Combining Probabilistic Ranking and Latent Semantic Indexing for Feature Identification

Denys Poshyvanyk[1], Yann-Gaël Guéhéneuc[2], Andrian Marcus[1*], Giuliano Antoniol[3], Václav Rajlich[1]

[1]*Department of Computer Science*
Wayne State University
Detroit Michigan 48202
denys@cs.wayne.edu,
amarcus@cs.wayne.edu
rajlich@cs.wayne.edu

[2]*GEODES – Research Group on*
Open, Distributed Systems,
Experimental Software Engineering
University of Montreal, Canada
guehene@iro.umontreal.ca

[3]*Département de Génie*
Informatique,
École Polytechnique de Montréal,
Canada
antoniol@ieee.org

Abstract

The paper recasts the problem of feature location in source code as a decision-making problem in the presence of uncertainty. The main contribution consists in the combination of two existing techniques for feature location in source code. Both techniques provide a set of ranked facts from the software, as result to the feature identification problem. One of the techniques is based on a Scenario Based Probabilistic ranking of events observed while executing a program under given scenarios. The other technique is defined as an information retrieval task, based on the Latent Semantic Indexing of the source code.

We show the viability and effectiveness of the combined technique with two case studies. A first case study is a replication of feature identification in Mozilla, which allows us to directly compare the results with previously published data. The other case study is a bug location problem in Mozilla. The results show that the combined technique improves feature identification significantly with respect to each technique used independently.

1. Introduction

Software evolution requires adding new functionalities to software systems, improving existing functionalities, and removing bugs, which can be considered as unwanted functionalities. Identifying the parts of the source code that correspond to a specific functionality is a prerequisite to evolution and is one of the most common activities undertaken by software engineers. In software engineering, this process is called *concept location* [19]. The concepts that represent a functionality of a system accessible and visible to the users, usually captured by the requirements explicitly, are called *features*.

While the developers often perform concept location manually, tool support is needed for large and complex software. Existing tools that support concept location rely on the information that is gleaned from the program by static and/or dynamic analysis. These analyses can provide a large number of facts and extracting the relevant facts is an information retrieval task, similar in nature with those from other fields like data mining or web searching. In this context, *precision* is used to measure the number of false positives returned by a query and *recall* is used to measure the number of false negatives. In all cases, the users judge the retrieved results.

Results often have low precision and/or recall, when querying repositories of purely static or of purely dynamic data. In particular, dynamic analyses are often unable to distinguish between overlapping features. Indeed, the same code region may contribute to several features (i.e., several features execute the same code). It is difficult to separate overlapping features, thus impacting precision. While static analyses may filter and organize the facts, they rarely identify entities contributing to a specific execution scenario exactly.

The research community has recognized the need to combine static and dynamic data to improve program understanding and to help in feature identification [2, 7, 9, 10, 21]. All current hybrid techniques share a common assumption: the fact that a statement belongs to an execution trace or that a module is activated by a feature are deterministic information, which are often expressed as Boolean values.

However, it is not always possible to state and quantify facts as deterministic quantities. Imprecision of the measure, uncertainty in the environment, perturbation phenomena, or simply the lack of knowledge, may require to express relations and facts in non deterministic terms [6]. When we execute a specified scenario, we have a deterministic relation between the trace and the scenario. Yet, we cannot be certain whether a called method and/or accessed field is

[*] A. Marcus is visiting in the Department of Mathematics and Computer Science at Babeş-Bolyai University Cluj-Napoca, Romania

Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC'06)
0-7695-2601-2/06 $20.00 © 2006 IEEE

relevant to the feature, thus there is a non-deterministic relation between the scenarios and the features.

Our goal is to use both certain and uncertain knowledge extracted with both static and dynamic analyses, filter it by probabilistic and information retrieval techniques, and in this way to identify features in source code. Static and dynamic knowledge are thought of as collaborating *experts* providing their valuable expertise and judgments in parallel. As in other domains, such as medical or financial domains, our goal is to combine the *subjective* judgments of these experts to improve our understanding of the facts.

In the presented work, we reformulate the feature identification problem as a decision-making problem in the presence of uncertainty. We are in the same position as a manager who must combine expert subjective forecasts [11, 26]. We have previously developed two techniques for feature identification. The developed techniques exploit different sources of information and provide complementary results. The first technique is based on Latent Semantic Indexing (LSI) [5] of the source code [16]. The second technique is a Scenario Based Probabilistic (SBP) ranking of events, observed while executing a program under given scenarios [2].

Using LSI, programmers can query static documents (i.e., classes, methods, and documentation), indexed via LSI, to obtain a ranked list of facts likely to be relevant to a feature. The role of the query is to capture some semantic characteristics of the feature of interest.

Using SBP, programmers can analyze dynamic traces of execution scenarios and get a list of entities (e.g., methods and classes) ranked according to their likelihood to be relevant to a feature exercised under the given scenarios.

The problem is that both techniques provide a different uncertain judgment and, to improve our knowledge, we must combine these judgments. The LSI and SBP ranking techniques could be considered our experts; the ranked lists are the judgments of these experts. We combine the respective ranked lists produced via an affine transformation where the affine coefficients express our confidence in the two experts and their ability to identify features correctly.

We compare the resulting combination with known results and thus gather evidence that the combination outperforms the judgments of each single expert. We perform the comparison in two case studies. First, we apply the combination to the scenarios presented in an earlier case study from [2]. Results clearly show the superiority of the combination. In the second case study, we identify methods and classes involved in a documented bug. We compare methods identified as being relevant to the bug with those actually contained in the official patch applied to fix the bug. We show that the combination indeed identifies the relevant methods with better precision and recall than either of the techniques does individually.

The remaining sections of this paper are organized as the following. Section 2 presents an overview of the related work on dynamic and static techniques for feature and concept location. It also briefly introduces necessary background information on our techniques. Section 3 presents our novel, hybrid technique for feature identification. The evaluation of the new technique via two case studies is presented in Section 4. The conclusions and future work are outlined in Section 5.

2. Previous Work

Existing techniques for concept location and feature identification fall into three broad categories: using static data, using dynamic data, and using both. We cover the main work on these techniques in subsection 2.1. Then, we present background information from our previous work on feature identification using the SBP ranking [2] in Section 2.2, and on concept location using LSI [16] in Section 2.3. Sections 2.2 and 2.3 introduce the formalism required to describe our novel technique in Section 3.

2.1. Related Work on Concept Location

The work by Wilde et al. [24] and Biggerstaff et al. [3] are the starting points of much of the work on concept and feature location, including the present paper.

Wilde and Scully [25] proposed the dynamic technique to identify features by analyzing execution traces of test cases. They use two sets of test cases to build two execution traces: an execution trace where the desired functionality is exercised and an execution trace where the functionality is not used. The two traces are compared to identify the parts of the program that implement the feature(s) associated with the functionality. In their work, the authors use only dynamic data to identify features; no static analysis on the program is performed. The method was recently extended [2, 6, 8].

Chen and Rajlich [4] proposed a semi-automated technique for static concept location based on searching on Abstract System Dependence Graph (ASDG). Maintainers identify features manually using the ASDG following a precise process.

Marcus et al. [16] proposed an information retrieval-based technique for static concept location. A comparison of several static concept location techniques is presented in [15].

Zhao et al. [27] proposed a static and non-interactive method for feature identification, which uses

information retrieval technique to reveal the basic connections between features and functionalities in the source code. A branch-reserving call graph is used to further recover both the relevant and the specific computational units for each feature.

Combining previous techniques, Eisenbarth et al. [7] used both static (i.e., dependencies) and dynamic (i.e., execution traces) data to identify features in software. They also used concept analysis techniques to relate features together.

Salah and Mancordis [20] also use both static and dynamic data to identify features in Java programs. They went beyond feature identification by creating feature-interaction views, which highlight dependencies among features.

2.2. Feature Identification using Probabilistic Ranking

The key steps of the process of identifying features and studying their evolution with a probabilistic ranking technique are summarized in this subsection. Static and dynamic analyses are used to extract data from executions of several releases of a system, following given scenarios.

We borrow from a previous work [2] key definitions and equations. A feature links program architecture with its dynamic behavior. Thus, a first step is to recover the program architecture. Second, a subset of the program architecture, a micro-architecture, must be identified as participating in the implementation of the functionality. Third, intra- and inter-feature relationships across releases are studied to highlight feature evolution.

We acquired and developed several parsers and tools to analyze existing software statically with reasonable precision. In particular, we developed our own C++ parser, which manages the previous degrees of imprecision, to generate AOL files [1]. AOL files are higher-level representations of object-oriented systems (classes, methods, relationships), which are simple to handle programmatically.

We assume that the source code is available and that a compiled version can be exercised under different scenarios. Dynamic analysis provides the necessary source of data to link functionalities (features) with software constituents and thus, to identify micro-architectures responsible for the specific implementation of functionalities.

We reuse our previous technique [2] inspired by Wilde's [6]. We instrument and generate traces of the executions of a software, given a certain scenario, which exercises a functionality of interest. We experimented with processor emulation on the Mozilla web-browser, using Valgrind[1], with satisfying results, to collect execution traces. We compared processor emulation with profiling techniques[2] and found that processor emulation collects more accurate data than profiling techniques, with little performance overhead.

We associate events in the execution trace with the functionality using a relevance index, a ranking quantifying the probability that an event is relevant to the functionality. We concur with Wilde [23, 25] to avoid the use of set operations. Avoiding set operations implies using thresholds and maintainer interactions to validate identified features. However, we can limit the use of thresholds and minimize maintainer interactions by ranking the events according to their relevance to the feature of interest.

Let F^* be a set of scenarios exercising a functionality of interest and F a set of scenarios not exercising the functionality. Execution of scenarios in F^* produces a set of intervals I^* containing events relevant to the functionality of interest. We mark these intervals as relevant via *Start/Stop* signals. Scenarios in F always produce intervals in I, intervals irrelevant to the functionality. However, intervals I^* *(I,* respectively) may contain irrelevant (relevant) events. Indeed, any scenario is likely to decompose in a few intervals in I^* surrounded by many intervals in I. If N_{I^*} and N_I are the overall numbers of events in the two sets I^* and I, then the frequency of a relevant event e_i in I^* is $f_{I^*}(e_i) = N_{I^*}(e_i)/N_{I^*}$ and its frequency in I is $f_I(e_i) = N_I(e_i)/N_I$. The relevance index of e_i is then:

$$r(e_i) = \frac{f_{I^*}(e_i)}{f_{I^*}(e_i) + f_I(e_i)} \quad \textbf{(1)}$$

Equation (1) is a *renormalization* of Wilde's equation, where events are re-weighted by population sizes (frequencies) to make events comparable directly. We use equation (1) to classify events in the I^* and I sets with respect to the functionality of interest. Then, we use equation (2) to keep the most relevant events with respect to a positive threshold t.

$$\mathrm{E}_t^* = \{e_i | r(e_i) > t\} \quad \textbf{(2)}$$

The size of the set E_t^* depends on t: 100.00% means that we only retain events that do not appear in intervals I and that, thus they are most relevant to the functionality.

[1] http://valgrind.kde.org/

[2] Profiling techniques do not collect traces but time spent in functions and methods along with callees and callers; such data can be used to build traces.

2.3. Concept Location using Latent Semantic Indexing

Developers usually use regular expressions when searching for the location of concepts, in the absence of sophisticated tools. They look for identifiers and comments, as these encode domain knowledge in the source code. We use LSI [5] to improve the search by allowing users to formulate more relaxed queries and to obtain ranked results at different levels of granularity.

Previous work covers in detail the use of LSI to index software elements for retrieval purposes [14, 16]. In summary, LSI works much like today's popular search engines (e.g., Google) by creating a signature (in this case a real valued vector) for each element of interest in the source code. We provide tool support [18] to define these elements of interest, which can be classes, methods, functions, interfaces, parts of documentation, etc. Once defined, these are extracted from the source code (i.e., in form of comments and identifiers) and used to generate a semantic space, which is used for the search. Users can write queries, which are also converted into vectors in the semantic space and the results of the search are returned as a list of software elements, ranked by their similarity to the user query.

We use as similarity measure between a query and a source code element, the cosine between their corresponding vectors. The cosine between two vectors v_q, corresponding to a query q, and v_i, corresponding to a source code element i (which can be a method or a class), in the semantic space, is the length-normalized inner product:

$$\cos(v_q, v_i) = \frac{\mathrm{v_q}^{\mathrm{T}}\mathrm{v_i}}{|\mathrm{v_q}|_2 \times |\mathrm{v_i}|_2} \qquad \textbf{(3)}$$

This similarity measure yields values between [-1, 1] for any pair of vectors, with 1 corresponding to identical documents. Negative values are associated to non-related documents.

We developed a set of tools [16, 18] and a methodology [15] to use the information retrieval technique for concept location in source code. We decompose the methodology in the following steps:

1. The semantic space is generated using the available tools, at a user defined granularity level.
2. Select a set of words/terms that describe the concept. This set of words constitutes the query.
3. Check whether a term is present in the vocabulary of the software system (generated by LSI). If the term is not present, then:
   a. Look up similar words using the vocabulary of the software system (e.g., use a spell-checker based on an editing distance function to suggest similar terms);
   b. Eliminate from the initial query the words that are not in the vocabulary. If the elimination of a word significantly alters the meaning of the query, go to step 2 and select additional words for the query.
4. Run the query with LSI on the search space. The query returns a list of source code elements ordered by their similarity to the query.
5. Examine the source code documents from the list in the order they appear in the results. For every document examined in the results a decision is required whether the document is part of the concept or not, if it is and it covers all the aspects of the concept, then stop. If it is not and the new knowledge obtained from the investigated documents helps to formulate a better query (e.g., narrow down the search criteria), go to step 2; else examine the next document in the list.

In essence, the similarity between a query expressing some semantic characteristics of a feature and a set of facts about the software (e.g., manual pages, documentation, classes or methods of a program), indexed via LSI, allows producing a ranking of entities relevant to the feature.

3. Combining the Experts

We introduce our novel technique to improve the precision of feature identification by combining the SBP [2] and LSI [16] ranking techniques. We consider the SBP and LSI rankings as the judgments of our experts. Experts provide expertise to solve the problem of identifying the feature precisely. We draw inspiration from Jacobs [11] to combine the SBP and LSI rankings. When n experts are present, judgments can be combined using the following equation:

$$f(x) = \sum_{i=1}^{n} \lambda_i \beta_i f_i(x) \qquad \textbf{(4)}$$

where $f_i(x)$ is the judgment of the i^{th} expert, λ_i is a weight expressing our confidence in the i^{th} expert and β_i is a re-normalization constant. We use re-normalization constants, because the different experts may express judgments that are not commensurable. We use these constants to map judgments and to allow meaningful combinations. In the more general case, β_i is a function of the input value x. The β_i coefficient can also be selected so that $f_i(x)$ lies in the interval [0, 1]. In any case, the judgments of the experts should be in a same interval, thus imposing the constraints that the weight λ_i defines an affine transformation: $\sum_{i=1}^{n} \lambda_i = 1$.

Our experts state judgments based on different information. The SBP expert grounds its judgment on the probabilistic ranking of dynamic events observed in execution traces. The LSI expert builds its judgment from a query on a set of documents, created with static data extracted from the source code and index with LSI. However, we ask, in different ways, both experts to answer the same question: we want to identify a set of particular facts related to a feature of interest for a maintainer. Thus, we combine the valuable expertise of both experts to obtain a more precise set of relevant events in order to minimize the maintainer's effort.

In this paper, we focus on identifying classes and methods relevant to a feature of interest. Thus, we are interested in combining our experts' judgments on methods that contribute to the feature. These methods are likely to be placed on top of both the SBP and LSI rakings, because we ask the two experts, the same query conceptually.

Let x be a method. For the sake of clarity, we denote with $sbp(x)$ and $lsi(x)$ the relevance scores assigned to x by our experts without detailing the actual query performed. $lsi(x)$ and $sbp(x)$ are not defined over a same domain. Thus, the combination of the two experts' judgments with equation (4) requires a re-normalization of the relevant scores. This normalization must not disrupt the LSI and SBP experts' judgments, because we want to promote methods that both experts consider highly relevant.

The $sbp(x)$ relevance score is defined on [0, 1] while the $lsi(x)$ score takes values in [-1, 1]. Among several possible renormalizations, we select two transformations: variable standardization and simple normalization. The latter transformation is grounded on the fact that LSI negative values are irrelevant. Thus the re-normalized SBP and LSI scores can be obtained as follows: ($r_{lsi}(x)$ has the same domain as $sbp(x)$)

$$r_{sbp}(x) = sbp(x)$$
$$r_{lsi}(x) = \begin{cases} lsi(x) & if\ lsi(x) > 0 \\ else \quad 0 \end{cases} \qquad \textbf{(5)}$$

For the former transformation of variable standardization, we remap $lsi(x)$ and $sbp(x)$ in a standard normal distribution with zero mean and unitary variance via the transformation [22]:

$$r_{sbp}(x) = \frac{sbp(x) - mean(sbp(x))}{stdev(sbp(x))}$$
$$r_{lsi}(x) = \frac{lsi(x) - mean(lsi(x))}{stdev(lsi(x))} \qquad \textbf{(6)}$$

This is equivalent to considering experts' judgments as being generated by Gaussian sources with different means and variances. Thus, we bring the relevance scores back to the tabulated N(0,1) before combining them. In the case studies, we did not observe any substantial difference on the rankings from the transformations for renormalization used to compute $r_{lsi}(x)$ and $r_{sbp}(x)$. Figure 1 shows the ranking for the first case study (see Section 4.3) based on the equations in (6), while the rest of the results in this paper are reported only for the equations in (5).

The combination of the two experts leads us to rewrite equation (4) as:

$$r_{combined}(x) = \lambda r_{sbp}(x) + (1-\lambda) r_{lsi}(x) \qquad \textbf{(7)}$$

where λ expresses our confidence on the ability of the SBP or LSI ranking experts to identify a feature correctly using their particular techniques. Equation (7) allows ranking methods based on the combined score. However, we may be interested to further reduce such a list. This can be done by retaining the k-top ranked methods or by imposing a threshold on $r_{combined}(x)$.

The combination of the techniques allows for some changes in the existing methodologies. The SBP ranking was originally done in combination with a knowledge-based filtering [2], using the previous experience of the user with the system under investigation. LSI ranking is in fact a mechanism that allows the user to gain knowledge on how elements of the software relate to each other. Thus, LSI ranking could replace the knowledge-based filtering in the combined technique.

4. Case Studies

We performed two case studies to assess the precision of the novel hybrid technique for feature identification. The first case study is a replication of a case study [2], which used the SBP ranking only. We compare previous results with the LSI ranking and our hybrid technique to assess their respective precisions.

The second case study focuses on bug location, which we consider as an undesirable feature. We apply the techniques (i.e., SBP ranking, LSI ranking, and the novel hybrid ranking) independently and compare their results.

These two case studies provide data to assess the precision of the combination of the SBP and LSI rankings and the help brought to the maintainer for identifying features in the source code.

In addition, to the authors' knowledge, these case studies are the largest (in terms of the size of the subject software) that were done on using LSI to index source code. In fact, the size of the software and its

vocabulary is one order of magnitude larger than natural language corpora used in previous experiments with LSI. Given this situation, we use these case studies to see how the LSI dimensionality reduction factor influences the results.

Table 1. Mozilla v1.6 size related statistics

Item	Count (MLOC)	Item	Count
Header files	8,055 (1.50)	Classes	4,853
C files	1,762 (0.90)	Methods	53,617
C++ files	4,204 (2.00)	Specializations	5,314
IDL files	2,399 (0.20)	Associations	17,362
XML files	283 (0.12)	Aggregations	6,727
HTML files	2,231 (0.19)		
Java files	56 (0.06)		

4.1. Object of the Case Studies

We use Mozilla, a large object-oriented multi-threaded web-browser [17], as the object of the case studies. Mozilla is an open-source web browser ported on almost all known software and hardware platforms. It is large enough to represent an industrial size program. It is developed mostly in C++ with C code (which accounts only for a small fraction of the program). We do not analyze parts of Mozilla written in other programming languages, like C, Java, IDL, XML, HTML, etc., to simplify the case studies.

The latest version of Mozilla includes more than 10,000 source files consisting of about 3.7 MLOC (millions lines of code), which are decomposed in about 3,500 different subdirectories. Mozilla consists of 90 modules maintained by 50 different module owners. In our case studies, we use version 1.6.

Table 1 shows some statistics for Mozilla v1.6. The reported figures should be considered as orders of magnitude rather than as exact values. Indeed, several factors influence these figures, such as the reverse engineering tools, the parsing techniques used [12] and the way in which we consider certain programming language features.

We choose to use conservative reverse-engineering techniques. We apply strict reverse-engineering rules such that we classify as classes only entities declared as such according to the C++ syntax. Moreover, we consider structures and complex templates (e.g., templates mixed with structures) as outside of the boundary of reverse-engineered models and do not recover their attributes, methods, and locations in source code files.

Table 2. LSI corpus vitals for Mozilla v1.6

MLOC	4.4
Vocabulary	85,439
Number of parsed documents	68,190
Number of methods	48,267
Number of functions	19,923

4.2. Design of the Case Studies

In each study, the use of the SBP ranking follows the pattern described in Section 2.2. The SBP ranking technique provides sets of ranked methods with equation (1) using their frequencies in the execution traces. Several methods may have the same frequencies in the traces and thus, the same ranking. For example, for the first case study, the SBP ranking provides a set of 274 methods ranked first, $r_{sbp}(e_i) = 1$ in equation (1). Thus, the recall of the SBP ranking is good but its precision is low.

We follow the methodology for LSI ranking described in Section 2.3. Creating the LSI search space, however, deviates from our previous experiences. We use the Mozilla source code to build and to index a semantic space to allow query-based searches for feature identification. LSI builds the semantic space corresponding to Mozilla, by projecting a 85,436×68,190 matrix (the vector space) onto a smaller one (a subspace), see Table 2. When applying LSI on natural language corpora, a space of 300 dimensionality is usually chosen [13]. On large software such as Mozilla, the size of the vocabulary is

Table 3. Results for locating the methods related to the "bookmark" feature. The position of the methods in the table (column 1) show the rank obtained by combining the two approaches. Numbers in parenthesis denote the ranks for those methods obtained with LSI alone. Columns 2-5 correspond to the different dimensionality of the LSI subspace.

Pos.	300	500	750	1,500
1	**CreateBookmark (3)**	**CreateBookmark (6)**	**AddBookmarkImmediately (1)**	**AddBookmarkImmediately (1)**
2	**AddBookmarkImmediately (4)**	**AddBookmarkImmediately (2)**	**CreateBookmark (14)**	**CreateBookmark (8)**
3	**CreateBookmarkInContainer(64)**	Flush	Flush	**CreateBookmarkInContainer(19)**
4	InsertResource	**CreateBookmarkInContainer(57)**	**CreateBookmarkInContainer(36)**	WriteBookmarks
5	ListenToEventQueue	InsertResource	WriteBookmarks	getFolderViaHint
6	Flush	WriteBookmarks	Observe	InsertResource

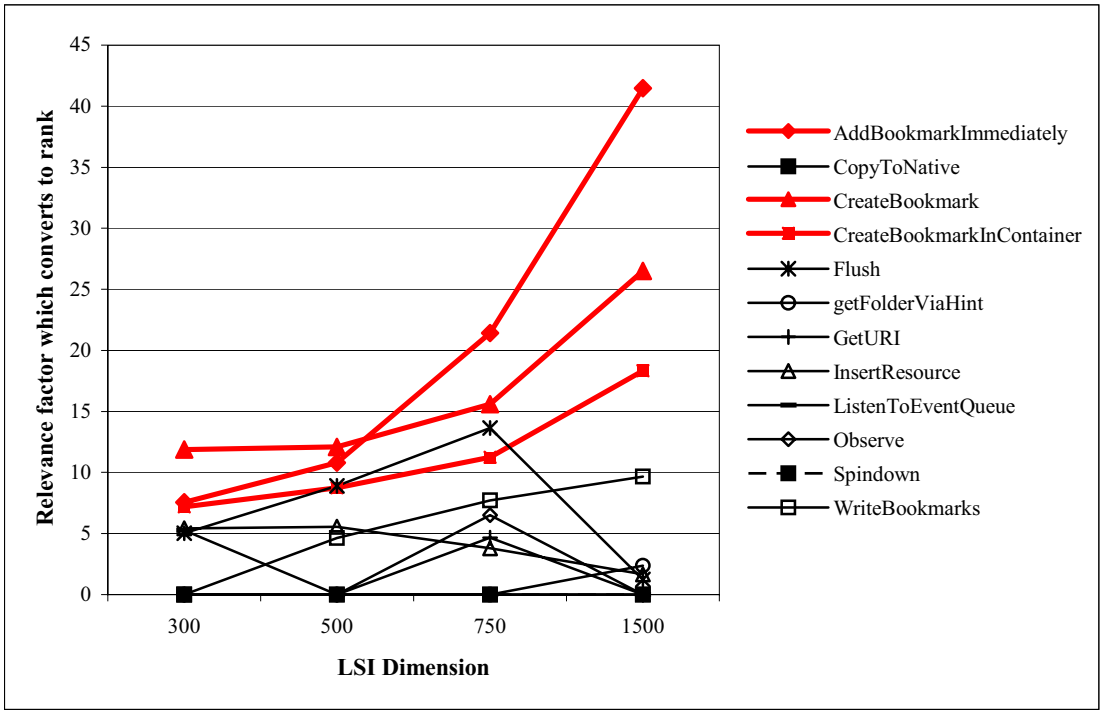

Figure 1. Rankings of relevant methods as the LSI dimensionality reduction factor increases for the first case study. The three relevant methods are highlighted with red color.

one order of magnitude larger than in natural languages (or in previously indexed source code). We adjust the dimensionality reduction factor and perform each query four times using as dimensionality reduction factors 300, 500, 750, and 1,500 respectively, to find the best (approximate) dimensionality reduction value by comparing the rankings. Any larger factor would generate a too large search space to be practical from a computational point of view on actual average computers. Table 3 and Figure 1 show how results improve in the first case study (see Section 4.3), as the LSI dimensionality reduction factor increases.

LSI, on the contrary to the SBP ranking, ranks each method differently and thus, has a good precision but may have a low recall. Since LSI creates the vector representation for each method based on its identifiers and comments, methods with small bodies and very few identifiers will not be ranked properly. Also, inheritance is not dealt with specifically here, so overridden methods in subclasses may also be miss-ranked by LSI in some cases. The SBP ranking is not perturbed in these situations.

4.3. The First Case Study

This first case study is partly a replication study of a previous case study published in [2]. We perform the same case study (same scenario, same feature identification task) to compare previous results from SBP ranking with new results from LSI and hybrid rankings. Such a partial replication is important because it allows comparing the three ranking techniques against one another. It is a partial replication study because we replicate the scenario and the task only, we do not apply again the SBP ranking, whose results are available elsewhere [2].

We consider two scenarios:

- *Scenario 1*: A user visits an URL. She opens Mozilla, clicks on a previously bookmarked URL, waits for the page to load, and closes the browser;
- *Scenario 2*: The user acts as before but, once the page is loaded, she saves the URL using the mouse right button and closes the web browser.

The feature identification task can be stated as: "*identify classes and methods in Mozilla that are part of the feature activated when a URL is saved in Scenario 1 with respect to Scenario 2*".

We apply the SBP ranking by running Mozilla according to the two scenarios and by collecting corresponding dynamic traces as detailed in [2]. We then apply equation (1) to produce sets of relevant methods to the feature of interest.

We apply the LSI ranking by formulating a query on the terms related to "*bookmark*" in the vocabulary of Mozilla generated during indexing of the corpus by LSI. We use our judgment to assess whether the terms

relate to the feature of creating a new bookmark. We create the following query: "*bookmark newbookmark bookmarkname bookmarkresource bookmarkadddate createbookmark insertbookmarkitem deletebookmark bookmarknode*". We do not have to spell-check the query terms because they are directly taken from the Mozilla vocabulary.

Table 3 summarizes the results obtained in identifying the feature in the first case study. The first column indicates the rank of the methods obtained when combining the SBP and LSI rankings with λ = 0.5. The methods in bold (i.e., CreateBookmark, AddBookmarkImmediately, and CreateBookmarkInContainer) are the methods realizing the functionality of interest, that we checked manually. The numbers in parenthesis show the LSI ranking of the methods obtained using r_{lsi} alone. Columns 2–5 correspond to the different dimensionality of the LSI subspace. Each of these three methods was ranked in the set of 274 methods with a relevance index (r_{sbp}) of 1 by the SBP ranking alone.

We obtain a better ranking of the relevant methods when combining the judgments of the two experts. The results also tend to improve overall when increasing the dimensions of the LSI space (see Figure 1), which supports our hypothesis that larger vocabularies warrant the usage of larger subspaces. The weight λ, for a given dimension, does not impact the ranking significantly. Different values for λ only re-order relevant methods.

Figure 1 shows that increasing the dimensionality reduction factor improves the ranking of the three relevant methods with respect to the top ten high-ranked methods, thus increasing the precision of our novel technique.

4.4. The Second Case Study

In the second case study, we locate a bug in Mozilla using our novel technique. We choose bug #182192[3], described as "*quotes (") are not removed from collected e-mail addresses*". Among possible bugs, the rationale to select this one was threefold. First, we were interested in a well known, documented, and reproducible bug. Second, to minimize the probability of obtaining good results by chance, we were interested in a bug that has no interaction with methods and classes involved in the first case study. Finally, we were looking for a bug with available and approved patch that was also actually fixed in recent Mozilla releases. Indeed, bug #182192 has all these characteristics; it was well known since early Mozilla releases; a patch was available; it was officially fixed in release 1.7. In this case, we also eliminate any potential bias given that none of the authors determine what part of the system corresponds to the feature of interest.

To apply SBP, we performed two scenarios:

- *Scenario1:* A user replies to an e-mail;
- *Scenario2:* A user performs the same action of Scenario 1, and on the same e-mail and, using the mouse, the user forces to collect the e-mail address of the sender.

The subsequent steps in obtaining the SBP ranking are the same as in the first case study. By comparing the two scenarios, methods and classes highly relevant to the process of e-mail address collection are spotted.

To obtain the LSI ranking, based on the bug description, we formulate the following query to retrieve related methods in Mozilla: "*collect collected sender recipient email name names address addresses addressbook*". We used the same technique as in the previous case study for the query formulation. LSI provides four lists of ranked methods for the four different dimensions.

The Bugzilla reports[4] contain the description of the bug fixes, including the methods CollectAddress and CollectUnicodeAddress (from nsAbAddressCollecter) responsible for the "unwanted" functionality that we are looking for in this case study.

Table 4. The top five methods related to the Mozilla bug after merging dynamic and LSI results with the corpus indexed using 1,500 dimensions; λ= 0.5; r_{lsi} is the rank of these methods obtained using LSI alone; methods highlighted in bold contain the fixes for the bug done by the developers.

Rank	Class name	Method name	r_{lsi}
1	nsMsgHeaderParser	ParseHeadersWithArray	2
2	nsMsgHeaderParser	ParseHeaderAddresses	4
3	***nsAbAddressCollecter***	***CollectAddress***	***37***
4	nsAddrDatabase	OpenInternal	36
5	***nsAbAddressCollecter***	***CollectUnicodeAddress***	***46***

Table 4 shows the ranking obtained with the combined methodologies (column 1) and the ranking obtained by using LSI alone (column 4). When using the SBP ranking 8,695 methods are retained and ranked; out of these, 206, including the CollectAddress, and CollectUnicodeAddress methods, obtain a score of 1.0. As in the previous case study, the SBP ranking provided high recall but low precision.

4.5. Discussion

The two case studies support our claim that combining expert judgments is effective in increasing

[3] https://bugzilla.mozilla.org/long_list.cgi?buglist=182192

[4] https://bugzilla.mozilla.org/attachment.cgi?id=147661&action=diff

the precision of feature identification in the large software systems. Our novel combination technique performs better than any one of the two techniques alone.

The case studies were carried out with the two different normalizations proposed in Section 3: the variable standardizations in equation (6) and the simple transformations in equation (5). We did not observe any substantial difference in ranking for the various LSI space dimensions. Ranking of relevant methods were exactly the same. Only in one experiment, for LSI using 1,500 dimensions, in the second case study, the rank of the CollectAddress method was exchanged with ParseHeaderAddresses (i.e., CollectAddress was ranked in position 2 and ParseHeaderAddresses in position 3). However, we do not have any empirical evidence to prefer one normalization technique over the other. More data and more experiments are needed to verify if there is really a significant difference between the two normalization techniques.

On our data set, we did not observe any major changes while ranging λ between 10% and 90%. Extreme values below 1% or above 99% obviously tend to perform more close to either the LSI or the SBP ranking.

5. Conclusions and Future Work

The main contributions of the paper can be identified as the followings:

- We proposed and defined a novel hybrid technique to combine an information retrieval based concept location technique with a dynamic technique for feature identification.
- We applied the proposed combination of techniques in a new case study for bug location in Mozilla. We also compared the results of the combined technique with our previous results on applying the dynamic technique only, through a replicated case study.
- The combined technique allows the elimination of the knowledge-based filtering, present in the previous SBP ranking technique. This has the advantage that the user does not need to acquire extensive knowledge of the target system a-priori.

The case studies showed that the two combined techniques, based on different analysis methods and data, are not only expressing different judgments in trying to identify features, but that these judgments are complementary. This is proved by the fact that the results obtained with the combined techniques betters any one of them used independently.

The work presented here also opens the door to future efforts in the area. More experiments are planned to compare results with other techniques. Also, a deeper investigation to determine heuristics that would identify the best value for the λ coefficient is under way. Last but not least, we are working on extending this approach such that it would combine several techniques for feature location.

6. Acknowledgements

This research was supported in part by grants from the National Science Foundation (CCF-0438970) and the National Institute for Health (NHGRI 1R01HG003491). G. Antoniol was partially supported from the Canada Research Chair program grant #950-202658. Václav Rajlich was partially supported from 2005 IBM Faculty Award.

7. References

[1] Antoniol, G., Fiutem, R., and Cristoforetti, L., "Using Metrics to Identify Design Patterns in Object-Oriented Software", in Proceedings of 5th IEEE International Symposium on Software Metrics (METRICS'98), Bethesda, MD, November 20-21 1998, pp. 23 - 34.

[2] Antoniol, G. and Gueheneuc, Y., "Feature Identification: A Novel Approach and a Case Study", in Proceedings of 21st IEEE International Conference on Software Maintenance (ICSM'05), Budapest, Hungary, September 25 2005, pp. 357-366.

[3] Biggerstaff, T. J., Mitbander, B. G., and Webster, D. E., "Program Understanding and the Concept Assignment Problem", *CACM*, vol. 37, no. 5, May 1994, pp. 72-82.

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

[5] Deerwester, S., Dumais, S. T., Furnas, G. W., Landauer, T. K., and Harshman, R., "Indexing by Latent Semantic Analysis", *Journal of the American Society for Information Science*, vol. 41, 1990, pp. 391-407.

[6] Edwards, D., Simmons, S., and Wilde, N., "An approach to feature location in distributed systems", Software Engineering Research Center 2004.

[7] Eisenbarth, T., Koschke, R., and Simon, D., "Locating Features in Source Code", *IEEE Transactions on Software Engineering*, vol. 29, no. 3, March 2003, pp. 210 - 224.

[8] Eisenberg, A. D. and De Volder, K., "Dynamic Feature Traces: Finding Features in Unfamiliar Code", in Proceedings of 21st IEEE International Conference on Software Maintenance (ICSM'05), Budapest, Hungary, September 25-30 2005, pp. 337-346.

[9] Eng, D., "Combining static and dynamic data in code visualization", in Proceedings of ACM SIGPLAN-SIGSOFT

Workshop on Program Analysis for Software Tools and Engineering (PASTE'02), Charleston, South Carolina, 2002, pp. 43-50.

[10] Greevy, O., Ducasse, S., and Girba, T., "Analyzing Feature Traces to Incorporate the Semantics of Change in Software Evolution Analysis", in Proceedings of 21st IEEE International Conference on Software Maintenance (ICSM'05), 2005, pp. 347-356.

[11] Jacobs, R., "Methods for combining experts' probability assessments", *Neural Computation*, vol. 7, no. 5, September 1995, pp. 867-888.

[12] Kollmann, R., Selonen, P., Stroulia, E., Systa, T., and Zundorf, A., "A study on the current state of the art in tool-supported UML-based static reverse engineering", in Proceedings of IEEE Working Conference on Reverse Engineering, Richmond, Virginia, October 29 - November 1 2002, pp. 22-33.

[13] Landauer, T. K., Foltz, P. W., and Laham, D., "An Introduction to Latent Semantic Analysis", *Discourse Processes*, vol. 25, no. 2&3, 1998, pp. 259-284.

[14] Maletic, J. I. and Marcus, A., "Supporting Program Comprehension Using Semantic and Structural Information", in Proceedings of 23rd International Conference on Software Engineering (ICSE'01), Toronto, Ontario, Canada, May 12-19 2001, pp. 103-112.

[15] Marcus, A., Rajlich, V., Buchta, J., Petrenko, M., and Sergeyev, A., "Static Techniques for Concept Location in Object-Oriented Code", in Proceedings of 13th IEEE International Workshop on Program Comprehension (IWPC'05), 2005, pp. 33-42.

[16] Marcus, A., Sergeyev, A., Rajlich, V., and Maletic, J., "An Information Retrieval Approach to Concept Location in Source Code", in Proceedings of 11th IEEE Working Conference on Reverse Engineering (WCRE'04), Delft, The Netherlands, November 9-12 2004, pp. 214-223.

[17] Mozilla, "Mozilla 1.6", URL, Date Accessed: October, http://www.mozilla.org, 2004.

[18] Poshyvanyk, D., Marcus, A., Dong, Y., and Sergeyev, A., "IRiSS - A Source Code Exploration Tool", in Industrial and Tool Proceedings of 21st IEEE International Conference on Software Maintenance (ICSM'05), Budapest, Hungary, September 25-30 2005, pp. 69-72.

[19] Rajlich, V. and Wilde, N., "The Role of Concepts in Program Comprehension", in Proceedings of IEEE International Workshop on Program Comprehension (IWPC'02), 2002, pp. 271-278.

[20] Salah, M. and Mancoridis, S., "A hierarchy of dynamic software views: from object-interactions to feature-interactions", in Proceedings of 20th IEEE International Conference on Software Maintenance (ICSM'04), Chicago, IL, September 11-14 2004, pp. 72-81.

[21] Salah, M., Mancoridis, S., Antoniol, G., and Di Penta, M., "Towards Employing Use-Cases and Dynamic Analysis to Comprehend Mozilla", in Proceedings of 21st IEEE International Conference on Software Maintenance (ICSM'05), Budapest, Hungary, September 2005, pp. 639-642.

[22] Snedcor, G. W. and Cochran, W. G., *Statistical Methods*, Iowa State University Press, 1989.

[23] Wilde, N., Buckellew, M., Page, H., Rajlich, V., and Pounds, L., "A Comparison of Methods for Locating Features in Legacy Software", *Journal of Systems and Software*, vol. 65, no. 2, February 15 2003, pp. 105-114.

[24] Wilde, N. and Gust, T., "Locating User Functionality in Old Code", in Proceedings of IEEE International Conference on Software Maintenance, Orlando, FL, November 1992, pp. 200-205.

[25] Wilde, N. and Scully, M., "Software Reconnaissance: Mapping Program Features to Code", *Software Maintenance: Research and Practice*, vol. 7, 1995, pp. 49-62.

[26] Winkler, R. and Clemen, R., "Multiple Experts vs. Multiple Methods: Combining Correlation Assessments", *Decision Analysis*, vol. 1, no. 3, September 2004, pp. 167-176.

[27] Zhao, W., Zhang, L., Liu, Y., Sun, J., and Yang, F., "SNIAFL: towards a static non-interactive approach to feature location", in Proceedings of 26th International Conference on Software Engineering (ICSE'04), May 2004, pp. 293-303.