Integrating Information Retrieval, Execution and Link Analysis Algorithms to Improve Feature Location in Software

Bogdan Dit, Meghan Revelle, and Denys Poshyvanyk
Computer Science Department
The College of William and Mary
Williamsburg, VA 23185
{bdit,meghan,denys}@cs.wm.edu

Abstract. Data fusion is the process of integrating multiple sources of information such that their combination yields better results than if the data sources are used individually. This paper applies the idea of data fusion to feature location, the process of identifying the source code that implements specific functionality in software. A data fusion model for feature location is presented which defines new feature location techniques based on combining information from textual, dynamic, and web mining or link analyses algorithms applied to software. A novel contribution of the proposed model is the use of advanced web mining algorithms to analyze execution information during feature location. The results of an extensive evaluation on three Java systems indicate that the new feature location techniques based on web mining improve the effectiveness of existing approaches by as much as 87%.

Keywords: *Concept location, feature identification, information retrieval, web mining, program comprehension, software evolution and maintenance*

1. Introduction

Software systems are constantly changing and evolving in order to eliminate defects, improve performance or reliability, and add new functionalities. When the software engineers who maintain and evolve a system are unfamiliar with it, they must go through the program comprehension process. During this process, they obtain sufficient knowledge and understanding of at least the part of the system to which a change is to be made. An important part of the program comprehension process is *feature* or *concept location* (Biggerstaff et al. 1994; Antoniol and Guéhéneuc 2006), which is the practice of identifying the source code that implements functionality, also known as a feature. Before software engineers can make changes to a feature, they must first find and understand its implementation.

For software developers who are unfamiliar with a system, feature location can be a laborious task if performed manually. In large software systems, there may be hundreds of classes and thousands of methods. Finding even one method that implements a feature can be extremely challenging and time consuming. Fortunately for software engineers in this situation, there are feature location techniques that automate, to a certain extent, the search for a feature's implementation.

Existing feature location techniques use different tactics to find a feature's source code. Approaches based on information retrieval (IR) leverage the fact that identifiers and comments embed domain knowledge to locate source code that is textually similar to a query describing a feature (Marcus et al. 2004). Dynamic feature location techniques collect and analyze execution traces to identify a feature's source code based on set operations (Wilde and Scully 1995) or probabilistic ranking (Antoniol and Guéhéneuc 2006). Static approaches to feature location rely on following or analyzing structural program dependencies (Chen and Rajlich 2000; Robillard 2008).

The state of the art in feature location involves integrating information from multiple sources. Researchers have recognized that combining more than one approach to feature location can produce better results than standalone techniques (Eisenbarth et al. 2003; Zhao et al. 2006; Hill et al. 2007; Liu et al. 2007; Poshyvanyk et al. 2007; Eaddy et al. 2008a). Generally in these combined approaches, information from one source is used to filter results from another. For instance in the SITIR approach to feature location (Liu et al. 2007), a single execution trace is collected, and then IR is used to rank only the methods that appear in the trace instead of all of the system's methods. Thus, dynamic analysis is used as a filter to IR, and filtering is one way to combine information from several sources to perform feature location. Instead of using filtering, PROMESIR (Poshyvanyk et al. 2007) combines the opinions of two "experts" (scenario-based probabilistic ranking (Antoniol and Guéhéneuc 2006) and IR (Marcus et al. 2004)) using an affine transformation. A comprehensive survey on existing feature location approaches is presented in (Dit et al. 2011).

The idea of integrating data from multiple sources is known as data fusion. The sources of data have their individual benefits and limitations, but when they are combined, those drawbacks can be minimized and better results can be achieved. Data fusion is used heavily in sensor networks and geospatial applications to attain better results in terms of accuracy, completeness, or dependability. For example, the position of an object can be calculated using an inertial navigation system (INS) or global positioning system (GPS). An INS continuously calculates the position of an object with relatively little noise and centimeter-level accuracy, though over time the position data will drift and become less accurate. GPS calculates position discretely, has relatively more noise, and meter-level accuracy. However, when data from an INS and GPS are used together in the proper proportions, the GPS data can correct for the drift in the INS data. Thus the fusion of INS and GPS data produces more accurate and dependable results than if they were used separately.

Inspired by the benefits of using data fusion to integrate multiple sources of information, this work applies data fusion to feature location. This work presents a data fusion model for feature location that is based on the idea that combining data from several sources in the right proportions will be effective at identifying a feature's source code. The data fusion model defines different types of information that can be integrated to perform feature location including textual, execution, and dependence. Textual information is analyzed by IR, execution information is collected by dynamic analysis, and dependencies are analyzed using link analysis algorithms. Applying web mining to feature location is a novel idea, but it has been previously used for other program comprehension tasks, such as identifying key classes for program comprehension (Zaidman and Demeyer 2008) and ranking components in a software repository (Inoue et al. 2005). Software lends itself well to web mining approaches, because like the World Wide Web, software can be represented by a graph, and that graph can be mined for useful information such as the source code that implements a feature.

This work makes the following contributions:

- A data fusion model for feature location is defined that integrates different types of information to locate features using IR, dynamic analysis, and web mining algorithms.

- An extensive evaluation of the feature location techniques defined in the model.
- Results that show that the new feature location techniques have better effectiveness than the state of the art in feature location. Statistical analysis indicates that this improvement is significant.

In addition, all of the data used in the evaluation is made freely available online[1], and other researchers are welcome to replicate this work. Making the data available will help facilitate the creation of feature location benchmarks.

The remainder of this article is structured as follows. Section 2 introduces the data fusion model for feature location. Section 3 outlines the evaluation methodology, and Section 4 discusses the results. Related work is summarized in Section 5, and Section 6 concludes.

2. Integrating Information Retrieval with Execution and Link Analyses

The feature location model presented here defines several sources of information, the analyses used to derive the data, and how the information can be combined using data fusion.

2.1. Textual Information from Information Retrieval

Textual information in source code, represented by identifier names and internal comments, embeds domain knowledge about a software system. This information can be leveraged to locate a feature's implementation through the use of IR. Information Retrieval is the methodology of searching for textual artifacts or for relevant information within artifacts. IR works by comparing a set of artifacts to a query and ranking these artifacts by their relevance to the query. There are many IR techniques that have been applied in the context of program comprehension tasks such as the Vector Space Model (VSM) (Salton and McGill 1983), Latent Semantic Indexing (LSI) (Deerwester et al. 1990), and Latent Dirichlet Allocation (LDA) (Blei et al. 2003). This work focuses on evaluating LSI for feature location, and the notation IR_{LSI} is used to denote that LSI is the method used to instantiate IR analysis in the model. IR_{LSI} follows five main steps (Marcus et al. 2004): creating a corpus, preprocessing, indexing, querying, and generating results.

Corpus creation. To begin the IR process, a document granularity needs to be chosen so a corpus can be formed. A document lists all the text found in a contiguous section of source code such as a method, class, or package. A corpus consists of a set of documents. For instance in this work, a corpus contains method-level granularity documents that include the text of each method in a software system.

Preprocessing. Once the corpus is created, it is preprocessed. Preprocessing involves normalizing the text of the documents. For source code, operators and programming language keywords are removed. Additionally, source code identifiers and other compound words are split (e.g., "featureLocation" becomes "feature" and "location"). Finally, stemming is performed to reduce words to their root forms (e.g., "stemmed" becomes "stem"), using the Porter stemmer (Porter 1980).

Index the corpus. The corpus is used to create a *term-by-document* matrix. The matrix's rows correspond to the terms in the corpus, and the columns represent

[1] http://www.cs.wm.edu/semeru/data/emse-link-analysis/ (verified on 05/30/2011)

documents (i.e., source code methods). A cell $m_{i,j}$ in the matrix holds a measure of the weight or relevance of the i^{th} term in the j^{th} document. The weight can be expressed as a simple count of the number of times the term appears in the document or as a more complex measure such as term frequency-inverse-document frequency. Singular Value Decomposition (SVD) (Salton and McGill 1983) is then used to reduce the dimensionality of the matrix by exploiting the co-occurrence of related terms, using a dimensionality reduction factor of 300 as in our prior work (Poshyvanyk et al. 2007).

Issue a query. A user formulates a natural language query consisting of words or phrases that describe the feature to be located (e.g., "print file to PDF format"). Each query extracted from the software repository goes through the same preprocessing techniques as the software corpus.

Generate the results. In the SVD model, each document corresponds to a vector. The query is also converted to a vector, and then the cosine of the angle between the two vectors is used as a measure of the similarity of the document to the query. The closer the cosine is to one, the more similar the document is to the query. A cosine similarity value is computed between the query and each document, and then the documents are sorted by their similarity values. The user inspects the ranked list, generally only reviewing the top results to decide if they are relevant to the feature.

2.2. Execution Information from Dynamic Analysis

Execution information is gathered via dynamic analysis, which is commonly used in program comprehension (Cornelissen et al. 2009) and involves executing a software system under specific conditions. For feature location, these conditions involve running a test case or scenario that invokes a feature in order to collect an execution trace. For example, if the feature of interest in a text editor is *printing*, the test case or scenario would involve printing a file. Invoking the desired feature during runtime generates a feature-specific execution trace.

Most existing feature location techniques that employ dynamic analysis use it to explicitly locate a feature's implementation by analyzing patterns in traces postmortem (Wilde and Scully 1995; Eisenbarth et al. 2003; Antoniol and Guéhéneuc 2006). The model presented in this work extends the dynamic analysis for feature location. Information collected from execution traces is combined with other data sources instead of being analyzed itself. Execution information is integrated with other information by using it as a filter, as in the SITIR approach (Liu et al. 2007) where methods not executed in a feature-specific scenario are pruned from the ranked list produced by IR_{LSI}.

The model in this work takes a similar approach to using execution information (denoted as "Dyn") as a filter. By extracting information from a single trace, the sequence of method calls can be used to create a graph where nodes represent methods and edges indicate method calls. This graph is a subgraph of a static call graph that only contains methods that were executed. The edges in the graph can be weighted or weightless. When weights are used, they can be derived from execution frequency information captured by a trace. For instance, Figure 1 shows a portion of an execution trace where method x calls method y two times and calls method z three times. This trace is represented by a graph where the weight of the edge from x to y is 2/5, and the weight of the edge from x to z is 3/5. Alternatively, instead of normalizing the edge weights, the values on the edge from x to y can be 2, and the weight of the edge from x to z can be 3. When dynamic execution information is used in either of these ways, it is denoted with the "freq" subscript, referring to the fact that execution frequency information is used. If no weights are placed

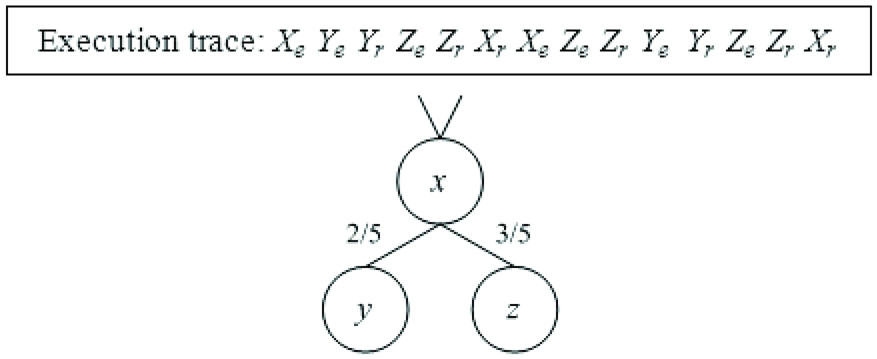

Execution trace: X_e Y_e Y_r Z_e Z_r X_r X_e Z_e Z_r Y_e Y_r Z_e Z_r X_r
x
2/5
3/5
y
z

Figure 1 An example of an execution trace translated into a call graph with execution frequency weights on the edges. $\boldsymbol{X_e}$ is the entry to method $\boldsymbol{X}$, and $\boldsymbol{X_r}$ is the return from method $\boldsymbol{X}$.

on the edges of a graph, this is denoted with the "bin" subscript, referring to the fact that only binary information about a method's execution is used.

2.3. Link Analysis Information from Web Mining

Web mining is a branch of data mining that concentrates on analyzing the structure of the World Wide Web (WWW) (Cooley et al. 1997). The structure of the WWW can be used to extract useful information. For instance, search engines use web mining to rank web pages by their relevance to a user's query. Web mining algorithms view the WWW as a graph. The graph is constructed of nodes, which represent web pages, and edges, which represent hyperlinks between pages.

Software can also be represented in graph form as a call graph. Nodes represent methods, and edges correspond to relationships or calls among methods. Therefore, web mining algorithms can be naturally applied to software to discover useful information from its structure, such as key classes for program comprehension (Zaidman and Demeyer 2008), component ranks in software repositories (Inoue et al. 2005), and statements that can be refined from concept bindings (Li 2009). This work explores whether web mining can also be applied to feature location, either as a standalone technique or used as a filter to an existing approach to feature location. Two web mining algorithms are discussed below.

2.3.1. HITS

The Hyperlinked-Induced Topic Search (HITS) (Kleinberg 1999) algorithm identifies hubs and authorities from a graph representing the WWW. Hubs are pages that have links to many other pages that contain relevant information on a topic. These pages with pertinent information are known as authorities. Good hubs point to many good authorities, and good authorities are pointed to by many hubs. Thus, hub and authority values are defined in a mutually recursive way. Let h_p stand for the hub value of page p and a_p represent the authority value of p. The hub and authority values of p are defined in Equation 1, where i is a page connected to p, and n is the total number of pages connected to p.

$$h_p = \sum_{i=1}^{n} a_i \text{ and } a_p = \sum_{i=1}^{n} h_i \qquad (1)$$

To start, HITS initializes all hub and authority values to one. Then, the algorithm is run for a given number of iterations (or until the values converge), during which the hub and authority values are updated according to Equation 1. The values are normalized after each iteration.

A slight variation of the HITS algorithm allows weights to be added to the links between pages. Weighted links denote relative importance. Let $w_{i\to p}$ represent the weight of the link between i and p. The formulas for hubs and authorities now become:

$$h_p = \sum_{i=1}^{n} w_{i\to p} \cdot a_i \text{ and } a_p = \sum_{i=1}^{n} w_{i\to p} \cdot h_i \qquad (2)$$

When using software to construct a graph instead of the WWW, the nodes and edges can be determined from a static call graph or dynamic execution trace. This work concentrates on constructing graphs from execution traces. Nodes in the graph correspond to methods, and edges represent dependencies (calls) between methods. If weights are placed on the graph edges, dynamic execution frequency can be used[2]. Otherwise, if no weights are used, binary dynamic information is used.

Using either frequency or binary dynamic information to construct a method call graph, the HITS algorithm can potentially be used for feature location in two ways. First, the methods in a graph can be ranked by extending the concepts of hubs and authorities to source code. Hub methods are those that call upon many other methods, while authority methods are called by a large number of other methods. Intuitively, hubs do not perform much functionality themselves but delegate to others, and authorities actually perform specific functionalities. Ranking methods in a software system by either their hub or authority values is a novel feature location technique. The notation $WM_{HITS(h,freq)}$, $WM_{HITS(h,bin)}$, $WM_{HITS(a,freq)}$, $WM_{HITS(a,bin)}$ is used, where WM refers to web mining, $HITS(h)$ and $HITS(a)$ stand for hub and authority scores respectively, and the "freq" and "bin" subscripts denote how dynamic information is used to weight the graph's edges.

The second way in which the HITS algorithm can be used for feature location is as a filter. Instead of directly using the hub and authority values to rank methods, those rankings can be combined with other information. From a theoretical point of view, when adapting the HITS algorithm from web pages to software, the intuition is that the methods with high hub values will be methods that are more general purpose in nature and not specific to a feature, i.e., methods in "god" classes. Conversely, methods with high authority values will be highly relevant to a feature. Therefore, top-ranked hub methods and bottom-ranked authority methods can be filtered from the results of other techniques such as $IR_{LSI}Dyn_{bin}$. The "top" superscript is used to represent when the top-ranked methods are filtered, and "bottom" superscript stands for the case when the bottom-ranked methods are filtered. Although these are the intuitive and expected results, our evaluation investigates filtering both bottom and top methods ranked by hubs and authorities scores in order to find the best method of filtering by hub and authority values.

2.3.2. PageRank

PageRank (Brin and Page 1998) is a web mining algorithm that estimates the relative importance of web pages. It is based on the random surfer model which states that a web

[2] The HITS algorithm does not require edge weights to be normalized, so the execution frequency values are used without normalization.

surfer on any given page p will follow one of p's links with a probability of β and will jump to a random page with a probability of $(1 - \beta)$. Generally, β = 0.85. Given a graph representing the WWW, let N be the total number of pages or nodes in the graph. Let $I(p)$ be the set of pages that link to p, and $O(p)$ be the pages that p links to. PageRank is defined by the Equation 3.

$$PR(p) = \frac{1-\beta}{N} + \beta \cdot \sum_{j \in I(p)} \frac{PR(j)}{|O(j)|} \tag{3}$$

PageRank's definition is recursive and must be iteratively evaluated until it converges.

Like HITS, PageRank can be applied to software if a system is represented by a graph where nodes are methods executed in a trace and edges are method calls. In the PageRank algorithm, edges always have weights. When binary execution information is used, the weight of all the outgoing edges from a node is equally distributed among those edges (e.g., if x has three outgoing edges, their weight will each be 1/3). Otherwise, execution frequency information can be used for the edge weights. PageRank requires normalized values, so the execution frequency values are normalized, as in the example in Figure 1.

Like HITS, PageRank can be used to directly rank and locate a feature's relevant methods or as a filter to other sources of information. When used directly as a feature location technique, it is denoted as $WM_{PR(freq)}$ or $WM_{PR(bin)}$, referring to the use of frequency or binary execution information to create a graph. PageRank, applied to software, is an estimate of the global importance of a method within the system. Therefore, methods that have global significance within a system will be ranked highly. Methods relevant to a specific feature are unlikely to have high global importance, so they may be ranked lower in the list. The evaluation examines PageRank as a feature location technique.

Since PageRank identifies methods of global importance, instead of using it as a standalone feature location technique, it can be used as a filter to be combined with other sources of information. Pruning the top-ranked PageRank methods from consideration may produce better feature location results. The "top" and "bottom" superscripts denote that the top and bottom results returned by PageRank are filtered. The evaluation explores the best way to use PageRank as a filter.

2.4. Fusions

Data fusion combines information from multiple sources to achieve potentially more accurate results. For feature location, this model has defined three information sources derived from three types of analysis: information retrieval, execution tracing, and web mining. This subsection outlines the feature location techniques instantiated within the model that are evaluated. **Table 1** lists all of the techniques.

Information Retrieval via LSI. This feature location technique, introduced by Marcus et al. (Marcus et al. 2004), ranks all methods in a software system based on their relevance to a query. Only one source of information is used, so no data fusion is performed. This approach is referred to as IR_{LSI}.

Information Retrieval and Execution Information. The idea of fusing IR with dynamic analysis is used by the SITIR approach (Liu et al. 2007) and is the state of the art of feature location techniques that rank program elements (e.g., methods) by their relevance to a feature. A single feature-specific execution trace is collected. Then, LSI ranks all the methods in the trace instead of all the methods in the system. Thus dynamic

Table 1 The feature location techniques evaluated.

IR & Dynamic Analysis	IR_{LSI}
	$IR_{LSI}Dyn_{bin}$
Link Analysis Algorithms	$WM_{HITS(h,bin)}$
	$WM_{HITS(h,freq)}$
	$WM_{HITS(a,bin)}$
	$WM_{HITS(a,freq)}$
	$WM_{PR(bin)}$
	$WM_{PR(freq)}$
IR, Dyn, & HITS	${IR_{LSI}Dyn_{bin}WM_{HITS(h,bin)}}^{top}$
	${IR_{LSI}Dyn_{bin}WM_{HITS(h,bin)}}^{bot}$
	${IR_{LSI}Dyn_{bin}WM_{HITS(h,freq)}}^{top}$
	${IR_{LSI}Dyn_{bin}WM_{HITS(h,freq)}}^{bot}$
	${IR_{LSI}Dyn_{bin}WM_{HITS(a,bin)}}^{top}$
	${IR_{LSI}Dyn_{bin}WM_{HITS(a,bin)}}^{bot}$
	${IR_{LSI}Dyn_{bin}WM_{HITS(a,freq)}}^{top}$
	${IR_{LSI}Dyn_{bin}WM_{HITS(a,freq)}}^{bot}$
IR, Dyn & PageRank	${IR_{LSI}Dyn_{bin}WM_{PR(bin)}}^{top}$
	${IR_{LSI}Dyn_{bin}WM_{PR,bin)}}^{bot}$
	${IR_{LSI}Dyn_{bin}WM_{PR(freq)}}^{top}$
	${IR_{LSI}Dyn_{bin}WM_{PR(freq)}}^{bot}$

information is used as a filter to eliminate methods that were not executed and therefore are less likely to be relevant to the feature. In this work, this technique is abbreviated $IR_{LSI}Dyn_{bin}$ and represents the baseline for comparison. Note that the $IR_{LSI}Dyn_{freq}$ approach is not evaluated. It filters the same methods as $IR_{LSI}Dyn_{bin}$ because it only matters whether a method was executed or not.

Web Mining. The HITS and PageRank algorithms can be used as feature location techniques that rank all methods in an execution trace using either binary or frequency information. Web mining has not been applied to feature location before; therefore all of the approaches involving web mining are novel. **Table 1** lists all the feature location techniques based on web mining.

Information Retrieval, Execution Information, and Web Mining. Applying data fusion, IR, execution tracing, and web mining can be combined to perform feature location. This work proposes the use of web mining as a filter to $IR_{LSI}Dyn_{bin}$'s results in order to eliminate methods that are irrelevant. **Figure 2** illustrates the process. Each web mining algorithm can be applied to binary or execution frequency information. To combine $IR_{LSI}Dyn_{bin}$ and web mining, the top or bottom web mining results can be pruned from $IR_{LSI}Dyn_{bin}$'s ranked list. If the results returned by a standalone web mining technique rank methods that are relevant to a feature at the top of the list, then methods at the bottom of the list can be filtered from consideration. However, since the standalone web mining techniques are based on a dynamically-constructed call graph, the resulting rankings could be similar across many different features, meaning the top-ranked results are not relevant to the feature. In this case, those top-ranked results are eliminated from consideration. For example, ${IR_{LSI}Dyn_{bin}WM_{HITS(h,bin)}}^{top}$ is a feature

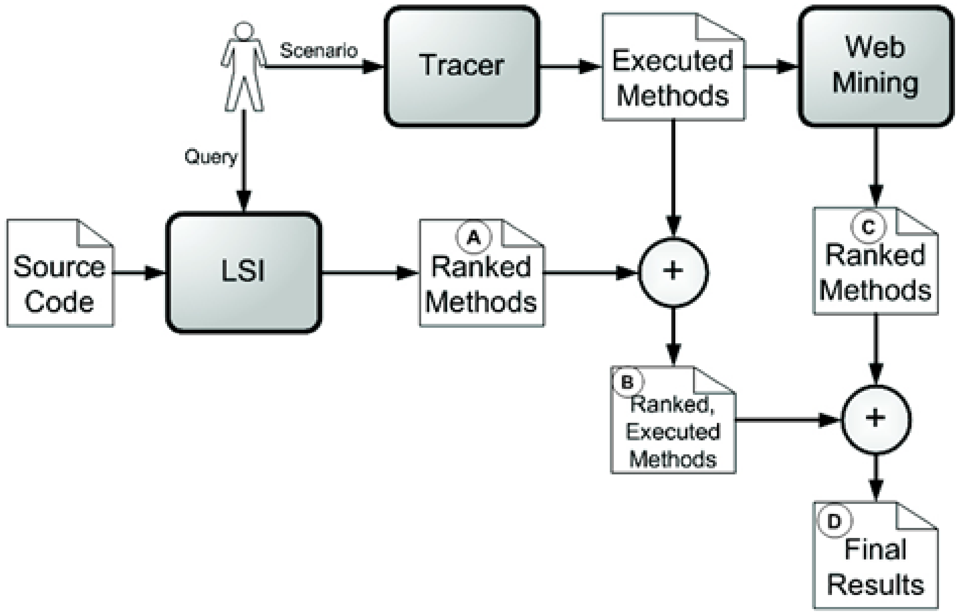


Figure 2 Combining textual analysis, dynamic analysis, and web mining for feature location. The ranked results of type A are: IR_{LSI} ; The ranked results of type B are: $IR_{LSI}Dyn_{bin}$; The ranked results of type C are: $WM_{HITS(h,bin)}$, $WM_{PR(bin)}$, etc.; The ranked results of type D are: $IR_{LSI}Dyn_{bin}\ {WM_{HITS(h,bin)}}^{top}$, $IR_{LSI}Dyn_{bin}\ {WM_{PR(bin)}}^{top}$, etc.

location technique that uses IR to rank all of the executed methods by their relevance to a query. A graph is constructed using binary execution information from a trace, and the methods in the graph are ranked according to their HITS hub values. Finally, the top methods from the HITS hub rankings are pruned from the $IR_{LSI}Dyn_{bin}$ results. In this technique, methods with high HITS hub values are filtered. **Table 1** lists all of the feature location techniques that filter $IR_{LSI}Dyn_{bin}$'s results using HITS or PageRank.

3. Experimental evaluation

This section describes the design of a case study to assess the feature location techniques defined by the data fusion model. The evaluation seeks to answer the following research questions:

RQ1: Does combining link analysis algorithms with an existing approach to feature location improve its effectiveness?

RQ2: Which web-mining algorithm, HITS or PageRank, produces better results?

The answers to these research questions will help reveal the best instantiation of the data fusion model.

3.1. Systems and Benchmarks

Table 2 Descriptive statistics on the execution traces. The columns represent the minimum, maximum, lower quartile, median, upper quartile, mean and the standard deviation. Forty-five traces were collected for Eclipse, 241 for Rhino, and 150 for jEdit.

		Min	Max	25%	Med	75%	Mean	μ
Eclipse	Methods	88K	1.5MM	312K	525K	1MM	666K	406K
	Unique	1.9K	9.3K	3.9K	5K	6.3K	5.1K	2K
	Size(MB)	9.5	290	55	98	202	124	83
	Nesting[5]	22	178	37	54	71	59	32
	Threads	1	26	7	10	12	10	5
Rhino	Methods	160K	12MM	612K	909K	1.8MM	1.8MM	2.3MM
	Unique	777	1.1K	870	917	943	912	54
	Size(MB)	18	1,668	71	104	214	210	273
	Nesting	25	37	28	27	28	28	1
	Threads	1	1	1	1	1	1	0
jEdit	Methods	4.9K	542K	30K	56K	91K	76K	75K
	Unique	227	1.9K	935	1.1K	1.3K	1.1K	310
	Size(MB)	0.55	227	6	16	42	30	39
	Nesting	4	491	53	102	196	137	106
	Threads	1	11	1	5	6	4	3

The evaluation was conducted on three open source Java software systems: Eclipse, Rhino and jEdit. Eclipse[3] is an integrated development environment. Version 3.0 has approximately 10K classes, 120K methods, and 1.6 million lines of code. Forty-five features from Eclipse were studied. The features are represented by bug reports submitted to Eclipse's online issue tracking system[4]. The bug reports are change requests that pertain to faulty features. The bug reports provide steps to reproduce the problem, and these steps were used as scenarios to collect execution traces. **Table 2** lists information about the size of the collected traces. The short descriptions in the bug reports were used as the IR queries. The bug reports also have submitted patches that detail the code that was changed to fix the bug. The modified methods are considered to be the "gold set" of methods that implement the feature. Since their code had to be altered to correct a problem with the feature, they are likely to be relevant to the feature. These gold set methods are used as the benchmark to evaluate the feature location techniques. This way of determining a feature's relevant methods from patches has also been used by other researchers (Liu et al. 2007; Poshyvanyk et al. 2007; Lukins et al. 2008).

The second system evaluated is Rhino, a Java implementation of JavaScript. Rhino[6] version 1.5 consists of 138 classes, 1,870 methods, and 32,134 lines of code. Rhino implements the ECMAScript specification[7]. The Rhino distribution comes with a test suite, and individual test cases in the suite are labeled with the section of the specification they test. Therefore, these test cases were used to collect execution traces for 241 features. The text from the corresponding section of the specification was used to formulate IR queries. For the gold set benchmarks for each feature, the mappings of

[3] http://www.eclipse.org/ (verified on 05/30/2011)
[4] https://bugs.eclipse.org/ (verified on 05/30/2011)
[5] Nesting is based on the average nesting level per feature.
[6] http://www.mozilla.org/rhino/ (verified on 05/30/2011)
[7] http://www.ecmascript.org/ (verified on 05/30/2011)

source code to features made available by Eaddy et al. (Eaddy et al. 2008b) were used. They considered the sections of the ECMAScript documentation to be features and associated code with each following the prune dependency rule which states: "A program element is relevant to a [feature] if it should be removed, or otherwise altered, when the [feature] is pruned" (Eaddy et al. 2008a). Their mappings are made publically available online[8] and have been used in several other research evaluations (Eaddy et al. 2008a; Eaddy et al. 2008b).

The third system evaluated is jEdit[9], a popular text editor written in Java. jEdit version 4.3 has 483 classes, 6.4K methods and 109 KLOC. We analyzed the SVN commits that were submitted between releases 4.2 and 4.3, and we associated these commits with issues from the jEdit bug tracking system, based on the identifiers found in the SVN commit log messages. The title and the description of the issue were used as the queries. In addition, the changes associated with each SVN commit were used to build the "gold set" of methods that were modified during that commit, and conversely for that issue or feature. We choose only 150 features that had steps to reproduce, and for which we could extract traces based on the issue's steps to reproduce. Among the 150 issues, 34 are in the feature category, 30 are patches, and 86 are bugs. As in the case of the previous two systems, the queries and gold sets are used to evaluate the feature location techniques.

The position of the first relevant method from the gold set was used as the primary means to evaluate the feature location techniques and is referred to as the *effectiveness measure* (Poshyvanyk et al. 2007). Techniques that rank relevant methods near the top of the list are more effective because they reduce the number of false positives a developer has to consider. The effectiveness measure is an accepted metric to evaluate feature location techniques. It is used here instead of precision and recall to be consistent with previous approaches (Liu et al. 2007; Poshyvanyk et al. 2007) and because feature location techniques have been shown to be better at finding one relevant method for a feature as opposed to many (Revelle and Poshyvanyk 2009). However, the evaluation also investigates how well the techniques locate all of a feature's relevant methods.

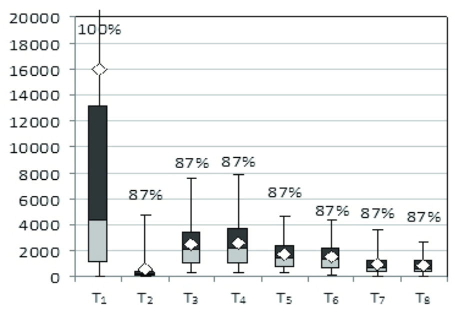
3.2. Hypotheses

Several null hypotheses were formed to test whether the performance of the baseline feature location technique improves with the use of web mining. The testing of the hypotheses is based on the effectiveness measure. Two null hypotheses are presented here; the other hypotheses can be derived analogously.

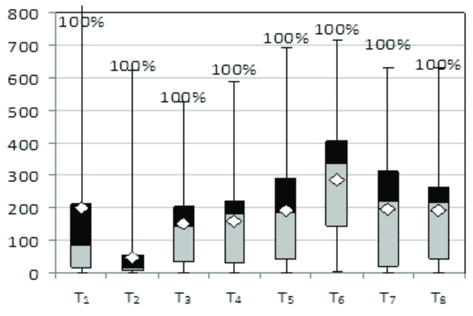
- $H_{0,WM_{PR(bin)}}$: There is no significant difference between the effectiveness of the baseline technique ($IR_{LSI}Dyn_{bin}$) and the $WM_{PR(bin)}$ technique.
- $H_{0,IR_{LSI}Dyn_{bin}WM_{PR(bin)}{}^{top}}$: There is no significant difference between the effectiveness of the baseline technique ($IR_{LSI}Dyn_{bin}$) and the technique $IR_{LSI}Dyn_{bin}WM_{PR(bin)}{}^{top}$

If a null hypothesis can be rejected with high confidence, an alternative hypothesis that states that a technique has a positive effect on the ranking of the first relevant method can be supported. The corresponding alternative hypotheses to the null hypotheses above are given. The remaining alternative hypotheses are formulated in a similar manner.

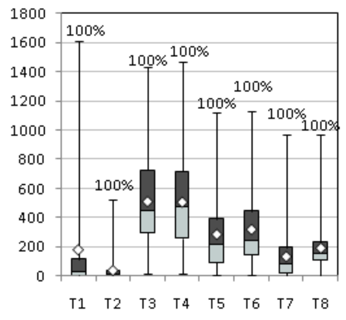
[8] http://www.cs.columbia.edu/~eaddy/concerntagger/ (verified on 05/30/2011)
[9] http://www.jedit.org/ (verified on 05/30/2011)



(a) Eclipse



(b) Rhino



(c) jEdit

T1: IR_{LSI}
T2: $IR_{LSI}Dyn_{bin}$
T3: $WM_{PR(freq)}$
T4: $WM_{PR(bin)}$
T5: $WM_{HITS(a,freq)}$
T6: $WM_{HITS(a,bin)}$
T7: $WM_{HITS(h,freq)}$
T8: $WM_{HITS(h,bin)}$

Figure 3 The effectiveness measure for the standalone web mining feature location techniques applied to 45 features in Eclipse, 241 features in Rhino and 150 features, bug reports and patches in jEdit. The values above the boxes represent the percentage of features for which the technique was able to locate at least one relevant method.

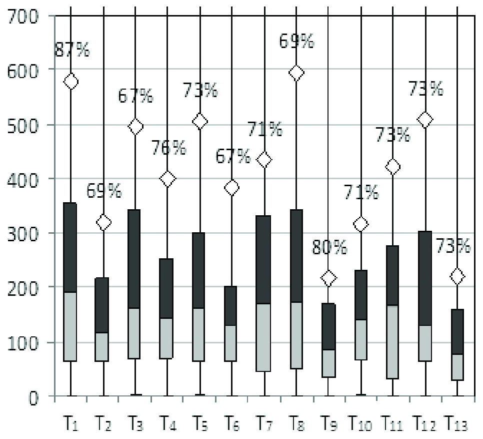
- $H_{A,WM_{PR(bin)}}$: The effectiveness of $WM_{PR(bin)}$ is significantly better than the baseline technique's effectiveness.
- $H_{A,IR_{LSI}Dyn_{bin}WM_{PR(bin)}^{top}}$: The effectiveness of $IR_{LSI}Dyn_{bin}WM_{PR(bin)}{}^{top}$ is significantly better than the baseline technique's effectiveness.

3.3. Data Collection and Analysis

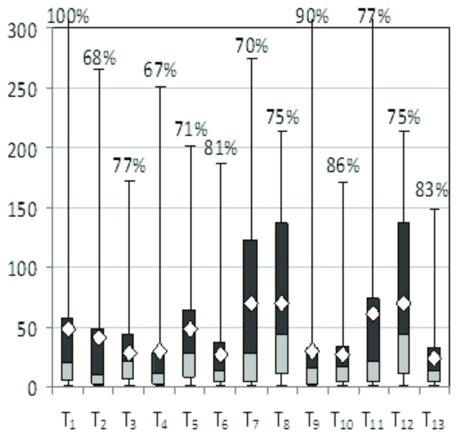
The primary data collected in the evaluation is the effectiveness measure. For each feature location technique, there are 45 data points for Eclipse, 241 for Rhino and 150 for jEdit. We use the following three criteria for comparing the feature location techniques. The first one is descriptive statistics of the effectiveness measure for each system which summarizes the data in terms of mean, median, minimum, maximum, lower quartile, and upper quartile, and displays it as box plots.

The feature location techniques can also be evaluated by how many features they can return at least one relevant result. Many of the techniques in the model filter methods from consideration, and some of those methods may belong to the gold set. It is possible for a technique to filter out all of a feature's gold set methods and return no relevant results. Therefore, the percentage of features for which a technique can locate at least one relevant method is reported. If a feature location technique ranks one of a feature's relevant methods closer to the top of the list than another technique, then the first approach is more effective. Every feature location technique can be compared to every other technique in this manner, and the percentage of times the first technique is more effective is reported. This is the second criteria for comparing the feature location techniques.

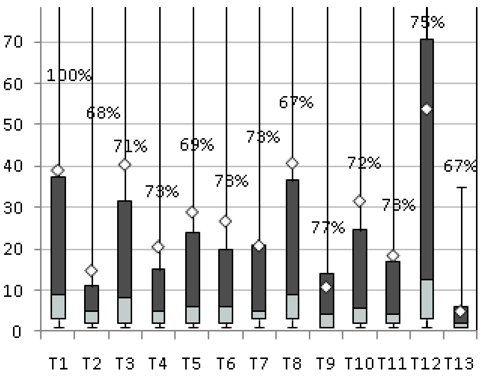
Data on whether one technique is more effective than another is not enough. The third comparison between feature location techniques consists of statistical analysis to determine if the difference between the effectiveness of two techniques is significant. The Wilcoxon Rank Sum test (Conover 1998) is used to test if the difference between the effectiveness measures of two feature location techniques is statistically significant. Essentially, the test determines if the decrease in the number of false positives reported by one technique as compared to another is significant. It is important to note that the input data for the Wilcoxon test considers for each feature the rank of the best method and ignores the ranks of all the other gold set methods, hence the test is not affected by the total number of false positives (i.e., for all the gold set methods). The Wilcoxon test is a non-parametric test that accepts paired data. Since a technique may not rank any of a feature's gold set methods, it would have no data to be paired with the data from another feature location technique. Therefore, only cases where both techniques rank a method are input to the test. In this evaluation, the significance level of the Wilcoxon Rank Sum test is $\alpha = 0.05$.



(a) Eclipse



(b) Rhino



(c) jEdit

T_1: $IR_{LSI}Dyn_{bin}$

T_2:$IR_{LSI}Dyn_{bin}WM_{PR(freq)}{}^{t[40,60,80]\%}$

T_3:$IR_{LSI}Dyn_{bin}WM_{PR(freq)}{}^{b[20,70,30]\%}$

T_4:$IR_{LSI}Dyn_{bin}WM_{PR(bin)}{}^{t[40,60,70]\%}$

T_5:$IR_{LSI}Dyn_{bin}WM_{PR(bin)}{}^{b[10,70,30]\%}$

T_6:$IR_{LSI}Dyn_{bin}WM_{HITS(a,freq)}{}^{t[30,70,40]\%}$

T_7:$IR_{LSI}Dyn_{bin}WM_{HITS(a,freq)}{}^{b[40,60,60]\%}$

T_8:$IR_{LSI}Dyn_{bin}WM_{HITS(h,freq)}{}^{t[10,70,20]\%}$

T_9:$IR_{LSI}Dyn_{bin}WM_{HITS(h,freq)}{}^{b[60,50,80]\%}$

T_{10}:$IR_{LSI}Dyn_{bin}WM_{HITS(a,bin)}{}^{t[20,70,40]\%}$

T_{11}:$IR_{LSI}Dyn_{bin}WM_{HITS(a,bin)}{}^{b[40,40,60]\%}$

T_{12}:$IR_{LSI}Dyn_{bin}WM_{HITS(h,bin)}{}^{t[10,70,20]\%}$

T_{13}:$IR_{LSI}Dyn_{bin}WM_{HITS(h,bin)}{}^{b[70,60,80]\%}$

Figure 4 The effectiveness measure for the feature location techniques that use web mining as a filter. The top and bottom percentages in brackets have two values. The first value is the percentage used in Eclipse, the second is the percentage used in Rhino, and the third is the percentage used in jEdit. The values above the boxes represent the percentage of features for which the technique was able to locate at least one relevant method.

4. Results and Discussion

This section presents the results of using the feature location techniques listed in **Table 1** to identify the first relevant method of 45 features of Eclipse, 241 features of Rhino and 150 features of jEdit. **Figure 3** and **Figure 4** show box plots representing the descriptive statistics of the effectiveness measure for Eclipse and Rhino. Low values in the box plots, which represent positions of relevant methods, suggest potentially less effort for developer to locate relevant methods, because the ranks are among the first results returned by the feature location techniques. The y-axis represents the effectiveness measure. The graphs for Eclipse and Rhino have different scales because Eclipse has more methods. **Figure 3** plots the feature location techniques based on IR (T_1), IR and dynamic analysis (T_2), and web mining as a standalone approach (T_3 through T_8). **Figure 4** shows the techniques that combine IR, dynamic analysis, and web mining (T_2 through T_{13}). $IR_{LSI}Dyn_{bin}$ is also included in this figure for reference since it represents the baseline for comparison. In **Figure 3** and **Figure 4**, the diamonds represent the average effectiveness measure. The dark grey and light grey boxes stand for the upper and lower quartiles, respectively, and the line between the boxes represents the median. The whiskers above and below the boxes denote the maximum and minimum effectiveness measure. In some cases, the maximum is beyond the scale of the graphs. The figures also report for each feature location technique, above the box plots, the percentage of features for which the technique was able to identify at least one relevant method.

The box plots in **Figure 3** show that using web mining as a standalone feature location technique produces results that are comparable to IR_{LSI} even though no query is used. However, these techniques are less effective than the state of the art, no matter the web mining algorithm used. The feature location techniques based on PageRank, HITS hub values, or HITS authority values are not as effectiveness as $IR_{LSI}Dyn_{bin}$. Overall, there is little difference between the use of binary and execution frequency information. It is surprising that ranking methods by their hub values is more effective than ranking them by their authority values, for all three systems. Intuitively, hubs are methods that delegate functionality to authorities which actually implement it. Therefore, authorities should be more valuable for feature location, but this was not observed. Another interesting observation is that for Eclipse and jEdit, PageRank was less effective than HITS, whereas for Rhino, PageRank returned comparable results to HITS. We suppose that this has to do with the structure, dependencies and the type of software. Eclipse and jEdit have many commonalities, such as graphical user interfaces, they are both editors, they have an architecture extensible for plug-ins, etc., whereas Rhino could be considered more like a library. Future research will confirm or infirm these preliminary observations.

Even though feature location techniques based on standalone web mining are not more effective than the state of the art approach, when web mining is used as a filter to IR, the results significantly improve in some cases. **Figure 4** presents box plots of the effectiveness measure of the techniques that used web mining to filter $IR_{LSI}Dyn_{bin}$'s results. The filters prune either the top or bottom methods ranked by a web mining algorithm. The threshold for the percent of methods to filter was selected for each technique individually such that at least one gold set method remained in the results for 66% of the features. In Eclipse, ${IR_{LSI}Dyn_{bin}WM_{HITS(h,freq)}}^{bot}$ (T_9 in Figure 4) had the best effectiveness measure on average. In Rhino and jEdit, ${IR_{LSI}Dyn_{bin}WM_{HITS(h,bin)}}^{bot}$ (T_{13} in Figure 4) was the most effective technique. In fact, all but one (T_8 in Figure 4) of the techniques that use web mining to filter IR are more effective than $IR_{LSI}Dyn_{bin}$ in Eclipse by 13% to 62% on average. In Rhino, most of the techniques that combine IR, dynamic analysis, and web mining have an average effectiveness of 1% to 51% better than $IR_{LSI}Dyn_{bin}$ with a few exceptions (T_7, T_8, T_{11} and T_{12} in Figure 4). Similarly for jEdit, with the exception of T_3, T_8 and T_{12}, all the techniques that use web mining as a filter have an average effectiveness of 19% to 87% (in the case of T_{13}) better than $IR_{LSI}Dyn_{bin}$. It should be noted that for all the three systems in our evaluation, ${IR_{LSI}Dyn_{bin}WM_{HITS(h,freq)}}^{top}$ (T_8) technique had lower effectiveness than the baseline technique, that is $IR_{LSI}Dyn_{bin}$, and for two of the systems (Rhino and jEdit) ${IR_{LSI}Dyn_{bin}WM_{HITS(h,bin)}}^{top}$ (T_{12}) had lower effectiveness than the baseline $IR_{LSI}Dyn_{bin}$ as well. This supports our observation that the methods of interest normally have higher HITS hub values, pushing them to the top of the list. Thus, removing those methods from the ranked list negatively impacts effectiveness. This observation is further emphasized by the fact that for all the three systems, ${IR_{LSI}Dyn_{bin}WM_{HITS(h,freq)}}^{bot}$ (T$_9$) and ${IR_{LSI}Dyn_{bin}WM_{HITS(h,bin)}}^{bot}$ (T$_{13}$) are the most effective techniques as compared to the baseline. In other words, removing methods with lower HITS hub values (usually those not relevant to the feature) improves effectiveness, because more irrelevant methods are filtered out and those methods of interest are ranked higher.

These results help answer **RQ1** because they lend strong support to the fact that integrating the ranking of methods using web mining with information retrieval is a very effective way to perform feature location. In regards to **RQ2**, the techniques based on HITS were generally more effective than the PageRank approaches, so HITS, used either as a standalone technique or as a filter, seems better suited to the task of feature location.

In addition to measuring the effectiveness of each of the feature location techniques, the new approaches based on web mining were directly compared to IR_{LSI} and $IR_{LSI}Dyn_{bin}$. **Table 3** shows for each new technique, the percent of times its effectiveness measure is better than that of the existing approaches[10]. The table shows a different view of the data presented in **Figure 3** and **Figure 4**. It shows on a case-by-case basis, which feature location technique is more effective. The data in this table is derived from the subset of methods that are ranked by both techniques, while **Figure 3** and **Figure 4** show data for all methods. In **Table 3**, if one approach ranks a method and another does not, the method is not included in the reported data. Also, the percentage of times the feature location techniques produce the same ranks for the same method is not explicitly included in the table, but it can be easily derived. The process of generating the values for Table 3 is better illustrated with an example (see Table 4). Feature location techniques A and B are used to locate eight features and the best rank of the methods from each feature is reported in the table. Features f_2 and f_3 will be discarded, since they produce a rank for just one system, and not for both (i.e., this situation may happen when all the relevant methods are filtered out). Among the remaining six features, for one feature the feature location technique A has better ranking than the feature location technique B (i.e., f_1), for two features both systems produce equal results (i.e., f_5 and f_6), and for three features B has better ranking than A (i.e., f_4, f_7, and f_8). When comparing A and B, we report these results as the pair 50% / 16.66%, which means that A produces better results in 50% of cases, and B produces better results in 16.66% of cases. In the remaining 100%-50%-16.66%=33.33% of cases both feature location techniques produce equivalent results. This pair formatting is presented in Table 3, where the left side of the pair represents the percentage of cases the feature location technique on the row is better than the technique on the left, and vice-versa, where the right side of the pair represents the percentage of cases where the technique on the column is better than the technique in the row.

The table shows that feature location techniques based solely on web mining never have better effectiveness than $IR_{LSI}Dyn_{bin}$. On the other hand, the techniques that use web mining as a filter routinely rank methods closer to the top of the list than $IR_{LSI}Dyn_{bin}$.

[10] The online appendix has data on the performance of each technique compared to all others.

Table 3 For each feature location technique, the left side of the values pair represents the percentage of times the effectiveness of the technique in the row is better than the technique in the corresponding column. The right side of the values pair represents the percentage of times the effectiveness of the technique in the column is better than the technique in the row.

	Eclipse		Rhino		jEdit	
	IR_{LSI}	IR_{LSI} Dyn_{bin}	IR_{LSI}	IR_{LSI} Dyn_{bin}	IR_{LSI}	IR_{LSI} Dyn_{bin}
$IR_{LSI}Dyn_{bin}$	97/3	X	84/9	X	83/1	X
$WM_{PR(freq)}$	59/41	10/87	48/51	17/80	12/88	2/98
$WM_{PR(bin)}$	59/41	10/90	43/56	17/81	13/87	2/98
$WM_{HITS(a,freq)}$	67/33	18/82	44/55	15/85	21/79	8/91
$WM_{HITS(a,bin)}$	56/44	18/82	24/75	6/94	17/83	3/97
$WM_{HITS(h,freq)}$	77/23	26/74	44/55	19/80	38/62	17/82
$WM_{HITS(h,bin)}$	77/23	26/74	39/59	19/78	26/72	9/89
$IR_{LSI}Dyn_{bin}WM_{PR(freq)}{}^{t}$	97/3	84/10	76/15	63/28	75/15	58/22
$IR_{LSI}Dyn_{bin}WM_{PR(freq)}{}^{b}$	100/0	77/17	78/17	56/37	75/16	59/25
$IR_{LSI}Dyn_{bin}WM_{PR(bin)}{}^{t}$	97/3	85/9	78/15	64/27	74/11	60/17
$IR_{LSI}Dyn_{bin}WM_{PR(bin)}{}^{b}$	97/3	88/6	78/18	45/46	78/13	61/23
$IR_{LSI}Dyn_{bin}WM_{HITS(a,freq)}{}^{t}$	97/3	80/10	81/12	66/26	77/13	65/17
$IR_{LSI}Dyn_{bin}WM_{HITS(a,freq)}{}^{b}$	97/3	91/6	78/18	44/47	81/10	62/19
$IR_{LSI}Dyn_{bin}WM_{HITS(h,freq)}{}^{t}$	97/3	74/6	67/28	31/60	74/19	51/29
$IR_{LSI}Dyn_{bin}WM_{HITS(h,freq)}{}^{b}$	97/3	94/3	87/7	74/13	84/4	72/6
$IR_{LSI}Dyn_{bin}WM_{HITS(a,bin)}{}^{t}$	97/3	84/6	80/15	61/32	78/15	60/21
$IR_{LSI}Dyn_{bin}WM_{HITS(a,bin)}{}^{b}$	97/3	85/9	66/27	42/40	82/6	63/14
$IR_{LSI}Dyn_{bin}WM_{HITS(h,bin)}{}^{t}$	97/3	70/6	67/28	31/60	73/18	47/31
$IR_{LSI}Dyn_{bin}WM_{HITS(h,bin)}{}^{b}$	97/3	94/3	83/11	71/19	83/3	73/6

Table 4 Example of computing percentages. The feature location techniques A and B are used to locate eight features and the values represent the ranks of the relevant methods in the list of the results for that feature

Feature Location Technique	Features							
	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8
A	10	-	12	20	18	9	21	19
B	17	14	-	15	18	9	16	4

The same trend, which suggests that HITS hubs rank relevant methods in the top results, is observed here. For example, for Eclipse system, $IR_{LSI}Dyn_{bin}WM_{HITS(h,freq)}{}^{top}$ produces better results than those of $IR_{LSI}Dyn_{bin}$ in 74% of cases, and $IR_{LSI}Dyn_{bin}$ produces better results in only 6% (in the remaining 100%-74%-6%=20% of cases both techniques produce equivalent ranks). However, when we compare $IR_{LSI}Dyn_{bin}WM_{HITS(h,freq)}{}^{bot}$ against $IR_{LSI}Dyn_{bin}$ for the Eclipse system we get even better results than for $IR_{LSI}Dyn_{bin}WM_{HITS(h,freq)}{}^{top}$ (94% vs. 74%). The same trend is observed for all the systems while using HITS hubs.

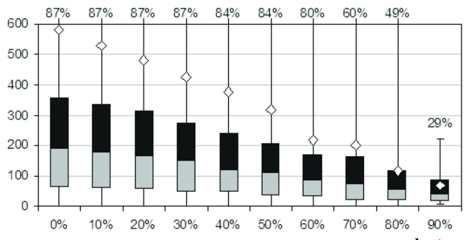
Table 5 The results of the Wilcoxon test.

	Eclipse	Rhino	jEdit	Null Hypothesis
$WM_{PR(freq)}$	1	1	1	Not Rejected
$WM_{PR(bin)}$	1	1	1	Not Rejected
$WM_{HITS(a,freq)}$	1	1	1	Not Rejected
$WM_{HITS(a,bin)}$	1	1	1	Not Rejected
$WM_{HITS(h,freq)}$	1	1	1	Not Rejected
$WM_{HITS(h,bin)}$	1	1	1	Not Rejected
${IR_{LSI}Dyn_{bin}WM_{PR(freq)}}^{top}$	**< 0.0001**	**< 0.0001**	**< 0.0001**	**Rejected**
${IR_{LSI}Dyn_{bin}WM_{PR(freq)}}^{bot}$	**0.004**	**0**	**0.003**	**Rejected**
${IR_{LSI}Dyn_{bin}WM_{PR(bin)}}^{top}$	**< 0.0001**	**< 0.0001**	**< 0.0001**	**Rejected**
${IR_{LSI}Dyn_{bin}WM_{PR(bin)}}^{bot}$	**< 0.0001**	0.74	**0.0004**	Not Rejected
${IR_{LSI}Dyn_{bin}WM_{HITS(a,freq)}}^{top}$	**0**	**< 0.0001**	**< 0.0001**	**Rejected**
${IR_{LSI}Dyn_{bin}WM_{HITS(a,freq)}}^{bot}$	**< 0.0001**	0.99	**< 0.0001**	Not Rejected
${IR_{LSI}Dyn_{bin}WM_{HITS(h,freq)}}^{top}$	**0**	1	0.066	Not Rejected
${IR_{LSI}Dyn_{bin}WM_{HITS(h,freq)}}^{bot}$	**< 0.0001**	**< 0.0001**	**< 0.0001**	**Rejected**
${IR_{LSI}Dyn_{bin}WM_{HITS(a,bin)}}^{top}$	**< 0.0001**	**< 0.0001**	**< 0.0001**	**Rejected**
${IR_{LSI}Dyn_{bin}WM_{HITS(a,bin)}}^{bot}$	**< 0.0001**	1	**< 0.0001**	Not Rejected
${IR_{LSI}Dyn_{bin}WM_{HITS(h,bin)}}^{top}$	**0**	1	0.144	Not Rejected
${IR_{LSI}Dyn_{bin}WM_{HITS(h,bin)}}^{bot}$	**< 0.0001**	**< 0.0001**	**< 0.0001**	**Rejected**

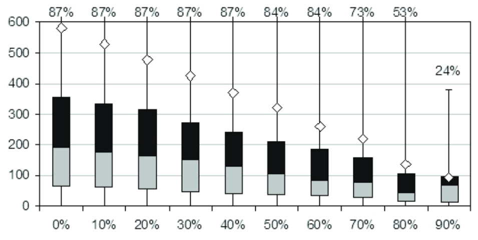
This finding also helps answer **RQ1,** combining web mining with existing approaches improves their effectiveness. **RQ2** addresses which of the two web mining algorithms is more effective. Based on the results in **Table 3**, the techniques based on HITS are more effective than the PageRank techniques.

4.1. Statistical Analysis

The Wilcoxon Rank Sum test was used to test if the difference between the effectiveness measures of two feature location techniques is statistically significant. **Table 5** shows the results of the test (p-values) for all of the techniques based on web mining as compared to $IR_{LSI}Dyn_{bin}$ and if the null hypotheses can be rejected based on the p-values. In the table, statistically significant results are presented in boldface. None of the approaches in which web mining is used as a standalone technique have statistically significant results. However in Eclipse, all of the feature location techniques that employ web mining as a filter to IR have significantly better effectiveness than $IR_{LSI}Dyn_{bin}$. Similarly in Rhino and jEdit, most of the approaches that use web mining as a filter have statistically significant results with a few exceptions. Therefore, the null hypotheses for these approaches that do not have significant results for all the systems cannot be rejected. However, for the techniques with statistically significant results for all the systems, their null hypotheses are rejected, and there is evidence to suggest that the corresponding alternative hypotheses can be supported. These feature location techniques have better effectiveness than the baseline technique.

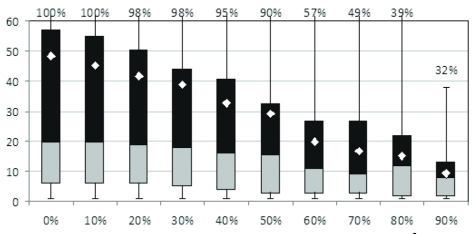


(a) $IR_{LSI}Dyn_{bin}WM_{HITS(h,freq)}{}^{bot}$

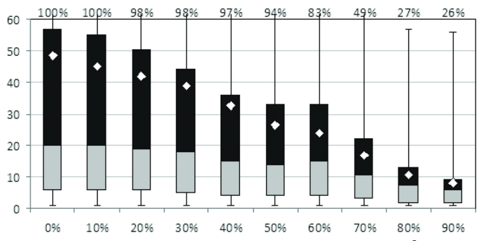


(b) $IR_{LSI}Dyn_{bin}WM_{HITS(h,bin)}{}^{bot}$

Figure 5 Summary of the effectiveness measure of $IR_{LSI}Dyn_{bin}WM_{HITS(h,freq)}{}^{bot}$ and $IR_{LSI}Dyn_{bin}WM_{HITS(h,bin)}{}^{bot}$ at different filtering thresholds for the 45 features of Eclipse. The values above the boxes represent the percentage of features for which the technique was able to locate at least one relevant method.

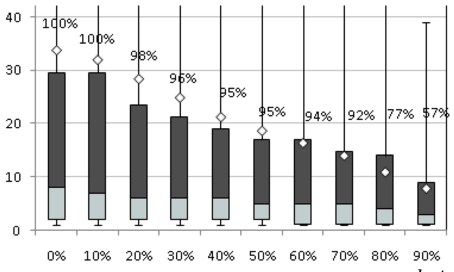


(a) $IR_{LSI}Dyn_{bin}WM_{HITS(h,freq)}{}^{bot}$

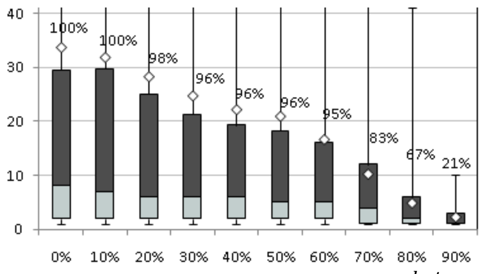


(b) $IR_{LSI}Dyn_{bin}WM_{HITS(h,bin)}{}^{bot}$

Figure 6 Summary of the effectiveness measure of $IR_{LSI}Dyn_{bin}WM_{HITS(h,freq)}{}^{bot}$ and $IR_{LSI}Dyn_{bin}WM_{HITS(h,bin)}{}^{bot}$ at different filtering thresholds for the 241 features of Rhino. The values above the boxes represent the percentage of features for which the technique was able to locate at least one relevant method.



(a) $IR_{LSI}Dyn_{bin}WM_{HITS(h,freq)}{}^{bot}$

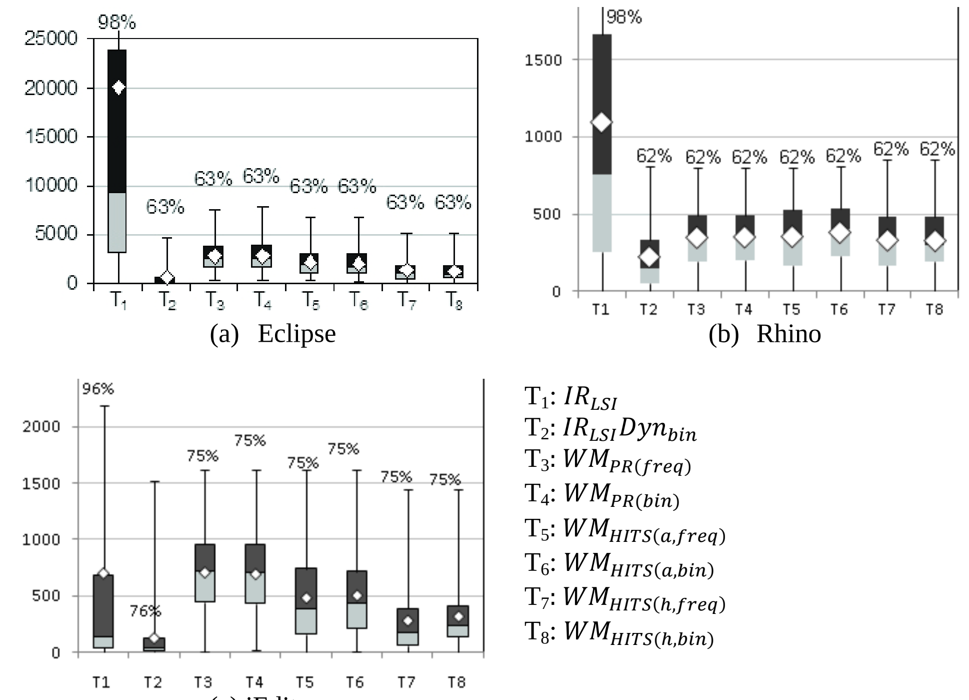


(b) $IR_{LSI}Dyn_{bin}WM_{HITS(h,bin)}{}^{bot}$

Figure 7 Summary of the effectiveness measure of $IR_{LSI}Dyn_{bin}WM_{HITS(h,freq)}{}^{bot}$ and $IR_{LSI}Dyn_{bin}WM_{HITS(h,bin)}{}^{bot}$ at different filtering thresholds for the 150 features of jEdit. The values above the boxes represent the percentage of features for which the technique was able to locate at least one relevant method.

4.2. Impact of the Selection of a Threshold

The results in the previous section for the techniques that use web mining as a filter present only one possible threshold for what percentage of the top or bottom web mining results to eliminate from the baseline results. The threshold was chosen such that a given feature location technique returned at least one relevant method for at least 66% of the



(a) Eclipse (b) Rhino

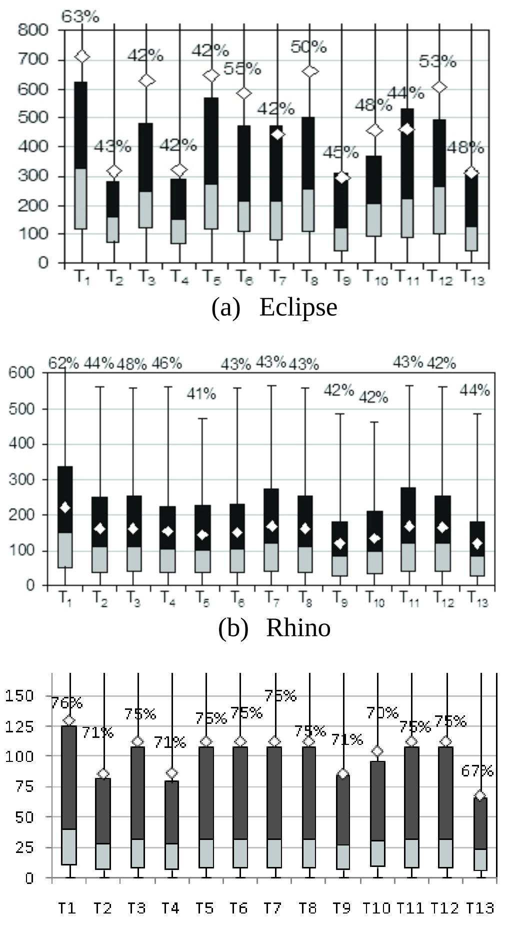
T1: IR_{LSI}
T2: $IR_{LSI}Dyn_{bin}$
T3: $WM_{PR(freq)}$
T4: $WM_{PR(bin)}$
T5: $WM_{HITS(a,freq)}$
T6: $WM_{HITS(a,bin)}$
T7: $WM_{HITS(h,freq)}$
T8: $WM_{HITS(h,bin)}$

(c) jEdit

Figure 8 Box plots of position of all gold set methods for the standalone web mining feature location techniques applied to 45 features of Eclipse, 241 features of Rhino and 150 features of jEdit. The values above the boxes represent the percent of all the gold set methods the technique could locate.

features studied. This section examines how varying the filtering threshold impacts the results, focusing on the techniques with the lowest average effectiveness, ${IR_{LSI}Dyn_{bin}WM_{HITS(h,freq)}}^{bot}$ and ${IR_{LSI}Dyn_{bin}WM_{HITS(h,bin)}}^{bot}$. **Figure 5** shows, for Eclipse, box plots of the average effectiveness of the two techniques with different filtering thresholds. **Figure 6** shows the results for Rhino and Figure 7 shows the results for jEdit.

Not surprisingly, the higher the filtering threshold, the lower the average effectiveness since more methods are eliminated from consideration. However, there is a tradeoff; the improvement in effectiveness comes at the cost of completeness. The values above the boxes in **Figure 5**, **Figure 6** and Figure 7 represent the percentage of features for which the technique was able to locate at least one relevant method. When a higher percentage of the HITS hubs results are filtered, the techniques find at least one relevant method for fewer features. For instance in Eclipse with ${IR_{LSI}Dyn_{bin}WM_{HITS(h,freq)}}^{bot}$, when the bottom 90% of the HITS hubs results are pruned from the baseline, the average effectiveness is 67, but the technique can identify a relevant method for only 29% of Eclipse's 45 features. Setting the threshold too high means methods that are relevant to a feature are considered false negatives and removed from the results. Therefore at least with the ${IR_{LSI}Dyn_{bin}WM_{HITS(h,freq)}}^{bot}$ and ${IR_{LSI}Dyn_{bin}WM_{HITS(h,bin)}}^{bot}$ techniques, a threshold of 50%–60% yields acceptable results. Automatically selecting appropriate thresholds for individual features remains part of our future work. In the meanwhile we



(a) Eclipse

(b) Rhino

(c) jEdit

T1: $IR_{LSI}Dyn_{bin}$

T2:$IR_{LSI}Dyn_{bin}WM_{PR(freq)}{}^{t[50,30,20]\%}$

T3:$IR_{LSI}Dyn_{bin}WM_{PR(freq)}{}^{b[20,30,0]\%}$

T4:$IR_{LSI}Dyn_{bin}WM_{PR(bin)}{}^{t[50,30,20]\%}$

T5:$IR_{LSI}Dyn_{bin}WM_{PR(bin)}{}^{b[20,40,0]\%}$

T6:$IR_{LSI}Dyn_{bin}WM_{HITS(a,freq)}{}^{t[20,30,0]\%}$

T7:$IR_{LSI}Dyn_{bin}WM_{HITS(a,freq)}{}^{b[40,30,0]\%}$

T8:$IR_{LSI}Dyn_{bin}WM_{HITS(h,freq)}{}^{t[10,30,0]\%}$

T9:$IR_{LSI}Dyn_{bin}WM_{HITS(h,freq)}{}^{b[60,40,20]\%}$

T10:$IR_{LSI}Dyn_{bin}WM_{HITS(a,bin)}{}^{t[20,40,10]\%}$

T11:$IR_{LSI}Dyn_{bin}WM_{HITS(a,bin)}{}^{b[40,30,0]\%}$

T12:$IR_{LSI}Dyn_{bin}WM_{HITS(h,bin)}{}^{t[10,30,0]\%}$

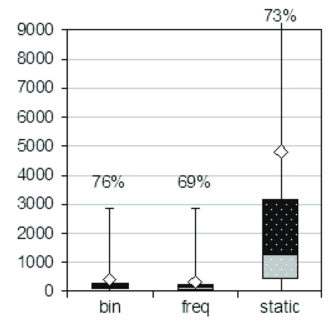
T13:$IR_{LSI}Dyn_{bin}WM_{HITS(h,bin)}{}^{b[60,40,30]\%}$

Figure 9 The average position of all gold set methods for the feature location techniques that use web mining as a filter applied to 45 features of Eclipse, 241 features of Rhino and 150 features of jEdit. The top and bottom percentages in brackets have three values. The first value is the percentage used in Eclipse, the second is the percentage used in Rhino, and the third is the percentage used in jEdit. The values above the boxes represent the percent of all the gold set methods the technique could locate.

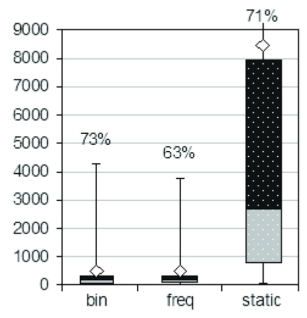
provide support for selecting these thresholds in a tool FLAT^3 (Savage et al. 2010) that implements proposed approaches for combining Information Retrieval with Execution and Link Analyses. Using FLAT^3, developers can select different ranking schemas based on link analysis (PageRank, HITS authorities, HITS hubs), different filtering mechanisms (top, bottom), and different thresholds.

4.3. Locating All of a Feature's Methods

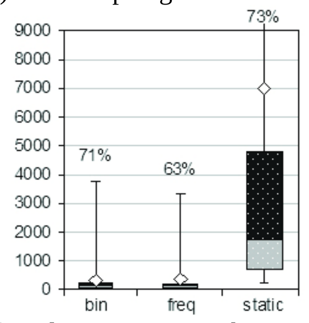
So far, this work has focused on the effectiveness of feature location only in terms of the position of the first relevant method (i.e., the effectiveness measure). However, since gold sets defining all the methods relevant to a feature were available, the feature location techniques can also be evaluated in terms of how well they locate all of a feature's



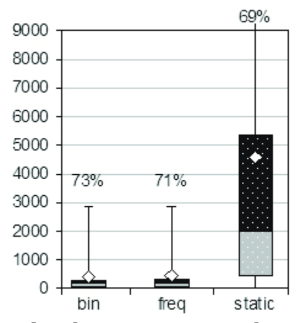
(a) Filter top PageRank results



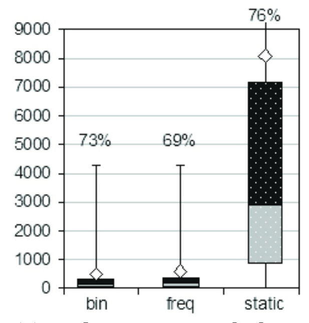
(b) Filter bottom PageRank results



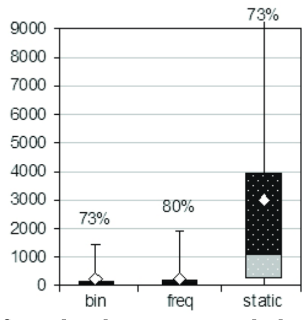
(c) Filter top HITS authority results



(d) Filter bottom HITS authority results



(e) Filter top HITS hub results

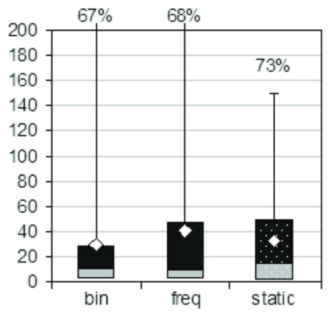


(f) Filter bottom HITS hub results

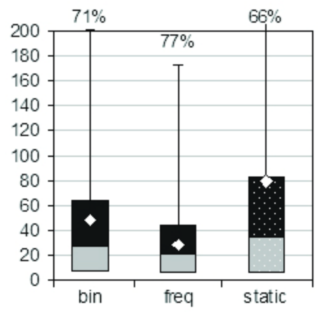
Figure 10 The effectiveness measure for the feature location techniques applied to 45 features of Eclipse. The values above the boxes represent the percentage of features for which the technique was able to locate at least one relevant method.

methods. **Figure 8** and **Figure 9** show box plots summarizing the positions of all of a feature's relevant methods. **Figure 8** presents the results for IR_{LSI}, $IR_{LSI}Dyn_{bin}$, and the standalone web mining feature location techniques, while **Figure 9** shows the results for the baseline and the techniques that use web mining as a filter.

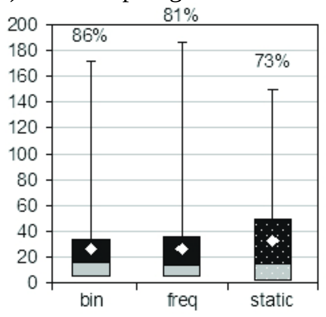
Figure 8 shows that the baseline approach, $IR_{LSI}Dyn_{bin}$ is the more effective at locating all of a feature's relevant methods than the standalone web mining techniques. However, using web mining as a filter improves the average effectiveness of locating all of the methods from a feature's gold set, as seen in **Figure 9**. As with the effectiveness measure results presented earlier, ${IR_{LSI}Dyn_{bin}WM_{HITS(h,freq)}}^{bot}$ and ${IR_{LSI}Dyn_{bin}WM_{HITS(h,bin)}}^{bot}$ were the two most effective techniques.



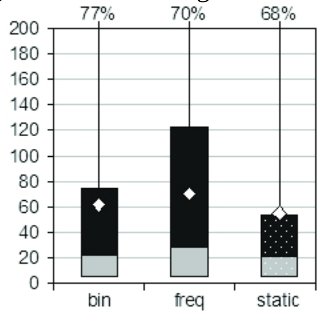
(a) Filter top PageRank results



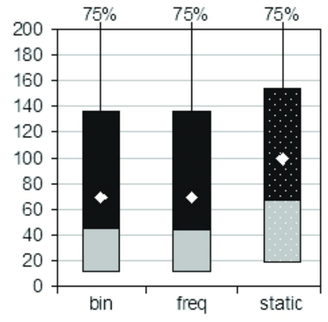
(b) Filter bottom PageRank results



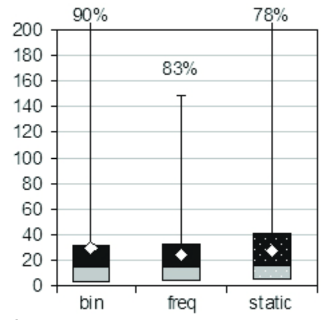
(c) Filter top HITS authority results



(d) Filter bottom HITS authority results



(e) Filter top HITS hub results



(f) Filter bottom HITS hub results

Figure 11 The effectiveness measure for the feature location techniques applied to 241 features of Rhino. The values above the boxes represent the percentage of features for which the technique was able to locate at least one relevant method.

4.4. Using a Static Call Graph

All of the feature location techniques investigated have leveraged a call graph that is constructed from execution traces specific to each feature. Collecting execution traces is computationally expensive and time consuming. This section explores whether comparable results can be achieved using a static call graph. The tradeoff is that only one static call graph is needed instead of a different dynamic call graph for each feature, but a static call graph is generalized and not feature-specific.

Figure 10 shows for Eclipse summaries of the effectiveness measure for each of the feature location techniques based on using web mining as a filter. **Figure 11** shows the results for Rhino. In each graph, the first plot represents using a dynamic call graph with binary weights and the second corresponds to using a dynamic call graph with execution frequency weights. The third patterned plot represents using a static call graph. For example, **Figure 10**(a) compares the results of ${IR_{LSI}Dyn_{bin}WM_{PR(bin)}}^{top}$, ${IR_{LSI}Dyn_{bin}WM_{PR(freq)}}^{top}$, and filtering PageRank's top-ranked methods from a static call graph from $IR_{LSI}Dyn_{bin}$'s results. Note that the threshold for the percentage of methods that were filtered from the static call graph was selected so that at least one method from the gold set remained in the results for at least 66% of the features.

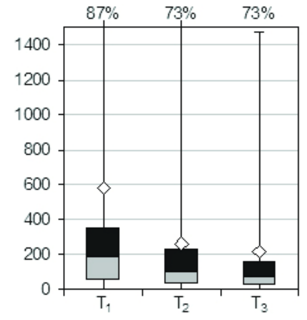
Figure 10 shows that in Eclipse, using a static call graph is not as effective as using a dynamically-constructed call graph. A static call graph includes all of a system's methods, not just those that were executed. Eclipse has over 84,000 methods, so using a static call graph significantly increases the number of methods that need to be ranked. This increase in the number of methods leads to a decrease in effectiveness because there are more false positives in the ranked list.

Figure 11 shows the results of using a dynamic call graph and a static call graph for each feature location technique that uses web mining as a filter in Rhino. Unlike the Eclipse results, using a static call graph in Rhino has comparable effectiveness. In general, the static approaches are not quite as effective as the dynamic ones, but the difference is not large. In Rhino, using a static call graph gives results that are close to those when using a dynamic call graph without the cost of collecting traces. Rhino is a smaller system than Eclipse, so ranking all of its methods instead of only those that were executed introduces fewer false positives. There may be other factors in why the static results are comparable to the dynamic results in Rhino but not Eclipse. Future work will include investigating the circumstances under which a static call graph might yield comparable results.

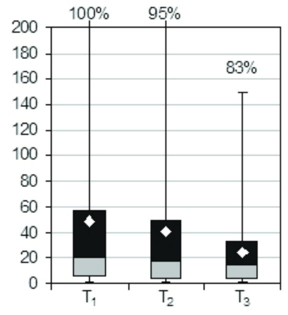
4.5. Discussion

The findings of the evaluation show that combining web mining with an existing feature location technique results is a more effective approach (**RQ1**). Additionally in the context of feature location, HITS is a more effective web mining algorithm than PageRank (**RQ2**). The most effective techniques evaluated were ${IR_{LSI}Dyn_{bin}WM_{HITS(h,freq)}}^{bot}$ and ${IR_{LSI}Dyn_{bin}WM_{HITS(h,bin)}}^{bot}$. The results indicate that filtering bottom-ranked hub methods from $IR_{LSI}Dyn_{bin}$'s results is the most effective approach from both the perspective of the position of the first relevant method and of all relevant methods. For instance, for one feature in Eclipse, IR_{LSI} ranked the first relevant method at position 1,696, and for $IR_{LSI}Dyn_{bin}$, the best rank of a relevant method was at position 61. On the other hand, ${IR_{LSI}Dyn_{bin}WM_{HITS(h,bin)}}^{bot}$, ranked the first relevant method to the feature at position 24. Filtering the bottom HITS hub methods eliminated 37 false positives from the results obtained by the state of the art technique. Examining the results in detail reveals why. Methods with high hub values call many other methods, while methods that do not make many calls have low hub values. These bottom-ranked hub methods are generally getter and setter methods or other methods that do not make any calls and perform very specific tasks. The ${IR_{LSI}Dyn_{bin}WM_{HITS(h,bin)}}^{bot}$ technique prunes these methods from the results since they are not relevant to the feature, thus improving effectiveness.

T$_1$: $IR_{LSI}Dyn_{bin}$
T$_2$: $IR_{LSI}Dyn_{bin}$ with getters and setters filtered
T$_3$: $IR_{LSI}Dyn_{bin}WM_{HITS(h,bin)}{}^{bot}$

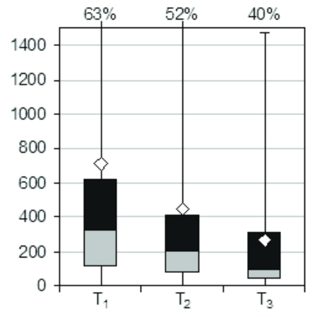


(a) Eclipse

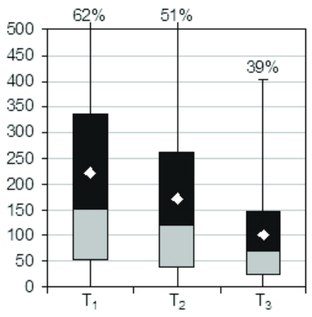


(b) Rhino

Figure 12 Comparing the effectiveness measure for the baseline technique (T$_1$), filtering getters and setters from the baseline (T$_2$), and one of the most effective techniques based on using web mining as a filter (T$_3$). The values above the boxes represent the percentage of features for which the technique was able to locate at least one relevant method.



(a) Eclipse

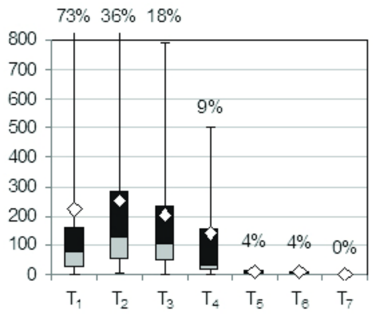


(b) Rhino

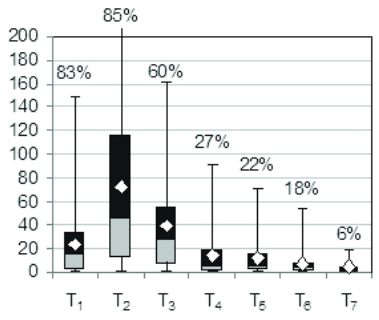
Figure 13 Comparing the average position of all gold set methods for the baseline (T$_1$), filtering getters and setters from the baseline (T$_2$), and one of the most effective techniques based on using web mining as a filter (T$_3$). The values above the boxes represent the percent of all the gold set methods the technique could locate.

The two most effective techniques remove bottom-ranked hub methods, and these methods tend to be getters and setters. To see if a simpler filtering heuristic is more effective that using web mining, a technique that filters out all getter and setter methods from $IR_{LSI}Dyn_{bin}$'s results was compared to two approaches: $IR_{LSI}Dyn_{bin}WM_{HITS(h,freq)}{}^{bot}$ and $IR_{LSI}Dyn_{bin}WM_{HITS(h,bin)}{}^{bot}$. **Figure 12** shows the average effectiveness measure of the baseline (*T$_1$*), the baseline with getter and setter methods pruned (*T$_2$*), and $IR_{LSI}Dyn_{bin}WM_{HITS(h,bin)}{}^{bot}$ (*T$_3$*). In both Eclipse and Rhino, removing getters and setters from $IR_{LSI}Dyn_{bin}$'s results is not as effective as $IR_{LSI}Dyn_{bin}WM_{HITS(h,bin)}{}^{bot}$. Similarly, when considering the ranks of all of a feature's relevant methods, the most effective technique is still $IR_{LSI}Dyn_{bin}WM_{HITS(h,bin)}{}^{bot}$, as seen in **Figure 13**. Therefore, using the HITS web mining algorithm and filtering bottom-

T1: $IR_{LSI}Dyn_{bin}WM_{HITS(h,bin)}{}^{bot}$
T2: $IR_{LSI}Dyn_{bin}$, filter fan-in ≤ 1
T3: $IR_{LSI}Dyn_{bin}$, filter fan-in ≤ 2
T4: $IR_{LSI}Dyn_{bin}$, filter fan-in ≤ 3
T5: $IR_{LSI}Dyn_{bin}$, filter fan-in ≤ 4
T6: $IR_{LSI}Dyn_{bin}$, filter fan-in ≤ 5
T7: $IR_{LSI}Dyn_{bin}$, filter fan-in ≤ 10

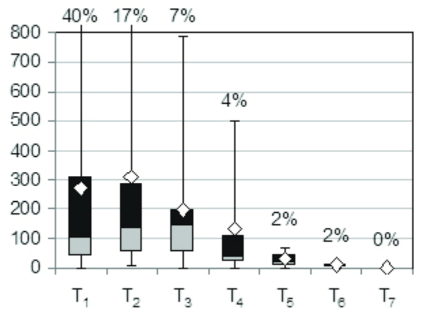


(a) Eclipse

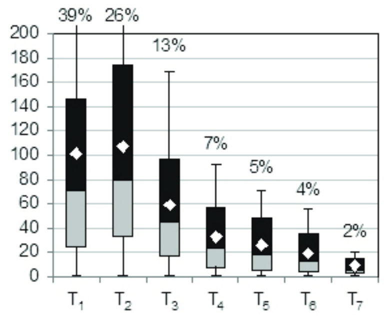


(b) Rhino

Figure 14 Comparing the effectiveness measure for one of the most effective techniques based on using web mining as a filter (T1) and techniques based on filtering methods with certain fan-in values from $IR_{LSI}Dyn_{bin}$'s results (T2 – T7). The values above the boxes represent the percentage of features for which the technique was able to locate at least one relevant method.



(a) Eclipse



(b) Rhino

Figure 15 Comparing the average position of all gold set methods for one of the most effective techniques based on using web mining as a filter (T1) and techniques based on filtering methods with certain fan-in values from $IR_{LSI}Dyn_{bin}$'s results (T2 – T7). The values above the boxes represent the percent of all the gold set methods the technique could locate.

ranked hub methods eliminates more false positives than simply pruning getter and setter methods.

In addition to investigating the filtering heuristic of eliminating getter and setter methods, another simplified heuristic was explored in which methods with certain fan-in values are pruned from $IR_{LSI}Dyn_{bin}$'s results. The fan-in of a module is defined as the number of locations from which control is passed in to the module (Henry and Kafura 1981) and is derived from a static call graph. Fan-in is similar to web mining. Both count the number of incoming links/calls to a page/method. The difference is that the web mining algorithms are more powerful because they incorporate indirect information. Not only are the number of incoming links counted, but the importance of those incoming links is considered as well. For instance, the PageRank of a page is based upon how many other pages link to the page and the PageRank of those pages. Similarly with HITS, page's authority score is based on how many hubs point to it, not just the total number of pages that link to it. The web mining algorithms are defined recursively (see Sections 2.3.1 and 2.3.2) to capture this indirect information. Another difference between this work and research on using fan-in is that web mining is applied to a dynamic call graph, while fan-in is computed from a static call graph.

Figure 14 compares the effectiveness of $IR_{LSI}Dyn_{bin}WM_{HITS(h,bin)}{}^{bot}$ to several techniques based on filtering methods with certain fan-in values from the ranked list produced by $IR_{LSI}Dyn_{bin}$. For instance, T_3 prunes all methods with a fan-in value less than or equal to 2. In both Eclipse and Rhino, the approaches that filter more methods have lower average effectiveness. However, these techniques are only able to locate at least one gold set method for a smaller percentage of all the features. The results are similar when the rankings of all of a feature's methods are considered, as seen in **Figure 15**. Therefore, using fan-in as a filtering heuristic is too naïve and simplistic because it eliminates too many of a feature's relevant methods, unlike using web mining.

Concerning the use of execution frequency or binary weights, the results do not show a significant difference between the two, nor is one consistently more effective than the other. However, one observation is that in Rhino, binary weights were more effective, likely because the Rhino traces had many loops which artificially inflated the execution frequencies of many of the methods. Using binary weights avoided this situation.

Each of the analyses used in the data fusion model have their own costs and overheads that must be weighed against the benefits of using the techniques. The main cost associated with LSI is indexing the corpus, which for large corpora can take several minutes, depending on many factors such as the size of the corpus and CPU speed. However, this is a one-time cost and can be performed incrementally when the source code changes (Jiang et al. 2008). Gathering execution information by collecting traces is probably the most expensive analysis used in the model in terms of both time and space. Tracing a program's execution can impose considerable overhead and significantly slow down execution speed (Cornelissen et al. 2009). Collecting a trace of a large system such as Eclipse could take an hour. Additionally, the collected trace will be large in size, possible over a gigabyte (see **Table 2**). Collecting multiple traces requires sufficient storage space to save them all. The final type of analysis used in the framework is web mining. Running the web mining algorithms can take several minutes for a large system. Like indexing with LSI, this is a one-time cost.

4.6. Threats to Validity

There are several threats to validity of the evaluation presented in this article. Conclusion validity refers to the relationship between the treatment and the outcome and if it is statistically significant. Since no assumptions were made about the distribution of the effectiveness measures, a non-parametric statistical test was used. The results of the test showed that the improvement in effectiveness of most of the web mining based feature location techniques over the state of the art is significant.

Internal validity refers to if the relationship between the treatment and the outcome is casual and not due to chance. The effectiveness measure is based on the position of a feature's first relevant method, and the relevant methods are defined by a gold set. In Eclipse, the gold set was defined by bug report patches. These patches may contain only a subset of the methods that implement a feature, and sometimes the methods were not implemented until a later version. In Rhino, the gold set methods were defined manually by other researchers who were not system experts. In jEdit, the gold set was extracted from SVN commits. Thus, relevant methods could be missing from the gold sets of each system. This threat is minimized by the fact that the patches were approved by the module owners and the Rhino data has been previously used by other researchers (Eaddy et al. 2008a; Eaddy et al. 2008b).

Another threat to internal validity pertains to the collection of data from IR and dynamic analysis. Information retrieval requires a query. The queries in this evaluation were taken directly from bug reports and documentation. It is possible that the queries used do not accurately reflect the features being located or that the use of different queries with vocabularies more inline with the source code would yield better results. However, using these default queries instead of formulating our own eliminated the introduction of bias. Similarly, execution traces were collected for each feature based on either the bug reports or test cases. The collection of these traces may not have invoked all of a feature's relevant methods or may have inadvertently invoked another feature. This is a threat to validity common to all approaches that use dynamic analysis. The use of test cases distributed with the software reduces this threat since the tests were created by the system's authors.

External validity concerns whether or not the results of this evaluation can be generalized beyond the scope of this work. Three open source systems written in Java were evaluated. Eclipse is large enough to be comparable to an industrial software system, but Rhino and jEdit are only medium-sized. Additional evaluations on other systems written in other languages are needed to know if the results of this study hold in general.

5. Related work

5.1. Feature Location

Existing feature location techniques can be broadly classified by the types of analysis they employ, be it static, dynamic, textual, or a combination of two or more of these. This section reviews some of the related work that is most relevant to the work presented here and explains the key differences between this work and the related work. For a comprehensive survey on feature location techniques, please refer to (Dit et al. 2011).

There are several static approaches to feature location. Chen and Rajlich (Chen and Rajlich 2000) proposed the use of Abstract System Dependence Graphs (ASDG) as a means of static feature location, whereby users follow system dependencies to find relevant code. Robillard (Robillard 2005; Robillard 2008) introduced a more automated static approach that analyzed the topology of a system's dependencies. Saul et al. (Saul et al. 2007) applied the HITS algorithm on subsets of the static callgraph in order to recommend methods related to a method given as a starting point. Harman et al. (Harman et al. 2002) used hypothesis-based concept assignment (HB-CA) (Gold and Bennett 2002) and program slicing to create executable concept slices and found that these slices can be used to decompose a system into smaller executable units corresponding to concepts (features) (Binkley et al. 2008). In this work, instead of using static information, textual and dynamic data are used to get results that are more tailored to a specific feature.

Software reconnaissance (Wilde and Scully 1995) is a well-known dynamic approach to feature location. Two execution traces are collected: one that invokes the feature of interest and another that does not. The traces are compared, and methods invoked only in the feature-specific trace are deemed relevant. SPR (Antoniol and Guéhéneuc 2006) is another dynamic feature location technique in which statistical hypothesis testing is used to rank executed methods. Our work employs dynamic information for feature location, but uses it as a filter to textual information instead of directly locating a feature from pure dynamic analysis.

Textual feature location was introduced by Marcus et al. (Marcus et al. 2004) when they applied LSI to source code. The approach has been extended to include relevance feedback (Gay et al. 2009), where users indicate which results are relevant, and a new query is automatically formulated from the feedback. Textual analysis of source code is not limited to LSI. Hill et al. (Hill et al. 2009) also use natural language processing (NLP) and the idea of query expansion and refinement in their approach to feature location. NLP analyzes the parts of speech of the words used in source code. Grant et al. (Grant et al. 2008) employ Independent Component Analysis (ICA) (Comon 1994) for feature location. ICA is an analysis technique that separates a set of input signals into statistically independent components. For each method, the analysis determines its relevance to each of the signals, which represent features. This work relies on LSI as opposed to other analyses because LSI is the *de facto* standard.

In addition to these techniques based on a single type of analysis, there are many hybrid approaches. Both SITIR (Liu et al. 2007) and PROMESIR (Poshyvanyk et al. 2007) combine textual and dynamic analysis. FLAT[3] (Savage et al. 2010) provides tool support for SITIR. Eisenbarth et al. (Eisenbarth et al. 2003) applied formal concept analysis (Ganter and Wille 1996) to execution traces and combined the results with an approach similar to ASDGs. This approach involves human input and does not produce ranked results, so it was not included it in the evaluation. Dora (Hill et al. 2007) and SNIAFL (Zhao et al. 2006) incorporate information from textual and static analysis. Rohatgi et al. (Rohatgi et al. 2009) proposed an approach that combines dynamic and static analysis. Cerberus (Eaddy et al. 2008a) is the only hybrid approach that combines static, dynamic, and textual analyses. Dora and Cerberus do not produce ranked results, but SNIAFL does, so future work involves comparing the new web mining based techniques to it.

There are some feature location techniques that are not based on textual, dynamic, or static analyses. Robillard and Dagenais (Robillard and Dagenais 2008; Robillard and Dagenais 2010) also use historical information from a repository for feature location. They use change history to identify clusters of program elements related to a task (i.e., a feature). Given a query of a set of program elements, their approach groups repository transactions by the number of nearest-neighbor program elements they share and returns a cluster of elements related to the query. Various filtering heuristics can be applied to the results to remove program elements that are unlikely to be related. For instance, if a program element is modified in a high percentage of all of the transactions in the repository, it can be ignored.

Hipikat (Cubranic and Murphy 2003; Cubranic et al. 2005) is a feature location approach that also makes use of archival information for feature location, but instead of identifying candidate program elements, Hipikat recommends artifacts from a project's archives such as online documentation, versions, bugs, or communications. Hipikat forms a group memory (Cubranic et al. 2004) from a project's history as recorded in source code repositories, issue trackers, communication channels, and web documents. Links between these artifacts are inferred using IR. For example, a source code version can be linked to a bug report if the bug's id is included in a repository commit log message. This history is used to find relevant artifacts in response to a user query. The query consists of an artifact, potentially a program element, for which the user wants recommendations of related artifacts. Hipikat responds with a list of artifacts ranked by their relevance.

No existing feature location techniques rely on web mining. However, web mining has been used for other program comprehension tasks. The HITS algorithm has been used on a dependence graph of a system weighted with dynamic coupling measures to identify the classes that are most important for understanding the software (Zaidman et al. 2006; Zaidman and Demeyer 2008). Saul et al. (Saul et al. 2007) also used HITS to recommend related API calls. SPARS-J (Inoue et al. 2005) is a system that analyzes the usage relations of components in a software repository using a ranking algorithm that is similar to PageRank. Components that are generic and frequently reused are ranked highly. Li (Li 2009) also uses a variant of PageRank called Vertex Rank Model (VRM) to refine concept bindings found using HB-CA. The VRM works on a dependence graph of concept bindings to identify statements that can be removed from the concept bindings without losing domain knowledge. Also, approaches based on Information Foraging Theory that rely on link analysis have been applied in the context of software maintenance tasks (Lawrance et al. 2007).

Aspect mining is closely related to feature location. The goal of aspect mining is to identify concerns[11] that are scattered throughout a system's modules so that they can be refactored in to their own modules known as aspects. The concerns are not known a priori, whereas in feature location, the features of interest are known before searching begins. Marin et al. (Marin et al. 2004; Marin et al. 2007) use fan-in to identify concerns that can be refactored in to aspects. Methods with high fan-in are called from many different locations within the system, and thus possibly represent a scattered concern. Other aspect mining approaches have employed the idea of data fusion by combining multiple techniques (Shepherd et al. 2005) including fan-in, clone detection (Bruntink et al. 2004; Shepherd et al. 2004; Bruntink et al. 2005), and natural language analysis (Shepherd et al. 2007).

5.2. Studies on Maintenance and Evolution

Feature location is an integral part of software maintenance and evolution. Many studies have been conducted investigating what programmers look for when performing change tasks and how they look for it. Robillard et al. (Robillard et al. 2007) performed an empirical study in which 23 programmers were asked to determine the source code that implements 16 concepts (features) in four different software systems written in Java. In analyzing the mappings produced by the programmers, Robillard et al. found that the amount of agreement for a feature ranged from 0% to 61%, with an average of 34%, meaning there is a large amount of variability in the source code programmers consider related to a feature.

Ferret (de Alwis and Murphy 2008) is a tool for answering conceptual queries, which are questions about a software system a programmer may have while performing maintenance and evolution. The model Ferret is based on supports the composition and integration of different sources of information into a queryable knowledge-base. A source of information is known as a sphere, and examples include structural relationships in source code, dynamic call information from an execution trace, and revision history recorded in a software repository. Ferret supports 36 different conceptual queries such as "What calls this method?", "What are this class' subclasses?", "What are all the fields

[11] A concern is an area of interest or focus in a system. Features can be concerns, but not all concerns are features.

declared by this type?”, and “What transactions changed this element?”. These types of queries represent questions programmers may have when investigating a software system in order to locate a feature’s implementation.

Sillito et al. (Sillito et al. 2008) also explored the kinds of questions programmers ask while performing software maintenance. They conducted two user studies with the goal of determining what programmers need to learn about a system before evolving it and how programmers find the information they need. Based on the results of their studies, they cataloged 44 types of questions programmers ask during maintenance tasks and described activities aimed at answering those questions.

Starke et al. (Starke et al. 2009) conducted a similar study focused on the challenges programmers face when trying to find information needed for a maintenance task and how tool support can be improved. From a case study involving ten programmers assigned change tasks on a large system, Starke et al. had five key observations. The observations most relevant to feature location are that the programmers only skimmed the search results instead of thoroughly investigating them all and only opened a small number of the results to view their source code. Based on their observations, they conclude that tools should support skimming and ranking of the results.

6. Conclusion

This work has introduced a data fusion model for feature location. The basis of the model is that combining information from multiple sources is more effective than using the information individually. Feature location techniques based on web mining and approaches using web mining as a filter to information retrieval were instantiated within the model. A large number of features from three open source Java systems were studied in order to discover if feature location based on combining IR and web mining is more effective than the current state of the art and which of two web mining algorithms is better suited to feature location.

The results of an extensive evaluation reveal that new feature location techniques based on using web mining as a filter are more effective than the state of the art, and that their improvement in effectiveness is statistically significant. Future work includes instantiating the model with different IR techniques and investigating when static call graphs are acceptable to use. All of the data used to generate the results presented in this work is made publically available to other researchers who wish to replicate these case studies.

7. Acknowledgements

We are grateful to the anonymous EMSE and ICPC 2010 reviewers for their relevant and useful comments and suggestions, which helped us in significantly improving the earlier versions of this paper. This work is supported in part by NSF CCF-0916260 and NSF CCF-1016868 grants. Any opinions, findings, and conclusions expressed herein are the authors’ and do not necessarily reflect those of the sponsors.

8. References

Antoniol G. and Guéhéneuc Y. G. (2006) Feature Identification: An Epidemiological Metaphor. IEEE Transactions on Software Engineering **32**(9): 627-641.

Biggerstaff T. J., Mitbander B. G. and Webster D. E. (1994) The Concept Assignment Problem in Program Understanding. 15th IEEE/ACM International Conference on Software Engineering (ICSE'94) 482-498.

Binkley D., Gold G., Harman M., Li Z. and Mahdavi K. (2008) An empirical study of the relationship between the concepts expressed in source code and dependence. The Journal of Systems and Software **81**: 2287–2298.

Blei D. M., Ng A. Y. and Jordan M. I. (2003) Latent Dirichlet Allocation. Journal of Machine Learning Research **3**: 993-1022.

Brin S. and Page L. (1998) The Anatomy of a Large-Scale Hypertextual Web Search Engine. 7th International Conference on World Wide Web, Brisbane, Australia, 107-117.

Bruntink M., van Deursen A., Tourwe T. and van Engelen R. (2004) An Evaluation of Clone Detection Techniques for Identifying Crosscutting Concerns. 20th IEEE International Conference on Software Maintenance (ICSM'04), Chicago, Illinois, IEEE Computer Society: Los Alamitos CA, 200-209.

Bruntink M., van Deursen A., van Engelen R. and Tourwe T. (2005) On the Use of Clone Detection for Identifying Crosscutting Concern Code. IEEE Transactions on Software Engineering (TSE) **31**(10): 804-818.

Chen K. and Rajlich V. (2000) Case Study of Feature Location Using Dependence Graph. 8th IEEE International Workshop on Program Comprehension (IWPC'00), Limerick, Ireland, 241-249.

Comon P. (1994) Independent Component Analysis, a New Concept? Signal Processing **36**(3): 287-314.

Conover W. J. (1998) Practical Nonparametric Statistics, Third Edition, Wiley.

Cooley R., Mobasher B. and Srivastava J. (1997) Web Mining: Information and Pattern Discovery on the World Wide Web. 9th IEEE International Conference on Tools with Articial Intelligence (ICTAI'97), 558-567.

Cornelissen B., Zaidman A., van Deursen A., Moonen L. and Koschke R. (2009) A Systematic Survey of Program Comprehension through Dynamic Analysis. IEEE Transactions on Software Engineering (TSE) **35**(5): 684-702.

Cubranic D. and Murphy G. C. (2003) Hipikat: Recommending pertinent software development artifacts. 25th International Conference on Software Engineering (ICSE'03), Portland, OR, 408-418.

Cubranic D., Murphy G. C., Singer J. and Booth K. S. (2004) Learning from Project History: a Case Study for Software Development. 2004 ACM Conference on Computer Supported Cooperative Work (CSCW'04), Chicago, Illinois, USA, ACM, 82-91.

Cubranic D., Murphy G. C., Singer J. and Booth K. S. (2005) Hipikat: A Project Memory for Software Development. IEEE Transactions on Software Engineering **31**(6): 446-465.

de Alwis B. and Murphy G. C. (2008) Answering Conceptual Queries with Ferret. 30th International Conference on Software Engineering (ICSE’08), Leipzig, Germany, 21-30.

Deerwester S., Dumais S. T., Furnas G. W., Landauer T. K. and Harshman R. (1990) Indexing by Latent Semantic Analysis. Journal of the American Society for Information Science **41**(6): 391-407.

Dit B., Revelle M., Gethers M. and Poshyvanyk D. (2011) Feature Location in Source Code: A Taxonomy and Survey. Journal of Software Maintenance and Evolution: Research and Practice (JSME): to appear.

Eaddy M., Aho A. V., Antoniol G. and Guéhéneuc Y. G. (2008a) CERBERUS: Tracing Requirements to Source Code Using Information Retrieval, Dynamic Analysis, and Program Analysis. 16th IEEE International Conference on Program Comprehension (ICPC'08), Amsterdam, The Netherlands, 53-62.

Eaddy M., Zimmermann T., Sherwood K., Garg V., Murphy G., Nagappan N. and Aho A. V. (2008b) Do Crosscutting Concerns Cause Defects? IEEE Transaction on Software Engineering **34**(4): 497-515.

Eisenbarth T., Koschke R. and Simon D. (2003) Locating Features in Source Code. IEEE Transactions on Software Engineering **29**(3): 210 - 224.

Ganter B. and Wille R. (1996) Formal Concept Analysis. Berlin, Heidelberg, New York, Springer-Verlag.

Gay G., Haiduc S., Marcus M. and Menzies T. (2009) On the Use of Relevance Feedback in IR-Based Concept Location. 25th IEEE International Conference on Software Maintenance (ICSM'09), Edmonton, Canada, 351-360.

Gold N. and Bennett K. (2002) Hypothesis-based Concept Assignment in Software Maintenance. IEE Proceedings-Software **149**(4): 103-110.

Grant S., Cordy J. R. and Skillicorn D. B. (2008) Automated Concept Location Using Independent Component Analysis 15th Working Conference on Reverse Engineering (WCRE'08), Antwerp, Belgium, 138-142.

Harman M., Gold N., Hierons R. and Binkley D. (2002) Code Extraction Algorithms which Unify Slicing and Concept Assignment. 9th Working Conference on Reverse Engineering (WCRE'02), Richmond, VA, 11-21.

Henry S. and Kafura D. (1981) Software Structure Metrics Based on Information Flow. IEEE Transactions on Software Engineering (TSE) **7**(5): 510-518.

Hill E., Pollock L. and Vijay-Shanker K. (2007) Exploring the Neighborhood with Dora to Expedite Software Maintenance. 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE'07), 14-23.

Hill E., Pollock L. and Vijay-Shanker K. (2009) Automatically Capturing Source Code Context of NL-Queries for Software Maintenance and Reuse. 31st IEEE/ACM International Conference on Software Engineering (ICSE'09), Vancouver, British Columbia, Canada.

Inoue K., Yokomori R., Yamamoto T., Matsushita M. and Kusumoto S. (2005) Ranking significance of software components based on use relations. IEEE Transactions on Software Engineering (TSE) **31**(3): 213- 225.

Jiang H., Nguyen T., Che I. X., Jaygarl H. and Chang C. (2008) Incremental Latent Semantic Indexing for Effective, Automatic Traceability Link Evolution Management. 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE'08), L'Aquila, Italy.

Kleinberg J. M. (1999) Authoritative Sources in a Hyperlinked Environment. Journal of the ACM **46**(5): 604-632.

Lawrance J., Bellamy R. and Burnett M. (2007) Scents in Programs: Does Information Foraging theory Apply to Program Maintenance? IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'07), IEEE, 15-22.

Li Z. (2009). Identifying High-Level Dependence Structures using Slice-based Dependence Analysis. King's College London, University of London. **Ph.D.**

Liu D., Marcus A., Poshyvanyk D. and Rajlich V. (2007) Feature Location via Information Retrieval based Filtering of a Single Scenario Execution Trace. 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE'07), Atlanta, Georgia, 234-243.

Lukins S., Kraft N. and Etzkorn L. (2008) Source Code Retrieval for Bug Location Using Latent Dirichlet Allocation. 15th Working Conference on Reverse Engineering (WCRE'08), Antwerp, Belgium, 155-164.

Marcus A., Sergeyev A., Rajlich V. and Maletic J. (2004) An Information Retrieval Approach to Concept Location in Source Code. 11th IEEE Working Conference on Reverse Engineering (WCRE'04), Delft, The Netherlands, 214-223.

Marin M., van Deursen A. and Moonen L. (2004) Identifying Aspects using Fan-in Analysis. 11th IEEE Working Conference on Reverse Engineering (WCRE'04), Delft, The Netherlands, 132-141.

Marin M., van Deursen A. and Moonen L. (2007) Identifying Crosscutting Concerns using Fan-in Analysis. ACM Transactions on Software Engineering and Methodology (TOSEM) **17**(1): 1-34.

Porter M. (1980) An Algorithm for Suffix Stripping. Program **14**(3): 130-137.

Poshyvanyk D., Guéhéneuc Y. G., Marcus A., Antoniol G. and Rajlich V. (2007) Feature Location using Probabilistic Ranking of Methods based on Execution Scenarios and Information Retrieval. IEEE Transactions on Software Engineering **33**(6): 420-432.

Revelle M. and Poshyvanyk D. (2009) An Exploratory Study on Assessing Feature Location Techniques. 17th IEEE International Conference on Program Comprehension (ICPC'09), Vancouver, British Columbia, Canada, 218-222.

Robillard M. (2005) Automatic Generation of Suggestions for Program Investigation. Joint European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering, Lisbon, Portugal, 11 - 20

Robillard M. P. (2008) Topology Analysis of Software Dependencies. ACM Transactions on Software Engineering and Methodology **17**(4): 1-36.

Robillard M. P. and Dagenais B. (2008) Retrieving Task-Related Clusters from Change History. 15th Working Conference on Reverse Engineering (WCRE'08), 17-26.

Robillard M. P. and Dagenais B. (2010) Recommending Change Clusters to Support Software Investigation: an Empirical Study. Journal of Software Maintenance and Evolution: Research and Practice **22**(3): 143-164.

Robillard M. P., Shepherd D., Hill E., Vijay-Shanker K. and Pollock L. (2007). An Empirical Study of the Concept Assignment Problem. Montreal, Quebec, Canada, McGill University.

Rohatgi A., Hamou-Lhadj A. and Rilling J. (2009) An Approach for Solving the Feature Location Problem by Measuring the Component Modification Impact. IET Software **3**(4): 292-311.

Salton G. and McGill M. (1983) Introduction to Modern Information Retrieval, McGraw-Hill.

Saul M. Z., Filkov V., Devanbu P. and Bird C. (2007) Recommending Random Walks. 11th European Software Engineering Conference held jointly with 15th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE'07), Dubrovnik, Croatia, 15-24.

Savage T., Revelle M. and Poshyvanyk D. (2010) FLAT^3: Feature Location and Textual Tracing Tool. 32nd ACM/IEEE International Conference on Software Engineering (ICSE'10), Cape Town, South Africa, 255-258.

Shepherd D., Gibson E. and Pollock L. (2004) Design and Evaluation of an Automated Aspect Mining Tool. Mid-Atlantic Student Workshop on Programming Languages and Systems (MASPLAS '04).

Shepherd D., Palm J., Pollock L. and Chu-Carroll M. (2005) Timna: a Framework for Automatically Combining Aspect Mining Analyses. 20th IEEE/ACM international Conference on Automated Software Engineering (ASE'05), Long Beach, CA, USA, 184-193.

Shepherd D., Pollock L. and Vijay-Shanker K. (2007) Case Study: Supplementing Program Analysis with Natural Language Analysis to Improve a Reverse Engineering Task. 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'07), San Diego, California, USA, ACM, 49-54.

Sillito J., Murphy G. C. and De Volder K. (2008) Asking and Answering Questions during a Programming Change Task. IEEE Transactions on Software Engineering (TSE) **34**(4): 434-451.

Starke J., Luce C. and Sillito J. (2009) Searching and Skimming: An Exploratory Study. 25th IEEE International Conference on Software Maintenance (ICSM'09), Edmonton, Alberta, Canada.

Wilde N. and Scully M. (1995) Software Reconnaissance: Mapping Program Features to Code. Journal of Software Maintenance: Research and Practice **7**: 49-62.

Zaidman A. and Demeyer S. (2008) Automatic Identification of Key Classes in a Software System using Webmining Techniques. Journal of Software Maintenance and Evolution: Research and Practice **20**(6): 387-417.

Zaidman A., Du Bois B. and Demeyer S. (2006) How Webmining and Coupling Metrics Improve Early Program Comprehension. 14th IEEE International Conference on Program Comprehension (ICPC'06), Athens, Greece, 74-78.

Zhao W., Zhang L., Liu Y., Sun J. and Yang F. (2006) SNIAFL: Towards a Static Non-interactive Approach to Feature Location. ACM Transactions on Software Engineering and Methodologies (TOSEM) **15**(2): 195-226.