

Information Retrieval Applications in Software Maintenance and Evolution

David Binkley* **Dawn Lawrie**

Loyola College

Baltimore, MD

21210-2699 USA

{binkley, lawrie}@cs.loyola.edu

Keywords Natural Language Processing, Software Engineering, Feature and Concept Location, Fault Prediction, Developer Identification, Program Comprehension, Impact Analysis, Traceability Links, Refactoring

Abstract

There is a growing interest in creating tools that can assist engineers in all phases of the software life cycle. This assistance requires techniques that go beyond traditional static and dynamic analysis. An example of such a technique applies information retrieval (IR), which exploits information found in a project's natural language. Such information can be extracted from the source code's identifiers and comments and in artifacts associated with the project, such as the requirements. The techniques described pertain to the maintenance and evolution phase of the software life cycle and focus on such problems as feature location and impact analysis. These techniques highlight the bright future that IR brings to addressing software engineering problems.

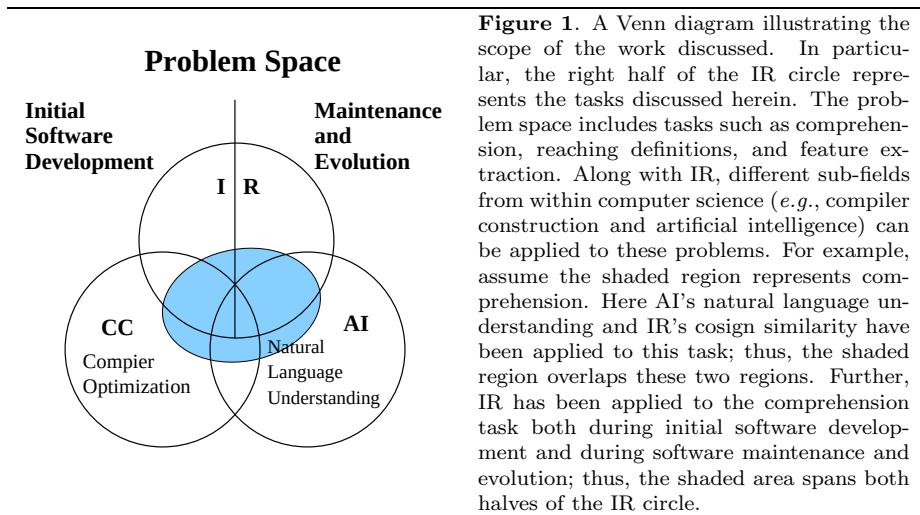
I INTRODUCTION

The goal of this article is to survey the application of Information Retrieval (IR) to the challenges encountered in SE during the *maintenance and evolution* of a software project – that is, problems encountered after the product’s initial release (*e.g.*, feature location, developer identification, and impact analysis). It is a companion to the previous chapter “Information Retrieval Applications in Software Development,” which considers the application of IR techniques to problems encountered during initial software development [1]. That chapter also provides some history, motivation, and forward-looking thoughts on the future of IR techniques in SE.

Figure 1 illustrates how the two articles relate to each other and to two other areas of computer science with overlapping goals. The left semi-circle of the circle labeled **IR** encompasses the application of IR techniques to the problems encountered during initial software development. The right semi-circle includes the applications considered in this chapter.

Most of the background terminology and glossary terms are the same and can be found at the beginning of the companion chapter. One term and one techniques, not required in the previous chapter, are defined here. First, the term *accuracy* denotes a measure of the total percentage of correct responses (the ratio of correctly predicted to all). Using the notation and example from the companion article, accuracy is defined as $A = (CN + CP)/(CN + FP + FN + CP)$ and in the example has the value 6/10. For large, diverse IR collections, this measure is often misleading as, for example, high accuracy can be attained by simply retrieving the empty set.

The technique not mentioned previously is *Latent Dirichlet Allocation* (LDA), which was introduced as a more theoretically sound and complex alternative version of LSI. In an LDA model, terms are automatically grouped together



into topics. The advantage that LDA brings is that each document is assumed to contain *multiple* topics. Furthermore, these topics have differing likelihoods (probabilities); thus, a document is represented using a collection of topic probabilities.

As with the companion chapter, this chapter favors techniques presented with sufficient technical detail to allow replication and is biased towards techniques that report results from empirical study. The applications are presented roughly in the order that these activities are found in the software life-cycle. Given the large literature base, the selected techniques and cited work should be taken as representative rather than exhaustive.

2 FEATURE / CONCEPT LOCATION

The following sections consider the application of IR to problems faced by SE during maintenance and evolution roughly in the order that they are encountered in the software life-cycle. Each section first briefly defines the problem

considered and then describes its importance and the current state of the art.

This is followed by a presentation of the relevant IR techniques.

To begin with, at present there is no widely agreed upon definition for feature location, concept location, or even the existence a difference between them. Some differentiate based on the analysis technique where concept location uses static analysis and feature location uses dynamic analysis (often of execution traces). Still others define features as those concepts captured by a use case and thus *stemming* from a functional requirement. Under this interpretation, all features are concepts, but not vise versa. Said another way, features are concepts associated with user visible functionality. Concepts that are not features stem from the problem or solution domain, for example, *security* (although to further complicate the issue some would argue that security is a *concern* and not a concept). While the remainder of this section uses the term feature location and the techniques described are dominated by static analyzes, it is agnostic as to the similarities and differences between feature, concept, and concern location. Feature location, traditionally performed as an intuitive and informal process, aims to identify parts of a software system that implement a specific functionality. It forms a prerequisite to many software evolution tasks, most notably program comprehension, and is thus one of the most common activities undertaken by maintainers.

Historically, most programmers use regular expression search (via **grep**, or an IDE variant) to help them locate items of interest in the source code and documentation. In this scenario, the feature extraction problem is still being solved at the line-by-line level. Recent tools for feature location have been based on static or dynamic analysis. These analyses often have low precision and recall. In particular, dynamic analyses are often unable to distinguish between overlapping features, while static analyses rarely identify entities contributing to a

specific execution scenario. To improve feature location, better tool support is needed for large and complex modern software. In fact feature location is one of the most common applications of IR to SE.

This section considers five techniques. The first is a clever “black-box” application. The next two use IR techniques as black boxes, but manipulate the results in some fashion. Finally, the last two approaches show increased integration of IR ideas into solutions for SE problems, a trend that is expected to continue.

The first technique uses LSI to accomplish feature location in an “off the shelf” technique [2]. In contrast to categorizing modules into concepts and non-concepts, this approach finds the modules related to a given feature based on a free-text user query. Queries can also be used to identify variables related to a term or phrase within the context of the software system. This avoids the user needing advanced knowledge of the terms used in the code.

By comparison with code-based methods, LSI is better able to identify words from the source code that relate to a user query within the context of the software system. In a case study using the 95,000 LoC C program, **mosaic**, the approach had an average precision of 12% (almost double existing dependence-based approaches) and outperformed text-based retrieval tools such as **grep**.

The second approach implements the serial combination of LSI, treated as a black-box, and Formal Concept Analysis (FCA) [3]. One advantage of FCA is that it provides an intentional description for each feature and offers the user useful cues through exploration of the resulting lattice.

The approach has seven steps. First, a corpus is created by extracting the identifiers and comments from each source code module. LSI is used to create an index for this corpus. A developer then provides a query (a set of terms that describe the feature of interest). Similarities between the user query and modules in the LSI space are used to return a ranked list of modules. In preparation for

FCA, the approach uses the top k descriptive terms from the first n modules in the ranked list, which are retained using Kuhn’s technique [3]. Using these modules as objects and descriptive terms as attributes, bottom-up FCA is used to build a concept lattice. Finally, the user can browse the lattice and refine the query as needed.

Testing with different configurations yields the recommendation that n range from 80 to 100 and k from 10 to 25. Fewer than 10 attributes produces weak clustering capacity, while more than 25 attributes creates too many lattice nodes. For example, locating the feature **print page** in the source code of Eclipse 3.16 returns the methods **getBounds**, **startPage**, **endPage**, **startJob**, **endJob**, and **cancelJob**. The following terms are selected from the identifiers and comments of these methods: **printer**, **print**, **page**, **job**, **device**, **paper** and **rendering** (these terms occur in the six methods but not in the rest of the source). The resulting lattice includes, for example, $C_1 = (\{\}, \{\textit{paper}\})$ at the top and $C_4 = (\{\textit{startJob}, \textit{endJob}, \textit{cancelJob}\}, \{\textit{print}, \textit{job}\})$ at the bottom. The technique was used to identify relevant features for two Eclipse bug fixes.

The third approach to feature location also uses existing techniques as black boxes but manipulates the output. In this instance, it casts the feature location problem as a decision-making problem in the presence of uncertainty [4]. The decision is based on the weighting of two ‘expert’ opinions. The first expert is LSI, where programmers can query static modules and obtain a ranked list of those modules likely to be relevant to a feature. LSI was chosen over other IR methods (*e.g.*, VSM) as it addresses the problems of polysemy and synonymy. The second expert is *Scenario Based Probabilistic* (SBP) extraction, where programmers can analyze dynamic traces of execution scenarios to extract a list of modules ranked according to their likely relevance to a feature exercised under the scenarios. SBP involves running the programs using two sets of scenarios,

one that exercises a functionality of interest and one that does not. For example, working on a bug related to saving a sender's email address involves two scenarios. First, the user replies to an email and saves the email address of the original sender. The second scenario is the same except the user does not save the email address.

Both techniques provide different *uncertain* judgments that are combined using an affine transformation where the affine coefficients express the confidence in the experts. One reason to expect the combination to work well is that the recall of the SBP ranking is good, but its precision is low, while LSI has good precision but low recall. Empirically tasked with finding modules relevant to a bug, combining the judgments produced higher ranking for relevant modules than either the LSI or SBP expert produced alone. Different values for the weights on the experts do not impact the resulting set of modules; they just produce different orderings.

The final two methods involve greater integration of IR techniques. In the first, Zhao et al. combine IR and static path analyses [5]. There are two inputs to this technique. The first is a collection of *all* the features in the software. These are used to identify *specific* and *relevant* modules for each feature. *Specific* modules are considered to be part of a single feature's implementation, while *relevant* modules, a superset of the specific modules, are involved in the implementation of one or more features.

The second input is the set of hard words automatically extracted from identifiers. (Soft words could be used, but at present they must be extracted manually [6].) To focus on the connection with other modules, only the 'important' identifiers (*e.g.*, a function's name and its formal parameters) are used.

The extracted hard words are used to form a VSM. The initial specific module sets for each feature are then generated in two steps. First, cosine similarity is

used to rank each module’s similarity to the feature. Then, for each feature list, a division-point is chosen as the largest difference between a consecutive pair of rankings. The modules before this division point form the initial set of specific modules used for the feature.

As a list of all relevant modules, the ranked list is inaccurate in two ways. First, it misses some modules due to the lack of corresponding descriptions in the feature, and second, it includes some unwanted modules because of the vagueness of natural language prose in describing a feature. These inaccuracies are overcome using the *branch-reserving call graph* (BRCG), which helps identify additional modules. The BRCG is an expansion of the CFG that includes branching and sequential information. Pruning the BRCG supports the efficient implementation of the following algorithm. Consider a call, c , to a method that is initially a specific module. Calls in the same block as c , and those in blocks within which it is nested, are relevant to c and thus to the corresponding feature. After determining the relevant modules of each feature, the final step is to determine the specific module sets. A module relevant to a feature and only to that feature is retained as a module specific to that feature.

An empirical study considered two programs. Of all 21 features in the first program, 66.67% were assigned the correct specific modules during the first step. This result improves to 85.71% after analyzing the relevant modules. Slightly better results were obtained with the second program. In comparing the two, it was clear that the feature descriptions in the second program were more precise than those of the first. IR techniques are highly subject to the quality of the input artifacts – in this case, program identifiers.

Compared to older static approaches, Zhao et al.’s method is fully automated. However, a maintainer must describe *all* the features before locating a specific feature. Furthermore, the approach is non-interactive, which prevents a main-

tainer from building up knowledge about the features while working with the system.

The second method to involve greater integration of IR techniques is unusual as it incorporates term proximity. Most IR techniques ignore the location of terms within an artifact because it usually does not improve performance enough to make it worth the computational expense. In this approach, Shepherd et al. mine aspects (cross cutting concerns) in requirements using a semi-automated approach that links the information about occurrences of verbs and their direct objects (DOs) [7]. In particular, a verb-direct object pair is defined to be two co-located identifiers in which the first identifier is an action or verb, and the second identifier is used as a direct object of the first.

To motivate the use of DOs, consider searching for a particular kind of *removal* in a program. Verbs such as *remove* frequently act on many different objects in a single program, such as *attribute*, *screen*, etc.

Verb and DO information is explicitly represented in the program's *action-oriented identifier graph* (AOIG), which has three levels of nodes. Each top layer node represents either a verb or a DO. The middle layer explicitly represents the pairing of a verb with a DO. The bottom layer includes a use node for each occurrence of a verb-DO pair in a module's comments or code.

An AOIG has two kinds of edges: a *pairing edge* connects a node in the top layer to the corresponding node in the middle layer, and a *use edge* connects verb-DO pair nodes in the middle layer to each use node found in the bottom layer. Although a verb or DO node may have edges to multiple verb-DO nodes, a verb-DO node has only two incoming edges: one from the verb node and one from the DO node involved in the relation.

A search takes as input two parallel queries: a *Verb Query* and a *Direct-Object Query*. Working with the associations in the AOIG, these two are expanded

(potentially with the aid of the user). For example, in an experiment starting from the query “**save**(verb query), **auctions**(DO query)”, the tool recommends adding the direct object **file**, as **file** is close to **auctions** in the AOIG. This recommendation leads to improved query effectiveness. Overall, the experiment demonstrated that the new approach was more consistent and more effective than Eclipse’s built-in lexical search and a modified Google Eclipse search.

3 FAULT PREDICTION

Fault prediction is the process of identifying faulty modules before an actual fault is detected. It permits, for example, engineers to concentrate their testing effort on the fault containing modules. Such concentration can reduce development costs and increase quality. Predicting fault-prone modules has been primarily addressed using software quality metrics. For example, the work of Gyimothy et al. correlates structural object-oriented metrics with faults [8]. In contrast, this section considers recently proposed IR-based solutions to this problem. These include two clever black-box approaches and one more involved technique.

The first approach applies spam email filter technology to the fault prediction problem [9]. Java source modules are treated as e-mail messages that are classified as *spam* when fault-prone and *ham* when not. Several classification strategies from the **CRM114** spam filtering software are used to predict faults. An experiment using the Java programs **ArgoUML** and **eclipseBIRT**, showed that the approach can classify up to 75% of software modules correctly. In more detail, the precision ranged from 52% to 75%, recall from 70% to 98%, and accuracy from 56% to 80%. One advantage of the IR-based approach is that there is no need to parse the source code in detail (in contrast, extracting metrics can be expensive).

The second approach applies the *QALP metric* to fault prediction [10]. This metric first builds a corpus where each module is split into two artifacts: one includes the soft words extracted from the source, while the other includes the header and in-line comments. The comments are stemmed and stopped using a standard English stop-list. The code is stopped using a programming language-specific stop-list that includes, for example, programming language keywords. Next, the cosine similarity between source code and corresponding comments is computed for each function using *tf-idf* weighting; thus, rating each on a scale from 0 to 1.

Linear mixed-effects regression is then used to build the fault-prediction models. These models predict the likelihood of a module being faulty. The results, while complex, show little correlation when applied to **Mozilla**. However, when applied to a proprietary program, statistically significant correlations exist. Of particular interest, the usefulness of the QALP metric in predicting faults increases as module size increases.

Finally, a variation on fault prediction is considered where bug-introducing changes are predicted [11]. This technique helps to prevent bugs *before* their introduction. The algorithm first extracts from a program's version history all changes that introduce bugs. This text is then used to train several machine learning algorithms. The output can be used to predict the probability that a subsequent change will introduce a bug.

Using the Weka tool-set, which provides a collection of machine learning algorithms (*e.g.*, K-Nearest Neighbors) [12], models were built to predict if a future change is likely to introduce a bug. Ten-fold cross-validation with the two small open-source systems **JHotDraw** and **DNSJava** supports the overall idea: precision ranged from 50% to 80% and recall from 10% to 58% [11].

4 DEVELOPER IDENTIFICATION

Developer identification seeks to identify the right developer for a task. It is useful primarily to project managers, especially when the number of developers is large (as is often the case with open source software). This section considers two different approaches to the problem.

The first approach applies two IR techniques. The first creates a *topic model* for a code using *Latent Dirichlet Allocation* (LDA) [13]. From this model, developer similarity is calculated using *Kullback-Leibler divergence* [13] based on reconstructing the probability distributions over topics for each author.

In more detail, a topic model is a *probability mass function* (a probability function where all the individual probabilities sum to one) that provides the probability that each word in the vocabulary relates to a given topic. Topic models represent topical divisions in the source code. The approach divides a program into k topics and thus k topic models. Matlab's *LDA-based AT* algorithm is then used to transform input *worked on* matrices, which identify the source code each developer has worked on, into two output matrices: an *artifact-topic* matrix and an *author-topic* matrix. These matrices associate different topics with artifacts and authors, respectively. Linstead et al. observe that (informally) the models produce a good topical division of their subject program, **Eclipse 3.0**.

The second step calculates developer similarity by converting the author-topic matrix to a similarity matrix that captures the pairwise “distance” between authors. Several methods were considered (*e.g.*, standard Euclidean measures and cosine similarity). The best results came from reconstructing the probability distributions over topics for each author using Kullback-Leibler divergence.

For the second technique, the best candidate developer is assumed to be the one that has completed similar requests in the past [14]. The algorithm creates a corpus of ‘artifacts’ for each candidate that includes past change requests

assigned to the developer and notes included by the developer during CVS commits. Change requests are then used to query the corpus using a probabilistic retrieval method with a complex term weighting function. The probability that an artifact a is relevant to query q , denoted $P(R|a, q)$, is defined as

$$P(R|a, q) = \sum_{t \text{ a term from } q} W(t) \quad (1)$$

where, after folding-in several tuning constants used by Canfora and Cerulo, the weight, W , for term t is given by

$$W(t) = \frac{2.2F(t)}{0.3 + 0.9RAL + F(t)} \log \frac{N}{NA(t)} \quad (2)$$

where $F(t)$ is the *frequency* of term t in the artifact, RAL is the *ratio* of the *artifact length* to the average artifact length, N is the number of artifacts in the collection, and $NA(t)$ is the number of artifacts in which term t appears.

Unlike most searches, in this application there is really only one relevant “document”: the developer assigned to the change. Therefore, the evaluation of the tool’s selection is done in terms of the percentage of change requests where the developer was ranked in the top N . Reported below are the results for $N = 1$, which is the strictest threshold because as N increases the retrieval can be “successful” even if the best developer appears lower in the ranking.

The evaluation considered two different programs, **KDE** and **Mozilla**. In each case half of the change requests were used as a training set and the remaining half were used as the test set. The correct developer was ranked first in 12% and 5% of the requests for **KDE** and **Mozilla**, respectively. The system does better with very active developers because there is more text ‘describing’ them. For example when restricted to the five most active developers, accuracy is 59% and 32%, respectively.

5 COMPREHENSION

Software comprehension tools assist engineers in understanding unfamiliar code. They are essential as economic pressure requires a maintenance engineer to quickly and effectively understand the subset of the code relevant to a maintenance request. Interest in the application of IR to comments and identifiers in support of program comprehension is largely due to the work of Maletic and Marcus [15]. Given this work is less than a decade old, techniques are just beginning to exploit natural language information; thus, IR (in particular natural language processing) has a bright future as a program comprehension aid.

This section highlights four comprehension techniques applicable during maintenance and evolution. IR also applies to comprehension tasks during initial software development. These uses are surveyed in a companion entry [1].

The first use of IR applies LDA ‘out of the box’ to extract domain level business concepts (*e.g.*, privacy), which often go undocumented (or poorly documented) making it hard to comprehend a system’s functionality [16]. What makes LDA appealing in this application is that it automatically produces three things: the topics, the words (hard words found in the source) that indicate topics, and the mixture topics found in each document. In this regard it generalizes techniques such as LSI as it supports multiple concepts within a document. In this application, the result is a set of topics and a distribution of these topics in each source file.

Empirical study finds the tool extracts a subset of the reasonable domain topics. Thus, LDA forms a satisfactory starting point for further manual refinement of topics. However, the authors note that “one disadvantage of LDA is that it does not derive any interrelationships between the extracted topics....” This comment provides one direction for future work.

The second technique creates a meaningful summary of a software module from extracted natural language information [17]. The summaries are then used to select classes for reuse. The algorithm begins by separately processing comments and code. Comments are parsed using a natural language parser. This task is simplified because comments are written in a limited number of sentence structures: present tense, simple past tense, or simple future tense. Some comments require minor modification (*e.g.*, prefixing them with the word *this*) to make them grammatically correct sentences. For the code, attribute and method names are extracted and then tagged with part-of-speech information before being combined with the parsed comments to construct a *concept graph*. The text associated with the internal nodes of this graph provide high-level descriptions of each corresponding concept. The tool was tested on three GUI software packages where the resulting descriptions largely matched those of experts. The third technique uses statistical properties of language to extract *key-concepts* that provide a concise word-based summary of a program [18]. This technique requires a simplification of *tf* and *idf* for a hard word *hw* where

$$tf(hw) = \begin{cases} 1 & \text{if } \exists a, tf(hw, a) > 1 \\ 0 & \text{otherwise} \end{cases}$$

$$idf(hw) = \begin{cases} 1 & \text{if } 1 \leq af(hw) \leq n \wedge !is_prefix(hw) \\ 0 & \text{otherwise} \end{cases}$$

and $tf(hw, a)$ is the term frequency of *hw* in artifact *a*, $af(hw)$ is the artifact frequency (the number of artifacts in which *hw* occurs), and $is_prefix(hw)$ is **true** iff *hw* is a prefix of all the identifiers in which it occurs.

A hard word is selected as a key-concept if it has a weight of one. This selection occurs iff its *tf* and *idf* scores are both non-zero. Thus, weights can be efficiently

computed because once a hard-word occurs in an artifact twice, its *tf* value is one and once it has been observed in two artifacts its *idf* is one.

To measure the effectiveness of the techniques, an educational operating system written by one of the authors was studied. This 5,000 LoC program includes 279 hard words manually classified into four categories: key-concepts (61), grouping words (18), attributes and less important words (70), and generic verbs (130). The algorithm finds 26% of the key-concepts with an accuracy of 57%. For comparison, traditional *tf-idf* finds 13% of the key-concepts with an accuracy of 28%; however, the traditional approach requires tuning the minimum threshold value – this value is not reported. As the technique is highly dependent on this threshold, a better comparison would present a range of threshold values.

The final technique uses LSI to create semantic clusters, which provide a developer with a first impression of an unfamiliar system [19]. Based on the hard words found in the identifiers and comments, LSI is used to determine the similarity between modules for the purposes of clustering. Term weights are computed using *tf-idf*; however, in the case of object-oriented code, inherited methods are added to the vocabulary of the modules at a reduced strength (*i.e.*, a weighting factor of 0.5 for each level of inheritance).

Modules are clustered in LSI-space using average-link *hierarchical agglomerative clustering* (HAC), which builds a dendrogram of the module clusters. Initially, each module is considered to be a singleton cluster. At each step, the two closest clusters are merged where distance is calculated as the average cosine-similarity of all pairs of modules in each of the clusters. The dendrogram is complete when all modules are merged into a single cluster. A threshold is applied to the dendrogram to determine the actual clusters that are created for a given project.

Each cluster is labeled with the top seven words calculated using the LSI-space to exclude common words in the project in favor of words that distinguish the cluster from other clusters. Terms are ranked using the following formula:

$$rel(t_0, C_0) = sim(t_0, C_0) - \frac{1}{|\mathcal{C}|} \sum_{C_n \in \mathcal{C}} sim(t_0, C_n)$$

where t_0 will have a high value in cluster C_0 if the similarity is high for cluster C_0 and low for all the other clusters of $\mathcal{C}$.

Finally, a Distribution Map is produced which colors each module based on the cluster it belongs to. The modules are then grouped by packages for Java applications to provide a visualization of the linguistic topics in each package. It should be noted that linguistic topics are not analogous to domain semantics. Instead they relate to application concepts and architectural components, as is demonstrated in a case study using **JEdit**. In addition, the terms selected as descriptions for the topics are highly dependent on the quality of identifiers since they are the source of the terms. Therefore, labels will be more informative to an engineer when good naming practices are followed.

6 IMPACT ANALYSIS

Impact analysis estimates the effort required to change a software system [20]; this task plays an important role in the software maintenance process because with knowledge of the effort required to make a change, management can better deploy resources. In addition, automating the identification of impacted modules simplifies the comprehension task for a maintainer. Non-IR approaches to impact analysis include the dependence-based approach of Kamkar et al., which uses static and dynamic slicing to identify affected modules [21]. This section

considers four IR-based techniques. The first three apply IR techniques “off the shelf” while the fourth is more involved.

The first approach begins by identifying the set of external artifacts that describe components directly or indirectly impacted by a change request [22]. From these the related source code modules can be identified (using, for example, the linking techniques described in Section 7). In a case study, the approach was investigated using the user’s manual of **LEDA** (Library of Efficient Data type and Algorithms) as the external documentation. Queries consisted of descriptions of maintenance tasks. Two retrieval methods were compared: VSM using cosine similarity to rank artifacts and a language model approach with a smoothed distribution of word frequencies. In the language modeling approach, artifact scores were computed as the probability that a word from the maintenance request would appear in the artifact. In the case study, recall peaked at 96% (where only artifacts with a score that exceeded 50% of the best possible score were retrieved), and precision was 40%.

The second technique uses Bugzilla, CVS, and previously implemented change requests to gather text related to a new change request [20]. Although Bugzilla and CVS are not formally indexed, common practice is to include the bug id in each commit comment. Therefore, it is straightforward to link change requests to source files. The underlying hypothesis behind this technique is that a new request will modify the same files as other similar requests. The retrieval method uses the probability of relevance as defined in Equation 1 from Section 4. Here the artifacts representing a module include all the change requests that have affected the module. Each change request includes commit comments, a short description, and a long description. Queries are comprised of a short description and an optional long description.

The implementation, as an Eclipse plug-in named **jimpa**, was evaluated on four different software projects: **Kcalc**, **Kpdf**, **Kspread**, and **Firefox**. In general, better retrieval in terms of recall and precision occurs with more text describing prior requests. For example, with **Kcalc**, when the ranked list was truncated at the first relevant artifact, precision was 78%. After 30 artifacts, recall was 98%. The third technique uses finer granularity to describe change requests [23]. Rather than considering all change requests for a module, each module is divided into sections using **diff**. An artifact is then created from the union of the change requests that impacted a particular section. Therefore, a module can be described by several artifacts. This variation requires modifying the scoring function of the probabilistic retrieval method to return the maximum score over all the individual artifacts related to the particular source file.

In a study involving **Gedit**, **Firefox**, and **ArgoUML**, this finer level of granularity led to a 10% to 20% increase in precision for the first artifact in the ranked list. However, it produced a dip in recall (at 100 artifacts) because a query that previously strongly matched an entire module by partially matching several of its sections only weakly matched the individual sections. In the extreme, this condition occurs when words of a query can be partitioned into those occurring in different sections; thus, the aggregate scores well, but not any of the individual sections. This dip is benign in cases where an engineer prefers the more selective retrieval.

The goal of the final impact analysis project is to refine **diff**'s output by classifying lines as deleted, added, or *changed* [24]. A line is considered changed when only a 'small' amount of the line is actually different. The approach first applies **diff** to identify ranges of differing code. **Diff** reports all differences as a deletion followed by an insertion. Classifying some pairs as changed is done in two steps. First, cosine similarity is applied to identify similar pairs. Pairs

whose Levenshtein Edit Distance falls below a specified threshold are considered changes. The order of these last two steps allows the comparatively inexpensive cosine similarity calculation to preselect candidates for the considerably more expensive edit distance calculation.

The technique was tested using 100 randomly selected samples from the open source project **ArgoUML**. Here, precision was defined to be the percentage of lines that the algorithm marked as *changed* where the oracle (an engineer) agreed, and recall was defined as the percentage of the lines that the oracle identified as *changed* where the algorithm agreed. It correctly identified changed lines with a precision of 96% and a recall of 95%.

7 TRACEABILITY LINKS

Traceability links tie together software artifacts from stake holders' initial requests to requirements specifications, design artifacts, models, reports, code, and test cases [25]. Maintaining these links is an arduous task. However, inadequate links is one of the main factors contributing to project cost over-runs and failures; thus, there is a need for tool support to establish or re-establish traceability links.

Given that link maintenance is a costly manual process, several IR-based automatic and semi-automatic techniques have been proposed. Like comprehension, IR finds application to this problem during initial software development as well as during maintenance and evolution. The initial software development applications are surveyed in a companion entry [1]. This section outlines three techniques that apply to software maintenance and evolution when traceability matrices were either not built or not kept up to date. It then considers a framework proposed to facilitate comparison and evaluation of traceability studies. The main focus of the work to date has been on how to report candidate links to

a user with maximum recall without sacrificing precision. LSI is the most popular retrieval technique used, although several other methods including VSM and probabilistic IR have been experimented with. No method has yet proved a clear favorite.

The first technique uses *Language Models* (LM) where an artifact is retrieved for a query if there is a high probability that the artifact's model would generate the query [26]. In the context of traceability, a unique LM is built for each section of the documentation. Then the source code and the documentation are fed into the Bayesian classifier that establishes the probability that a particular manual section would generate the words found in the source. Each module is linked to the manual section that has the highest probability of generating it.

An implementation builds LMs by counting the number of occurrences of a particular event. In particular, bi-gram LMs are built where the occurrence of a word in the model is conditioned based on the prior word: let $C(s)$ be the number of times the sequence of words s occurs. The probability $P(w_j|w_i)$ that w_j occurs given w_i precedes it in the text, is $\frac{C(w_iw_j)}{C(w_i)}$.

An implementation was tested on the C++ program **LEDA**. However, only 110 of **LEDA**'s 208 classes are described in its 78 manual sections. For example, abstract super-classes, which are not described by the manual, get linked to a section describing a subclass. Thus, when associating all classes with the most similar section, precision is low (38.94%). However, recall is quite high (82.65%). The second technique is LSI-based [27]. It treats manual sections as queries to produce a ranked list of source files based on cosine similarity. In comparison with the LM-based approach, LSI produced better precision and similar recall for **LEDA**. In particular, returning just the top two ranked pairs for each section produced 53.98% precision and 83.33% recall. The LSI implementation was also applied to a second program, **Albergate**, a student-built Java system whose

documentation shares considerably less vocabulary with the source code. Again using the top two ranked pairs, the results showed the expected drop in precision (28.45%) and recall (57.89%).

The third technique performs incremental traceability recovery using *relevance feedback* (a technique that attempts to move a query vector away from non-relevant answers and towards relevant answers) [28]. For example, one relevance feedback method employed is the Rochio Standard, which re-weights the original vector by increasing the weight of words found in correct links and decreases the weight of words found in false positives.

This modification is achieved by updating the weight w for each term t in the vector based on three tuning constants α , β , γ where

$$w_{new}(t) = \alpha \cdot w(t) + \beta \cdot \overline{CL_t} - \gamma \cdot \overline{FP_t}$$

where $\overline{CL_t}$ is the average weight of the terms correctly linked to t and $\overline{FP_t}$ is the average weight of the terms linked to t that are false positives. The three constants are used to emphasize positive or negative feedback as well as the importance of the original weight. Empirically, good results were achieved with $\alpha = 1$, $\beta = 0.75$, and $\gamma = 0.25$ (of particular importance was positive feedback being given three times the importance of negative feedback).

Two systems were used to evaluate the approach: **EasyClinic** and the NASA Imaging Spectrometer **MODIS** software. For **EasyClinic** relevance feedback improved performance using LSI by about 1% on one subset of the data but degraded performance by about 5% on another. The authors speculate the relevance feedback has the greatest impact when the artifact used as a query is short.

For **MODIS**, relevance feedback was applied to both a VSM and an LSI model. It improved precision by 0.5% for VSM and about 2% for LSI (both at 100% recall).

In addition, the experiments with **MODIS** call into question the superiority of LSI for retrieving candidate links, as the precision values fluctuate back and forth at different levels of recall.

Finally, Huffman and Dekhtyar describe a four phase framework for comparing traceability techniques [29]. The phases are *definition*, *planning*, *realization*, and *interpretation*. Each phase is broken down into multiple parts. The planning phase, for example, is broken down into three parts: experimental design, measurement, and product. Each part provides specific information. Measurement, for example, defines the metrics used, there validation, objectivity, and scale. The other phases are similarly broken down into multiple parts. By casting studies into this rigid framework, all studies use a similar vocabulary and structure, which facilitates comparison. The paper compares four IR-based traceability recovery techniques (including the second and third considered above). The authors conclude by noting the need for larger standardized data-sets (with answer-sets) and the study of the human factors associated with the tracing process.

8 REFACTORING

Refactoring attempts to improve software’s internal structure, maintainability, and comprehensibility. With object-oriented code, this task is often accomplished by splitting and merging classes and migrating partial functionality from one class to another. A consequence of refactoring is that it may be difficult to trace the lifetime of a feature across different versions. This shortcoming has a negative impact on maintainability [30].

Refactoring is necessary when a program’s internal structure degrades, often as a result of prior maintenance and evolution. Although IR has not yet been applied to the task of refactoring itself, it has been used to compensate for the

lack of configuration management and the loss of traceability between related classes. This section describes an approach that automatically identifies and documents evolutionary discontinuities.

First, Antoniol et al. trace features of a class from version to version [30]. Their goal is to find classes that have been split, merged, or replaced. To accomplish this goal, classes are represented in a VSM using *tf-idf*. Cosine similarity finds initial connections between classes across different versions. Class *B* is assumed to be a replacement for Class *A* if

1. *A* and *B* have different names.
2. *A* and *B* have a high cosine similarity.
3. *A* and *B* belong to different and subsequent releases n and $n + 1$ where Class *B* is not present in release n and Class *A* is not present in release $n + 1$.

A threshold is used to determine replacement. It is calibrated to system size taking into account the expected impact of false positives. In a large system the threshold needs to be high (0.8 to 0.9), but in a smaller system there will be fewer false positives so a lower threshold (0.6 to 0.7) can be tolerated in an attempt to improve recall.

A case study using the Java Domain Name Server (**dnsjava**) found that the same class was involved in several different types of refactoring in the same release. Although there were false positives reported, the high cosine similarity scores were generally helpful in determining developer's past actions.

9 SUMMARY

This article illustrates the variety of ways in which IR techniques have been used to address problems that software engineers face during software maintenance

and evolution. These techniques are only a beginning. However, the initial success that IR has found in SE is a testament to the wealth of information stored in the natural language of a program and its supporting artifacts.

Looking forward, continued exploitation of this information should increase the value that IR brings to software engineers. Moreover, IR researchers continue to develop new retrieval methods. For example, *Latent Dirichlet Allocation*, a recent replacement for LSI, allows artifacts to be generated from a mixture of topic models rather than a single topic [31]. SE researchers need to continually look for opportunities to exploit such new ideas and developments in solving current and future software engineering problems.

10 ACKNOWLEDGMENTS

Special thanks Andrian Marcus for his comments, suggestions, and discussions, and to the reviewers for their thoughtful and detailed comments. Our appreciation to the following for responding to the request for current work and ideas for future research: Gerardo Canfora, Maurice Carey, Luigi Cerulo, Andrea De Lucia, Prem Devanbu, Letha Etzkorn, Bill Frakes, Gerald Gannod, Emily Gibson, Jane Hayes, Rocco Oliveto, Denys Poshyvanyk, Neil Rowe, and David Shepherd. The expertise of Craig Allen was invaluable in preparing this survey. This work is supported by National Science Foundation grant CCR0305330.

References

- [1] D. Binkley and D. Lawrie. Applications of information retrieval to software development. *Encyclopedia of Software Engineering*, V(N), January 2008.
- [2] A. Marcus, A. Sergeyev, V. Rajlich, and J. Maletic. An information retrieval approach to concept location in source code. In *IEEE Working*

Conference on Reverse Engineering, Delft, The Netherlands, November 2004.

- [3] D. Poshyvanyk and A. Marcus. Combining formal concept analysis with information retrieval for concept location in source code. In *15th International Conference on Program Comprehension, Banff, Alberta, Canada*, June 2007.
- [4] D. Poshyvanyk, Y. Gael-Guéhéneuc, A. Marcus, G. Antoniol, and V. Rajlich. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE Transactions on Software Engineering*, 33(6), June 2007.
- [5] W. Zhao and L. Zhang. Sniafl: Towards a static non-interactive approach to feature location. *ACM Transactions on Software Engineering and Methodology*, 15(2), 2006.
- [6] H. Feild, D. Binkley, and D. Lawrie. An empirical comparison of techniques for extracting concept abbreviations from identifiers. In *Proceedings of IASTED International Conference on Software Engineering and Applications (SEA 2006)*, Dallas, TX, November 2006.
- [7] Z. Fry, S. Shepherd, E. Hill, L. Pollock, and K. Vijay-Shanker. Analysing source code: looking for useful verb–direct object pairs in all the right places. *IET Software*, 2(1), Feb 2008.
- [8] T. Gyimóthy, R. Ferenc, and I. Siket. Emperical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software Engineering*, 31(10), October 2005.
- [9] O. Mizuno, S. Ikami, S. Nakaichi, and T. Kikuno. Spam filter based approach for finding fault-prone software modules. In *Proceedings of Fourth*

International Workshop on Mining Software Repositories, Dubrovnik, Croatia, May 2007.

- [10] D. Binkley, H. Feild, D. Lawrie, and M. Pighin. Software fault prediction using language processing. In *Proceedings of 2nd Testing: Academic and Industrial Conference (TAIC-PART)*, Windor, UK, September 2007.
- [11] L. Aversano, L. Cerulo, and C. Del Grosso. Learning from bug-introducing changes to prevent fault prone code. In *Proceedings of International Workshop On Principles of Software Evolution*, Dubrovnik, Croatia, September 2007.
- [12] I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, second edition, 2005.
- [13] E. Linstead, P. Rigor, S. Bajracharya, C. Lopes, and P. Baldi. Mining eclipse developer contributions via author-topic models. In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, Dubrovnik, Croatia, May 2007.
- [14] G. Canfora and L. Cerulo. Supporting change request assignment in open source development. In *Proceedings of SAC 2006*, Dijon, France, April 2006.
- [15] J. Maletic and A. Marcus. Supporting program comprehension using semantic and structural information. In *Proceedings of the 23rd IEEE/ACM International Conference on Software Engineering*, Toronto, Canada, 2001.
- [16] G. Maskeri, S. Sarkar, and K. Heafield. Mining business topics in source code using latent dirichlet allocation. In *1st India Software Engineering Conference*, Hyderabad, India, February 2008.

-
- [17] L. Etzkorn, L. Bowen, and C. Davis. An approach to program understanding by natural language understanding. *Natural Language Engineering*, 5(1), 1999.
 - [18] M. Ohba and K. Gondow. Toward mining "concept keywords" from identifiers in large software projects. In *Proceedings of the Proceedings of the Second International Workshop on Mining Software Repositories*, St Louis, MO, May 2005.
 - [19] A. Kuhn, S. Ducasse, and T. Gîrba. Semantic clustering: Identifying topics in source code. *Information and Software Technology*, 49(3), Mar 2007.
 - [20] G. Canfora and L. Cerulo. Impact analysis by mining software and change request repositories. In *Proceedings of the Eleventh IEEE International Software Metrics Symposium*, Como, Italy, September 2005.
 - [21] M. Kamkar. An overview and comparative classification of program slicing techniques. *J. Syst. Softw.*, 31(3), 1995.
 - [22] G. Antoniol, G. Canfora, G. Casazza, and A. De Lucia. Identifying the starting impact set of a maintenance request: A case study. In *Proceedings of the Fourth Conference on Software Maintenance and Reengineering*, Zurich, Switzerland, February 2000.
 - [23] G. Canfora and L. Cerulo. Fine grained indexing of software repositories to support impact analysis. In *Proceedings of the Third International Workshop on Mining Software Repositories*, Shanghai, China, May 2006.
 - [24] G. Canfora, L. Cerulo, and M. Di Penta. Identifying changed source code lines from version repositories. In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, Minneapolis, May 2007.

-
- [25] A. De Lucia, F. Fasano, R. Oliveto, and G. Tortora. Recovering traceability links in software artefact management systems using information retrieval methods. *ACM Transactions on Software Engineering and Methodology*, 16(4), September 2007.
 - [26] G. Antoniol, G. Canfora, A. De Lucia, and E. Merlo. Recovering code to documentation links in oo systems. In *Proceedings of the Sixth Working Conference on Reverse Engineering*, Atlanta, Georgia, USA, October 1999.
 - [27] A. Marcus and J. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Proceedings of the 25th IEEE/ACM International Conference on Software Engineering*, Portland, OR, May 2003.
 - [28] A. De Lucia, R. Oliveto, and P. Sgueglia. Can information retrieval techniques effectively support traceability link recovery? In *Proceedings of the 20th IEEE International Conference on Software Maintenance*, Philadelphia, USA, September 2006.
 - [29] J. Hayes, A. Dekhtyar, and J. Osborne. A framework for comparing requirements tracing experiments. *Journal of Software Engineering and Knowledge Engineering*, 15(5), 2005.
 - [30] G. Antoniol, M. Di Penta, and E. Merlo. An automatic approach to identify class evolution discontinuities. In *Proceedings of the IWPSE*, Kyoto, Japan, September 2004.
 - [31] X. Wei and W. B. Croft. Lda-based document models for ad-hoc retrieval. In *Proceedings of the Twenty-Ninth International ACM Conference on Research and Development in Information Retrieval*, Seattle, WA, August 2006.