositories. Open Hub supports keyword searches on function definitions, fields, and other language constructs. While keyword queries are a popular way to search [3, 9, 10, 12, 13], code search engines now support querying by test cases to find code that functionally meets the query [48]. Even incomplete code snippets can be submitted to code search engines to autocomplete them [32, 15].

However, considering only relevance in ranking code search results can cause problems for programmers. When the top ten results are similar, other different results – that have lower relevance scores based on the query the developer entered – are hidden, even though they may in fact be closer to what programmers seek. For example, the top ten results can all interpret a keyword query in the same way – leaving the entire top ten useless to some programmers. Imagine submitting the keyword query "file writer" and the top ten results returned use the Apache library because they contain the words "file" and "writer" more often. These results are not useful for programmers looking for examples of writing to file with the JDK. In this case, the results are not diverse enough in the libraries they use.

Further, the top ten results can all similarly contain unnecessary code or not complete code. Consider submitting the query "quick sort" and the top ten results bury the needed code in 2,000 lines of unneeded code or only return empty classes named QuickSort. We refer to these results as in-concise.

Work in the area of information retrieval has considered some problems of sameness in the search engine results. They have developed several techniques to make document results diverse [26, 44]. These techniques avoid returning results that interpret the query the same way [17], and help give the user a better idea of the information space being searched [26].

The key insight underlying our work is that the sameness between results can be controlled by utilizing social and technical properties of the code in ranking results. In this paper we explore this with four new ranking algorithms: (1) diversifying results across their language constructs, complexity, author, and others (ST), (2) by utilizing a new heuristic, constraining results to be concise (MoreWithLess), (3) combining the previous two ways to make results concise and diverse (MWL-ST Hybrid), and (4) diversifying results by the LDA [19] topics that describe them (KL).

To compare these algorithms to existing approaches that use only relevance, we selected three existing algorithms to serve as controls: a standard TF-IDF approach, an enhanced TF-IDF approach from Lemos et al. [34], and Open Hub's ranking algorithm.

We evaluated each algorithm in a search engine infrastructure, taking keyword queries as input and returning a set of matching classes. We chose keywords because they are the most common way to search and classes are a common granularity in code search.

978-0-7695-5594-2/15 $31.00 © 2015 IEEE
DOI 10.1109/MSR.2015.15

We took each of the seven algorithms and asked 21 programmers to compare pairs of top ten search results produced by each. Each participant was asked to make 21 comparisons, where each comparison presents the top ten results from two pairs of algorithms for a distinct query. Programmers read through the search results and selected the set of search results they preferred.

We found that programmers strongly and significantly preferred the results from MoreWithLess (chosen 67% of the time) and MWL-ST Hybrid (chosen 64% of the time). Participants reported preferring MoreWithLess because it gave "clear and concise" results and MWL-ST Hybrid because it gave a "larger variety" of results. Other algorithms were preferred less than 50% of comparisons.

The remainder of this paper is organized as follows. Section 2 presents relevant background material in code search and information retrieval. Section 3 introduces each of the controls and our sameness ranking algorithms. In Section 4, we present the details of our evaluation. Section 5 presents our results. Section 6 discusses threats to validity. Section 7 discusses related work. Finally, we conclude in Section 8 with an outlook on future work.

II. BACKGROUND

Research in code search engines has extensively considered ranking algorithms. Typically keyword queries are given to the search engine and the search engine uses some form of TF-IDF [16] to return a ranked list of results. TF-IDF is a common relevancy criteria in information retrieval, where it scores how descriptive the query is for a document versus the rest of the corpus indexed. A document containing a high frequency of words in a query, where those words are rarer in the rest of indexed corpus will be given a high TF-IDF score.

However, source code is different than the unstructured natural language documents in information retrieval. As such, new approaches to code search have been developed. Sourcerer [21] was among the first code search engines to harness the structure of the code indexed to handle rich upfront language construct queries and rank code by the amount other code refers to it (a type of page rank score). S6, CodeGenie, CodeCojurer [48, 35, 37] accept test cases as queries to search for code that passes the test cases. Automatic query reformulation search engines take a query and expand it with synonyms in natural language and code related terms in order to retrieve relevant results that would otherwise be left behind [34, 45, 22]. Specialized code mapping search engines have also been developed to take queries that specify a class A and class B and finds code that converts objects of type A to objects of type B [20, 49]. Even the code being written by the programmer is used to form queries to find code to auto-complete it [32, 15] or to push recommendations to the programmer [51, 40].

However, existing ranking algorithms consider only the relevance of the search results to the query (other than just checking if results are clones or not). This can be problematic. Keyword queries are inherently ambiguous and multiple interpretations are common. If the top ten code results all take the same interpretation of the query, then some users will find them useless. For example, if the keyword query is "Observer", the top results might all be code using the Apache `FileAlterationObserver` class where the variable names are "observer". Yet, the user may have wanted to see examples using the Observer interface in the JDK. Further, it is not ideal if the top 10 results have high TF-IDF scores but the results bury the needed code in thousands of lines or if many results are simplistic and offer little as API examples or algorithm implementations.

Information Retrieval has done some work on the sameness of results. In particular, work has considered the problem of returning results that interpret a query one way, and address it by returning a diverse set of results [17]. The idea is that this would increase the chance of returning a result with the right interpretation. Other work has found that diversity helps users in obtaining an understanding of the information space in the search results [26]. The most well-known diversity algorithm is called Maximal Marginal Relevance (MMR) [26]. Since some of our sameness algorithms build on MMR, we discuss it in more detail here. Formally MMR is defined as:

$$\mathbf{MMR} \stackrel{\text{def}}{=} \mathbf{Arg}\max_{\mathbf{D_i} \in \mathbf{R \backslash S}} \left[\lambda(\mathbf{Sim_1}(\mathbf{D_i}, \mathbf{Q})) - \left((1-\lambda) \max_{\mathbf{D_j} \in \mathbf{R \backslash S}} \mathbf{Sim_2}(\mathbf{D_i}, \mathbf{D_j}) \right) \right]$$

This definition specifics a procedure to re-rank search engine results so that they are both relevant to the query and different from each other.

MMR starts with an empty set $\boldsymbol{S}$ and the set of results $\boldsymbol{R}$ returned from the search engine given query $\boldsymbol{Q}$ where $\boldsymbol{R} = \{\boldsymbol{D_1}, \ldots, \boldsymbol{D_i}, \ldots, \boldsymbol{D_n}\}$ and $\boldsymbol{D_i}$ denotes an arbitrary document, **n** is the number of documents found, and $\mathbf{1} \leq \boldsymbol{i} \leq \boldsymbol{n}$. First, MMR arbitrarily puts the first result in $\boldsymbol{R}$ in $\boldsymbol{S}$ [44]. The next document put into **S** is chosen to be the document in $\boldsymbol{R \backslash S}$ (in $\boldsymbol{R}$ but not $\boldsymbol{S}$) that is the least same to any of the documents in $\boldsymbol{S}$. The amount of sameness between two documents is calculated by measuring their similarity. Similarity is defined by a function called $\boldsymbol{Sim_2}$ and can be any function that takes in two documents and returns a numeric score, where high scores mean very similar and low scores indicate very different. MMR continues until $\boldsymbol{S}$ is at a desired size. However, there is more to MMR. It also offers an option to weigh, with parameter $\boldsymbol{\lambda}$, the relevancy scores produced from the search engine with the similarity scores produced by $\boldsymbol{Sim_2}$. In this way, the impact of $\boldsymbol{Sim_2}$ on the search engine's ranking algorithm can be controlled. The function $\boldsymbol{Sim_1}$ denotes the search engines ranking function, where it takes in a query and document and returns a ranking score. For example, $\boldsymbol{Sim_1}$ could be TF-IDF. When making use of $\boldsymbol{\lambda}$, MMR selects the next document to put into $\boldsymbol{S}$ as the document in $\boldsymbol{R \backslash S}$ with the highest score for:

$$\lambda(\mathbf{Sim_1}(\mathbf{D_i}, \mathbf{Q})) - \left((1-\lambda) \max_{\mathbf{D_j} \in \mathbf{R \backslash S}} \mathbf{Sim_2}(\mathbf{D_i}, \mathbf{D_j}) \right).$$

The weight parameter $\boldsymbol{\lambda}$ allows the MMR diversification procedure to be "tuned" to find a best balance between the importance of relevancy and diversity. For example, if $\boldsymbol{\lambda} = \mathbf{1}$,

then diversity is not considered and all documents are ranked by relevance scores. If $\boldsymbol{\lambda} = \mathbf{0}$, then relevance scores are ignored (other than working within the set returned from the search engine) and only diversity scores are used to rank documents.

Diversifying results is not the only way to avoid problems of sameness, however. Another approach is to select those that are the same in favorable ways (like conciseness). We explore both approaches in this paper.

III. Ranking Algorithms

In our paper we evaluate seven ranking algorithms. Four of the seven ranking algorithms are intentionally built to control the sameness of the code results either by varying them, making them all concise, or both varied and concise. Otherwise these four algorithms are exactly the same. The formal definition of sameness differs across each algorithm and will be described in detail.

Three of the seven ranking algorithms are controls. They are a standard TF-IDF ranking algorithm, Lemos' et al. WordNet query expansion [34], and Open Hub's ranking algorithm. We chose these three controls because they are examples of an established and popular ranking approach (TF-IDF), a recent example from the literature that was shown to improve performance of TF-IDF (WordNet), and an established and popular standard in practice (OpenHub).

All together we have six algorithms (four sameness algorithms and two controls) that vary in only the ranking algorithm and allow us to avoid confounds. However, for the other algorithm (OpenHub's ranking algorithm), we do not have access to the internals of OpenHub, so we cannot say for certain all that contributes to its final ranked results.

For purposes of our experiment, we constructed a search engine infrastructure in which we could easily swap out ranking algorithms while leaving everything else the same. We describe this first.

A. Code Search Infrastructure

The code search infrastructure relies on four primary components: a List Server, a Code Miner, Sameness Algorithm Ranker, and Automatic Reformulator. Figure 1 presents the overall architecture as a data flow diagram of the primary information that flows among the components. The search engine itself takes queries as HTTP Get requests and returns results in XML and JSON.

The List Server contains a list of repository URLs (i.e., different projects) on GitHub. We obtained this list by using GitHub's API to identify the URLs of 602,244 repositories that included only Java code, a list we obtained between February 4, 2014 and February 12, 2014. The List Server simply passes a single, not-yet-mined repository URL to the Code Miner each time the Code Miner has completed mining a previous repository.

Once the Code Miner has received a repository URL, it clones the repository from GitHub and analyzes it to create a social and technical summary of each of the source code files in that repository (this is utilized in our sameness algorithms

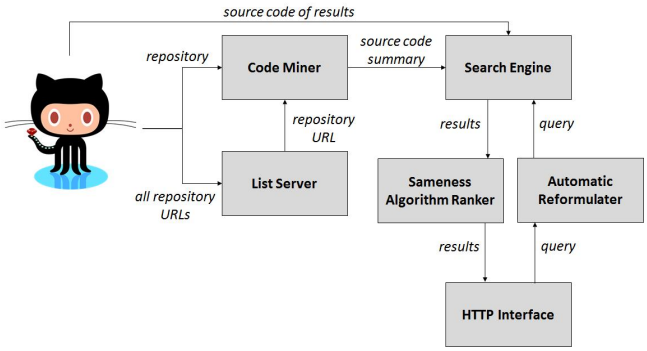

Figure 1. Infrastructure architecture.

explained later). To do so, it extracts a variety of information, including method and variable names, import statements, method invocations, code length, code complexity, and other items. This summary is stored in a database that the Search Engine maintains.

The Search Engine uses the Apache Solr framework [2] to index the source code. When it receives a keyword query, it uses Solr to identify the source files that match the search criteria. It ranks the results using a variant of TF-IDF and returns them together with the original source files that it obtains from GitHub.

The Sameness Algorithm Ranker receives the results from the search engine and then applies a sameness algorithm on the results and returns a top ten. It is configurable to use any algorithm we need and can be toggled off to receive the original results from the search engine.

The Automatic Reformulator receives a query from the HTTP interface and applies the WordNet reformulation algorithm on it. It too is configurable to use any algorithm and can be toggled off to keep the original query unchanged.

From the 602,244 repositories on GitHub, we mined over 300,000 projects. Overall, the search engine database used in our experiments consists of 10M classes, 150M methods, and 253M method calls. A publicly available interface to our code search infrastructure is available at *codeexchange.ics.uci.edu*.

B. Control Ranking Algorithms

We briefly explain the TF-IDF algorithm and the WordNet algorithm below. Since we do not have access to the internals of Open Hub, we cannot explain how it works. To use OpenHub's ranking algorithm, we submitted queries to it (with the Java language filter applied) and downloaded the top ten results for each query.

1) TF-IDF

The TF-IDF ranking algorithm used is the Lucene variant of TF-IDF [11] which is what many Solr implementations use. To use only TF-IDF in our experiment we simply toggle off the Sameness Algorithm Ranker and Automatic Reformulator.

The Lucene TF-IDF scoring function for ranking is defined as:

$$score(q,d) = \sum_{t\ in\ q}\big(tf(t\ in\ d)\cdot idf(t)^2\cdot t.getBoost()\cdot norm(t,d)\big)\cdot coord(q,d)\cdot queryNorm(q).$$

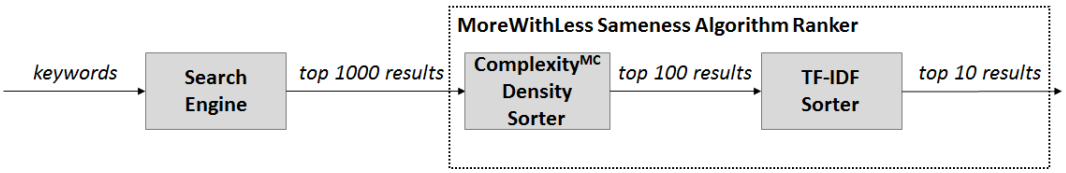

Figure 2. MoreWithLess data flow.

This function takes in a query $\boldsymbol{q}$ and a document $\boldsymbol{d}$ and returns a numeric score. For each term $\boldsymbol{t}$ in $\boldsymbol{q}$, it sums the frequency of $\boldsymbol{t}$ in $\boldsymbol{d}$ (with function $\boldsymbol{tf}$), multiplies this frequency by the inverse document frequency (with function $\boldsymbol{idf}$), multiplies this by any boosts for a term (with function $\boldsymbol{getBoost}$), and finally multiplies this by a normalization factor that weighs documents with less terms higher than documents with more terms (with function $\boldsymbol{norm}$).

The $\boldsymbol{getBoost}$ function is used to weigh documents containing some terms heavier. We do not make use of this function, so all terms are weighted equally. The normalization function, however, is important, because it ranks shorter documents higher. The assumption behind this is that the importance of a term in a document is inversely proportional to how many terms the document has in total.

The entire summation is weighted by the **coord** and **queryNorm** function values. The **coord** function is used to score documents higher that have more of the terms in query **q**. Finally, **queryNorm** is used to normalize the document score so that it can be compared across queries and indexes.

2) WordNet

WordNet [1] is a publically available thesaurus that provides synonyms and antonyms to English words. Our WordNet control is taken from Lemos' et al. [34], which presents several ways to expand a method interface query. Such a query specifies the return type, keywords in the method name, and the parameter types. They had success with expanding their queries with WordNet, a type thesaurus, combining WordNet with a code thesaurus and a type thesaurus, but had no success with the code thesaurus alone. Since they had significant success with WordNet and since we focus only on keywords (not types), we chose to use the WordNet expansion.

The WordNet algorithm is described as follows. Given a query $\boldsymbol{Q} = \{t_1, \ldots, t_k\}$, where t_i is a term in the query, they automatically reformulate it such that $\boldsymbol{Q}$ becomes:

$$(t_1\ OR\ s_{11} OR \ldots\ OR\ s_{1m_1})\ AND \ldots\ AND\ (t_k\ OR\ s_{k1} OR \ldots\ OR\ s_{km_k})$$
$$AND\ NOT(a_{11}\ OR \ldots\ OR\ a_{1o_1})\ AND \ldots\ AND\ NOT(a_{n1} OR \ldots\ OR\ a_{no_n}).$$

Here, s_{ij} is a synonymn from WordNet for term t_i, $1 \le j \le m_i$, and where there are m_i synonyms for term t_i. Further, a_{ig} is an antonym from WordNet for term t_i, $1 \le g \le o_i$, and where there are o_i antonyms for term t_i. Terms are created by splitting words in the query by case change and white space. We used the JWI Java library [7] to interface with WordNet.

C. Sameness Ranking Algorithms

All of our sameness ranking algorithms were developed to take the top 1000 results returned from the TF-IDF ranking algorithm, and then re-rank the results to produce a new top 10. In this way, the algorithms use TF-IDF to get documents that have some relevancy to the query, and then they consider the sameness between these results.

1) MoreWithLess

Code results can often contain unneeded code or are not complete. Making them laborious to read or simply not useful. We developed MoreWithLess as a way to return concise results. Its full data flow is displayed in Figure 2.

First, MoreWithLess receives the top 1000 results from the search engine. Next, it selects 100 code results with the highest scores for a heuristic we developed called Complexity^{MC} Density, and then among these 100 returning 10 with highest TF-IDF values. A code result's Complexity^{MC} Density is defined as:

$$(complexity)\left(\frac{1}{|object\ method\ calls|}\right)\left(\frac{1}{|characters|}\right).$$

```
    @Override
    public void sort() {
        partition(list, 0, list.length - 1);
    }

    private void partition(int[] list, int i, int j) {
        if (j - i + 1 < 2) // number of elements < 2
            return;
        int pivot = list[i + r.nextInt(j - i)]; // in range [i
,j]
        int l = i;
        int r = j;
        while (l < r) {
            while (list[l] < pivot)
                l++;
            while (list[r] > pivot)
                r--;
            swap(list, l, r);
        }
        partition(list, i, l - 1);
        partition(list, l + 1, j);
    }

    private void swap(int[] l, int i, int j) {
        int tmp = l[i];
        l[i] = l[j];
        l[j] = tmp;
    }
```

Figure 3. High Complexity^{MC} Density (.0056).

```
        protected AlgorithmWindow algoWindow;
        final QuickSortAnimationWindow this$0;

        public PartitionMethod(String name, boolean state, QuickSort aQuickSort,
String dataDir, int aPartitionMethod, String filename,
                QuickSortAnimationWindow quickSortAnimationWindow, AlgorithmWind
ow algorithmWindow)
        {
            this$0 = QuickSortAnimationWindow.this;
            super(name, state);
            quickSort = aQuickSort;
            partitionMethod = aPartitionMethod;
            animWindow = quickSortAnimationWindow;
            algoWindow = algorithmWindow;
            ladderTree = CodeCanvas.getLadderTreeFromFile(dataDir, filename,
getLogger(), getBreakPoint());
            addItemListener(new ItemListener() {

                public void itemStateChanged(ItemEvent e)
                {
                    animWindow.resetPartitionButtons();
                    setState(true);
                    quickSort.setPartitionMethod(partitionMethod);
                    algoWindow.setLadderTree(ladderTree);
                }
```

Figure 4. Low Complexity^{MC} Density (.000052).

Table 1: Schema for ST algorithm.

Property Type Name	Value Description
Imports	List of the top level names in the package members imported
Variable words	List of the words occurring in variable names (each word is delimited by case change)
Class name	Name of Java Class
Author name	Name of author of last commit
Project name	Name of the GitHub project class is in
Method call names	List of all the names of method calls
Method declaration names	List of all the names of method declarations
Size	The number of characters
Number of imports	The number of import statements
Complexity	The cyclomatic complexity
Parent class name	The name of the class extended
Interfaces	The names of all interfaces implemented
Package	The package the class is in
Number of fields	The number of fields
Number of method declarations	The number of method declarations
Number of method calls	The number of method calls
Generic	A Boolean value (true if a generic class, false otherwise)
Abstract	A Boolean value (true if abstract class, false otherwise)
Wild card	A Boolean value (true if it has Java wild cards, false otherwise)
Owner Name	Name of owner of the GitHub project this class is in

Here, complexity is the cyclomatic complexity [50] of the result divided by the number of object method calls (or 1 if it equals 0) and number of characters (or 1 if it equals 0) in the result. The higher the score for Complexity^{MC} Density the more logic there is in the code result and the fewer the number of object method calls and characters. Lower scoring results would have lower levels of logic and more object method calls or number of characters. The superscript MC in Complexity^{MC} denotes that we are dividing complexity by number of object method calls (M) and number of characters (C).

The motivation for counting only object method calls (we include static method calls) and not method calls defined in the same class is because we did not want to penalize recursive code. The motivation to penalize by object method calls is to rank classes lower that make large amount of calls to other classes, where many of those calls are not needed by the programmer. The motivation to penalize by size is to rank classes lower that are not efficiently written. The motivation to reward points for complexity is to score code that does more in its source code and avoid code like incomplete abstract classes or code that only makes a few method calls and does nothing with the return values.

To illustrate code that is ranked higher and lower with MoreWithLess, we show two quick sort examples taken from GitHub. Consider the quick sort code presented in Figure 3 – this is a class named QuickSorter (class header and constructor not shown). This class has a complexity of 4, 711 characters, and 1 object method call. The Complexity^{MC} Density score for this code is .0056. This example shows a typical implementation of the quick sort algorithm.

Now consider the code in Figure 4. Only part of a class named `QuickSortAnimationWindow` is shown. This class has a complexity of 5, 1615 characters, and 59 object method calls. The Complexity^{MC} Density score for this code is .000052. The code in Figure 4 uses the Java AWT library to create an interactive UI that animates quick sort being performed. The quick sort algorithm is defined in another class. Since the Complexity^{MC} Density in Figure 3 is 100 times higher than that in Figure 4, it will be ranked much higher by MoreWithLess.

Complexity^{MC} Density is different than the CMAXDENS density metric in the software maintenance literature [23]. There they define CMAXDENS as the cyclomatic complexity per 1000 lines of code and suggest that higher values lead to lower levels of productivity – a negative. Our heuristic is different in two ways. One, it is mathematically a different kind of density at the class level. Two, higher values of density are intended to indicate more concise code – which we hypothesize programmers prefer.

2) ST

ST is a "kitchen sink" approach, selecting the most diverse top ten in terms of results' language constructs and social data

$$sim_2^{ST}(D_i, D_j) = \sum \widehat{|authorName(D_i) \cap authorName(D_j)|},\ \widehat{|className(D_i) \cap className(D_j)|},\ \widehat{\left(\frac{1}{\max\left(abs\left(complexity(D_i) - complexity(D_j)\right), .5\right)}\right)},\ \widehat{\left(\frac{1}{\max\left(abs\left(|fields(D_i)| - |fields(D_j)|\right), .5\right)}\right)},\ \widehat{|hasWildCard(D_i) \cap hasWildCard(D_j)|},\ \widehat{|isAbstract(D_i) \cap isAbstract(D_j)|},\ \widehat{|isGeneric(D_i) \cap isGeneric(D_j)|},\ \widehat{|imports(D_i) \cap imports(D_j)|},\ \widehat{\left(\frac{1}{\max\left(abs\left(|imports(D_i)| - |imports(D_j)|\right), .5\right)}\right)},\ \widehat{|methodCallNames(D_i) \cap methodCallNames(D_j)|},\ \widehat{|methodDecNames(D_i) \cap methodDecNames(D_j)|},\ \widehat{|ownerName(D_i) \cap ownerName(D_j)|},\ \widehat{|package(D_i) \cap package(D_j)|},\ \widehat{|parentClass(D_i) \cap parentClass(D_j)|},\ \widehat{|projectName(D_i) \cap projectName(D_j)|},\ \widehat{\left(\frac{1}{\max\left(abs\left(size(D_i) - size(D_j)\right), .5\right)}\right)},\ \widehat{|variableWords(D_i) \cap variableWords(D_j)|}$$

Equation 1. Sim_2 for ST.

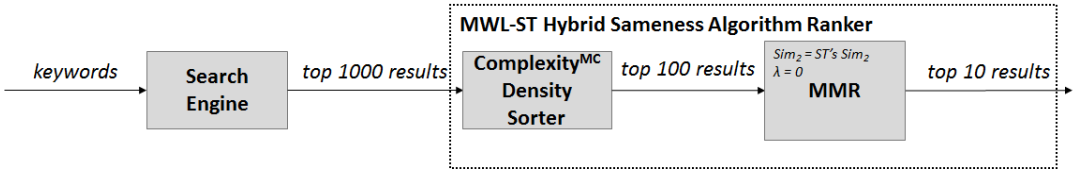

Figure 5. MWL-ST Hybrid data flow.

pulled from their GitHub repository. This ranking algorithm builds on MMR (described in Section 2). In this case, $\boldsymbol{Sim_1}$ is the TF-IDF function used by our search engine. To get the most diverse set, ST sets $\boldsymbol{\lambda = 0}$. However, ST goes one step further than MMR. Recall MMR finds the most diverse set where the first result is in that set. ST finds the most diverse set among all possible sets of ten in the results.

The $\boldsymbol{Sim_2}$ function for ST is defined as the similarity between social and technical properties of two code results. The social and technical properties of a code result are defined by the schema in Table 1. The fields in the schema were chosen because they can easily be obtained from walking the AST of a code result, and extracting meta-data from its GitHub repository using either Git or the GitHub API.

Function $\boldsymbol{Sim_2}$ for ST is formally defined in Equation 1. It calculates the differences between two documents by measuring the overlap in the values for their social-technical schemas. Note that the hat notation on top of each term means that its value is normalized, functions return values in sets when used with set operators, and the superscript "ST" on $\boldsymbol{Sim_2}$ denotes it as the one used in ST.

3) MWL-ST Hybrid

This ranking approach attempts to take the best of MoreWithLess and ST. The motivation behind this approach is to first select a result set that is concise and then select a variety of interpretations among them by diversifying with ST. MWL-ST Hybrid first selects the top 100 Complexity^{MC} Density scoring results and then selects the ten most diverse results with ST. Figure 5 presents the data flow of MWL-ST Hybrid. The configuration of MMR is presented in the upper left corner of the MMR component in the data flow diagram.

4) KL

This ranking algorithm also builds on MMR (described in Section 2), and configures MMR as ST did. However, KL's $\boldsymbol{Sim_2}$ function is defined as the similarity between two code result topic distributions produced from LDA [19].

LDA generates topics describing a large corpus of documents. Each topic is a distribution of words, where high probability words are more descriptive than others for that topic and can be used to interpret it. For example, if a topic's highest probability words are chicken, plow, cows, and horses, then that topic can be reasonably interpreted as farming and we can conclude some portion of the documents in the corpus have something to do with farming. LDA lets us figure out which portion, because it models documents as topic distributions. In this way we can see which topics are more descriptive of a document than others.

KL defines $\boldsymbol{Sim_2}$ as the KL-divergence [47] between two code result topic distributions. KL-divergence measures how similar two topic distributions are, where higher scores mean more similar and lower scores mean less similar. So if two code results are described by very different topics, then their KL-divergence score will be lower than two code results that are described by the same topics.

To implement $\boldsymbol{Sim_2}$, KL first creates the LDA topic distributions of the code results. KL works with preprocessed code results returned from the search engine as shown in Table 2. This schema defines the values of the code results that LDA is run on. This schema was chosen so that the topics would be in terms of social and technical properties of the code results and avoid standard terms in the Java language. KL runs LDA on the results using the Mallet implementation [8]. KL configures LDA to run for 100 iterations, produce 10 topics, and uses the Mallet defaults for hyper-parameters ($alpha = 1$ and $beta = .01$). Once done, KL can run its configuration of MMR on the code result topic distributions, where $\boldsymbol{Sim_2}$ takes the KL-divergence between any two code result topic distributions. KL-divergence is also implemented in Mallet.

IV. Evaluation

We conducted a pair-wise preference survey to examine which of the search result ranking algorithms programmers preferred. The survey was approximately 2.5 hours long and done individually and remotely. Each question displayed a different query and the top ten results from two different ranking algorithms for that query. Participants reviewed each of the top ten for the query and made their selection with the

Table 2. Schema for KL algorithm.

Property Type Name	Value Description
Class name	The class name
Author name	The name of the author of last commit
Project name	The GitHub project name
Owner name	The GitHub project owner name
Parent class name	The parent class name
Interfaces	The list of names of interfaces implemented
Package	The package name of the class
Imports	The names of all package members imported
Variable words	List of the words occurring in variable names (each word is delimited by case change)
Method call names	List of all the names of method calls
Method declaration names	List of all the names of method declarations
Method call arguments	List of all the values to method calls
Field types	List of all fields types

Table 3. Queries.

Query	Category	Source
database connection manager	A (100%)	Sourcerer
ftp client	I (75%)	Sourcerer
quick sort	I (100%)	Sourcerer
depth first search	I (100%)	Sourcerer
tic tac toe	I (75%)	Sourcerer
api amazon	A (100%)	Koders
mail sender	A (100%)	Koders
array multiplication	I (75%)	Koders
algorithm for parsing string integer	I (100%)	Koders
binary search tree	I (75%)	Koders
file writer	A (100%)	Koders
regular expressions	A (100%)	Mica
concatenating strings	A (75%)	Mica
awt events	A (100%)	Mica
date arithmetic	I (100%)	Mica
JSpinner	A (100%)	Mica
prime factors	I (100%)	CodeExchange
fibonacci	I (100%)	CodeExchange
combinations n per k	I (100%)	CodeExchange
input stream to byte array	A (100%)	CodeExchange
spring rest template	A (100%)	CodeExchange

survey interface. We asked them to treat a top ten as a set rather than a ranked list, because we wanted to evaluate their preference for the contents of the top ten rather than order of top ten. We next discuss the survey interface, how the experiment was constructed, where we obtained the queries, and how we selected participants.

The survey interface is shown in Figure 6. The query "Fibonacci" is displayed in the middle and the top ten from ranking algorithm "A" are on the left and the top ten from ranking algorithm "B" on the right. The labels "A" and "B" are used to hide the real names of the ranking algorithms, however we keep track of what "A" and "B" really are on our servers. The users can scroll down to see all the results and can scroll inside each of the editors to see the code entirely. Each editor color codes the syntax of the code to make it easier to read. The

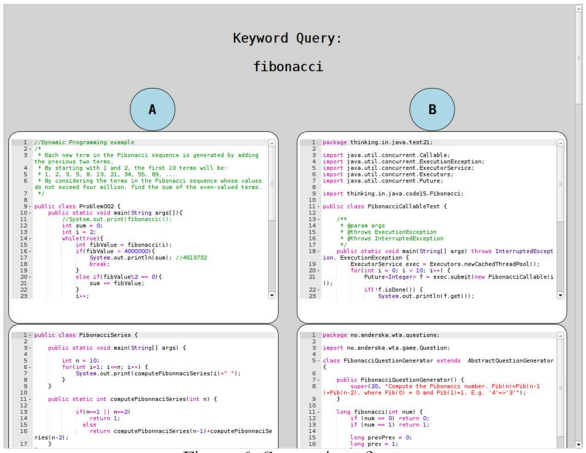

Figure 6. Survey interface.

editors also serve to provide a uniform interface across the top ten produced from each of the seven algorithms. The subjects indicate their preference by selecting the button labeled as "A" or "B" occurring above the top ten they prefer. Once they make their selection a popup box appears where they are asked to explain why they made their selection. After the last question, they were prompted with open ended questions.

To prevent bias, comparisons were structured so that each combination of ranking algorithms (21 pairs total) was presented to each user once and evaluated for each query once. For the 21 comparisons, this requires 21 participants and 21 queries. In this way each algorithm comparison was evaluated by each person and query once, yielding a total of 441 distinct questions total. To remove ordering bias, the order in which each person was presented questions was randomized, the order of the results in a top ten was randomized, and the left/right placement of a top ten is done at random.

To obtain realistic queries and in an unbiased manner, we selected them from four different code search engine logs. Five were taken from Sourcerer logs [21], five from Mica logs [30], Six from Koders' logs [18], and five from the logs of CodeExchange [4] (we released CodeExchange in August 2014 and have since logged over 4,000 keyword queries). Further the queries were categorized by three independent experts as either algorithm/data structure implementation queries (I) or as API/library example queries (A). For 76% of the queries, the experts had a 100% level of agreement; for the remainder, they had an agreement level of 75%. This yielded 10 API/library queries and 11 algorithm/data structure implementation queries. Table 3 lists each query, its category (with agreement percentage), and its source.

We recruited participants that had work experience as a programmer and were skilled in Java. To do so, we sent out an advertisement on UCI's Institute for Software Research [6] mailing list. Both professional programmers and students subscribe to this list. We recruited 21 participants who self-reported an average of 7.8 years of work experience and a high skill level in Java of 7.5 (1 being novice and 10 being expert). Table 4 give a more in depth report of the self-reported demographics of our participants.

V. Results

After collecting all 441 answers from participants we created the "win" table shown in Table 5. Each row is identified by a rank algorithm and each cell shows how many times it won against the rank algorithm identified by its column header. For example, MoreWithLess beat Open Hub 14 times.

Table 4. Subject demographics.

	All	Student	Non-Student
Avg. age	32	30.3	34.1
Avg. years worked	7.8	4.8	11.7
Avg. Java skill (1-10)	7.5	7.3	7.8
Male	19	10	9
Female	2	2	0
Number of subjects	21	12	9

Since each ranking algorithm competes against the other 21 times, this means MoreWithLess beat Open Hub 67% of the time (win percentages appear beside the win totals). The total wins for each ranking algorithm appear down the total column. MoreWithLess and MWL-ST Hybrid have the highest winning percentage by far (67% and 64% respectively). The others all have below a 50% winning percentage. This indicates that programmers prefer the results from MoreWithLess and the MWL-ST Hybrid much more than the others. To understand how large a difference there is and if these results were due to chance, we conducted a chi-squared test on the win-loss table of MoreWithLess and MWL-ST Hybrid. Table 6 shows the results, with the Chi Squared results below each table. Both chi squared test tables show the effect size (w), population (N), degrees of freedom (df), significance levels (sig. level), and power (power). Population here is 126 comparisons (each ranking algorithm is compared 126 times total). The significance level for both are well below .05 and let us conclude the win-loss record for both are not due to chance. Further both effect sizes are above .40. The effect size is a way of measuring the strength of a phenomena and typically a Cohen's w above .3 is considered a medium effect and anything above .5 is considered large effect [27]. So, in the cases above, the outcomes not due to chance and the MoreWithLess and MWL-ST Hybrid had a medium to large effect on the programmer's preference.

We found that for 2 of the queries, 9 results for OpenHub were displayed instead 10. For these 2 queries, OpenHub lost 9/12 times. We are unsure why this happened, but this slight difference does not change the overall outcome. Awarding OpenHub those 9 wins and removing them from the corresponding winning ranking algorithms, still results in MoreWithLess and MWL-ST Hybrid being the only significantly preferred algorithm and strongly preferred algorithms.

A. Participant Explanations

To help better understand why the programmer's like MoreWithLess and MWL-ST Hybrid much more, we examined the programmers' explanations of why they chose them and why they did not. We also looked at the explanations to help explain why MWL-ST Hybrid beat MoreWithLess when they were being compared. Participants explain that their preference was due to the relevancy, diversity, and conciseness of the results.

Table 6. Chi squared results for win loss tables.

MoreWithLess win loss table		
	win	loss
observed	84	42
expected	63	63

MWL-ST Hybrid win loss table		
	win	loss
observed	81	45
expected	63	63

Chi Squared Test
w = 0.4349398
N = 126
df = 1
sig.level = 0.0106
power = 0.99

Chi Squared Test
w = 0.4000399
N = 126
df = 1
sig.level = 0.03046
power = 0.99

There were many cases when the participants selected MoreWithLess because it was/had "*simpler examples*", "*more concise and clear*", "*more readable*", "*more useful*", "*more math-oriented*", or "*better examples*". For example, when comparing MoreWithLess with Open Hub one participant said: "*Set [MoreWithLess] contains more concise examples of how to convert an input stream into a byte array.*" When comparing MoreWithLess with WordNet one participant explained MoreWithLess had *"…more short focused bits about prime factors.*" Other times participants simply say they were more "relevant" and a few times it was a "*close call*".

For the cases when MoreWithLess was not selected, participants felt the other ranking algorithm had more "*relevant*" code or "*better examples*" or had more "*variety*" and other times it was "*hard to tell*". For example, when comparing MoreWithLess with WordNet, one participant said "*Set [WordNet] seemed to have a few more easily adapted examples.*" When comparing ST with MoreWithLess one participant said "*set [ST] has a wider variety of implementations*".

MWL-ST Hybrid was often selected because it had "*more unique samples*", "*more instances*", "*larger variety*", and "*more relevant*" code. For example, when comparing MWL-

Table 5. Win table for ranking algorithms.

	MoreWithLess	MWL-ST Hybrid	Open Hub	WordNet	TF-IDF	KL	ST-Schema	total
MoreWithLess	0(0%)	8(38%)	14(67%)	13(62%)	17(81%)	16(76%)	16(76%)	84(67%)
MWL-ST Hybrid	13(62%)	0(0%)	12(57%)	14(67%)	13(62%)	15(71%)	14(67%)	81(64%)
Open Hub	7(33%)	9(43%)	0(0%)	10(48%)	10(48%)	12(57%)	13(62%)	61(48%)
WordNet	8(38%)	7(33%)	11(52%)	0(0%)	12(57%)	9(43%)	12(57%)	59(47%)
TF-IDF	4(19%)	8(38%)	11(52%)	9(43%)	0(0%)	11(52%)	12(57%)	55(44%)
KL	5(24%)	6(29%)	9(43%)	12(57%)	10(48%)	0(0%)	10(48%)	52(41%)
ST-Schema	5(24%)	7(33%)	8(38%)	9(43%)	9(43%)	11(52%)	0(0%)	49(39%)

ST Hybrid with Open Hub one participant said: "*I prefer getting the set of [MWL-ST Hybrid] because the set of [Open Hub] only provides codes for DOM/SAX writers.*" When Comparing MWL-ST Hybrid with WordNet one participant explained "*as a whole, set [MWL-ST Hybrid] gave me more different ideas on how to implement a tic tac toe game including a simple ai to play against...*" In other cases, participants explained that they had trouble deciding. It appears the variety of examples returned by MWL-ST Hybrid could in part explain why it was chosen.

In cases where MWL-ST Hybrid was not selected, participants reported it was not to the "*point*" or not "*concise*" enough, or that they simply felt the other ranking algorithm provided more "*relevant*" or "*better examples*". For example when comparing MWL-ST Hybrid with Open Hub one participant said "*Group [Open Hub] was more to the point for the given Keyword Query*". When comparing MWL-ST Hybrid with WordNet, one participant explained "*Results from Set [MWL-ST Hybrid] had lower relevance and higher complexity.*" A few times they said it was hard to make the decision.

It is interesting to note that while MoreWithLess wins most often in total, MWL-ST Hybrid beats MoreWithLess 62% of the time. For six of the comparisons, participants said MWL-ST Hybrid gave more relevant code. For four comparisons, they said it was a tough choice. For the other three comparisons, MWL-ST Hybrid was chosen because it had more variety. Some explanations were: "*The set [MWL-ST Hybrid] results seem to contain more varieties…*" and "*Set [MWL-ST Hybrid] seemed to have a larger variety of examples relevant to doing different tic tac toe things.*" So, it appears that a part of the reason why MWL-ST Hybrid beat MoreWithLess is because it offers more variety and relevant results. It also appears though it was not always easy to choose between the two.

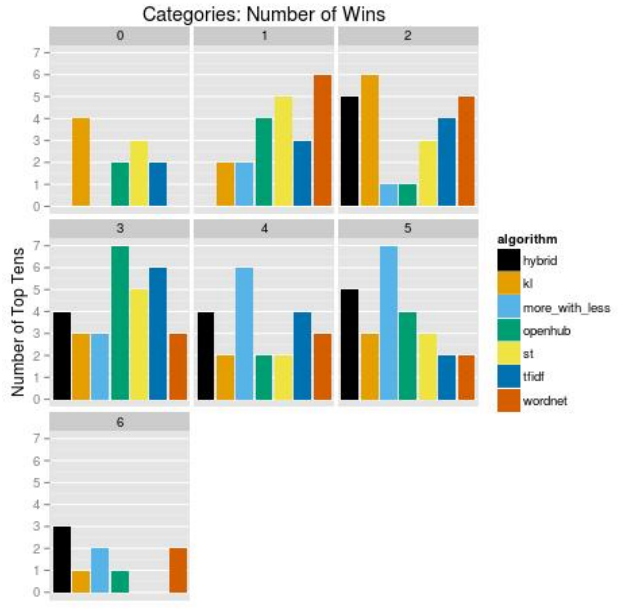

Figure 7. Algorithm vs. number of top tens vs. number of wins.

Finally, other approaches were sometimes selected because they had concise examples or offered variety. For example, in a case where KL beat WordNet a participant reported that "*results from Set [KL] were concise and as expected.*" ST beats MWL-ST Hybrid once because "*Set [ST] contains a greater variety of verbose examples than set [MWL-ST Hybrid]*." Given that MoreWithLess and MWL-ST Hybrid were more often preferred, it appears these ranking algorithms met the participants' preference of relevancy, diversity, and conciseness more often than the other ranking algorithms.

B. Properties of Top Ten by Number of Wins

While MoreWithLess and MWL-ST Hybrid were strongly preferred by the participants, we found other algorithms also provided a substantial set of highly preferred top ten sets. This can be seen in Figure 7, where it shows seven different grouped bar charts, where each is placed in a category. Each category identifies the number of wins a top ten received (labeled as 0, 1, 2, 3, 4, 5, or 6) , the bar color identifies the algorithm producing that top ten, and the y-axis identifies the number of top ten sets produced. For example, Figure 7 shows that the MWL-ST Hybrid algorithm had 3 top ten sets that won 6 times (the max a top ten set can win). Figure 7 shows that algorithms other than MoreWithLess and MWL-ST Hybrid produced many top ten sets winning 4 or more times.

We looked to see if higher scores for Complexity^{MC} Density and diversity were not just common to preferred top ten sets from MoreWithLess and MWL-ST Hybrid, but common in all preferred top ten sets. Additionally, we looked to see if variations and parts of Complexity^{MC} Density could be found in preferred top ten sets. To do so, we plotted the number of wins of top ten sets to the average of these properties - Table 7 shows the results. Using Spearman's rho, we found that there is a statistically significant and medium sized positive correlation (rho around .3 is often interpreted as medium sized) between more wins and higher levels of Complexity^{MC} Density and complexity over amount of object method calls (we call it Complexity^{M} Density). That means, on average, there is a correlation between more highly preferred top ten sets and higher complexity densities. Further, we found two weak to

Table 7. Correlation analysis with Spearman's rho.

x-axis	y-axis	rho	p
number of wins	average complexity/|method calls|	.3242	.00003
number of wins	average Complexity^{MC} Density	.3187	.00004
number of wins	average |method calls|	-.2740	.00039
number of wins	average group Levenshtein distance	-. 2447	.0014
number of wins	average complexity/size	.2286	.00267
number of wins	average size	-.0967	.1219
number of wins	average group technical similarity	-.0156	.4255
number of wins	average complexity	-.0030	.4852

medium sized negative correlations as well – higher preferred top ten sets are correlated to a lower number of object method calls and lower Levenshtein distances, on average. Levenshtein distance is the number of edits required to transform one character sequence into another (a type of similarity metric). Here the average Levenshtein distance of a top ten set means the average number of edits to change one code result into another in that top ten set – note that, average similarity distances are calculated with group average similarity [39]. Interestingly, then, more highly preferred top ten sets are closer in their edit distance.

We found no correlation between the number of wins of top ten sets and its average size, complexity, or group technical similarity (measured by the Sim_2 function for ST but without the social values in the ST schema). We could not include the social values for measuring similarity because some are not available for the Ohloh top ten sets. Finding no correlation between the average group technical similarity and highly preferred top ten sets suggests further research needs to be done in understanding why they were preferred.

VI. Threats to Validity

First, while our participants all had substantial work experience, 12 of them were students. However the average work experience among the students was almost 5 years – we thought this acceptable experience.

Second, while we selected 21 queries from four different sources and are reasonable representatives of implementation and API exemplar queries, these are by no means all possible kinds of queries. It is important then that the results in this paper are scoped to the kinds and actual queries used in the survey.

Third, our controlled experiment setup allowed us to control for confounding variables, however it also meant compromising on the realism of the experiment. A more realistic experiment would be complementary to ours – offering realism but losing control of confounding variables. Consequently, more complete evidence can be obtained by combing data from both controlled and realistic experiments. In more realistic experiments, programmers would need to be able to type their own keywords and be given specific code search tasks. These tasks would need to be a mix of both ill-defined and precise tasks that are varied across intents [25, 38].

VII. Related Work

Similar ideas to conciseness and diversity have occurred in the mining API documentation literature [41, 28, 31]. This literature covers research in enhancing API documentation with exemplar code snippets illustrating different usage examples. Often these are extracted by mining common patterns occurring in source code repositories and formatting them to make them more readable.

In the API mining literature (and in a recent method level code search tool paper [24]), conciseness is framed as size or number of irrelevant lines of code [28, 24, 41]. Yet, this is an incomplete description of conciseness. After all, an empty class called QuickSort has no irrelevant lines of code and spans one line, but this does not make it a concise example of quick sort. Concise examples have to be complete and brief. Not surprisingly then, in [24] they had worse results ranking code results by size. Further, as we showed in Section 5, Lucene's TF-IDF performed poorly which also ranks smaller sized code higher. Instead, our work takes a step closer to concise code results by considering the logic in the code, its size, and its reliance on outside classes. Indeed, as the results with MoreWithLess show, doing so resulted in the best performing ranking algorithm.

While the word diverse is not always used, similar concepts such as producing API documentation examples that are not duplicates [31, 28] and cover the API are considered [31]. Certainly one of the goals of a diverse top ten is to avoid duplicates, but diversity in the top ten is also a very different problem than coverage of an API. An API doc provides a list of all method calls that can be used to check the coverage of a set of examples against. However, code search has no lists to check against. For example, there is no way currently to check if a search engine can return all the possible ways of implementing tic tac toe, a ftp client, quick sort, and so on. While our work represents a step toward producing diverse examples in the top ten, much research remains in understanding the meaning of diverse beyond just relevant results that are not duplicates and different.

VIII. Conclusion

Considering only ranking the top ten results by their relevancy to the query overlooks the problems of sameness in the results. This paper presents four new sameness ranking algorithms as the first step toward addressing sameness problems. By leveraging the social and technical information of the code results, we created two ranking algorithms to make results more concise (MoreWithLess) and both concise and diverse (MWL-ST Hybrid). We found that the participants in our survey strongly preferred these two algorithms over our other sameness algorithms and three controls. Further, the participants explained they preferred a relevant, concise, and diverse top ten. Finally, we found correlations with highly preferred top ten sets and higher values of complexity density and lower values for number of object method calls and edit distance.

There is much future work to be done. In particular, there is opportunity to research more in depth when MWL-ST Hybrid performs better than MoreWithLess , the meaning of diversity in the top ten, and interface implications of letting users control $\boldsymbol{\lambda}$ in MMR ranking algorithms. Also, we plan to perform further studies where we evaluate sameness ranking algorithms' effectiveness for supporting programmers' code search tasks.

Acknowledgment

This work was sponsored by NSF grant CCF-1321112. Special thanks to Martín Medina for helping with the experiment.

REFERENCES

[1] "About WordNet - WordNet - About WordNet." [Online]. Available: http://wordnet.princeton.edu/wordnet/. [Accessed: 14-Feb-2015].

[2] "Apache Lucene - Apache Solr." [Online]. Available: http://lucene.apache.org/solr/. [Accessed: 14-Feb-2015].

[3] "Code Search · GitHub." [Online]. Available: https://github.com/search. [Accessed: 14-Feb-2015].

[4] "CodeExchange." [Online]. Available: http://codeexchange.ics.uci.edu/. [Accessed: 14-Feb-2015].

[5] "GitHub · Build software better, together." [Online]. Available: https://github.com/. [Accessed: 14-Feb-2015].

[6] "Institute for Software Research." [Online]. Available: http://isr.uci.edu/. [Accessed: 14-Feb-2015].

[7] "JWI 2.3.3." [Online]. Available: http://projects.csail.mit.edu/jwi/. [Accessed: 14-Feb-2015].

[8] "MALLET homepage." [Online]. Available: http://mallet.cs.umass.edu/. [Accessed: 14-Feb-2015].

[9] "Open Hub Code Search." [Online]. Available: http://code.openhub.net/. [Accessed: 14-Feb-2015].

[10] "searchcode | source code search engine." [Online]. Available: https://searchcode.com/. [https://searchcode.com/.].

[11] "Similarity (Lucene 2.9.4 API)." [Online]. Available: https://lucene.apache.org/core/2_9_4/api/all/org/apache/lucene/search/Similarity.html. [Accessed: 14-Feb-2015].

[12] "Sourcegraph." [Online]. Available: https://sourcegraph.com/. [Accessed: 14-Feb-2015].

[13] "Stack Overflow." [Online]. Available: http://stackoverflow.com/. [Accessed: 14-Feb-2015].

[14] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung, "An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information During Software Maintenance Tasks," IEEE Trans. Softw. Eng., vol. 32, no. 12, pp. 971–987, Dec. 2006.

[15] A. T. Nguyen, H. A. Nguyen, T. T. Nguyen, and T. N. Nguyen, "GraPacc: A Graph-based Pattern-oriented, Context-sensitive Code Completion Tool," Proceedings of the 34th International Conference on Software Engineering, Piscataway, NJ, USA, 2012, pp. 1407–1410.

[16] C. D. Manning, P. Raghavan, and H. Schütze, Introduction to Information Retrieval. New York, NY, USA: Cambridge University Press, 2008.

[17] C. L. A. Clarke, M. Kolla, G. V. Cormack, O. Vechtomova, A. Ashkan, S. Büttcher, and I. MacKinnon, "Novelty and Diversity in Information Retrieval Evaluation," in Proceedings of the 31st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, New York, NY, USA, 2008, pp. 659–666.

[18] C. Lopes, S. Bajracharya, J. Ossher, P. Baldi (2010). UCI Source Code Data Sets [http://www.ics.uci.edu/~lopes/datasets]. Irvine, CA: University of California, Bren School of Information and Computer Sciences.

[19] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," J. Mach. Learn. Res., vol. 3, pp. 993–1022, Mar. 2003.

[20] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman, "Jungloid Mining: Helping to Navigate the API Jungle," in Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, New York, NY, USA, 2005, pp. 48–61.

[21] E. Linstead, S. Bajracharya, T. Ngo, P. Rigor, C. Lopes, and P. Baldi, "Sourcerer: Mining and Searching Internet-scale Software Repositories," Data Min. Knowl. Discov., vol. 18, no. 2, pp. 300–336, Apr. 2009.

[22] F. A. Durão, T. A. Vanderlei, E. S. Almeida, and S. R. de L. Meira, "Applying a Semantic Layer in a Source Code Search Tool," in Proceedings of the 2008 ACM Symposium on Applied Computing, New York, NY, USA, 2008, pp. 1151–1157.

[23] G. K. Gill and C. F. Kemerer, "Cyclomatic complexity density and software maintenance productivity," IEEE Transactions on Software Engineering, vol. 17, no. 12, pp. 1284–1288, Dec. 1991.

[24] I. Keivanloo, J. Rilling, and Y. Zou, "Spotting Working Code Examples," in Proceedings of the 36th International Conference on Software Engineering, New York, NY, USA, 2014, pp. 664–675.

[25] J. Brandt, P. J. Guo, J. Lewenstein, M. Dontcheva, and S. R. Klemmer, "Two Studies of Opportunistic Programming: Interleaving Web Foraging, Learning, and Writing Code," Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, New York, NY, USA, 2009, pp. 1589–1598.

[26] J. Carbonell and J. Goldstein, "The Use of MMR, Diversity-based Reranking for Reordering Documents and Producing Summaries," in Proceedings of the 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, New York, NY, USA, 1998, pp. 335–336.

[27] J. Cohen, Statistical Power Analysis for the Behavioral Sciences, 2 edition. Hillsdale, N.J: Routledge, 1988.

[28] J. Kim, S. Lee, S. Hwang, and S. Kim, "Towards an Intelligent Code Search Engine," in Twenty-Fourth AAAI Conference on Artificial Intelligence, 2010.

[29] J. Singer, T. Lethbridge, N. Vinson, and N. Anquetil, "An Examination of Software Engineering Work Practices," in Proceedings of the 1997 Conference of the Centre for Advanced Studies on Collaborative Research, Toronto, Ontario, Canada, 1997, p. 21–.

[30] J. Stylos and B. A. Myers, "Mica: A Web-Search Tool for Finding API Components and Examples," IEEE Symposium on Visual Languages and Human-Centric Computing, 2006. VL/HCC 2006, 2006, pp. 195–202.

[31] J. Wang, Y. Dang, H. Zhang, K. Chen, T. Xie, and D. Zhang, "Mining succinct and high-coverage API usage patterns from source code," in 2013 10th IEEE Working Conference on Mining Software Repositories (MSR), 2013, pp. 319–328.

[32] M. Bruch, M. Monperrus, and M. Mezini, "Learning from Examples to Improve Code Completion Systems," Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, New York, NY, USA, 2009, pp. 213–222.

[33] M. Umarji, S. E. Sim, and C. Lopes, "Archetypal Internet-Scale Source Code Searching," in Open Source Development, Communities and Quality, B. Russo, E. Damiani, S. Hissam, B. Lundell, and G. Succi, Eds. Springer US, 2008, pp. 257–263.

[34] O. A. L. Lemos, A. C. de Paula, F. C. Zanichelli, and C. V. Lopes, "Thesaurus-based Automatic Query Expansion for

Interface-driven Code Search," in Proceedings of the 11th Working Conference on Mining Software Repositories, New York, NY, USA, 2014, pp. 212–221.

[35] O. A. L. Lemos, S. K. Bajracharya, J. Ossher, R. S. Morla, P. C. Masiero, P. Baldi, and C. V. Lopes, "CodeGenie: Using Test-cases to Search and Reuse Source Code," in Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering, New York, NY, USA, 2007, pp. 525–526.

[36] O. Barzilay, O. Hazzan, and A. Yehudai, "Characterizing Example Embedding as a software activity," in ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation, 2009. SUITE '09, 2009, pp. 5 –8.

[37] O. Hummel, W. Janjic, and C. Atkinson, "Code Conjurer: Pulling Reusable Software out of Thin Air," IEEE Software, vol. 25, no. 5, pp. 45–52, Sep. 2008.

[38] R. E. Gallardo-Valencia and S. E. Sim, "What Kinds of Development Problems Can Be Solved by Searching the Web?: A Field Study," in Proceedings of the 3rd International Workshop on Search-Driven Development: Users, Infrastructure, Tools, and Evaluation, New York, NY, USA, 2011, pp. 41–44.

[39] R. Feldman and J. Sanger, The Text Mining Handbook: Advanced Approaches in Analyzing Unstructured Data. Cambridge University Press, 2007.

[40] R. Holmes, R. J. Walker, and G. C. Murphy, "Approximate Structural Context Matching: An Approach to Recommend Relevant Examples," IEEE Trans. Softw. Eng., vol. 32, no. 12, pp. 952–970, Dec. 2006.

[41] R. P. L. Buse and W. Weimer, "Synthesizing API Usage Examples," in Proceedings of the 34th International Conference on Software Engineering, Piscataway, NJ, USA, 2012, pp. 782–792.

[42] S. D. Fleming, C. Scaffidi, D. Piorkowski, M. Burnett, R. Bellamy, J. Lawrance, and I. Kwan, "An Information Foraging Theory Perspective on Tools for Debugging, Refactoring, and Reuse Tasks," ACM Trans. Softw. Eng. Methodol., vol. 22, no. 2, pp. 14:1–14:41, Mar. 2013.

[43] S. E. Sim, M. Agarwala, and M. Umarji, "A Controlled Experiment on the Process Used by Developers During Internet-Scale Code Search," Finding Source Code on the Web for Remix and Reuse, S. E. Sim and R. E. Gallardo-Valencia, Eds. Springer New York, 2013, pp. 53–77.

[44] S. Guo and S. Sanner, "Probabilistic Latent Maximal Marginal Relevance," in Proceedings of the 33rd International ACM SIGIR Conference on Research and Development in Information Retrieval, New York, NY, USA, 2010, pp. 833–834.

[45] 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, Piscataway, NJ, USA, 2013, pp. 842–851.

[46] S. K. Bajracharya and C. V. Lopes, "Analyzing and mining a code search engine usage log," Empir Software Eng, vol. 17, no. 4–5, pp. 424–466, Aug. 2012.

[47] S. Kullback and R. A. Leibler, "On Information and Sufficiency," The Annals of Mathematical Statistics, vol. 22, no. 1, pp. 79–86, Mar. 1951.

[48] S. P. Reiss, "Semantics-based Code Search," Proceedings of the 31st International Conference on Software Engineering, Washington, DC, USA, 2009, pp. 243–253.

[49] 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.

[50] T. J. McCabe, "A Complexity Measure," IEEE Transactions on Software Engineering, vol. SE-2, no. 4, pp. 308–320, Dec. 1976.

[51] Y. Ye and G. Fischer, "Supporting Reuse by Delivering Task-relevant and Personalized Information," Proceedings of the 24th International Conference on Software Engineering, New York, NY, USA, 2002, pp. 513–523.