How Developers Use Multi-Recommendation System in Local Code Search

Xi Ge[1] David Shepherd[2] Kostadin Damevski[3] Emerson Murphy-Hill[1]
[1]Department of Computer Science, NC State University, Raleigh, NC, USA
[2]ABB Inc., Raleigh, NC, USA
[3]Mathematics and Computer Science Department, Virginia State University, Petersburg, VA, USA
xge@ncsu.edu david.shepherd@us.abb.com kdamevski@vsu.edu emerson@csc.ncsu.edu

***Abstract*—Developers often start programming tasks by searching for relevant code in their local codebase. Previous research suggests that 88% of manually-composed queries retrieve no relevant results. Many searches fail because existing search tools depend solely on string matching with a manually-composed query, which cannot find semantically-related code. To solve this problem, researchers proposed query recommendation techniques to help developers compose queries without the extensive knowledge of the codebase under search. However, few of these techniques are empirically evaluated by the usage data from real-world developers. To fill this gap, we studied several query recommendation techniques by extending Sando and conducting a longitudinal field study. Our study shows that over 30% of all queries were adopted from recommendation; and recommended queries retrieved results 7% more often than manual queries.**

I. Introduction

Many programming tasks start with a search [1], [2]. However, searching code is difficult for developers: many are forced to use out-of-date tools, like regular expression based search tools, to search millions of lines of code they have never seen [3]. Since these tools depend solely on developers composing queries that exactly match unfamiliar strings, 88% of all queries using these tools return no relevant results [2]. Because developers spend as much as 40% of their time searching, navigating, reading, and understanding source code [1], [2], these failed searches waste a lot of developers' time, potentially increasing the cost of software development.

Local code search tools help developers find a starting point for programming tasks. These tools are now distributed as part of popular integrated development environments (IDEs): InstaSearch [4] and Sando [5] are available for Eclipse [6] and Visual Studio [7], respectively. These tools take developers' queries as input and retrieve code elements from the currently opened project in the IDE. Researchers have shown local code search tools to be more effective than regular expression search tools [5]. Unlike regular expression search tools like grep, local code search tools: (1) retrieve code elements, such as methods, classes and fields, instead of lines of text; (2) return ranked results, making more relevant results more accessible; and (3) index the codebase before developers need to search, making search almost instantaneous [5].

Despite these advantages, local code search tools suffer from some of the same issues that plague regular expression search tools. For instance, imagine a developer wants to search for how usernames are stored in an authentication system. They may search for "user file" or "user element". However, the implementation may refer to storage as a "user document". The developer would not think to compose this query without knowledge of the exact terms used in the codebase. Neither regular expression search tools nor local code search tools will help a developer in this scenario.

To address this problem, researchers propose multiple query recommendation techniques [8], [9], [10]. However, few of these techniques are empirically studied in the real-world setting. Without observing developers' actual interaction with the recommendation techniques, the usability and usefulness of these techniques are unclear. To fill this gap, we implemented Coronado, an extension to the Sando search tool that integrates various recommendation techniques to help developers compose new queries and refine failed ones. Taking advantage of the popularity of Sando, we were able to evaluate these recommendation techniques by collecting field data. In summary, the contributions of this paper are:

- An implementation of multiple query recommendation techniques as an extension to Sando, a local code search tool for Visual Studio [5], which we discuss in Section II.
- The result of longitudinal field study, presented in Section III, investigating Coronado's usability and usefulness. This study collected usage data from 274 unique Sando users for 7 months. Our study suggests that the queries recommended before manual search fails are equally useful to the recommendations afterwards; and recommending the identifiers in the codebase is the most effective technique.

II. Sando Recommendations

Before delving into the recommendation techniques, we first introduce a local code search tool called Sando on which our techniques was implemented and evaluated. As a Visual Studio plug-in, Sando allows developers to search code snippets in their local codebase by issuing queries similar with those accepted by Google. Sando supports searches over C and C# code. Sando treats the different levels of software entities, including classes, methods and fields, as documents. Search results are a list of documents that contain the terms in the input query. Sando orders the results based on Inverse Document Frequency scores that reflect a term's importance to a document in a collection of documents [11]. Figure 1 is a screen shot of Sando, where the search box is at the top (A) and the list view at the bottom (B) presents search results. Near

978-1-4799-4035-6/14/$31.00 ©2014 IEEE

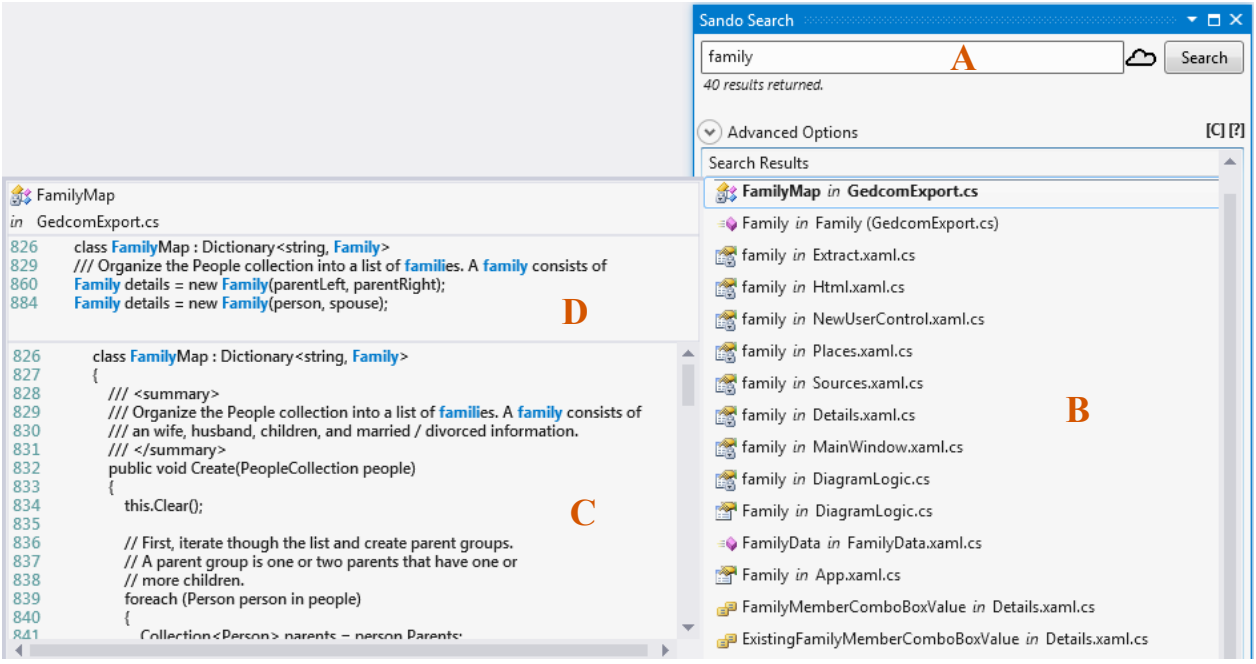


Fig. 1: Sando screenshot.

the search box, the cloud button invokes a recommendation technique based on tag clouds as illustrated in Section II-B, the "[C]" button clears the search history, and the "[?]" button provides help.

When a users single clicks one of the search results, Sando displays a pop up window to give her an overview of the corresponding code element. The lower half of the pop up window (C) has the entire code element, and the upper half contains the exact lines containing the queried terms (D). To open the file containing a search result, the user double clicks the result.

We used Sando as our platform to implement and evaluate the recommendation techniques for the following reasons: (1) Sando is representative of local code search tools, as it implements similar functionality as the search tools in other integrated development environments; (2) Sando is an open source project with extensible APIs, so we can easily add features to it; (3) Users have downloaded Sando over five thousand times[1], which provides us a significant number of users from which to collect feedback. We next detail the recommendation over the following two subsections. Section II-A describes the components used in our techniques; Section II-B and Section II-C describe how we generate and present the recommended queries.

A. Components

To bridge the cognitive gap between local code search users and the codebase under search, we implemented an extension to Sando named Coronado that is based on five components: **codebase terms**, **term co-occurrence matrix**, **Verb-Direct-Object repository**, **software engineering thesaurus**, and **English thesaurus**. Before we explain how Coronado uses these components to recommend queries, we first discuss each component and how Coronado collects and maintains those contents.

1) Codebase terms: The first component of Coronado contains the terms in the codebase under search. Coronado collects these terms when Sando's indexer traverses the programmer's project. Sando's indexing process breaks a source code element into a set of terms, including raw identifier names and terms extracted by splitting identifiers that use camel case. For instance, Sando indexes a method name "parseFile" as three terms: "parse", "file", and "parseFile". C and C# keywords are not indexed.

Sando indexes the codebase in two stages: the initial indexing and the incremental indexing. Sando performs the initial indexing only when the developer opens a project whose index information has not been cached previously by Sando. Sando performs incremental indexing whenever the developer changes a project whose index information is cached. Initially indexing a project of $10,000$ lines of code takes Sando about twenty seconds, and incrementally indexing a changed C# file takes about twenty milliseconds. By reusing the terms collected during indexing, Coronado builds and maintains the set of terms in the codebase under search.

2) Term co-occurrence matrix: The second component of Coronado is a matrix that records the number of co-occurrences of every two terms that appear together. For each pair of terms $[t_1, t_2]$ that occur together in a source document, the matrix builder increments the current value of the element at $[t_1, t_2]$ in the matrix by one. To give an example, consider the following code snippet:

```
PathManager CreatePathManager(string path) {
  if(Path.HasExtension(path))
    return new PathManager(Path.GetDirectoryName(path));
  else
    return new PathManager(path);
}
```

[1] goo.gl/jjuhP1

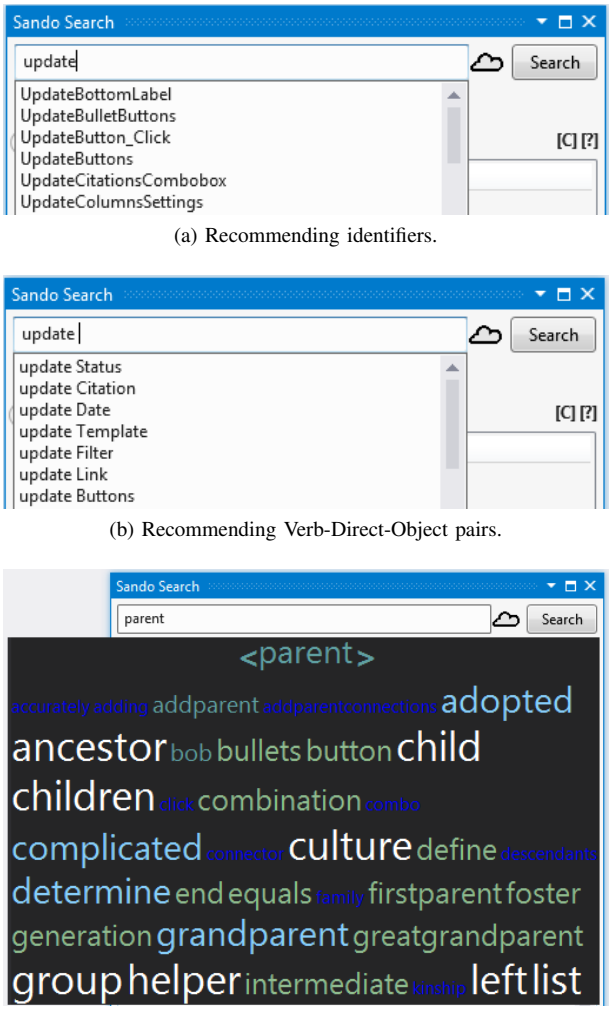
The method contains the following twelve terms: *path*, *manager*, *PathManager*, *create*, *CreatePathManager*, *has*, *extension*, *HasExtension*, *get*, *directory*, *name*, and *GetDirectoryName*. Therefore, for this method, the co-occurring pairs are the combinations of any two terms; that is, 144 $(12 * 12)$ in total. When the developer adds this method to the codebase, the term co-occurrence matrix increments the current value of each element corresponding to each pair by one, for example, elements of $\{$*create, path*$\}$, $\{$*path, manager*$\}$ and so on.

One caveat is that keeping the co-occurrence matrix in memory is inefficient. For instance, consider Sando itself, a project with about 10,000 lines of code and about 2000 terms. For such a project, the number of elements in the matrix is about 4 million $(2000 * 2000)$. To improve memory efficiency, we exploited the fact that the co-occurrence matrix is usually sparse; that is, according to our tool usage data, over 90% of the elements in the matrix contain the value of 0. Thus, we represent the co-occurrence matrix by using the Yale format, a data structure that efficiently stores sparse matrices without substantially impacting lookup and insertion speed [12].

3) Verb-Direct-Object repository: Another component in our Coronado technique is Verb-Direct-Object repository. Based on the observation that source code is often about performing actions on objects, Fry and colleagues proposed a technique that combines natural language processing and source code analysis [13]. Their technique extracts Verb-Direct-Object pairs from a codebase, such as "open file" or "close stream". By adopting this technique, Coronado periodically analyzes the codebase under search, extracting the Verb-Direct-Object pairs and caching them for future recommendations.

4) Software engineering thesaurus: Coronado also uses a thesaurus of terms that are frequently used in software development. Software engineering has developed its own set of terms, synonyms and abbreviations that do not exist in conventional English, such as "imap". Some software engineering terms have specific meanings that are different than their conventional English meanings, such as "class". Thus, simply reusing a general thesaurus cannot help interpret field-specific information, and can even be detrimental to client software tools [14]. To build a field-specific thesaurus, we reuse Gupta and colleagues' work that mines the relationship between different terms in source code and generates 1724 pairs of semantically related terms. Among these pairs, 91% are synonyms specific to the field of software engineering [14]. The examples of these field-specific synonyms include "execute"–"invoke", "load"–"initialize" and "instantiate"–"create". In addition to the related terms, their work also quantifies the commonality of the pairs of synonyms. For instance, "create" has been found to be related to several terms, including "make", "do", and "construct". However, "make" is more closely related to "create" than it is to "construct" or "do".

5) English thesaurus: The final component of Coronado is a general English thesaurus. To build this thesaurus, we reused Miller's lexical database WordNet [15]. WordNet represents related concepts as a graph, where nodes are words and synonym edges connect words with similar meanings. Thus, finding synonyms using WordNet takes constant time because Coronado simply looks for neighbors of a given word. Although WordNet obtains a complete set of English



(a) Recommending identifiers.

(b) Recommending Verb-Direct-Object pairs.

(c) Frequently co-occurring terms.

Fig. 2: UI of pre-search recommendations.

words, keeping the whole data set in memory is costly. Hence, we reduced the size of WordNet to include the $100k$ most frequently used words in English [16]. We speculate that the $100k$ words are sufficient for most queries, although how often a user queries with unusual words remains an open question because, for privacy reasons, we do not collect data about what terms Sando users are searching for.

B. Pre-search recommendation

We next describe the how Coronado uses these components. Coronado issues two types of recommendations: the pre-search ones and the post-search ones; the former type reminds the developer of the information that she may forget, while the latter type corrects erred queries issued from the developer. Whenever a developer types something into Sando's search box, Sando issues *pre-search recommendations*. These recommendations come from three sources: identifiers, Verb-Direct-Object pairs, and co-occurring terms.

1) Identifiers: The first source of the pre-search recommendations are identifiers. When the programmer types a string into the search box, Coronado looks through the codebase

terms component to determine whether that string is a prefix of any identifier in the local terms. Each identifier is then recommended to complete the developer's search string, and displayed in a drop-down menu below the search box. For example, Figure 2a displays what search terms are recommended to the developer when she types the string "update" when searching over the Family.Show codebase, an open source project that allows users to build family trees [17].

2) Verb-Direct-Object pairs: When the developer hits spacebar after typing a verb in the search box, Sando retrieves the Verb-Direct-Object pairs whose verb is the given search term, from the Verb-Direct-Object repository component. If Sando finds multiple Verb-Direct-Object pairs, Sando ranks them by using the co-occurrence matrix; the more often the verb and direct object appear together, the higher up the pair will appear in the drop-down box. The drop down menu in Figure 2b exemplifies the Verb-Direct-Object recommendations when the developer inputs "update" to the search box; because update co-occurs most often with Status than any of the other objects in the Verb-Direct-Object list in Family.Show, "update Status" is the most highly recommended search query.

3) Frequently co-occurring terms: The third type of pre-search recommendation suggests frequently co-occurring terms by using the term co-occurrence matrix. After inputting one or more terms in the search box, the developer can click the cloud button near the search box to show a tag cloud. The terms in the tag cloud are those that co-occur most frequently with the term or terms in the search box. The bigger the font size in the tag cloud, the more co-occurrences the term has with the terms in the search box. Figure 2c shows an example – this tag cloud presents terms that co-occur frequently with "parent" in Family.Show. The term "children" co-occurs more frequently with "parent" than "define" does. Clicking on a term in the tag cloud adds that term to the end of the original search query.

For frequently co-occurring terms, we chose to visualize the terms in a tag cloud, rather than a conventional drop-down list, like we used for identifier and Verb-Direct-Object recommendations. We made this choice because our early experiments suggested that for a given codebase, there are more co-occurring terms than either identifiers or Verb-Direct-Object pairs. Thus, because tag clouds use screen space more efficiently than drop-down lists, we used them for recommending term co-occurrence. In later discussion, we also refer to the frequently co-occurring terms as the tag cloud recommendation.

C. Post-search recommendation

In addition to helping the developer complete queries, Coronado also issues recommendations after the developer clicks the search button and no search results were returned. In this case, the developer's query contains terms that do not exist in the codebase. To help her compose a new query, Coronado uses the codebase terms, software engineering thesaurus, and English thesaurus to recommend semantically similar terms in the codebase under search. Supposing the original query has several space-separated terms, Coronado first queries the codebase terms component to check whether each term exists in the codebase. If a term does not exist in the codebase, and thus the search fails, Coronado uses the following three steps to generate a new query.

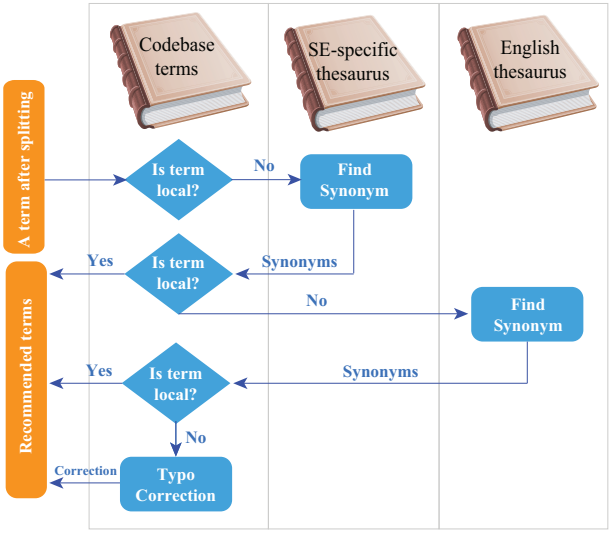


Fig. 3: Post-search recommendation.

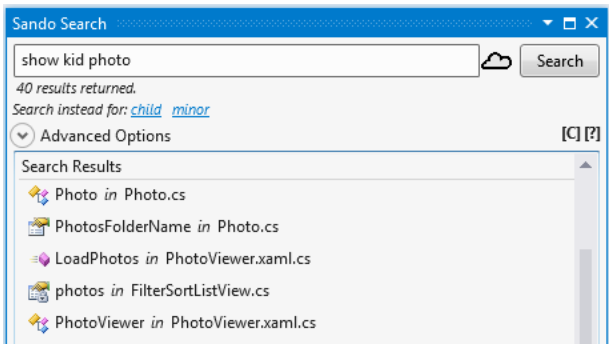


Fig. 4: UI of post-search recommendations.

1) Pre-processing: If a term does not exist in the codebase, some parts of the term might. Thus, Coronado greedily splits the term and incorporates the parts that exist in the local terms to the recommended queries. More specifically, for a given term in the query, Coronado tries to find its longest prefix and suffix that exist in the local terms, and in turn, the codebase itself. Next, the technique recursively splits the middle part of the term until no in-codebase prefix or suffix can be found.

For example, suppose the developer searches for "getelementname", which does not exist in the codebase. If the local terms contain "get" and "name", splitting the searched term leads to three new terms: "get", "element", and "name".

2) Synonym recommendation: After splitting, if the split terms do not exist in the local terms, Coronado tries to find synonyms for them in the thesauri. Coronado starts with the software engineering thesaurus due to its higher relevance than the software engineering thesaurus. If no synonyms in the software engineering thesaurus are found, Coronado tries to find synonyms in the English thesaurus. After retrieving the synonyms, the technique next excludes those synonyms that are not in the local terms. The remaining synonyms are recommended to the developer as the replacements to the original term. If finding multiple synonyms to recommend, Coronado uses the term co-occurrence matrix stated in Section II-A to

rank them. The synonyms that occur more frequently with the other terms of the input query rank higher. We illustrate the process of finding synonyms in Figure 3.

3) Typo correction: For those terms that neither exist in the local terms by themselves nor have synonyms in the thesauri, Coronado considers them as the typos of a term in the local terms. Therefore, Coronado uses these terms to correct them. The algorithm for correction adopts 2-gram indexing to quickly find the terms that spell similarly with a given typo [18].

More specifically, this correction algorithm first creates a correction table with 416 $(26 * 26)$ rows where each row is labeled by a pair of letters (from "aa", "ab", "ac" to "zz"). Next, for every term in the local terms, Coronado inserts the term to those rows in the table whose label is a part of the term. For instance, the term "example", assuming it is in the local terms, is inserted to the rows of "ex", "xa", "am", "mp", "pl" and "le". After inserting all of the local terms, Coronado is ready to use this table to correct a given typo by going through the following steps:

- For the typo, we first calculate the rows it will be inserted into as if the typo were a term.
- We next analyze the existing terms in these rows, and find out the term that shares the most common rows with the typo as its correction. If finding multiple such terms, we further select the one whose edit distance to the typo is the smallest as the correction to the typo.

After calculating the post-search recommendations, Sando displays the recommended queries under the search box; each query is a hyperlink, the click on which leads to Sando's searching the corresponding query, as illustrated in Figure 4.

III. Field Study

We conducted a longitudinal study to answer the following research questions:

- RQ1: How often do developers use Coronado's recommendations during their normal work to improve their queries?
- RQ2: Do developers seem satisfied with queries constructed from Coronado's recommendations?

A. Study Setting

We evaluated Coronado through a collection of anonymous usage data from Sando users in the wild. Sando automatically uploads usage data to S3 cloud storage [19] for users that give their permission. Coronado was instrumented to log two types of events, recommendation *review* and recommendation *acceptance*. More specifically, the events that we included in the log are:

- *The developer reviews a pre-search recommendation.* An event is written to the log when a user reviews a pre-search recommendation, either by highlighting a drop-down recommendation (i.e., highlighting a recommendation in Figure 2a) or displaying a tag cloud (Figure 2c).
- *The developer accepts a pre-search recommendation.* An event is written to the log when a developer accepts a pre-search recommendation, either by choosing a recommendation from the drop-down list or by clicking on a term in the tag cloud.
- *A user reviews post-search recommendations.* Each time a Sando user submits a query containing terms that do not exist in the codebase under search, Sando issues a set of post-search recommendations, displayed as clickable links (see Figure 4). Each time Sando displays these links, an event is logged.
- *The developer accepts a post-search recommendation.* After Sando issues post-search recommendations, the user may accept these recommendations by clicking on the hyperlink. Each time a link is clicked an event is written to the log.

In addition to the aforementioned events, Sando also logs events that apply to all queries. These events include:

- *The developer submits a query.* Each time a user searches using Sando, we log the number of results returned.
- *The developer examines a search result.* Sando provides two ways to examine a search result, via a preview pop-up summary of the result or by opening the relevant file in the code editor. We log both of these events.

B. Results

We released a version of Sando with Coronado recommendations to the Visual Studio Gallery site [20] and continuously gathered usage data for approximately 7 months. In total, we collected logs submitted by 274 unique Sando users[2]. All known Sando and the SrcML.NET developers were excluded from the data set (SrcML.NET is a service used by Sando).

As shown in part (b) of Figure 5, the majority of the users issued few queries, while several power-users issued upward of 200 queries using the tool. There was a steady stream of queries gathered over the time period and the dataset was not dominated by queries gathered over a short time span. As more developers downloaded Sando over the collection period, the number of collected queries steadily increased from month to month. Each developer issued a range of between 1 and 235 Sando queries, while the average number of queries per developer was 9.35 with a standard deviation of 20.98 queries, indicating that the data set was also not influenced by a small group of developers.

The remaining parts of Figure 5 serve as an overview of the results that are presented and discussed below in the context of RQ1 and RQ2.

RQ1. Use of recommendations. During the collection period, Sando users executed 2563 queries, 820 of which utilized recommendations (32%), as shown in Figure 5(a). We observe that the recommendation usage rate is fairly high,

[2]Data for a 25 day period, consiting of 44 separate users issuing 363 queries, is available at https://github.com/abbiss/SandoRecommendationStudyData/

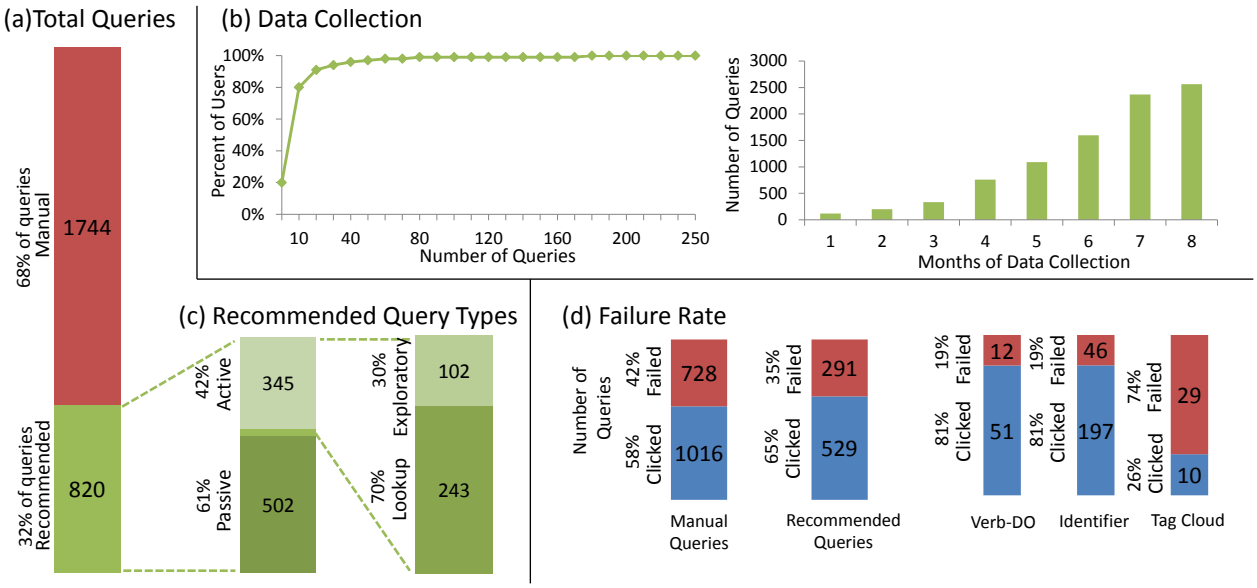


Fig. 5: Summary of results using Sando users' data, collected in the wild.

considering that developers are working on their own systems, that they are likely not accustomed to code search tools with recommendations, and that there was no in-tool documentation of the recommendation feature.

> Both the adoption rate of recommendations, used in about 32% of all queries, and the raw number of recommendations utilized, 820, indicate that recommendations are having a real impact on developers in the field.

We categorized the observed recommendations into two basic categories, pre-search and post-search. Developers used pre-search recommendations 345 times while they employed post-search recommendations 502 times (see Figure 5(c)). This indicates that developers actively sought recommendation usage in 42% of the recommended queries while they accepted recommendations after the query in 61% of the cases.

> The balance between pre-search and post-search recommendations suggests that both types of recommendations in Coronado are equally useful.

Of the pre-search recommendations (345 in total) 63 were Verb-Direct-Object pairs, 243 were identifiers, and 39 were from tag clouds. Heavy use of identifier recommendations is an indication of searches for the purpose of information lookup, where the developer likely has a specific part of the code (e.g. a specific method or class name) in mind. On the other hand, using the Verb-Direct-Object pairs and tag cloud recommendations are likely to be used in more exploratory searches, as exemplified in Figure 5(c), where the developer only vaguely knows what she is looking for.

> The distribution of recommendation types suggests that a substantial part (70%) of recommendations usage in the field is during exploratory searches.

RQ2. Quality of search results for recommended queries.

When investigating user satisfaction with a search query it is valuable to measure cases where searches completely fail, as they are a straightforward indicator of the inadequacy of the result set returned by the search tool. A large number of inadequate result sets translates to anticipated dissatisfaction of the users with the code search tool. In our data, we considered a search to have failed if there was no further interaction with the search UI (e.g., no clicks on any kind, single or double, on the results) after the query. During our collection period, of the 820 queries from recommendations 291 resulted in failure (35%) and of the 1744 manual queries 728 resulted in failure (42%), as shown in Figure 5(e).

> Recommendation-based queries received a click more often than manual queries.

Using failed queries as our metric, we can examine the effectiveness of each of the recommendation techniques that constitute our Coronado approach. Of the 63 queries where the users relied on the Verb-Direct-Object recommendation, only 12 (or 19%) were unclicked (i.e. failed) queries. Similarly, for identifer recommendations, 46 out of 243 (or 19%) queries failed. Tag cloud recommendations fared a lot worse, 29 out of 39 queries (or 74%) yielding no interactions with the result set. The post-search recommendation techniques in Coronado yielded a success rate of 42% (212 failed queries out of 502 issued), matching the success rate of manual queries.

> Identifier and Verb-Direct-Object recommendations were most successful when used by developers in the field.

C. Implications

Based on the data from the longitudinal study, several implications may help researchers improve the usability and the usefulness of query recommendation techniques for code search. We next summarize these implications.

Both the preventive and the corrective recommendations are important. As the results of RQ1 suggest, pre-search and post-search recommendations are equally useful, suggesting that developers need both reminders and corrections to compose effective queries. Existing query recommendation

techniques integrates no feedback loop from the retrieved results, thus they cannot correct the failing queries [8], [9].

Simply recommending identifiers in the codebase can reasonably assist developers. Recommending the identifiers in codebase may be the most straightforward technique we implemented. However, according to the collected data, developers adopted this recommendation more frequently than more sophisticated ones, suggesting that the basic program entity serves as an important clue for developers' retrieval of the relevant code snippets.

Verb-Direct-Object pairs are effective for exploratory searches. The low failure rate and significant adoption rate of Verb-Direct-Object pairs are especially encouraging as these recommendations, unlike identifier recommendations, occur on searches where the developer is unfamiliar with (a segment of) the codebase and does not have a specific program element in mind. Such exploratory searches can be more challenging for a code search technique and therefore the low measured failure rate for these queries should be considered a successful outcome for this technique.

The readability of the recommended queries matters. Different recommendation techniques lead to the varied readability of the recommended queries. For instance, aligning with how developers explain code snippets, the recommended identifiers and the Verb-Direct-Object pairs are more readable and understandable than the queries recommended from the other techniques. The difference explains the former two's higher adoption rate. As another example, the tag cloud recommendation, despite effectively capturing the related terms in the codebase, attracts less adoption due to the distance between the fragmented information and the developers' perception of the codebase.

Recommending queries followed by manual selection leads to more useful results than using either one alone. Existing query expansion techniques silently add terms to developers' original queries without explicitly interacting with the developers [8], [9], [10]. Our study shows that developers' explicit selection improves the usefulness of the recommended queries, leading to more promising results retrieved.

D. Threats to validity

Our log file analysis is potentially susceptible to several threats, both internal and external. One internal threat is how we measure user satisfaction of retrieved results. Conventional measurements of the quality of the retrieved results, such as precision and recall [21], cannot be collected by analyzing the log files. However, failed queries, or the clickthrough rate, which is the name for this metric in the Internet search community has consistently been shown to produce a reliable measure [22], albeit somewhat course-grained, of user satisfaction with a result set.

Externally, we collected data only from 274 users for a period of about 7 months. The results drew from the limited number of users and time length may be not generalisable to other Sando users for a longer period of time.

IV. Related Work

A large body of existing work is related to ours. In this section, we summarize them from two different themes, namely query reformulation and improving code search.

Query reformulation. In information retrieval systems, queries of higher quality can lead to more relevant results. The original queries provided by users may not be good enough; hence researchers proposed multiple ways to reformulate them. In general, query reformulation techniques either expand [8], or reduce [9], given queries. Specific to the field of software engineering, Haiduc and colleagues proposed a recommender called Refoqus that suggests query expansion by analyzing a set of attributes of given queries [23]. Semi-automatic query reformulation techniques integrate developers' feedback to improve the original queries. For example, De Lucia and colleagues proposed a technique that uses developers' feedback to incrementally discover traceability [24]. Coronado differs from all these techniques in that Coronado recommends related search terms and refine failed ones; that Coronado guarantees that the recommended terms lead to some results; and that Coronado uses software engineering-specific lexical information to find related terms.

Improving Code Search. To better assist developers retrieve relevant code snippets, researchers proposed various techniques. For instance, Yang and Tan leverage the context of words to mine semantically related terms in the codebase [25]. Ali and colleagues combine software repository mining results with information retrieval techniques to improve the accuracy of the later [26]. Sisman and Kak enrich developers' queries by injecting terms appearing in the artifacts drawn from their manual queries [10]. Bajracharya and colleagues proposed Structural Semantic Indexing that associates code snippets by the APIs they used, thus the developers can easily retrieve the usage examples of certain APIs [27]. Marcus and Maletic apply latent semantic indexing to recover the traceability link between documentation and source code [28]. Different from these works, this paper aims at evaluating the usefulness and the usability of existing techniques instead of proposing new ones.

V. Future Work

Future research can build on our work by improving and refining search recommendation techniques. For instance, extensions of this work could apply the degree-of-interest model to better rank recommended queries [29], using context information to promote search terms related to developers' recent activity.

Another way to improve the state of the art in recommendation techniques is to suggest queries to developers based on their colleagues' successful queries. Such a technique could leverage collaborative filtering using data from developers who work on the same codebase to recommend better and more relevant queries [30].

Before developing better recommendation techniques, conducting empirical studies of local code search tool users is helpful. Analyzing log files, as we did in our field study, can show usage patterns, but cannot uncover the causes of developers' behavior. For instance, we know that users use pre-search recommendations less often than post-search recommendations, but we do not know the causes without empirical study.

To enrich the study, we also plan to compare Sando with other search engines used by developers. For example, Brandt and colleagues studied developers' usage of online resources as well a web search engine that is integrated to IDEs [31], [32]; the comparison potentially leads to the insight of how developers behave differently when seeking information through various channels.

VI. Conclusion

Local code search tools help developers easily find the code they want to reuse or edit. However, composing the right queries can be difficult for developers who are not familiar with the codebase under search. To help developers compose queries that return relevant parts of the codebase, researchers proposed various query recommendation techniques. To investigate the usability and the usefulness of these recommendation techniques, we integrated several of them to the Sando search tool and conducted a longitudinal field study. We found that, in the field, developers issued 32% of all their queries by taking recommendations; preventive and corrective recommendations are equally useful; and that developers tend to adopt recommendations that are more readable.

Acknowledgment

This research was conducted during the first author's internship at ABB Research. We thank the participants for helping us conduct the experiments. We thank the help from Vinay Augustine, Patrick Francis, and Will Snipes. We also thank the comments from the Development Liberation Front group members Titus Barik, Michael Bazik, Brittany Johnson, Kevin Lubick, John Majikes, Yoonki Song and Jim Witschey.

References

[1] J. Singer, T. Lethbridge, N. Vinson, and N. Anquetil, "An examination of software engineering work practices," in *Proceedings of the conference of the Centre for Advanced Studies on Collaborative research*, 1997, pp. 21–36.

[2] 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 Transactions on Software Engineering*, vol. 32, no. 12, pp. 971–987, 2006.

[3] J. Sillito, G. C. Murphy, and K. De Volder, "Questions programmers ask during software evolution tasks," in *Proceedings of the international symposium on Foundations of software engineering*, 2006, pp. 23–34.

[4] "InstaSearch - Eclipse plug-in for quick code search," https://code.google.com/a/eclipselabs.org/p/instasearch/, 2013.

[5] D. Shepherd, K. Damevski, B. Ropski, and T. Fritz, "Sando: an extensible local code search framework," in *Proceedings of the International Symposium on the Foundations of Software Engineering*, 2012, pp. 15:1–15:2.

[6] "The Eclipse Foundation," http://www.eclipse.org/, 2013.

[7] "Microsoft Visual Studio," http://www.microsoft.com/visualstudio/, 2013.

[8] C. Carpineto and G. Romano, "A survey of automatic query expansion in information retrieval," *ACM Computing Surveys*, vol. 44, no. 1, pp. 1:1–1:50, 2012.

[9] N. Balasubramanian, G. Kumaran, and V. R. Carvalho, "Exploring reductions for long web queries," in *Proceedings of the international conference on Research and development in information retrieval*, 2010, pp. 571–578.

[10] B. Sisman and A. C. Kak, "Assisting code search with automatic query reformulation for bug localization," in *Proceedings of the Working Conference on Mining Software Repositories*, 2013, pp. 309–318.

[11] H. C. Wu, R. W. P. Luk, K. F. Wong, and K. L. Kwok, "Interpreting tf-idf term weights as making relevance decisions," *ACM Transactions on Information Systems*, vol. 26, no. 3, pp. 13:1–13:37, 2008.

[12] M. H. S. S. C. Eisenstat, M. C. Gursky and A. H. Sherman, "Yale sparse matrix package i: The symmetric codes," *International Journal of Numerical Methods in Engineering*, vol. 18, pp. 1145–1151, 1982.

[13] Z. Fry, D. Shepherd, E. Hill, L. Pollock, and K. Vijay-Shanker, "Analysing source code: looking for useful verb-direct object pairs in all the right places," *IET Software*, vol. 2, no. 1, pp. 27–36, 2008.

[14] S. Gupta, S. Malik, L. Pollock, and K. Vijay-Shanker, "Part-of-speech tagging of program identifiers for improved text-based software engineering tool," in *Proceedings of the International Conference on Program Comprehension*, 2013.

[15] G. A. Miller, "Wordnet: A lexical database for english," *Communications of the ACM*, vol. 38, pp. 39–41, 1995.

[16] "Frequent Word Lists," http://invokeit.wordpress.com/frequency-word-lists/, 2013.

[17] "Family.Show open source project," https://familyshow.codeplex.com/, 2013.

[18] K. Kukich, "Techniques for automatically correcting words in text," *ACM Computing Surveys*, vol. 24, no. 4, pp. 377–439, 1992.

[19] "Amazon S3 Cloud Storage," http://aws.amazon.com/s3/, 2013.

[20] "Visual Studio Gallery," http://visualstudiogallery.msdn.microsoft.com, 2013.

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

[22] M. Richardson, E. Dominowska, and R. Ragno, "Predicting clicks: Estimating the click-through rate for new ads," in *Proceedings of the International Conference on World Wide Web*, 2007, pp. 521–530.

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

[24] A. De Lucia, R. Oliveto, and P. Sgueglia, "Incremental approach and user feedbacks: a silver bullet for traceability recovery," in *Proceedings of the International Conference on Software Maintenance*, 2006, pp. 299–309.

[25] J. Yang and L. Tan, "Inferring semantically related words from software context," in *Proceedings of the Working Conference on Mining Software Repositories*, 2012, pp. 161–170.

[26] N. Ali, Y. Gueheneuc, and G. Antoniol, "Trustrace: Mining software repositories to improve the accuracy of requirement traceability links," *IEEE Transaction on Software Engineering*, vol. 39, no. 5, pp. 725–741, 2013.

[27] 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*, 2010, pp. 157–166.

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

[29] M. Kersten and G. C. Murphy, "Mylar: a degree-of-interest model for ides," in *Proceedings of the international conference on Aspect-oriented software development*, 2005, pp. 159–168.

[30] G. Linden, B. Smith, and J. York, "Amazon.com recommendations: Item-to-item collaborative filtering," *IEEE Internet Computing*, vol. 7, no. 1, pp. 76–80, 2003.

[31] 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," in *Proceedings of the Conference on Human Factors in Computing Systems*, 2009, pp. 1589–1598.

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