My Repository Runneth Over: An Empirical Study on Diversifying Data Sources to Improve Feature Search

Sukanya Ratanotayanon Hye Jung Choi Susan Elliott Sim
Department of Informatics
University of California, Irvine
Irvine, USA
{sratanot, hchoi7, sesim}@uci.edu

***Abstract*—**

Research on feature location that apply information retrieval techniques have experimented the kinds of inputs to the corpus and the algorithms that could be used. At first, only source code was used. Later extraction techniques were improved, and data from other software tools and analyses were used to expand or augment the repository. But, does having more diverse data in the repository always produce better results? In this paper, we report on an empirical study to examine the effect of increasing data diversity to improve feature location through search. In particular, we looked at the effect of including: i) change sets from revision control system, ii) tickets from issue trackers, and iii) elements from a Static Dependency Graph (SDG). We searched for three features of *Jajuk*, an open source Java jukebox, and two features of *jEdit*, an open source Java text editor. We used four different corpuses built with a combination of the above data. We used *Eclipse*'s code search and an index built with source code as baseline conditions. We found that it is not always better to have more diverse data. Adding SDG data to changesets increased recall, but drove down precision. Adding data from issue trackers had little effect and in one case lowered recall. We also found that large-scale refactoring of the code decreases the effectiveness using changesets for feature location.

Keywords-component; feature location; code search; program comprehension; change sets

I. Introduction

Software developers frequently perform searches on source code to help them find where a feature is located [20]. Unfortunately, there is a large gap between the problem description (which uses vocabulary from the problem domain) and the strings in source code (which uses vocabulary from the solution domain) [2, 9, 14]. Much work has been done in the area of feature location to address this problem. Feature location tools help developers to find where a feature is located in the source code.

A number of feature location techniques have been created. Many of them leverage techniques from information retrieval (IR), which deals with how to retrieve unorganized diverse data effectively and efficiently [5].These techniques are valuable, because feature location involves locating relevant information from a large body of source code. Research on feature location that apply IR techniques have varied both the kinds of inputs to the corpus and the algorithms used.

The kinds of data in the repositories for feature location have changed over time. At first, only source code was used, but later extraction techniques were improved, and data from other software tools and analyses were used to expand or augment the repository.

A number of approaches have been developed for turning source code and specifications into an indexed repository of documents [1, 12, 15]. More recently, changesets from revision control systems have been found to be helpful. A changeset contains information related to a commit made to a revision control system. It usually contains conceptual-level information that is difficult to find in the source code. More importantly, it provides explicit links from the conceptual description of a task to the implementation. Building on earlier success with IR techniques, changeset data was input into the repository instead of source code. In addition, this type of corpus can be improved using information from other sources, such as issue trackers [6], either as documents or metadata to improve indexes.

Results from static and dynamic analysis tools provide useful information about relationships among program elements [21]. For instance, relationships from a static dependency graph (SDG) can be used to expand search results with program units that are relevant, but are not part of a changeset.

The trend to include more data in repositories suggests that having more information may produce better results. In this paper, we report on an empirical study to investigate whether this is the case. In other words, does increasing data diversity necessarily improve feature location?

In our study, we built searchable repositories from different combinations of data sources: changesets, issue trackers and dependency graphs. We chose features from two subject systems, Jajuk (three features) and jEdit (two features). To evaluate the search results, we studied the software closely and manually created a set of authoritative implementation locations for each feature. The independent variable in the experiment was the type or types of data in the repository. This variable had four levels, each with a different combination of data, plus two control conditions. The four combination of data are: i) only changesets, ii) changeset plus dependency graphs, iii) change set and information from a tracker and iv) all information including changesets, information from trackers and dependency graph.

As our control conditions, we also performed the search for these features using Eclipse regular expression search and a searchable corpus built from source code. For our dependent variables, we used precision and recall.

Compared with our baseline conditions, the results show that the repository built with changeset provides results with better precision, but lower recall. However, augmenting the repository with the SDG yields comparable recall rates but still have better precision than baseline.

Among the various combinations of data sources, we found that adding information from trackers had little effect, and in one case lowered recall. The combination of the data sources that provided the best balance of the precision and recall was changesets with bug reports and feature requests. Adding information from the SDG on the other hand improved the recall at the cost of the precision. To use this type of information effectively, an accurate ranking mechanism is needed. We conclude that it is not always better to have more diverse data in a repository for feature location.

Our results also indicated that although changesets are helpful for locating features, but large-scale refactoring of the code limits their effectiveness.

The paper proceeds as follow: Section 2 reviews previous work in feature location on including various kinds of data in the repository. In Section 3, we give an overview of the approach used in our study. Section 4 presents our research questions. Section 5 describes the feature location platform that we used in our study. Sections 6 and 7 detail the study design and results, followed by discussion and threats to validity in Sections 8 and 9. Future work and conclusions are given in Section 10 and 11 respectively.

II. Improving Feature location with Diverse Data

To comprehend a program, developers need to know where features that they are interested are located in source code. A feature is sometimes scattered or tangled in source code [3, 7] so locating a feature is not easy but essential to understand source code. Searching source code with conceptual keywords only provides a limited support for locating program elements relevant to a feature. Vocabulary from the problem domain is not usually present in the source code [2, 9, 14]. Although the vocabulary sometimes appears in comments, they are not present in all relevant locations.

A number of feature location techniques have been created using techniques from information retrieval (IR), which have been created to deal with the problem of retrieving information from large collections of unorganized data [5]. Initially, these repositories tended to be populated using only source code and the matches returned needed only to be lexically close to the terms in the search specification. Advances involved the creation of novel techniques for creating the search corpus from source code and specifications [12, 15].

Information from additional sources improved searching for features with conceptual keywords. We focus on data from two sources: software tools in use on the project and inferences from analysis tools. Leveraging data from software tools, such as issue trackers and revision control systems, is effective because this information is readily available and does not place a burden on the developers. In particular, we are interested in the following three data sources for improving feature location: changesets, data from an issue tracking system, and static dependency graphs.

More recently, research found that using changesets from revision control systems has proven to be helpful. A changeset is the record from a single commit transaction to the revision control system, consisting of the names of the files that have been changed, the changes to the files, and a commit comment. Consequently, a changeset can be used to provide explicit links between domain concepts and lines of code [4], or links between program elements [16, 23]. These links can be effective for locating relevant program units. Several tools take advantage of commit comments in revision control systems so that software developers could look for source code using conceptual keywords. Some researchers used machine learning techniques to find patterns by mining change patterns from commit transaction and predict likely changes of relevant program elements [23]. Building on the success of IR approaches, changeset data can be used as documents for indexing instead of source code. In addition, this type of corpus can be improved using information from other sources.

Data from issue trackers have also been used to help developers understand code. Many software projects adopt an issue tracking system to keep track of bugs or feature requests. The items in these tracker systems provide detailed description of features at the conceptual level as opposed to a short description commonly provided in the changesets. What makes this type of information useful is that developers often provide a link from a bug report or feature request to the changeset implementing it. Therefore, we can use this information to provide even more details to sections of code using a combination with changesets. Some tools that have utilized bug reports or feature requests to supplement their corpus for conceptual-level searching include Hipikat [6] and ConcernTagger [8]. Bug reports or feature requests have also been used to identify features that developers should be interested in and to create queries for searching [15, 17].

Finally, software analysis tools can provide information about program elements. While the previous two types of information are good sources of metadata, they do not increase the completeness of the matches returned. A changeset may miss program units that were related to the task, but were not modified or may contain only a facet of the feature that was touched by the task. We need a mean to extend program elements identified as relevant to other related program elements. A static dependency graph, containing program dependency information among program elements, can be used for this purpose. Program elements implementing the same feature are most likely to have relationships and depend on each other. Therefore, we should be able to discover missing program elements using their dependency relationship with the program elements that are already found. Previously, static analysis has been successfully combined with IR techniques for feature location. Examples of successful tools are: SNIAFL [22], JQuery [11], and JIRiSS [13].

III. Research Questions

There are several assumptions made in the usage of information from data sources presented in the previous section to improve feature location. The main assumption is that information from these data sources would improve the connection between conceptual keywords and source code. Therefore, allow more relevant results to be returned when searched with conceptual keywords.

To evaluate these assumptions, we performed an empirical study by searching different features using search corpuses enhanced with information from these data sources. The specific questions that we aim to investigate are as followed:

- *Does increasing data diversity improve feature location?*
  Information from different data sources improves search mechanism in different aspects. Adding information from changesets and tracker items to the search corpus improves the possibility that the keyword will be matched to relevant members. Using static dependency graph, we can retrieve more relevant program elements using their relationships. Therefore, we expected that adding more types of information to enhance the search mechanism would result in more completed and relevant results returned when searching with conceptual keyword.
- *What combination of data provides better results?*
  Because different data sources improve the search in different ways, the combination of them may yield different results. We aim to evaluate which combination of data source provides the best results when used together.

IV. Feature Location Platform

To perform the evaluation, we used a prototype search platform, Kayley, to search for various features in subject software systems. The two main features of Kayley are: 1) the ability to create a searchable repository from changesets; and 2) the ability to enhance the repository with diverse data from arbitrary sources. These features allow us to create search corpuses for a subject software system using different combinations of data sources. In this study, we incorporated the information from an issue tracker items and a static dependency graph to the searchable repository of changesets. The features of Kayley and its usages are discussed in detail below.

A. Creating a searchable corpus of changesets

The prototype retrieves changesets by importing the commit history of a software system from Subversion (SVN). To create a search corpus, instead of using a source file as a document, each changeset is treated as a document. We used Jakarta Lucene[1], a high-performance, full-featured, Java-based text search engine, to index the changesets. Although the changesets associates concepts to lines, Kayley returns the enclosing program element. The index of the corpus is built using the following information: comments of each changeset, author, creation date, and signatures of associated program elements. The indexed information is tokenized in lower case, and reduced to the word root before being indexed.

B. Adding diverse information to the search corpus

Kayley can incorporate various combinations of data from diverse providers. In this study, we included two types of data: information from a tracker tool and a static dependency graph.

Kayley uses information from trackers as additional domain level vocabulary. The additional, longer description can help improve the accuracy of the index. Information from the tracker is input as an XML file, and where possible Kayley will match the track identification number with the commit comment in the changesets. When a match is found, the description of the tracker item will be added to the changeset and indexed along with other metadata.

The static dependency graph can be used to expand the set of relevant program elements. Kayley checks each method in a changeset, and only expands the ones that exist in the current version of the code. For each of these filtered methods, a static dependency graph is used to discover the following program elements.

- All callee methods in the subtree of the dependency graph rooted at the method being checked. Callee methods from binary files are excluded as they usually come from libraries.
- Definition of public fields that are used by each method identified.

The indexed changesets are then extended to associate with these program elements.

C. Using the Platform

The first step is to create a search corpus of changesets by providing a SVN repository path of the project to Kayley. When creating a search corpus, a user can specify additional data to be incorporated as discussed above.

Kayley is intended to be used interactively. The user can query Kayley using keywords describing the feature. Kayley will present the user with the top 50 changesets that match the keyword. At this point, the user needs to examine the returned changesets and select a set of relevant ones. The program elements associated to the selected changesets will be validated for their existence in the code. The program elements existing in the current version of the code will be returned as search results.

V. Design of The Experiments

This section discusses the design of our experiment to investigate whether increasing data diversity in a searchable

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

repository improves feature location. We performed feature location for three features of Jajuk and two features of jEdit using indexes built with four combinations of data sources. The methodology of our study is described in subsection A. The subject systems and features that we are interested in are presented in subsection B.

A. *Method*

1) *Treatment and Baseline Conditions*

In our experiment, we used six treatment conditions to assess which condition produced the best result. There are four combinations of data sources and two other tools used as baseline conditions.

- *Changesets (C1)*
  In this condition, the index that we used was built with only changesets. For each subject system, we created an index using changesets from their respective version histories using Kayley.
- *Changesets + SDG (C2)*
  We augmented the index from C1 with the static dependency graph so that program elements that do not appear in the change sets will also be returned.
- *Change sets + bug reports + feature requests (C3)*
  We used a crawler to gather bug reports and feature requests from the issue trackers and added these to the repository from C1.
- *Change sets + bug reports + feature requests + SDG (C4)*
  This condition includes every data source used previously.
- *Regular expression search using Eclipse IDE (C5)*
  This is our first baseline condition. Text search is the most common way that developers search source code. We conducted the search using regular expression search in Eclipse IDE.
- *Source Code Only (C6)*
  This is our second baseline condition. A number of previous approaches have built the repository using source code rather than change sets. We used the FLAT[3] platform[2] by Savage et al. [17] for these searches.

2) *Performing the Searches*:

A search for every feature was performed under every condition. We input into Kayley the same search specification for a particular feature. From the changesets that were returned, the two first authors agreed on which were the relevant ones. These changesets were then used to retrieve program elements that implemented the desired feature.

In C5, the search specification was entered into Eclipse. This search returned results at the line level, so we recorded the enclosing program elements.

In C6, search specification was entered into the FLAT[3] platform and all the returned program elements were recorded.

[2] http://www.cs.wm.edu/semeru/flat3/index.html

B. *Subject Systems*

For our evaluation, we chose two subject systems, Jajuk and jEdit. We chose them because they were moderate sized Java projects, with revision control histories and feature tracking data. They have also been used previously in empirical studies.

Jajuk[3] is an open-source music player, which consists of 471 classes, 1,346 methods and 53,097 LOC in total. The version that we used is 1.8.3 with the revision number 5571. Robillard et al. previously conducted an experimental study using Jajuk and created features (concerns) and their mappings using ConcernMapper [16]. We used their study and their features as a guideline for ours.

jEdit[4] was selected to compare results with Jajuk. jEdit is an open-source text editor, which consists of 836 classes, 5,154 methods and 98,662 LOC in total. We used version 4.3.1, revision 17000. jEdit has been frequently used as a subject system due to its size, popularity, and active community. We chose two of the features that Revelle and Poshyvanyk [14] investigated.

C. *Features Included in the Study*

1) *Jajuk Features*

We selected three features of Jajuk: Add a song, Shuffle mode and Sort collection. These features were previously examined in a study performed by Robillard [16]. The descriptions of these features are as followed

a) *Add a song (F1)*

There are several ways to add a song in Jajuk. For our study, we only focused on adding by dragging and dropping. After some experimentation, we decided to specify the search as “drag drop playlist.” Initially, we tried “add song,” but this retuned too many results Our final search specification provided a more manageable number of results. For Eclipse regular expression search, we used the regular expression keyword, “drag*drop*playlist.”

b) *Shuffle mode (F2)*

In shuffle mode, the songs are played in a random order. There are three ways to turn on this feature: selecting the Shuffle Mode option from the menu, pressing the Shuffle mode icon, or using Ctrl-h shortcut. We chose “shuffle mode” as keywords for the search. For the Eclipse regular expression search, “shuffle*mode” was used.

c) *Sort collection (F3)*

This feature allows users to sort their entire music collection according to different parameters, such as Album, Genre, Artists, Year, Discovery date, Rate, and Hits. A user can access this feature by using a “sort by” dropdown list in the Track Tree panel. We chose “sort collection” as the search specification. For the Eclipse condition, we used “sort*collection”.

A summary of our keywords for the features of Jajuk is presented in Table I.

[3] http://www.jajuk.info/index.php/Main_Page
[4] http://www.jedit.org/

TABLE I. FEATURES AND KEYWORDS FOR JAJUK

Feature	***Description***	***Keywords***
F1	Add a song	drag drop playlist / drag*drop*playlist
F2	Shuffle mode	shuffle mode / shuffle*mode
F3	Sort collection	sort collection / sort*collection

2)jEdit Features

We selected two features from jEdit: Thick Caret and Edit History. Similar to the features of Jajuk, these were used in a previous study performed by Revelle and Poshyvanyk [14].

a) Thick Caret (F4)

This feature allows a user to enable a thicker caret (or cursor) in the text area. We defined the scope of this feature limits to an option to enable thick caret and the operation to paint thick caret in the content area of jEdit. Although the term "thick cursor" was more familiar to us, we used the keyword "thick caret" for this feature to follow the idiom used by developers on the project. Under C5, we used a regular expression "thick*caret" for Eclipse regular expression search.

b) Edit History (F5)

This feature allows users to edit the history of previous searches in "Find" dialog box. We limit our scope of the feature to the history popup and actions to load and display the history list. We did not include the ability to save a search history as part of this feature. The keyword for searching for F5 is "edit history" for conditions C1 through C4 and "edit*history" for C5.

TABLE II. FEATURES AND KEYWORDS FOR JEDIT

Feature	***Description***	***Keywords***
F4	Thick caret	thick caret / thick*caret
F5	Edit history	edit history / edit*history

D. Oracles

We needed an oracle so that we could evaluate the experimental results and to calculate precision and recall of each result. Two researchers studied the software closely and manually created a set of authoritative implementation locations for each feature.

The first two authors individually studied the subject systems and determined relevant program elements for each feature. Subsequently, they reconciled their solutions and came to a consensus on the gold standard. This gold standard, the list of program elements, is used as an oracle to determine the relevancy of retrieved results. The total number of relevant program elements per features in this authoritative list is presented in Table III.

TABLE III. THE NUMBER OF PROGRAM ELEMENTS OF EACH FEATURE

	F1	**F2**	**F3**	**F4**	**F5**
Oracle	7	17	18	13	20

When creating the oracle, we did not limit ourselves to a specific number of relevant program elements for each feature. However, the number of relevant program element per feature is relatively small because we limited them to ones that are considered necessary. Also, both researchers needed to agree to put a program element on the list. The two solutions initially had an inter-rater reliability score of Cohen's $\kappa = 0.99$.

E. Variables and measures

Independent variables of our study are the conditions (C1 to C6). The conditions represent different combination of data sources.

The dependent variables are precision and recall, two metrics from IR, to provide a high level characterization of performance. Precision is the proportion of the total number of elements retrieved that are also relevant. Recall is the proportion of relevant elements that are also retrieved.

$$precision = \frac{|\{relevant\_documents\} \cap \{retrieved\_documents\}|}{|\{retrieved\_documents\}|}$$

$$recall = \frac{|\{relevant\_documents\} \cap \{retrieved\_documents\}|}{|\{relevant\_documents\}|}$$

VI. RESULTS

We found that using different data sources for building search index led to different characteristic of returned results. Table IV and Table V present the precision and recall of each condition of Jajuk and jEdit respectively. For conditions C1 to C4, performance on jEdit was much higher than performance on Jajuk. Further inspection revealed that diminished performance on Jajuk could be attributed to a refactoring that was applied to the source code. We will discuss this issue in detail in the discussion section.

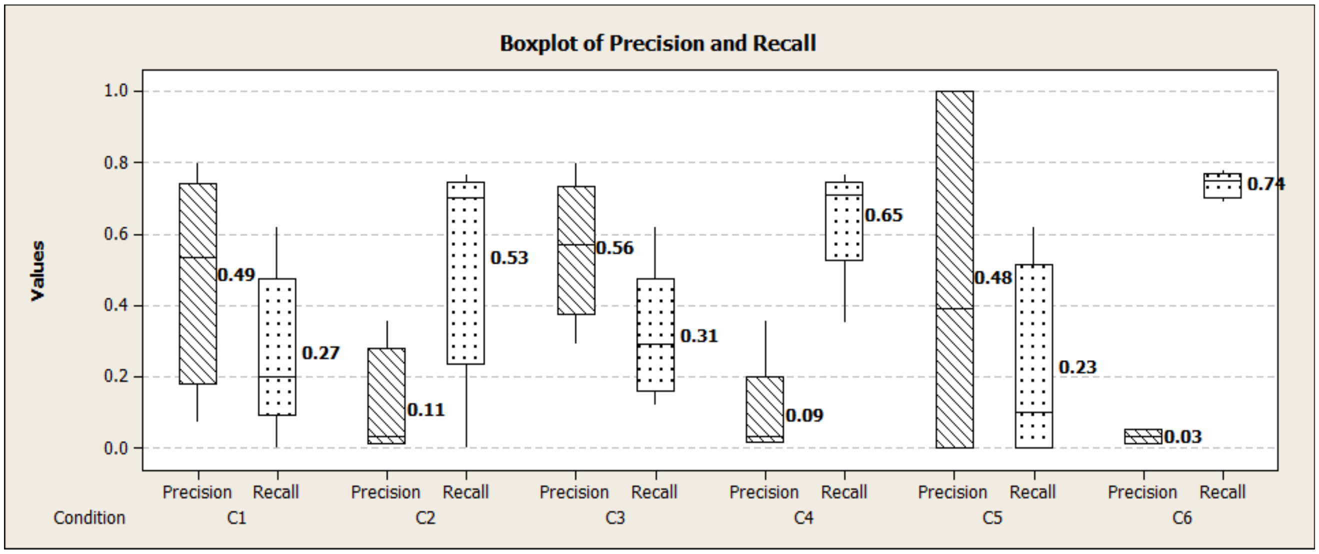

Figure 1. Boxplot of Precision and Recall of both Jajuk and jEdit.

Compared with our baseline conditions, the results show that a repository using changesets can return better search results than regular expression search or a repository using source code in some cases.

TABLE IV. PRECISION AND RECALL OF JAJUK.

	F1		F2		F3		Average	
	P	***R***	***P***	***R***	***P***	***R***	***P***	***R***
C1	N/A	0	0.1	0.18	0.5	0.33	0.29	0.17
C2	N/A	0	0	0.47	0.04	0.72	0.03	0.4
C3	0.29	0.29	0.7	0.12	0.46	0.33	0.47	0.25
C4	0.02	0.71	0	0.35	0.04	0.72	0.03	0.59
C5	0	0	0.4	0.41	0	0	0.13	0.14
C6	0.01	0.7	0.05	0.77	0.05	0.78	0.03	0.75

Note: ***P*** represents a rate of precision, and ***R*** represents a rate of recall.

TABLE V. PRECISION AND RECALL OF JEDIT

	F4		F5		Average	
	P	***R***	***P***	***R***	***P***	***R***
C1	0.8	0.62	0.57	0.2	0.69	0.41
C2	0.01	0.77	0.36	0.7	0.19	0.74
C3	0.8	0.62	0.57	0.2	0.69	0.41
C4	0.01	0.77	0.36	0.7	0.19	0.74
C5	1	0.62	1	0.1	1	0.36
C6	0.03	0.69	0.01	0.75	0.02	0.72

The distribution of precision and recall for each condition, aggregated over all the features, are illustrated in Figure 1. The vertical boxes show the interquartile range of the metrics, while values outside this range are shown using vertical lines centered on the box. The horizontal line inside the box depicts the median.

For C5, the results show that the effectiveness of regular expression search depends on the search terms that exist in the code. Regular expression search worked well with F4 “Thick Caret,” but could not retrieve anything for F1 “Add Song” and F3 “Sort Collection,” because the keywords in the specified patterns do not exist in the code. The augmented corpuses in C3 and C4 are more effective than using regular expression search.

In C6 (search using a repository of source code in FLAT[3]), recall is generally higher and the precision is lower, meaning that more of the relevant program elements are retrieved, but so are a lot of irrelevant ones. On average, the results from C2 and C4 have comparable recall values, but higher precision, meaning that they retrieve approximately the same proportion of the relevant program elements, but fewer irrelevant ones. This tendency is stronger for jEdit than for Jajuk. In contrast, C1 and C3 have much higher precision, but many program elements are missing. Finally, performance on C6 is more predictable, as indicated by the short boxes in the plot.

Repositories populated with changesets can be augmented with data from other sources as well. We will now look the effect of including diverse data sources.

C1 and C3 tended to return a small number of results, which led to lower recall, but higher precision than the other conditions. These two combinations were based primarily on the changesets, and did not include data from the SDG.

Using SDGs helps in retrieving the missing members, as seen in the improvement of recall of C2 and C4 when compared with their counterparts C1 and C3. This result suggests that with a good ranking algorithm, this information can be very useful for augmenting a searchable repository.

On the average, the information from the issue trackers yields small improvement in both the precision and recall of the results. In C3, we augmented the changeset index with only information from tracker. This condition shows the best overall precision and recall, without retrieving a large number of program elements. Table VI shows the number of returned program elements, and the number of returned and relevant program elements for each search.

TABLE VI. EXPERIMENTAL RESULTS OF JAJUK AND JEDIT

	F1		**F2**		**F3**		**F4**		**F5**	
	RE	***CO***	***RE***	***CO***	***RE***	***CO***	***RE***	***CO***	***RE***	***CO***
C1	0	0	41	3	12	6	10	8	7	4
C2	0	0	329	8	306	13	1009	10	39	14
C3	7	2	3	2	13	6	10	8	7	4
C4	207	5	177	6	306	13	1009	10	39	14
C5	2	0	18	7	2	0	8	8	2	2
C6	376	5	273	13	285	14	334	9	2208	15

Note: ***RE*** stands for Retrieved, and ***CO*** stands for Correct.

The results show that using dependency graphs (C2 and C4) and source code (C6) in the repository consistently retrieved a large number of program elements. This occurred with both subject systems. Compared to the small number of elements in the oracle, the results from these conditions can be overwhelming and potentially unusable. Although they help to retrieve missing members, they also add the problem of sifting through the results to find the most relevant ones. This problem points to the need for accurate ranking algorithms to help bring attention to the relevant results.

These results suggest an answer to our initial research question. Increasing data diversity in the searchable repository does not necessarily improve feature location. Sometimes, including additional data comes at a cost. When using SDGs, the improvement in recall values comes at the price of lower precision.

VII. DISCUSSION

In this section, we explore a number of unexpected effects in the data: the choice of keywords in searches, including data from issue trackers, and the effect of refactoring on using change sets for feature location.

A. *Searching with Alternate Keywords*

When a developer is not familiar with the vocabulary used on a software project, she might refer to the feature with other synonyms. In this study, we encountered this situation with F4, "Thick Caret." Caret is used in jEdit to represent the blinking line marker that shows where the next character will be inserted in the text area. However, the term "cursor" is much more common when discussing such a marker. It is possible that a developer may use the keyword "cursor" to search for the feature instead of "caret".

We performed another search for F4 to examine how well the additional data help in the case that synonyms are used to search for the same feature. We used the term "thick cursor" in this search. The result is shown in Table VII.

TABLE VII. SEARCH RESULT OF F4 USING ALTERNATIVE KEYWORDS

	"thick caret"		**"thick cursor"**	
	Precision	***Recall***	***Precision***	***Recall***
C1	0.8	0.62	1	0.62
C2	0.01	0.77	0.05	0.77
C3	0.8	0.62	1	0.62
C4	0.01	0.77	0.05	0.77
C5	1	0.62	0	0
C6	0.03	0.69	0.19	0.62

The new search specification, "thick cursor", improved the results from conditions C1 to C4. In contrast, regular expression search (C5) did not return any results, because the cursor was called caret throughout the code. Also, the term was not used near the term "thick" as we specified in the regular expression pattern. However, the term "cursor" was used when providing commit comments. This shows that one way additional data sources help with searches is by contributing alterative keywords that are not in the code.

It is interesting that for the condition C6 (IR over source code), the performance using both sets of keywords was similar. Recall was slightly worse, but the precision was improved. Because the term "cursor" appears infrequently in the source code, the new query produced fewer results, while still retrieving relevant program elements containing the term "thick."

We find this result encouraging, because it suggests to us that using changesets in the repository is a promising approach. It allows users to search with familiar vocabulary from the problem domain, rather than limiting them to vocabulary from the solution domain. In other words, developers do not need to know exactly what they are looking for in order to search for it.

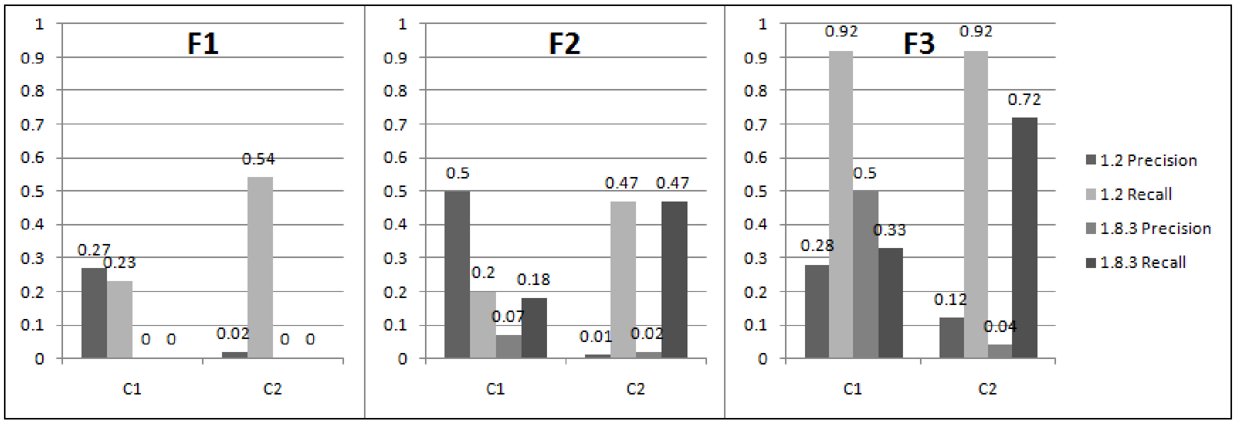

Figure 2. Comparing Precision and Recall of Search Results at the Version 1.2 and the Version 1.8.3

B. *Only Small Improvements from Tracker Information*

Overall, including information from issue trackers led to only small improvements in search performance.

We had expected that including data from issue trackers would always return more results, meaning better recall. Because we used tracker information as additional metadata, searches in condition C3 would retrieve the same program elements as searches on C1 plus others. However, this was not the case, due to the high quality of commit comments.

This lack of impact can be seen by comparing the results from the condition C1 to C3 and C2 to C4, as shown in Table IV. Here, we focus on comparing C1 and C3, because the other pair has similar properties. Further inspection of changesets returned by our queries in condition C1 versus C3 showed that most of the relevant changesets are returned in both conditions. Therefore, the performances of both conditions are similar. The reason for this is that developers on both Jajuk and jEdit diligently comment their commit transactions. Although the comments provided are short, they describe the task and the feature touched by the tasks with the same keywords that we would use for searching for the feature. Therefore, any changeset with an associated work ticket number would be returned with or without the aid of additional metadata.

Most of relevant changesets we selected from the query results for both conditions are the same. Kayley returns only the top 50 results, so some changesets that were returned in one condition were not present in the other. However, these different changesets have little effect in improving or worsening the recall of results in both conditions.

C. *Effect of Large Scale Refactoring*

Refactoring is a known problem for program comprehension tools and the usefulness of the information that they provide. We had an opportunity to observe this problem in action with feature F1 “Add Song” from Jajuk.

Not surprisingly, we found that refactoring has a negative impact on the effectiveness of using changeset information for feature location. In this case, refactoring modified the identifiers of program elements associated with the change set or removed program elements entirely. Consequently, these program elements were not returned by Kayley.

Searches for program elements implementing feature F1 returned no results, as was shown in the Table VI. This result startled us, because the query returned several changesets with highly relevant description. For example, one of changesets returned was “Revision 233: Drag and drop in playlist editor and some methods renamed”. When examining the returned changesets, we also found that the program elements associated with them seemed relevant.

Further inspection of the commit history shows that the class handling the drag and drop events for the F1 features were refactored in previous versions. The selected changesets were associated to the refactored program elements. Therefore, the links to these program elements were not useful because they do not exist anymore.

To investigate how much refactoring affected the search results, we performed an additional experiment on Jajuk’s features using a version before the refactoring. The comparison of results between the two versions is shown in Table VIII. We selected version 1.2, because it was used in a previous study by Robillard et al. [17]. Using the same procedure as before, we created search corpuses for conditions C1 and C2 using changesets up to version 1.2, and created new oracles for the features. Conditions C3 and C4 were omitted as they provided similar results.

From the results shown in Figure 2, we can see that the precision and recall were worse in the version 1.8.3, the one used in the main study. We found that for all features, there were changesets with a highly relevant description, but contained relevant refactored program elements, especially F1 and F3. Therefore, searches on version 1.2 performed much better because these program elements still exist in that version of the code. The results for F2 in the two versions are comparable, because there was a changeset post refactoring that touched the implementation of the feature. This changeset provided links to the new set of relevant program elements.

TABLE VIII. COMPARING PRECISION AND RECALL OF SEARCH RESULTS AT THE VERSION 1.2 AND THE VERSION 1.8.3

		Version 1.2		**Version 1.8.3**	
		Precision	***Recall***	***Precision***	***Recall***
F1	***C1***	0.27	0.23	0	0
	C2	0.02	0.54	0	0
F2	***C1***	0.5	0.2	0.07	0.18
	C2	0.01	0.47	0.02	0.47
F3	***C1***	0.28	0.92	0.5	0.33
	C2	0.12	0.92	0.04	0.72

All three of our feature searches were affected by refactoring, which shows that refactoring to existing features is common, especially for older features. Refactoring usually affects many features, so the ensuing changeset touches too many parts of the code to be useful. In addition, when refactoring takes place, the comments specifying which features were touched tend not to be provided. If there is no task touching the feature after refactoring is done, the links to refactored program elements are usually lost. In order to use changeset information effectively, additional algorithms are needed to trace evolution and refactoring of program elements.

VIII. THREATS TO VALIDITY

Threats to the external validity of our study come from the characteristics of our subject systems and work practices of the developers on the projects. The ability to generalize the results depends on how representative are the subject systems, jEdit and Jajuk, of other software systems.

Both jEdit and Jajuk are well-known open source software projects. Their characteristics and work practices are typical of active, popular, open source projects [19]. The size of software, frequency of commits, comments provided to each commit and usages of issues tracker are representative of open source projects. In addition, both projects have been frequently used in previous research in feature location [15, 17]. Using these systems allows us to compare our result to theirs.

Threats to internal validity include how the features were selected and how oracle for each feature was created. The selection of feature could be biased in favor of searches using certain tools or combinations of data. To mitigate this threat, we selected features that have been used in previous studies. The researchers who conducted those studies could not have anticipated our study, so no bias is possible. Again, the advantage of using these concerns is that we can compare our result with existing data.

Lastly, the creation of oracle of relevant code for each feature can affect the result of our study because the relevancy of program element to a feature depends on expert judgment. To mitigate this threat, two researchers independently created oracles, and discussed them until they could agree on each member to be included or excluded. The researchers spent time studying the running system and source code, so that they could make an informed decision when judging relevance of program elements. We achieve a high rate of inter-rater reliability, indicating that the oracles are trustworthy.

IX. FUTURE WORK

Our results point to a number of directions for future work. We discuss three of them here.

We would like to conduct further experiments with using various combinations of data. We obtain high recall and low precision when we use IR over source code, but when using changesets we obtain low recall and high precision. Perhaps we could find a way to use the results from searching on changesets to filter or reduce the matches provided by an index built with source code.

Work is needed to find a more effective ranking algorithm for the matches that are returned by a search. Currently, we use the default Lucene ranking, which uses TF/IDF [10]. The results from including data from SDGs and issue trackers indicate that we can do better.

Adding the data from issue trackers decreased recall in one case, because some good matches no longer appeared in the top 50. An SDG helps to locate more relevant program elements, but also tend to retrieve too many elements. Being able to rank returned elements accurately would help reduce the effort required by developers focus on the best ones.

A good ranking algorithm would order search results so that more relevant and important program elements are placed higher in results. A known problem in ranking is that commonly used program elements are ranked higher. For example, library or utility modules, user interface classes, driver classes, or getter/setter methods appear high in the list because they are called frequently, not because they are more relevant. Providing weights for ranking and filtering is difficult, because each package or component is often tangled with a lot of different source code with different purposes. Further research on efficient ranking algorithm is so necessary that we can provide better results to developers.

Work is also needed to both study and mitigate the effects of large scale refactoring. We found that large scale refactoring in Jajuk limited the effectiveness of searching over changesets. We would like to conduct further studies of refactoring effect to better understand the extent of this phenomenon. In other words, how many other projects are affected by large scale refactoring and by how much? How does this influence our ability to use changesets for feature location? Depending on the extent and severity of the problem, it may be necessary to develop additional tools and algorithms to automatically update the locations referred to by changesets.

X. CONCLUSION

A central idea in feature location is to minimize the gap between conceptual problem description and formal source code and to aid developers' comprehension of a software system. In this paper, we report on an empirical study on the effect of including different combinations of diverse data sources for feature location. We attempted to locate five features from two subject systems, Jajuk and jEdit under six conditions using various data sources, such as changesets, bug reports, feature requests, and SDGs. Each feature under different condition was tested using our feature location platform. We used regular expression search and feature location using IR techniques on source code (via FLAT^3) as baseline conditions for comparison.

We found that using changesets to populate the repository tended to provide higher recall, but lower precision in comparison to using source code. We also found that precision and recall when searching on changesets was more consistent and flexible than regular expression search and IR on source code.

Changesets are helpful for locating features by providing more conceptual-level information, but historical records on changesets tell us that large-scale refactoring of the code severely limits their effectiveness because a lot of refactored source codes do not exist in the current version of the source code, so developers cannot locate features.

We learned that including more diverse data does not always result in better performance. Adding data from SDGs to changesets drives down precision, but increases recall. Adding data from issue trackers did little to improve performance and in one case actually lowered performance.

We conclude that more data diversity is not always better and it is possible for a repository to runneth over. We plan to find ways to improve on algorithms for ranking matches, in order to find a happy balance between precision and recall.

Acknowledgment

Special thanks to Rosalva E. Gallardo-Valencia for her assistance in preparing this manuscript, to Phitchayaphong Tantikul for writing a web crawler, and to Roy Tiburcio, Derek Raycraft and Eugen Nistor for their valuable advice in implementation.

References

[1] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo, "Recovering traceability links between code and documentation," *IEEE Transactions on Software Engineering,* vol. 28, pp. 970-983, 2002.

[2] T. J. Biggerstaff, B. G. Mitbander, and D. Webster, "The concept assignment problem in program understanding," in *Proceedings of the 15th international conference on Software Engineering* Baltimore, Maryland, 1993, pp. 482-498.

[3] G. Canfora and L. Cerulo, "How Crosscutting Concerns Evolve in JHotDraw," in *Proceedings of the 13th International Workshop on Software Technology and Engineering Practice*, 2005, pp. 65-73.

[4] A. Chen, E. Chou, J. Wong, A. Y. Yao, Z. Qing, Z. Shao, and A. Michail, "CVSSearch: searching through source code using CVS comments," in *Proceedings of IEEE International Conference on Software Maintenance*, 2001, pp. 364-373.

[5] B. Cleary, C. Exton, J. Buckley, and M. English, "An empirical analysis of information retrieval based concept location techniques in software comprehension," *Empirical Software Engineering,* vol. 14, pp. 93-130, 2009.

[6] D. Cubranic and G. C. Murphy, "Hipikat: recommending pertinent software development artifacts," in *Proceedings of the 25th International Conference on Software Engineering* Portland, Oregon, 2003, pp. 408-418.

[7] M. Eaddy, A. Aho, and G. C. Murphy, "Identifying, Assigning, and Quantifying Crosscutting Concerns," in *Proceedings of the First International Workshop on Assessment of Contemporary Modularization Techniques*, 2007, p. 2.

[8] M. Eaddy, T. Zimmermann, K. D. Sherwood, V. Garg, G. C. Murphy, N. Nagappan, and A. V. Aho, "Do Crosscutting Concerns Cause Defects?," *IEEE Transactions on Software Engineering,* vol. 34, pp. 497-515, 2008.

[9] G. Fischer, S. Henninger, and D. Redmiles, "Cognitive tools for locating and comprehending software objects for reuse," in *Proceedings of the 13th International Conference on Software Engineering* Austin, Texas, 1991, pp. 318-328.

[10] E. Hatcher and O. Gospodnetic, *Lucene in Action*: Manning Publications Co., 2004.

[11] D. Janzen and K. D. Volder, "Navigating and querying code without getting lost," in *Proceedings of the 2nd international conference on Aspect-oriented software development* Boston, Massachusetts: ACM, 2003, pp. 178-187.

[12] A. Marcus and J. I. Maletic, "Recovering documentation-to-source-code traceability links using latent semantic indexing," in *Proceedings of the 25th International Conference on Software Engineering* Portland, Oregon, 2003, pp. 125-135.

[13] D. Poshyvanyk, A. Marcus, and Y. Dong, "JIRiSS - an Eclipse plug-in for Source Code Exploration," in *Proceedings of the 14th IEEE International Conference on Program Comprehension*, 2006, pp. 252-255.

[14] V. Rajlich and N. Wilde, "The role of concepts in program comprehension," in *Proceedings of the 10th International Workshop on Program Comprehension*, 2002, pp. 271-278.

[15] M. Revelle and D. Poshyvanyk, "An exploratory study on assessing feature location techniques," in *Proceedings of the 17th International Conference on Program Comprehension*, 2009, pp. 218-222.

[16] M. P. Robillard, "Topology analysis of software dependencies," *ACM Transactions on Software Engineering and Methodology,* vol. 17, pp. 1-36, 2008.

[17] M. P. Robillard, D. Shepherd, E. Hill, K. Vijay-Shanker, and L. Pollock, "An Empirical Study of the Concept Assignment Problem," McGill University SOCS-TR-2007.3, 2007.

[18] T. Savage, M. Revelle, and D. Poshyvanyk, "FLAT3: Feature Location and Textual Tracing Tool," in *Proceedings of the 32nd International Conference on Software Engineering,* Cape Town, South Africa, 2010.

[19] W. Scacchi, "Free and Open Source Development Practices in the Game Community," *IEEE Softw.,* vol. 21, pp. 59-66, 2004.

[20] 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.

[21] N. Wilde, M. Buckellew, H. Page, V. Rajlich, and L. Pounds, "A comparison of methods for locating features in legacy software," *Journal of Systems and Software,* vol. 65, pp. 105-114, 2003.

[22] W. Zhao, L. Zhang, Y. Liu, J. Sun, and F. Yang, "SNIAFL: Towards a static noninteractive approach to feature location," *ACM Transactions on Software Engineering Methodology,* vol. 15, pp. 195-226, 2006.

[23] T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller, "Mining Version Histories to Guide Software Changes," in *Proceedings of the 26th International Conference on Software Engineering*, 2004, pp. 563-572.