On the use of Relevance Feedback in IR-based Concept Location

Gregory Gay[1], Sonia Haiduc[2], Andrian Marcus[2], Tim Menzies[1]

[1]Lane Department of Computer Science,
West Virginia University
Morgantown, WV, USA

[2]Department of Computer Science
Wayne State University
Detroit, MI, USA

gregoryg@csee.wvu.edu; sonja@wayne.edu; amarcus@wayne.edu; tim@menzies.us

Abstract

Concept location is a critical activity during software evolution as it produces the location where a change is to start in response to a modification request, such as, a bug report or a new feature request. Lexical based concept location techniques rely on matching the text embedded in the source code to queries formulated by the developers. The efficiency of such techniques is strongly dependent on the ability of the developer to write good queries. We propose an approach to augment information retrieval (IR) based concept location via an explicit relevance feedback (RF) mechanism. RF is a two-part process in which the developer judges existing results returned by a search and the IR system uses this information to perform a new search, returning more relevant information to the user. A set of case studies performed on open source software systems reveals the impact of RF on the IR based concept location.

1. Introduction

Biggerstaff et al. [3] defined the *concept assignment problem* as "… discovering human oriented concepts and assigning them to their implementation instances within a program …". The problem has been rephrased in the research community in the past decades as *concept location* in software. The redefinition relates it to the problem of *feature location* in software [28], as *features* are regarded as (a set of) concepts associated with a software system's functional requirements, reflected in user visible functionality. Biggerstaff et al.'s early definition of the problem needs to be instantiated for practical use. It needs a well defined context, which in turn helps define the operational goal and scope of concept location. The context is defined by establishing the software engineering task supported by concept location, such as, change management, fault localization, traceability link recovery, etc. This in turn helps define the input and output parameters for concept location: how are the human oriented concepts described and what are the "implementation instances within a program" referred to by Biggertsaff? For example, Wilde et al.'s approach "is not expected to cover all possible functionalities nor always to discover all code segments associated with a functionality. It is only intended to provide starting points for more detailed exploration of the code" [28].

In our research, we define concept location in the context of software change. The software change process [24] starts with a modification request and ends with a set of changes to the existing code and addition of new code. The software maintainer undertakes a set of activities to determine the parts of the software that need to be changed: concept location, impact analysis, change propagation, and refactoring. *Concept location starts with the change request and ends when the developer finds the location in the source code where the first change must be made (e.g., a class or a method)*. The other activities start with the result of concept location and establish the extent of the change.

In our operating context (i.e., software change), the developers must decide where in the code they will start the change, hence their involvement is critical. The challenge here is to make sure the tools and methodologies work equally well for users with a wide range of expertise and abilities.

Marcus et al. [20] proposed an Information Retrieval (IR) based approach to concept location. The idea of the approach is to treat source code as a test corpus and use IR methods to index the corpus and build a search engine, which allows developers to search the source code much like they search other source of digital information (e.g., the internet). The methodology was further refined in [22] and also combined with other feature location techniques [10, 14, 16, 21, 29]. This family of approaches relies on the user to formulate a query and if she does not identify fast enough the location of the change, then she rewrites the query and restarts the search.

Two issues remained constant in all these methods: (1) each approach is highly sensitive to the ability of

the user to write good queries; and (2) the knowledge gained by the user during the location process is not captured explicitly. Specifically, developers start the process from a change request and they either use it as is as a query or they extract a set of words from the change request and use them as a query. Extracting a good query from a change request depends on the experience of the developer and on her knowledge of the system. Previous work showed that developers tend to write queries with significantly different performance starting from the same change request [14]. As the developer investigates the results of the first search, she learns more about the system and eventually decides to improve the query by adding or removing words. There is a gap between the source code representation as classes and methods (or other decomposition units) and the words in a query and some developers can fill this gap easier than others.

In this paper we propose and evaluate an approach aimed at addressing these two issues, based on explicit relevance feedback (RF) provided by the user. We call the approach IRRF (Information Retrieval with Relevance Feedback) based concept location. The approach is motivated by similar work on traceability [8, 11] in software.

We present a case study where we explore under what circumstances IRRF improves the classic IR based concept location. We reenact changes associated with several bug fixes in three open source projects.

2. Concept location and relevance feedback

Concept location is sometimes regarded as an instance of an information seeking activity, where developers search and browse the source code to find the location to start the change. Different tools target the searching or the navigation aspects of the process. Regardless of the tool support, most concept location methodologies are interactive and iterative, as they require the user to decide whether a certain element of the source code, recommended by a tool, is relevant or not to the change. These decisions affect subsequent steps in the process (i.e., navigation or new search).

Our new concept location methodology combines the IR based concept location [20] with explicit RF from the user, which is used to formulate new queries. This section describes each technology and the proposed combination.

2.1. IR based concept location

Lexical based static concept location considers source code a text corpus and leverages the information encoded in identifiers and comments from the source code to guide the search. As such, it can be seen as a classic IR problem: given a document collection and a query, determine those documents from the collection that are relevant to the query. Given that relevance is defined with respect to a textual query, user involvement is necessary to convert this relevance measure into a change related decision.

IR based concept location, proposed in [20], is based on the above ideas. It implies the use of an IR method to index the corpus extracted from the software and uses the index to compute a similarity measure between the document and a query. It is composed of five major steps, which may be instantiated differently based on the type of IR method used and how the corpus is created:

1. *Corpus creation.* The source code is parsed using a developer-defined granularity level (i.e., methods or classes) and documents are extracted from the source code. Each method (or class) will have a corresponding document in the corpus. Natural language processing (NLP) techniques and other filtering techniques can be applied to the corpus.
2. *Indexing*. The corpus is indexed using the IR method and a mathematical representation of the corpus is created. Each document (hence each method or class) has a corresponding index.
3. *Query formulation*. A developer selects a set of words that describe the concept to be located. This set of words constitutes the query. The tool checks whether the words from the query are present in the vocabulary of the source code. If a word is not present, then the tool eliminates the word from the initial query. If filtering or NLP was used in the corpus creation, the query will get the same treatment.
4. *Ranking documents*. Similarities between the query and every document from the source code are computed. The similarity measure depends on the IR method used. Based on these measures the documents in the corpus are ranked with respect to the query.
5. *Results examination*. The developer examines the ranked list of source code documents, starting with documents with highest similarities. For every source code document examined, a decision is required whether the document will be changed or not. If it will be changed, then the search succeeded and concept location ends. Else, if new knowledge obtained from the investigated documents helps formulate a better query (e.g., narrow down the search criteria), then step 3 should be reapplied, else the next document in the list should be examined.

2.2. Relevance feedback in IR

Relevance feedback analysis is a technique to utilize user input to improve the performance of retrieval algorithms. Relevance feedback has been one of the successes of information retrieval research for the past 30 years [17]. For example, the Text Retrieval

Conference[1] (co-sponsored by the National Institute of Standards and Technology - NIST and the U.S. Department of Defense) has a relevance feedback track. While the applications of relevance feedback and type of user input to relevance feedback have changed over the years, the actual algorithms have not changed much. Most algorithms are either pure statistical word based, or are domain dependent. There is no general agreement of what the best RF approach is, or what the relative benefits and costs of the various approaches are. In part, that is because RF is hard to study, evaluate, and compare. It is difficult to separate out the effects of an initial retrieval run, the decision procedure to determine what documents will be looked at, the user dependent relevance judgment procedure (including interface), and the actual RF reformulation algorithm. Our case study aims at evaluating only a subset of these aspects of RF.

There are three types of feedback: explicit, implicit, and blind ("pseudo") feedback. In our approach, we chose to implement an explicit RF mechanism. Explicit feedback is obtained from users by having them indicate the relevance of a document retrieved for a query. Users may indicate relevance explicitly using a binary or graded relevance system. Binary relevance feedback indicates that a document is either relevant or irrelevant for a given query. Graded relevance feedback indicates the relevance of a document to a query on a scale using numbers, letters, or descriptions (such as "not relevant", "somewhat relevant", "relevant", or "very relevant").

Classic text retrieval applications of RF make several assumptions [17], which are not always true in the case of source code text and make the problem more challenging for us:

- The user has sufficient knowledge to formulate the initial query. This is not always the case when it comes to software, as developers might be unfamiliar with a software system or they might not have enough knowledge about a particular problem domain.
- There are patterns of term distribution in the relevant vs. non-relevant documents: (i) term distribution in relevant documents will be similar; (ii) term distribution in non-relevant documents will be different from that in relevant documents (i.e., similarities between relevant and non-relevant documents are small). There is no evidence so far that this is true for source code.

RF has some known limitations, some of which we are also faced with:

- It is often harder to understand why a particular document was retrieved after applying relevance feedback. We found this to be quite true in the case of source code based corpora.
- It is easy to decrease effectiveness (i.e., one irrelevant word can undo the good caused by lots of good words). This is hard to judge in our case, but quite likely.
- Long queries are inefficient for a typical IR engine and we found that source code based corpora tends to increase query length significantly and rapidly.

2.3. IRRF based concept location

The IRRF based concept location combines the IR based concept location described in Section 2.1 with an explicit RF mechanism. The new concept location methodology is defined as follows:

1. *Corpus creation* – same as IR.
2. *Indexing* – same as IR.
3. *Query formulation* – same as IR.
4. *Ranking documents* – same as IR.
5. *Results examination*. The developer examines the top N documents in the ranked list of results. For every source code document examined, a decision is required whether the document will be changed or not. If it will be changed, then the search succeeded and concept location ends. Else, the user marks the document as relevant or irrelevant. After the N documents are marked a new query is automatically formulated and the methodology resumes at step 4. If several rounds of feedback do not result in reaching the result, then the query may be reformulated manually by the user and resume at step 4.

In order to provide tool support for the IRRF methodology, several options are available. There are several options on how to generate the corpus from the source code, such as: document granularity (e.g., method, class, etc.), identifier splitting (i.e., keep original identifier or not), stop word removal, keyword removal, stemming, and comments inclusion. Some of these options are programming language specific. Section 3 provides details on the options we used in the evaluation. Indexing can be done with a variety of IR methods. The initial query formulation can be done by the developer or automatically extracted from the change request (i.e., use the entire change request as the query).

2.3.1. Vector space model

The original IR based concept location technique [20] was built around Latent Semantic Indexing (LSI) [13], an advanced IR method. Researchers investigated the use of other IR techniques , such as Vector Space Models (VSM) [27], Latent Dirichlet Allocation (LDA), or Bayes classifiers for concept location and other related activities (e.g., traceability link recovery). So far, there is no clear winner among these techniques. In consequence, we decided to use

[1] http://trec.nist.gov/

here a VSM technique, implemented in Apache Lucene[2]. Our choice is motivated also by the fact that traditional RF methods were developed specifically for this type of IR technique.

In VSM, a vector model of a document *d* is a vector of words weights that span over the number of words in the corpus. The weights for each word are computed based on the *term frequency* of that word in *d* and the *inverse document frequency*. The similarity between two documents is computed as the cosine between their corresponding vector models.

2.3.2. RF with Rocchio

There are several options to implement an RF mechanism. One of the most popular approaches is the Rocchio relevance feedback method [26], used in conjunction with a VSM indexing technique. We implemented our own version of Rocchio, which integrates with Apache Lucene and works in the following way. When analyzing the top ranked methods, the user is asked to judge the current method as relevant, irrelevant, or neutral to the current change task. Given a set of documents D_Q encompassed by query Q, let R_Q be the subset of relevant documents and I_Q the set of irrelevant documents to the query. The original query Q can be then transformed by adding terms from R_Q and removing terms from I_Q. This mechanism is meant to bring the query closer to the relevant documents and drive it away from the irrelevant documents in the vector space. The new query is formulated as follows:

$$Q' = \alpha Q + \frac{\beta}{|R_Q|}\sum_{d \in R_Q} d - \frac{\chi}{|I_Q|}\sum_{d \in I_Q} d \qquad (1)$$

where α, β, and χ are weighting parameters and *d* represents a document and its associated vector. The relevance feedback can be given by the user in several feedback rounds and the query is updated after each round.

The Rocchio technique is recursive. Each round, a new query is generated based on the query generated in the previous round. The three constants α, β, and χ are provided so that a level of importance can be specified by the user for the initial query, the relevant documents and the irrelevant documents. According to [9], placing emphasis on the relevant documents may improve the recall (new relevant documents may be found) while emphasizing the irrelevant documents may affect precision (false positives may be removed). Joachims recommends weighting the positive information four times higher than the negative [12]. De Lucia et al. [8] advocates using α=1, β=0.75, and χ=0.25 (i.e., relevant documents are three times more important than irrelevant ones). Our implementation follows a similar line of thought, setting values of α=1, β=0.5, and χ=0.15 for the three weighting parameters. We tried other sets of weights (α=1, β=0.75, and χ=0.25 and α=1, β=1, and χ=1), but the final set of weights seemed to yield the best performance.

This type of query expansion is still prone to noise as certain common, but unimportant, terms may be added to the query. To filter this noise and improve precision, the system used in this paper only allows terms to be added to the query if they appear in less than 25% of the corpus.

3. Case study

As mentioned before, we developed IRRF to address several issues we learned in our experience with concept location. IR based concept location assumes the developers read the source code and reformulate the query if they did not locate the target methods. We found that most programmers are good at judging whether a method is relevant to a change or not, but their ability to formulate a good natural language query based on their knowledge of the software varies quite a bit. Developers have a hard time deciding what is wrong with their previous query and how to make it better. IRRF eliminates this step in the process and allows developers to focus on what they know best (i.e., source code rather than queries) by reformulating the queries automatically. Earlier work on traceability [7, 11] showed that RF improves an IR task, but not in all cases. Our assumption is that IRRF improves the IR based concept location (i.e., reduces developer effort) and the goal of the case study we performed is to investigate under what circumstances this is true, given that our approach has a different context than the traceability work.

3.1. Methodology

The case study consists of the reenactment of past changes in open source software (i.e., we know which methods were modified in response to the change request). The modified methods form the *change set*, and we call these methods *target methods*. This methodology has been used in previous work on evaluating concept location techniques [14, 16, 21]. In our case study, one developer performed the concept location reenactment using IRRF, given a set of change requests. He has seven years of programming experience (five in Java) and he was not familiar with the source code used in the study. For each change request his task was to locate one of the methods from the change set, based on the following scenario:

- He starts by running a query based on the change request, called the *initial query*.
- If any one of the target methods is among the top 5 methods in the ranked list of results, then he stops and

[2] http://lucene.apache.org/java/docs/

selects another change request, as RF is not needed in this case (i.e., IR based concept location will reach the method fast enough).

- Else he provides RF in several rounds. In each round, the developer marks the *N* top ranked methods as being *relevant* or *irrelevant*. If he cannot judge the relevancy of a method, the he marks the document as *neutral* and proceeds to the next document, increasing the size of the set of marked methods set by one.
- IRRF automatically reformulates the query based on the feedback provided by the developer and another round of feedback begins. We keep track of the number of methods marked by the developer.
- After each query is run, based on the positions of the target methods in the ranked list of search results and on the number of methods marked, the following decisions are made:
  a. If any of the target methods is located in the top *N* documents, then STOP; consider IRRF successful and a target method found.
  b. If for two consecutive feedback rounds the positions of the target methods declined in the ranked list of results, then STOP; consider that IRRF failed (i.e., the developer needs to reformulate the query manually).
  c. If more than 50 methods were marked by the developer, then STOP; consider that IRRF failed (i.e., the developer needs to reformulate the query manually).

The values used for *N* vary and the performance of IRRF depends on it. The most commonly used values range from 1 to 10. We investigated the results of IRRF for three values of *N*: 1, 3, and 5, which are recommended values in recent studies for presenting lists of results to developers for investigation [25]. Each reenactment was done three times by the developer, the difference from case to case was the number *N* of marked methods in one round.

3.1.1. Data - software and change sets

We chose as the objects of the case study three open source systems: Eclipse[3], JEdit[4], and Adempiere[5]. All three systems have an active community and a rich history of changes. They all have online bug tracking systems, where bugs are reported and patches are submitted for review.

Eclipse is an integrated development environment developed in Java. For our case study, we considered version 2.0 of the system, which has approximately 2.5 millions lines of code and 7,500 classes. JEdit is an editor developed for programmers and it comes with a series of plugins which add extra features to its core functionality. It is developed in Java and version 4.2 used in this case study has approximately 300,000 lines of code and 750 classes. Adempiere is a commons-based peer-production of open source enterprise resource planning applications. It is developed in Java and it has approximately 330,000 lines of code and 1,900 classes in version 3.1.0, which was used in our case study.

We used the history of a software system in order to extract real change sets from the source code. Specifically, we used approved patches of documented bugs for extracting the change sets. The bug descriptions are considered to be the change requests. This approach has been used in previous work on evaluating concept location techniques [14, 16, 21]. Some changes involve the addition of new methods. We do not, however, include these methods in the change sets, as they did not exist in the version that a developer would need to investigate in order to find the place to implement the change. For each of the systems, we analyzed their online defect tracking systems and manually selected a set of bugs to extract change sets for our case study.

The Eclipse community uses the open-source bug tracking system BugZilla[6] to keep track of bugs in the system. Each bug has an associated bug report, which consists of several sections, one of which is the bug description. Sometimes the patches used to fix the bugs are also contained in the bug report, as attachments. They are usually in the form of *diff* files, containing the lines of code that changed between the version of the software where the bug was reported and the version where the bug was fixed. For our case study, we chose an initial set of 10 bugs reported in version 2.0 of the system, for which the patches were available in their bug reports.

For jEdit[7] and Adempiere[8], we analyzed the bug tracking systems hosted on the projects' sourceforge.net website. Both projects have systems that keep track of the patches submitted for known bugs in the source code. In these trackers, each patch has an associated report where the changes implemented in the patch are described in a *diff* file attached to the report. We selected for each system 10 initial patches for which a good description of the bug fixed by the patch was available, either in the description of the patch or in a separate bug report. All the patches we selected for jEdit were submitted and their corresponding bugs reported after version 4.2 of the system was released. For Adempiere patches were selected after the release of version 3.1.0.

[3] http://www.eclipse.org/
[4] http://www.jedit.org/
[5] http://www.adempiere.com/
[6] https://bugs.eclipse.org/bugs/
[7] http://sourceforge.net/tracker/?group_id=588&atid=300588
[8] http://sourceforge.net/tracker/?atid=879334&group_id=176962

Based on the patches reported for the three systems, we constructed the 10 change sets for each system. All change sets contained between one and six target methods.

3.1.2. Corpus creation

We extracted a corpus for each of the three systems. We used the version of the software in which the bugs chosen in the previous step were reported. We mapped each method in the source code to a document in our corpus. The Eclipse corpus has 74,996 documents, the JEdit corpus has 5,366 documents, and the Adempiere corpus has 28,622 documents. By comparison, the size of the corpora used in previous work on RF in traceability [6, 11] is in the few hundreds of documents range.

The corpora were built in the following manner:

- We extracted the methods using the Eclipse built in parser. The comments and identifiers from each method implementation were extracted.
- The identifiers were split according to common naming conventions. For example, "setValue", "set_value", "SETvalue", etc. are all split to "set" and "value". We kept the original identifiers in the corpus, which would favor any query containing an identifier already known by the user.
- We filtered out programming language specific keywords, as well as common English stop words[9].
- We used the Porter stemmer[10] in order to map different forms of the same lexeme to a common root.

3.1.3. Query formulation

As mentioned before, the goal of IRRF is to allow the developer not to write manually defined queries. Hence, in the case study the developer used as the initial query the bug description and bug title contained in the bug or patch reports (i.e., he copied the bug description and title). However, we eliminated any details referring to the implementation of the bug fix contained in these descriptions, prior to the study. The query was then automatically transformed following the same steps as the corpus (i.e., identifier splitting, stop word removal, stemming).

3.1.4. Assessment

Tool support in concept location is geared towards reducing developers' effort in finding the starting point of a change. Previous work on concept location [14, 16, 21] defined and used as an efficiency measure the number of source code documents that the user has to investigate before locating the point of change. We use here the same measure with an added advantage. In previous work the cost for (re)formulating a query was never considered in evaluation (i.e., assumed to be zero). In our case, the cost of (re)formulating a query is indeed almost zero, as the initial query is copied from the bug description and title, whereas subsequent queries are formulated automatically. The number of methods investigated is automatically tracked as they are explicitly marked by the developer.

In order to avoid the bias and the cost associated with user formulated queries, we do not consider manual query formulation as part of the methodology (see the first part of this section). We eliminate this step from both methodologies (i.e., IR and IRRS based concept location) for this case study.

For each change request, the base line is provided by the IR based concept location without query reformulation. The initial query is run and the baseline efficiency measure is the highest ranking (k) of any of the target methods. This means the user would have to investigate k methods to reach the target. For the IRRF case the efficiency measure is the number of methods marked one way or another (i.e., relevant, irrelevant, or neutral) before the target was found or until the method fails (see the methodology described above) plus the last rank of the target method. IRRF is considered to improve the baseline if its efficiency measure is lower than the baseline efficiency (i.e., fewer methods are investigated).

3.2. Results and discussion

The results presented and discussed here are aggregated and omit several intermediary steps from the study, such as: actual markings in each round for all cases, queries, intermediary ranking of target methods, complete change sets, etc. The complete data collected during the case study is available online at www.cs.wayne.edu/~severe/IRRF.

As explained in Section 3.1.1, we selected 10 changes for each system (i.e., 30 total). In 12 cases, at least one of the target methods was ranked in top 5 after the initial query, hence we did not use RF in those cases. **Error! Reference source not found.** aggregates the results of the concept location for the three systems considered, reflecting the changes for which we used RF: 7 for Eclipse, 6 for jEdit, and 5 for Adempiere.

The *Baseline* column shows the positions of the target methods in the result list after the initial query. The best rank in each case where there is more than one target method is bold. This is the efficiency measure in the baseline case (i.e., how many methods would the user need to investigate to find the best ranked target).

The IRRF columns show the positions of the target methods at the end of the IRRF location process, whether it succeeded or not (see Section 3.1 for details). *N* indicates the number of marked methods in

[9] www.dcs.gla.ac.uk/idom/ir_resources/linguistic_utils/stop_words
[10] http://tartarus.org/~martin/PorterStemmer/

Table 1. Concept location results for Eclipse, jEdit and Adempiere

No.	Defect Report#	Baseline	IRRF with N=1	IRRF with N=3	IRRF with N=5
Eclipse					
1	Bug #13926	**54**	1 (**16**m/15r)	11 (51m/16r)	36 (50m/10r)
2	Bug #23140	**17**,42,47	99, **1**, 2 (**9**m/8r)	4, **1**, 2 (**7**m/3r)	6, **4**, 14 (**9**m/2r)
3	Bug #19691	1K+, 368, 531, 1K+, **108**, 139	1K+, 1K+, 1K+, 1K+, 1K+, 1K+ (2m/2r)	1K+, 1K+, 1K+, 1K+, 1K+, 1K+ (7m/2r)	1K+, 1K+, 1K+, 1K+, 1K+, 1K+ (11m/2r)
4	Bug #12118	**9**	1 (**5**m/5r)	1 (23m/8r)	4 (10m/2r)
5	Bug #17707	**8**	1 (**2**m/2r)	1 (**4**m/2r)	2 (**7**m/2r)
6	Bug #19686	**428**	448 (5m/5r)	3 (**48**m/16r)	5 (**46**m/9r)
7	Bug #21062	583,**56**	1K+, 781 (2m/2r)	604, **1** (**37**m/13r)	1K+, 1K+ (20m/4r)
jEdit					
1	Patch #1649033	40,87,**22**	70, 60, 50 (8m/7r)	39, 7, 42 (22m/7r)	30**, 5**, 33 (**26**m/5r)
2	Patch # 1469996	**296**	1 (**37**m/36r)	289 (12m/4r)	5 (**41**m/9r)
3	Patch #1593900	**7**	1 (**6**m/4r)	1 (**5**m/2r)	1 (7m/2r)*
4	Patch # 1601830	**47**	216 (2m/2r)	242 (9m/3r)	146 (10m/2r)
5	Patch #1607211	**354**	98 (5m/5r)	3 (**36**m/12r)	3 (**28**m/6r)
6	Patch # 1275607	**151**	238 (4m/4r)	38 (48m/16r)	35 (50m/10r)
Adempiere					
1	Patch #1605419	**15**,550	**1**, 11 (**8**m/7r)	**3**, 109 (**17**m/5r)	**1**, 81 (**12**m/3r)
2	Patch #1599107	**122**	613 (6m/3r)	1K+ (8m/2r)	1K+ (12m/2r)
3	Patch #1599116	**7**	1 (**3**m/2r)	1 (**5**m/2r)	1 (7m/2r)*
4	Patch #1612136	**58**	141 (4m/3r)	1 (**13**m/5r)	1 (**16**m/4r)
5	Patch #1628050	**52**	1 (**3**m/3r)	2 (**5**m/2r)	2 (**7**m/2r)

Green – IRRF retrieves results more efficiently | **Yellow** – IRRF retrieves a better cumulative ranking of the target methods.

*IRRF performs as efficiently as the baseline

each round during IRRF. Note that only the methods for which relevance was given are counted as one of the *N* methods ranked in a feedback round (i.e., methods marked neutral are not counted towards the *N*, yet they count towards the efficiency measure). The number of methods analyzed by the developer before he stopped (i.e., the efficiency measure), either because a target method was found or because IRRF failed is reported in parenthesis (marked with *m*). This number includes all the methods marked by the developer in all the rounds of feedback, including also the methods marked as neutral, plus the rank of the target method in the final round. If one of the target methods was not ranked in the top 1,000 results, we denote its position as 1K+. To complete the picture, the number of feedback rounds is also reported in parenthesis (denoted with *r*), including the (incomplete) round when the target is found.

For example, row #2 in Eclipse, reads as follows. Baseline (**17**, 42, 47) means there are three target methods and the best ranked is on position 17. IRRF with *N*=3 (4, **1**, 2 (**7m**/3r)) means that one of the target methods (i.e., the second) was ranked #1 on the third round and the user marked a total of 7 methods to reach it (including the target method in the 3rd round). The two numbers to compare here are: 17 in the baseline vs. 7 in the RF case. We consider that IRRF improves here and highlight the table cell with green. Cells marked with yellow show no improvement of IRRF, but they are interesting as the cumulative ranking of methods is better than in the baseline (the number of investigated methods needs to be added here to the ranks of the target methods for a proper comparison). White cells indicate cases where IRRF does not improve the baseline. The stars in the white cells indicate the cases when IRRF failed in finding the target method in a reasonable amount of steps.

The data reveals that IRRF brings improvement over the baseline in 13 of the 18 changes (i.e., at least one cell in the row is green). In 3 changes, the improvement is observed for all values of *N* (i.e., all three RF cells are green in these rows). RF with *N*=1 improved in 9 cases, RF with *N*=3 improved in 9 cases, and RF with *N*=5 improved in 8 cases, not all the same.

More specifically, in Eclipse for 6 out of the 7 change sets reported, IRRF retrieved one of the target methods more efficiently than the baseline. In jEdit, the ratio was 3 to 3, and in Adempiere IRRF performed better in 4 out of 5 cases. We did not observe a pattern of when one of the values of *N* performs better than the

other ones, nor about the magnitude of the IRRF improvement over the baseline. So, we can not formulate at this time rules such as "*N*=5 is a better choice than *N*=3 or *N*=1". Nor we can state that there is a correlation between the initial query and IRRF improvements.

One interesting phenomenon that we observed is that for one change set in jEdit and for one in Adempiere IRRF did not improve the efficiency of the baseline (based on our working definition), however it achieved a better cumulative ranking of the target methods. These two cases are marked with yellow in the table. We highlight these cases as we believe is still an indication that RF brings some added benefit in these situations.

Another interesting and rather unexpected phenomenon is that in some cases where the there are more target methods the baseline favors one of them, whereas IRRF helps retrieve another one faster. See Bug #23140 in Eclipse and Patch #1649033 in JEdit (the yellow cell). We do not speculate on this issue here, as it is orthogonal to the goal of the study, but it opens up an avenue for future research.

We identified cases when neither the ranking of the first target method, nor the cumulative ranking of IRRF was better than in the case of the baseline (i.e., all white rows in the table). Our initial assumption was "if the initial query is really poor, RF does not help much". Unfortunately, this is not true as there were several cases where the initial query was worse, yet IRRF improved drastically (i.e., by one order of magnitude). For example, see Bug #19686 in Eclipse, Patch # 1469996, and Patch #1607211 in JEdit.

We investigated the cases with poor IRRF performance. For example, in the case of Bug #19691 in Eclipse, we found that the methods the developer would consider as being relevant based on the bug description would in fact not be relevant, even if they contained related terms from the bug description. The bug description is about exporting preferences for the team, whereas the target methods just contained "ignore" settings in the team preferences. This case highlights the difficulty of concept location in practice. Change requests are often formulated in terms different that the source code, both linguistically and logically. We can safely conclude that IRRF brings improvements over IR based concept location in many cases, but it is far from being a silver bullet.

3.3. Threats to validity

In our case study, we reenacted changes already performed in software systems and automated the process of query formulation. In practice, the user may reformulate the query along the way and IR may retrieve better results with the user re-formulated query. Simply put, the case study approximates the situation when the developer is not good at writing queries. We argue that the opposite case does not need RF. In fact, as the results revealed, 12 of the 30 bug descriptions produced great initial queries. It is important to clearly establish the cases where explicit RF helps.

Our results are based on the feedback provided by only one user. Different people might give different feedback to IRRF.

We used only three values of *N* (i.e., 1, 3, and 5) in the case study and a single weighting scheme in the IRRF implementation. We are aware of the fact that other values used of *N* might retrieve different results. However, these values are within the range of values usually adopted in the implementation of explicit relevance feedback and represent a reasonable amount of information for a user to analyze in one round of feedback. The current set of weights used in our Rocchio implementation was chosen based on empirical evidence. Other weights could lead to slightly different results.

4. Related work

Approaches to concept location in software differ primarily on what type of information is used to guide the developer while searching the code. Dynamic techniques are based on the analysis of execution traces and focus on identifying features (i.e., concepts associated with user visible functionality of the system). Static techniques use the textual information embedded in source code (i.e., comments and identifiers) and/or structural information about the software (e.g., program dependencies). There are combined methods that use combinations of dynamic and static techniques. Given that IRRF is meant to augment lexical based static concept location, we will only refer to them in this section. A more comprehensive overview of concept location techniques is available in [21] and static techniques are discussed in [19].

Lexical base static concept location techniques rely on matching a user query to the text in the source code. Traditional searching methods are built using regular expression matching tools, such as grep. Such techniques limit user quires to be formulated as regular expressions and do not provide a ranking list of results, but rather a simple list of matches.

More sophisticated techniques rely on the use of IR methods and have the advantage (over regular expression matching) that allow the user to formulate natural language queries and the results are ranked. Marcus et al. [20] introduced such a technique, using LSI as the IR method. Our approach is based on this technique, as described in Section 2. The method was

later extended by using formal concept analysis to cluster the results of the search [22]. A different implementation of the method was done using Google Desktop Search [23], as the underlying IR engine. More recently, Lukins at al. proposed a variation of the LSI based technique, using LDA [16]. SNIAFL [29] is a related approach, which combines an IR technique with call graph information and falls under the category of combined concept location techniques.

Related to IR based concept location is work on traceability link recovery, impact analysis, and recommendation systems. We do not list here all these works, but rather explain how they relate and how they differ from our work. Software artifacts are represented in different formats and textual information is the common denominator for all of them, hence IR methods have become widely used in tools that support traceability link recovery [1, 7, 11, 15, 18] and in many recommendation systems [5]. In impact analysis [2, 4] the textual information indexed with the IR method is usually used in conjunction with structural information or historical information about changes. On of the main differences between these applications of IR vs. concept location is that they use queries extracted from different artifacts, rather than user written. This allows for automation in many cases and elimination of the user from the process. By definition these techniques retrieve usually sets of documents, hence evaluation is different than in the case of concept location.

Of particular interest to our work is the work of Hayes et al. [11] and De Lucia et al. [6, 8], which introduced RF in the context of traceability link recovery. At technology level, our work is similar to these approaches, as we use same IR methods and similar RF implementations. The difference is in the context, application, methodology, and evaluation. In [11] the problem is to reduce the number of false positives when high level requirements are traced to low level requirements. The context is the same in [8] except that several types of artifacts are considered there. In both cases, there are no user queries and the results consist of traceability matrices, which are manually evaluated. Basically, those approaches use RF to improve an essentially automated process. Due to the manual effort needed in the evaluation and availability of data, experimental data is restricted to corpora with hundreds of documents at best. In our application, we use corpora several order of magnitude larger, which is more typical for IR applications.

5. Conclusions and future work

We defined a new methodology for lexical based static concept location, which combines the IR based concept location with explicit relevance feedback. The goal of the methodology is to alleviate the burden of query formulation on the developer.

Our case study revealed that in most cases RF reduces developer effort over the IR based concept location, in the absence of manual query reformulation. It also showed that in some cases, especially when the initial query is poor, RF does not really help. Our results are in line with previous work using RF [8, 11] in traceability link recovery. In each work RF was found to improve the performance in many case, but not across the board in all cases. Although our context and corpora are different than the classic use of RF on text retrieval and in requirements traceability, this application is subject to some of the same limitations.

Future work will focus on specific issues. We will compare different IR methods (e.g., LSI, LDA, VSM, etc.) used in IRRF. We expect our results might be different than previous comparisons, where small corpora were used, which is often unsuitable for some statistical IR techniques. We will also experiment with more system, more change request, and different ways to build corpora. For example, we expect that if we eliminate the comments from the corpus, the queries will grow slower. On the other hand, information will be lost. We plan to analyze the trade-offs. Different weighting options in RF will be also investigated. Intuitively, favoring irrelevant documents over relevant documents should help in concept location, yet our current results support the opposite.

As far as the methodology is concerned we need to define a better heuristic that tells the user when to switch from RF to manual reformulation. During reenactment we knew the end result, so the heuristic we used in the experiment is based on this. In practice that would not work as is.

We focused here on studying methodologies rather than users. An obvious next step is to extend our research to study how developers perform RF. After all, concept location is a user driven activity.

6. References

[1] Antoniol, G., Canfora, G., Casazza, G., De Lucia, A., and Merlo, E., "Recovering Traceability Links between Code and Documentation", *IEEE Transactions on Software Engineering*, 28, 10, October 2002, pp. 970 - 983.

[2] Antoniol, G., Canfora, G., Casazza, G., and Lucia, A., "Identifying the Starting Impact Set of a Maintenance Request: A Case Study", in *Proceedings 4th European Conference on Software Maintenance and Reengineering*, Zurich, Switzerland, Feb 29 - March 03 2000, pp. 227-231.

[3] Biggerstaff, T. J., Mitbander, B. G., and Webster, D. E., "The Concept Assignment Problem in Program Understanding", in *Proceedings 15th IEEE/ACM International Conference on Software Engineering (ICSE'94)* May 17-21 1993, pp. 482-498.

[4] Canfora, G. and Cerulo, L., "Impact Analysis by Mining Software and Change Request Repositories", in *Proceedings 11th IEEE International Symposium on Software Metrics (METRICS'05)*, September 19-22 2005, pp. 20-29.

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

[6] De Lucia, A., Fasano, F., Oliveto, R., and Tortora, G., "Can Information Retrieval Techniques Effectively Support Traceability Link Recovery?" in *Proceedings 14th IEEE International Conference on Program Comprehension (ICPC'06)*, Athens, Greece, June 14-16 2006, pp. 307-316.

[7] De Lucia, A., Fasano, F., Oliveto, R., and Tortora, G., "Recovering Traceability Links in Software Artefact Management Systems", *ACM Transactions on Software Engineering and Methodology*, 16, 4, 2007

[8] De Lucia, A., Oliveto, R., and Sgueglia, P., "Incremental Approach and User Feedbacks: a Silver Bullet for Traceability Recovery", in *Proceedings IEEE International Conference on Software Maintenance (ICSM'06)*, Philadelphia, Pennsylvania, 2006, pp. 299-309.

[9] Dekhtyar, A., Hayes, J. H., and Larsen, J., "Make the Most of Your Time: How Should the Analyst Work with Automated Traceability Tools?" in *Proceedings 3rd International Workshop on Predictor Models in Software Engineering*, Minneapolis, MN, May 20 2007

[10] Eaddy, M., Aho, A. V., Antoniol, G., and Guéhéneuc, Y. G., "CERBERUS: Tracing Requirements to Source Code Using Information Retrieval, Dynamic Analysis, and Program Analysis", in *Proceedings 17th IEEE International Conference on Program Comprehension (ICPC'08)*, Amsterdam, The Netherlands, 2008, pp. 53-62.

[11] Hayes, J. H., Dekhtyar, A., and Sundaram, S. K., "Advancing candidate link generation for requirements tracing: the study of methods", *IEEE Transactions on Software Engineering*, 32, 1, January 2006, pp. 4-19.

[12] Joachims, T., "A probabilistic analysis of the Rocchio algorithm with TFIDF for text categorization", in *Proceedings 14th International Conference on Machine Learning*, 1997, pp. 143-151.

[13] Landauer, T. K., Foltz, P. W., and Laham, D., "An Introduction to Latent Semantic Analysis", *Discourse Processes*, 25, 2&3, 1998, pp. 259-284.

[14] Liu, D., Marcus, A., Poshyvanyk, D., and Rajlich, V., "Feature Location via Information Retrieval based Filtering of a Single Scenario Execution Trace", in *Proceedings 22nd IEEE/ACM International Conference on Automated Software Engineering*, Atlanta, GA, November 5-9 2007, pp. 234-243.

[15] Lormans, M. and Van Deursen, A., "Can LSI help Reconstructing Requirements Traceability in Design and Test?" in *Proceedings 10th European Conference on Software Maintenance and Reengineering*, 2006 pp. 47-56.

[16] Lukins, S. K., Kraft, N. A., and Etzkorn, L., "Source Code Retrieval for Bug Localization Using Latent Dirichlet Allocation", in *15th IEEE Working Conference on Reverse Engineering*. Antwerp, Belgium, 2008, pp. 155-164.

[17] Manning, C. D., Raghavan, P., and Schütze, H., *Introduction to Information Retrieval*, Cambridge University Press, 2008.

[18] Marcus, A., Maletic, J. I., and Sergeyev, A., "Recovery of Traceability Links Between Software Documentation and Source Code", *International Journal of Software Engineering and Knowledge Engineering*, 15, 4, Oct. 2005, pp. 811-836.

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

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

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

[22] Poshyvanyk, D. and Marcus, D., "Combining Formal Concept Analysis with Information Retrieval for Concept Location in Source Code", in *Proceedings 15th IEEE International Conference on Program Comprehension (ICPC'07)*, Banff, Alberta, Canada, June 2007, pp. 37-48.

[23] Poshyvanyk, D., Petrenko, M., Marcus, A., Xie, X., and Liu, D., "Source Code Exploration with Google ", in *Proceedings 22nd IEEE International Conference on Software Maintenance (ICSM'06)*, Philadelphia, PA, 2006, pp. 334 - 338.

[24] Rajlich, V. and Gosavi, P., "Incremental Change in Object-Oriented Programming", in *IEEE Software*, 2004, pp. 2-9.

[25] Robillard, M., "Topology Analysis of Software dependencies", *ACM Transactions on Software Engineering and Methodologies (TOSEM)*, 17, 4, 2008, pp. 18-53.

[26] Rocchio, J. J., "Relevance feedback in information retrieval", in *The SMART Retrieval System - Experiments in Automatic Document Processing*, Prentice Hall, 1971, pp. 313-323.

[27] Salton, G. and McGill, M., *Introduction to Modern Information Retrieval*, McGraw-Hill, 1983.

[28] Wilde, N., Gomez, J. A., Gust, T., and Strasburg, D., "Locating User Functionality in Old Code", in *Proceedings IEEE International Conference on Software Maintenance (ICSM'92)*, Orlando, FL, November 1992, pp. 200-205.

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