Empir Software Eng (2017) 22:1063–1102
DOI 10.1007/s10664-016-9486-9

Eye movements in software traceability link recovery

Bonita Sharif[1] · John Meinken[1] · Timothy Shaffer[1] · Huzefa Kagdi[2]

Published online: 3 December 2016

Abstract Information Retrieval (IR) approaches, such as Latent Semantic Indexing (LSI) and Vector Space Model (VSM), are commonly applied to recover software traceability links. Recently, an approach based on developers' eye gazes was proposed to retrieve traceability links. This paper presents a comparative study on IR and eye-gaze based approaches. In addition, it reports on the possibility of using eye gaze links as an alternative benchmark in comparison to commits. The study conducted asked developers to perform bug-localization tasks on the open source subject system JabRef. The *iTrace* environment, which is an eye tracking enabled Eclipse plugin, was used to collect eye gaze data. During the data collection phase, an eye tracker was used to gather the source code entities (SCE's), developers looked at while solving these tasks. We present an algorithm that uses the collected gaze dataset to produce candidate traceability links related to the tasks. In the evaluation phase, we compared the results of our algorithm with the results of an IR technique, in two different contexts. In the first context, precision and recall metric values are reported for both IR and eye gaze approaches based on commits. In the second

Communicated by: Patrick Mäder, Rocco Oliveto and Andrian Marcus

✉ Bonita Sharif
bsharif@ysu.edu

John Meinken
jmeinken@student.ysu.edu

Timothy Shaffer
trshaffer@student.ysu.edu

Huzefa Kagdi
kagdi@cs.wichita.edu

[1] Department of Computer Science and Information Systems, Youngstown State University, Youngstown, OH 44555, USA

[2] Department of Electrical Engineering and Computer Science, Wichita State University, Wichita, KS 67260, USA

context, another set of developers were asked to rate the candidate links from each of the two techniques in terms of how useful they were in fixing the bugs. The eye gaze approach outperforms standard LSI and VSM approaches and reports a 55 % precision and 67 % recall on average for all tasks when compared to how the developers actually fixed the bug. In the second context, the usefulness results show that links generated by our algorithm were considered to be significantly more useful (to fix the bug) than those of the IR technique in a majority of tasks. We discuss the implications of this radically different method of deriving traceability links. Techniques for feature location/bug localization are commonly evaluated on benchmarks formed from commits as is done in the evaluation phase of this study. Although, commits are a reasonable source, they only capture entities that were eventually changed to fix a bug or resolve a feature. We investigate another type of benchmark based on eye tracking data, namely links generated from the bug-localization tasks given to the developers in the data collection phase. The source code entities relevant to subjected bugs recommended from IR methods are evaluated on both commits and links generated from eye gaze. The results of the benchmarking phase show that the use of eye tracking could form an effective (complementary) benchmark and add another interesting perspective in the evaluation of bug-localization techniques.

Keywords Eye-tracking · Software traceability link recovery · Continuous traceability · Eye-gaze benchmark

1 Introduction

Software traceability (Cleland-Huang et al. 2012) involves the ability to trace requirements across the software lifecycle. We must first generate, discover, or recover traceability links among software artifacts. In order for software traceability to be practical, we need to constantly maintain and evolve existing links. As software evolves (with bug fixes, and new feature additions), existing links could become suspect and possibly decay to a point where they do not accurately reflect the state of the system. The two broad areas in software traceability are traceability link recovery/generation and traceability link maintenance. Information retrieval (IR) approaches such as Latent Semantic Indexing and Vector Space Model (VSM) have been used to recover software traceability links (De Lucia et al. 2007). Most of these techniques need to be re-run on the system artifacts as the source code is modified, which takes a tremendous amount of effort and time to maintain existing links. IR methods also suffer from a large number of false positives (Asuncion et al. 2007), which is one of the reasons software traceability is largely missing in practice (Ramesh and Jarke 2001; Mader et al. 2009).

In 2007, an alliance of investigators from academia and industry identified several grand challenges (Coest.org 2007) to be undertaken in order for software traceability to become more prevalent and sustainable in software development. Our work is directly motivated by the grand challenges document and addresses several of the key problems in software traceability, one of the most important being reducing developer effort. In an effort to make software traceability more effortless (Cleland-Huang et al. 2011) and avoid the problems stated above, we devised an approach (Walters et al. 2014) that allows traceability links to be recovered by using a developer's eye gaze while they are solving a typical software task. All of the eye gaze collection is done unobstrusively while the developer is working on tasks

such as bug fixing or adding a new feature. This approach can also be used to evolve traceability links while the developer is working. It requires minimal input from the developer. This method is much more lightweight and developer-centric than the traditional IR methods for traceability link recovery and evolution, which tend to be artifact-centric. Eye trackers have become accessible to researchers who are using them to gain additional insights into software development activities (Sharafi et al. 2015). Modern eye trackers implicitly collect subjects' (e.g., developers) eye gaze data on the visual display (stimulus) in an unobtrusive way while they are performing a given task. This eye movement data could provide much valuable insight as to how and why developers arrive at a certain solution. We believe these measures can add a new additional dimension in supporting software traceability tasks.

In a prior pilot study published as a new and emerging idea (Walters et al. 2014), we found that eye-tracking data collected during bug-fixing tasks could help identify source code entities (SCE's) related to manually injected, single-line bugs. We collected eye gaze data via the eye-tracking enabled Eclipse plugin, *iTrace* (Shaffer et al. 2015) - the first of its kind to capture eye data within an IDE. The data collected in (Walters et al. 2014) was on a small system. The results from our pilot study show that strong recall is achieved when the developers accurately perform bug-localization tasks. We extend this prior work by running a study on a larger more realistic system and corresponding bug reports to determine if the initial results apply to larger systems. In addition, we also compare our results from developer gaze with existing state of the art IR methods. We believe this is a fair comparison to show that eye tracking can indeed generate quite useful data that can be used by traceability researchers to augment existing state-of-art methods. A comparison to IR methods was not conducted in our pilot study (Walters et al. 2014).

IR techniques find relevant locations in artifacts after which the developer records or vets the link locations. With the eye-gaze approach, *iTrace* is used by the developer to find the link as he/she works and at the end of the session *iTrace* captures and documents the link. These two tasks are different and one might argue are not head-to-head comparable. Since IR techniques are well known and are the current state-of-the-art in retrieving links, we were interested in seeing how they match up when compared with our eye gaze approach with respect to the oracle which in our case are the commits made to fix the bugs.

In practice, traceability information is recorded into a (manually created) traceability matrix (Davis 1993). As the number of artifacts increases, traceability matrices become extremely hard to manipulate. Apart from this scalability issue, they need substantial developer effort (not cost-effective). There is no silver bullet or single good approach when it comes to recovering traceability links (Lucia et al. 2006). There is a dearth of traceability information available to traceability researchers to validate their proposed techniques. The same (few) traceability datasets are frequently used, which exposes threats to external validity. With the *iTrace* environment, newly created eye-gaze datasets on developers' interactions with software artifacts will open a larger test bed, and enable other traceability researchers to validate their approaches. This approach can certainly complement previous approaches to traceability, including serving as a mechanism for cross validation. The eye tracking enriched datasets serve as a new platform to further confirm or refute previously established conjectures and results in software traceability.

The research questions we seek to answer in this paper are:

– RQ1: Is eye gaze a viable and feasible source to recover traceability links for features in real-world bug reports?

- RQ2: How do the results of the gaze-based algorithm compare to results from an IR method for traceability link recovery in terms of precision and recall and their usefulness to fix a bug?
- RQ3: How does eye gaze data compare with commit data as an oracle for bug localization tasks?
- RQ4: Do different IR methods such as VSM and LSI behave the same when compared to commit data and eye tracking data for bug localization tasks?

The first two RQs relate to how gaze can be used to recover traceability links. The last two RQs seek to compare IR techniques to gaze data and commit data in the hope of proposing a new data set based on eye gaze for comparison studies in the future. The intention behind each of these research questions is explained below.

The first question (RQ1) is to determine whether eye tracking data is a viable source to help identify SCEs related to concepts in more true-to-life bugs. In other words, we were trying to determine if we can recover traceability links to features in a bug report and source code using eye gaze. To this end, we conducted a study and asked five developers to investigate real bug reports on an open-source system and perform feature location tasks. The eye-tracking data from their sessions was analyzed using our gaze-link algorithm to generate a list of source code entities (SCE's) for each task/feature represented in the bug report. These are our candidate links.

The second question (RQ2) is to determine if our method produces more useful (in terms of fixing the bug) results than currently used IR methods. We compared each bug report against the source code using LSI (Latent Semantic Indexing) and VSM (Vector Space Model) methods. We report these results in two contexts. First the precision and recall of LSI, VSM and gaze approach is compared with respect to commits on a class-level granularity. Second, the most successful of the two IR methods was then selected to test against our gaze-link algorithm to determine usefulness of the generated links. A ranking of usefulness of the IR-generated links and the gaze-link algorithm's generated links was completed by a second group of developers in a blind setup. In a majority of the tasks, the gaze-link algorithm's links were considered to be more useful (for the bug fix) to the second group of developers. IR methods take as input, source code and bug reports to find a list of candidate links using a similarity measure. Our method takes as input, source code, bug reports and eye gazes on source and bug reports to find a list of candidate links using a algorithm we devise. This additional eye tracking information helps our approach achieve more useful results.

Empirical evaluation is an accepted and expected form of validation in the software maintenance and evolution research community. Building effective benchmarks has always been a critical component. Using human experts to assess the results of a particular technique, i.e., human validation, has a long history. In the last decade or so, the widespread availability of software repositories has made possible to use software changes (e.g., commits) for benchmarking. Maintenance techniques such as feature location, impact, analysis, and bug localization have relied heavily on these forms of benchmarks. The fundamental premise in validity using commits is that if the relevant units of source code recommended by these techniques are found in commits, the technique is deemed appropriately accurate. Although, commits offer one benefit, they do not necessarily and sufficiently capture all the relevant entities. For example, they capture only the entities that were eventually changed to fix a bug in a feature, but not all the other entities that also implement the feature or were important in

locating the final changes. We posit that this limitation of commits does not depict an accurate picture of the effectiveness of automated maintenance techniques. To help address this issue, we seek to investigate another validation methodology using eye tracking data (RQ3 and RQ4). We compared different information retrieval methods on two data sets: commits and eye gaze data. The eye gaze data consisted of the links generated from our gaze-based approach (results of RQ1). This proposed benchmark that uses eye gaze data is compared with commit data at both the method and class level granularities.

The paper is organized as follows. The next section discusses two traceability scenarios that are possible with eye gaze. This is followed by a discussion on the role of eye gaze in traceability research. In Section 4 we first present an overview of the *iTrace* Eclipse plugin and the gaze-link traceability approach with an example on how links are generated. This is followed by an overview of IR methods in Section 5. In Section 6, we describe the study conducted and provide the study results in Section 7. A discussion and implication of the results is provided in Section 8. Threats to validity are presented in Section 9. Finally, the last two sections present related work followed by conclusions and future work.

2 Traceability Scenarios

A quest for ubiquitous traceability was presented by Gotel et al. (2012) where a roadmap was provided to help bring together diverse approaches to solve the problems that plague traceability adoption in practice. We believe using eye tracking is one such approach that is very different from traditional methods of how traceability is currently done. The eye-tracking enabled software traceability environment *iTrace*, will collect pertinent eye gaze data necessary to generate a software traceability model and generate candidate traceability links. This entire process is conducted without interrupting the developer's workflow. In *iTrace*, the links are automatically collected for only pertinent artifacts for each stakeholder (developer, tester, project manager). We can tailor links to individual stakeholders for their assigned tasks. We illustrate two possible scenarios of how our approach could be instantiated and realized.

2.1 Traceability Link Recovery - The Canonical Approach

The forward engineering support for traceability imposes an additional burden of establishing traceability links on the developers, who are often faced with demanding delivery schedules. Therefore, traceability often downgrades to a low-priority activity with no obvious immediate benefits to the developers. With the eye tracking approach, a developer could be building the software from scratch, i.e., writing requirements, building design diagrams, and writing code; all of which will be tracked by *iTrace* with links generated as the software is being engineered. A developer might be reading the requirements specification and implementing the necessary code. *iTrace* would monitor all the developer interaction and eye movements while a developer is performing this task. Pertinent information such as duration of eye gazes and patterns of eye gazes on and between different artifacts are captured. This information is analyzed to establish link candidates between the implemented requirements and code. The granularity of artifacts may range from sentence to documents for textual documents, and from statements to file for source code. Once the user has

finished a session, a link validator in *iTrace* would prompt the user to verify the established link candidates (as is done in current state-of-the-art approaches).

2.2 Continuous Traceability

xIn a maintenance environment, *iTrace* would monitor the developer activities associated with performing a maintenance task (e.g., bug fix). Here, a traceability link would be established between a specific bug report (description of the bug) and the source code fix (changes to the source code). Our pilot study (Walters et al. 2014) and the study in this paper is conducted on this latter scenario. We started with this scenario as it is the most common. Developers typically work on existing systems to fix bugs or modify existing features. Given the unobtrusive nature of the *iTrace* approach, it is reasonable to imagine a future in which such technology is used to capture trace links under the hood.

3 The Role of Eye Gaze in Traceability Research

The use of eye tracking (via *iTrace*) in software traceablity research will bring forward two broad possibilities: linking eye tracking data across models, and using eye tracking data to augment existing link recovery techniques. It is well established in eye tracking literature that comprehension occurs during eye fixations, the frequency of fixations implies importance of an entity, and fixation sizes may imply bottlenecks (Duchowski 2003).

First, with respect to linking eye-tracking data across models as shown in the study presented here, *iTrace* would use frequency, duration and patterns of eye movements to determine which source code element for example, should be linked to another model such as bug reports or design diagrams. In order to perform such a linking mechanism, eye tracking on source code and other artifacts are necessary. *iTrace* can be extended to include support for other software artifacts such as design diagrams, test cases, bug reports, or any other form of semi-structured text. *iTrace* will collect eye tracking data while developers are working with different types of software artifacts in an IDE such as Eclipse. A link retrieval algorithm for *iTrace* will then analyze the eye tracking data to determine if a link between the software artifacts exists (e.g., source code and design diagrams, source code and requirements, or source code and bug reports). This algorithm would have a mechanism to negate the factors such as gaze information that did not contribute to the developer's task (e.g., outlier analysis – developers looking at things they are not really working on). Our initial implementation of the algorithm handles this filtering quite well.

The second possibility with *iTrace* is to use eye-tracking measures as metadata to enhance existing traceability link recovery algorithms such as LSI and probabilistic approaches. As an example, in Poirot Tracemaker (Lin et al. 2006), we could use the eye gaze information such as the number of fixations to determine the most viewed class or method in a class diagram and plug that into the traceability link finder (e.g., as meta data). The same can be done for the ADAMS (De Lucia et al. 2009) traceability link recovery tool. This possibility uses existing eye gaze behavior of individuals to feed the link recovery process. *iTrace* is envisioned to support traceability among a wide spectrum of artifacts. An algorithm is already developed and evaluated to establish links between requirements (in the form of bug reports) and source code. Ali et al. (2012) look into using eye gaze to help improve weighting in IR methods. We discuss their approach in Section 10 on related work.

4 The Gaze-Link Traceability Approach

We give a brief description of *iTrace* – an Eclipse plugin used to gather eye tracking information while a developer is working on tasks. Next, we present an overview of how traceability links are generated from developer eye gaze using a running example.

4.1 An Overview of iTrace

As stated earlier, the novelty of our approach is that it relies on eye gaze (gathered on software artifacts) as well as the developer skill and knowledge to find links. In our study, the links are determined between source code and the feature location tasks. *iTrace* (Walters et al. 2014; Walters et al. 2013), allows us to capture line-level data in source code of where a developer is looking. Furthermore, it tells us the particular source code entity (SCE) that the eye gaze on a line maps to. The *iTrace* plugin receives raw eye gaze data in the form of (x, y) coordinates from the eye tracker while a developer is at work. These coordinates are then mapped to line and columns within Eclipse to give us line-level gaze data. One of the biggest benefits that *iTrace* provides is support for scrolling while accurately tracking developer's eye movements. This feature implies that we are no longer limited to small code snippets to run eye tracking studies on source code such as reported by Ali et al. (2012). Our study is based on a large real-world software system and the developer is able to freely move between any number of files in Eclipse with the assurance that their gaze will be recorded while they scroll both horizontally and vertically (a very realistic scenario) and navigate the huge code base. In summary, *iTrace* provides the SCE-level granularity of a developer's eye gaze while they are performing a task.

4.2 How Links are Generated Using Eye Gaze

In order to infer pertinent information from the large amount of raw data collected from the eye tracker, we devise an algorithm that parses through the data and outputs the most relevant SCEs (source code entities that the developer looked at the most and focused on the most towards the end of the session). The first version of this algorithm was presented in (Walters et al. 2014). In the first version of our algorithm, we used *srcML* to map the line and column numbers in the gaze to the appropriate qualified names for source code entitites such as method names, variables, and class names. See Fig. 1 for a brief overview of how the link generation process worked at the time the study was conducted. The most recent version of the algorithm does not use *srcML*. Instead, we do the mapping of which

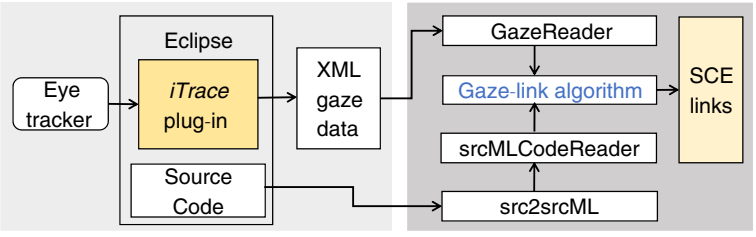


Fig. 1 The main components (simplified) in the gaze-link approach to recovering traceability links

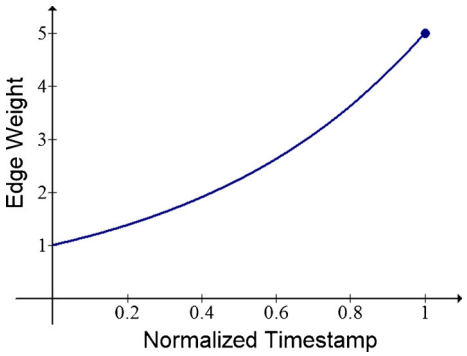


Fig. 2 Weighting function

SCE the raw gaze maps to directly within Eclipse using the Eclipse AST model. This way we do not have to do the post processing of running this through *srcML* and map the SCE information in real time. This difference only affects where the mapping actually occurs. Without *srcML*, the mapping is done on the fly and online right after the gaze is received. We moved to an online model to reduce the steps in post processing the eye gaze data.

The gaze link algorithm takes as input the raw gaze data from one or more developer sessions. For example, links can be generated with respect to one developer after they are done with performing their task or with respect to a set of developers. We refer the reader to Walters et al. (2014) for the initial pseudocode of the algorithm and the updated implementation online.[1]

We now give a short description of how the links are generated. For each gaze, *iTrace* records a timestamp. We normalize these timestamps between 0 and 1 to make processing easier and consistent across developers. The output of our gaze algorithm is a weighted digraph with SCEs (which may be methods or member attributes) as vertices and the gaze path that the developer took between SCEs represented as edges between these vertices. When an edge (gaze path between two SCEs) is encountered in the gaze data, a weight is added to that edge. The weight is calculated from the normalized timestamp using an exponential equation. More information about this function and how we tested it with different values is given in Walters et al. (2014). We give more weight to the elements that occur towards the end of the session. When an edge is encountered, its weight is set using the exponential function:

$$f(x) = a^x \tag{1}$$

If an edge is encountered multiple times (meaning the developer went back and forth between two SCEs), a new weight is assigned. If the base of our exponential function is set to five, an edge encountered at the end of the session will have a weight of five times an edge encountered at the beginning of the session. We set our base to 100 for the purposes of weight calculation after testing it empirically on several data sets.

The value for a can be adjusted to set how much additional weight is given to edges encountered later in the session. Setting a to 1 results in a flat line, adding equal weight to

[1] https://github.com/seresl/itrace-gazelink-algorithm

all edges regardless of when they are encountered. Figure 2 shows a graph of this function for $a = 5$. For the present study, a was assigned a value of 100. We tested this function with values: 1, 10, 100, and 1000 and selected the best one ($a = 100$). There was no improvement when 1000 was used.

The edge weights are then normalized to [0.0, 1.0]. At this point the algorithm has determined source-to-source links from the developer's gaze data. Finally, to determine the links i.e., list of SCEs that are most relevant, the scores for each SCE need to be calculated. The SCE scores are extracted from the remaining edges (after running a high-pass filter to eliminate weak links) by summing the weights of all edges to which a given SCE is connected. These scores are also normalized to [0.0, 1.0]. At this point, the most important SCEs to the current task have been determined for each developer. The cutoff values were determined empirically. If an SCE appears in multiple developers' results, the scores are summed. See Fig. 3 for an example of a graph that was generated for one particular developer for bug id: 1542552. The numbers shown are the weighted scores.

Finally, the maximum and minimum scores are used to normalize the result to [0.0, 1.0] and a high-pass filter with a cutoff of 0.6 for classes and methods and 0.1 for variables is used. These SCEs and scores represent links from the SCEs to the current task and their related use cases with a strength proportional to the score. See Fig. 4 for an example of the link generation process.

The SCE scores from multiple developers on the same task can optionally be merged to create a single set of SCE scores. Merging is done by combining SCE lists from all developers into a single list. If an SCE only appears in one developer's results, the score for that SCE is used directly (i.e., sum of 1 developer). If an SCE appears in multiple developers' results, the score for that SCE is set to the sum of the individual scores. See Fig. 5 for a graphic representation of the merging of SCE's.

4.3 Algorithm Modifications

There are some important differences between this algorithm and the one introduced earlier (Walters et al. 2014). Previously, the timestamp was used directly without normalizing to calculate edge weights. Without normalizing, we realized that the final SCE weights produced by the algorithm could be slightly different depending on the start time. The

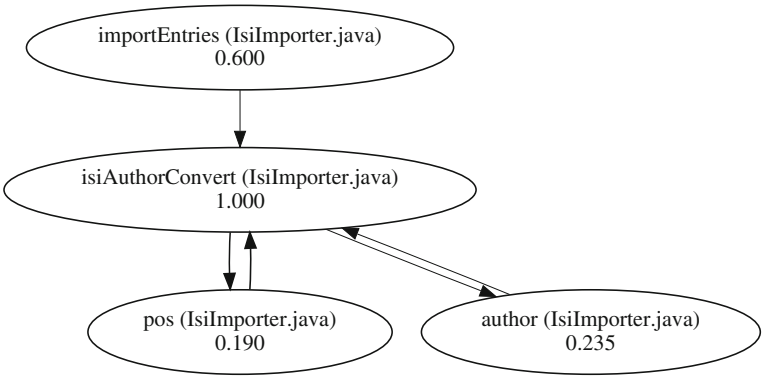


Fig. 3 The SCE's with their corresponding weights for bug ID 1542552 (Wrong author import from Inspec ISI file) for one developer

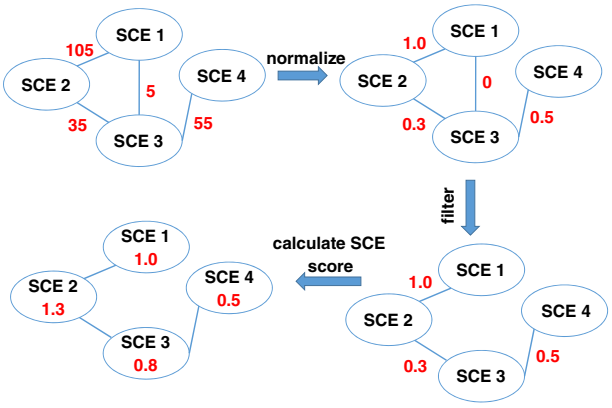


Fig. 4 An example of how the score is calculated for each link (SCE). Edge weights are normalized between (0 and 1) and a highpass filter of 0.1 is applied. To generate an SCE score, weights of all edges attached to that SCE are added. The scores are then normalized between (0 and 1) and a highpass filter of 0.6 is applied to classes and methods and 0.1 is applied to variables. Note that the scores shown in the figure are not normalized. The normalized scores for each link to the bug report are SCE 1: 0.7, SCE 2: 1.0 SCE 3: 0.6, SCE 4: 0.3 for the above example

previous algorithm rewarded later discovery of an edge by assigning the edge weight as the square of the timestamp on the first time an edge was encountered. For subsequent encounters of the same edge, the timestamp was added directly without squaring. We decided that using the square of the timestamp was too arbitrary, especially since the timestamp values were not being normalized. Instead, weights are now calculated using an exponential function of the normalized timestamp as given above. Our new method does not differentiate between the first encounter of an edge and subsequent encounters. Instead, we assign higher weights to all edge encounters that occur later in the session. This method is used to decay older links. The changes described above resulted in a much wider spread of edge and vertex weights. While the process of using high-pass filters to eliminate edges and vertices with low-relevance has not changed, the values for those filters have been adjusted to accommodate the wider spread of weights.

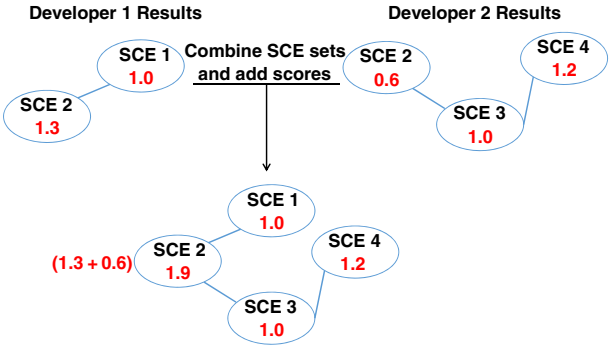


Fig. 5 An example of how the scores are merged for multiple developers

5 Information Retrieval Approaches - VSM and LSI

Information Retrieval techniques work by performing a text comparison of one document against a collection of documents to identify similarity matches. Popular IR methods for trace generation include Vector Space Model (VSM) and Latent Semantic Indexing (LSI).

5.1 Vector Space Model

The Vector Space Model (VSM) (Manning et al. 2008) is a popular scoring scheme because of its efficiency and versatility. Efficient implementations of vector and matrix operations are readily available and can be used for bulk searching of large collections of documents. In VSM, each document is represented as a vector. This vector representation discards term order and only records term frequencies in each document. If each term in a document lies on its own axis in the vector space corresponding to the document, the dot product of two document vectors can be used to compare them. The dot product is easy and efficient to compute, and returns the cosine of the angle between the two document vectors. Similar documents would have cosine similarities close to 1, while dissimilar documents would have cosine similarities near 0.

This similarity metric is preferred over simpler techniques such as directly comparing term frequency magnitudes because it is effective regardless of document lengths. Computing the magnitude of the vector difference between two documents would only correctly measure similarity between documents of similar lengths. The same cosine similarity can be trivially adapted to searching by representing queries as vectors in the vector space of the documents and calculating the cosine similarity between the query and each document.

Term frequencies are generally calculated using a weighting scheme that takes the total number of documents into account, such as tf-idf. In tf-idf, the weight of a term t in a document d is given by

$$tf - idf = tf_{t,d} \times idf_t$$

where $tf_{t,d}$ is the number of times term t occurs in document d and $idf_t = \log \frac{N}{df_t}$ where N is the total number of documents in the collection and df_t is the number of documents in the collection containing term t. tf-idf corrects for terms that occur frequently throughout the collection so that a term that occurs only once carries more weight than a term that occurs in every document.

5.2 Latent Semantic Analysis

Latent Semantic Analysis (LSA) or Latent Semantic Indexing (LSI) (Deerwester et al. 1990) is an extension of VSM used in natural language searching. LSI represents the collection of document vectors as a matrix and works from an approximation with fewer dimensions. The approximate form of the document collection matrix captures some word meaning and allows searches to find synonyms, e.g. a query for ‘car’ might return a document containing ‘automobile’.

LSA is a computationally expensive process and is based on singular value decomposition (SVD), an optimization technique that reduces the size of document collection matrices while preserving structure. The SVD for a collection of documents must be precomputed, and addition/removal/change of a document requires recomputation. With our approach using gazes, no recomputation is required and *iTrace* will eventually list links as the developer is working or right after they are done with the task.

6 The Study

In this paper, we seek to compare the gaze-link method with IR methods to determine if the gaze-link method is indeed superior to the conventional IR methods for traceability link recovery. We conduct a comparative study to determine how the gaze-link method of deriving traceability links fairs against IR methods in two contexts. The first context referred to as the *precision and recall context* deals with comparing each approach to how the developer fixed the bug (used as the ground truth/oracle). The second context referred to as the *usefulness context* deals with how useful another set of developers found the links generated by each approach. The design of the study is described next.

6.1 Study Design Overview

The study design is shown in Table 1. In the first *data collection phase* of our study, we asked a group of five developers to perform bug localization tasks for real bug reports on an open source system and tracked their eyes unobtrusively while they performed the tasks. Next, we fed the eye gaze data to our gaze-link algorithm to give us a set of links directly related to the particular bug reports. In the general sense, a bug report can be compared to a

Table 1 Study overview. The first four rows related to RQ1 and RQ2. The last row relates to RQ3 and RQ4

Steps	Main Details
1. Data Collection Phase (RQ1 and RQ2)	Conducted an eye tracking study with five Java developers using the Tobii X60 eye tracker. Main Task: Identify the classes, methods, and variables relevant to fixing the bug.
2. Generate links using IR methods (RQ1 and RQ2)	Run VSM and LSI on the bug reports and source code to generate a ranked list of SCEs and their scores. VSM performed better than LSI and was used in the rest of the analysis. Top 4 results are used. This was done on Top-5 and Top-10 threshold levels and at class and method level granularity.
3. Generate links using gaze-link algorithm (RQ1 and RQ2)	Run gaze-link algorithm on the gaze data from all developers in Step 1. above to generate a ranked list of SCEs and their scores. Top 4 results are used.
4. Evaluation Phase (RQ1 and RQ2)	*Precision and Recall Context*: Calculate Precision and Recall for IR and gaze-link algorithm based on how JabRef developer solved the bug in commits (oracle). *Usefulness Context*: Conducted an online questionnaire with eight Java developers (no overlap with developers in Step 1. above). Main Task: Rank the randomized list of SCEs (VSM and gaze-link) in a blinded setting based on how useful they are in solving the bug.
5. Benchmarking Phase (RQ3 and RQ4)	A set of graphs and tables based on precision and recall w.r.t commits and eye gaze are presented to show trends in the way different IR techniques work on eye gaze data and commit data as benchmarks. An overlap analysis of the two benchmarks is also presented.

use case/requirements document in which we would be generating requirements-to-source-code traceability links. We also run several IR methods (VSM and LSI) with the bug report against the source code.

After the gaze data was collected (and before doing the evaluation phase), it is analyzed offline using our gaze-link algorithm. Results from multiple developers on the same task were merged using the approach defined in the algorithm described resulting in a single SCE list for each task. Prior to the study, the bug reports used in the study were analyzed using Latent Semantic Indexing (LSI) and Vector Space Model (VSM) generating a ranked list of entities at both the method and class level granularity.

In the *evaluation phase* of our study we asked a different group of developers to compare the usefulness of the IR results and the gaze-link algorithm's results in a blind ranking exercise. In addition, we also report on the standard measures of precision and recall with respect to a class-level oracle. The oracle refers to how JabRef developers fixed the bug in the repository. More information on evaluation is presented in Section 6.10.

In the benchmarking phase (last row in Table 1), IR results are compared to commits and gaze-link approaches and an overlap analysis on both benchmarks is performed. During the description of the study and the results we provide context by mentioning which phase of our study we are referring to. The phases (data collection, evaluation, and benchmarking) are listed in Table 1 along with a short summary of what comprises each phase. All study material with a replication package is available online.[2]

6.2 Subject System

We used *JabRef* (v. 1.8.1, release date 9/16/2005), a graphical application for managing bibliographic databases as the subject system in the study. JabRef is approximately 38KLOC in 311 files. It uses the standard LaTeX bibliographic format BibTeX, though it can import and export many other formats. It is an open-source, Java-based system available on SourceForge (http://jabref.sourceforge.net/).

6.3 Tasks

We used eight real bug reports as tasks for our study. The bug reports were presented verbatim to the participants. The bug reports used were submitted soon after JabRef v. 1.8.1 was released and were considered fixed and closed when the study was conducted. By using an earlier version of the software, we were able to see exactly what the open-source developers did to patch each bug. Eight bugs were selected to give a wide range of scope and difficulty levels. The simplest bugs were fixed on a single line of code, while the most complex bugs involved changes to multiple methods in multiple classes. We classified bugs as easy, medium or difficult based on how the JabRef developer solved it. We based our rating on how complex the bug was and how hard we thought it would be to solve it (given that we already knew the solution as implemented in the repository).

In the data collection phase, a developer was exposed to four bugs (one was given six bugs to solve) with an overlap of at least 2 bugs across all five developers bringing the total number of bugs tested for to 8. Two of the bugs were solved by all five developers. There were eight tasks in total, but in the data collection phase, each developer only completed

[2] http://www.csis.ysu.edu/~bsharif/itrace-IRvsGaze

Table 2 JabRef bug reports used in the study

Task ID	Bug ID	Date Submitted	Title
1	1548875	8/29/2006	download pdf produces unsupported filename
2	1436014	2/21/2006	No comma added to separate keywords
3	1594123	11/10/2006	Failure to import big numbers
4	1489454	5/16/2006	Acrobat Launch fails on Win98
5	1594169	11/10/2006	Entry editor: navigation between panels
6	1540646	8/15/2006	default sort order: bibtexkey
7	1631548	1/9/2007	Open last edited DB at startup depends on the working dir
8	1542552	8/18/2006	Wrong author import from Inspec ISI file

some (between four and six) of the tasks. We did this to avoid fatigue in our experiment and to keep the entire session less than an hour for most. Two of the tasks were completed by all five developers while the other six tasks were completed by at least two developers each. The two developers who completed all the tasks were experts. The only criterion was that we wanted one easy and one difficult task to be represented as additional tasks. Information about each bug is provided in Tables 2 and 3. The developers were shown the tasks in a random order to minimize any order effects.

6.4 Participants

In the eye gaze data collection phase, five developers participated. One of them was a novice with 1-2 years programming experience in Java. The other four developers had at least 5 years of programming experience in Java except for one developer who had 3 years of experience in Java. All developers rated themselves as somewhat familiar with the Eclipse IDE. Developers rated their bug-fixing experience as average or better with the exception of the programming novice, who rated their skills as below average. All developers had at least occasional experience with bug-fixing tasks. None of the developers were familiar with the subject system prior to the study. They were given a one page description on the subject system prior to the session by email and again at the start of the session. The participants

Table 3 Details on the bug reports used in the study along with the developers assigned to each bug task

Task ID	Bug ID	Difficulty Rating	Scope of Solution	Developers Assigned
1	1548875	moderate	single method	1,3
2	1436014	difficult	multiple classes	4,5
3	1594123	easy	single method	4,5
4	1489454	easy	single method	1,3
5	1594169	difficult	multiple classes	1,2
6	1540646	moderate	single method	1,2
7	1631548	easy	single method	1,2,3,4,5
8	1542552	difficult	multiple methods	1,2,3,4,5

The difficulty rating was determined based on the number of changes required and how complex the bug was

Table 4 Task ID assigned to participants - grouped by task difficulty. Tasks 7 and 8 were completed by all participants

	Task Difficulty		
Participants	Easy	Moderate	Difficult
1	4, 7	1, 6	5, 8
2	7	6	5, 8
3	4, 7	1	8
4	3, 7		2, 8
5	3, 7		2, 8

in this phase were senior graduate students and experienced faculty at Youngstown State University. See Table 4 for the tasks that were assigned to each participant grouped by task difficulty.

For the evaluation phase, a separate set of eight developers participated. They were senior undergraduate and graduate students at Wichita State University and Kent State Universit. Three participants had more than 5 years of Java and programming experience, three had between 1-2 years of experience in Java and the rest had 1-2 years of programming experience. All developers were faced with bug fixing tasks at least occasionally with one exception who had only 1-2 years of experience and rarely fixed bugs. Developers all rated their own object-oriented programming skills as at least average or above.

6.5 Data Collection Phase for the Gaze-Link Algorithm

In the data collection phase, five developers completed their sessions in the eye-tracking lab at Youngstown State University. Gaze data was collected using the Tobii X60 eye tracker and the *iTrace* Eclipse plugin. The font size was set at 14 points. This value was chosen so that the Tobii X60 could accurately track each line. The Tobii X60 has an accuracy of about 0.5 degrees that corresponds well to the chosen font size. Video and Audio was also recorded, and the session was monitored by a moderator to ensure collection of accurate data throughout the study. The Tobii X60 eye tracker does not require the developer to sit completely still. It allows for some normal head movement within a bounding box.

A week before the study, the participants were sent a short 1-page description on JabRef. This was a high level overview of the features JabRef provides. Since none of the participants read this one page overview prior to the study, they were asked to read the short JabRef description before they began the study. Please refer to the replication package for the description presented to participants.

On the day of the study, the developers worked with the JabRef source code in the Eclipse IDE. They also had access to a working executable version of JabRef 1.8.1 so that the bugs were reproducible. They were not required to compile the source code or use debug mode, but they were allowed to use native Eclipse search tools. For each task, developers were given the full text of a bug report and were asked to identify the classes, methods or variables relevant to fixing the bug. They were not required to completely solve the bug or edit any code. The main idea behind the tasks was to determine if developers can find relevant SCEs that help towards a feature in which the bug might be reported in with the ultimate goal of fixing the bug in the future. Hence concept/feature location was the most important aspect of this task. Therefore the main task across all bug reports was: *Identify the classes, methods, or variables relevant to fixing the bug*.

6.6 Generating Traceability Links Using IR Methods

The bug reports are evaluated against the source code using two IR methods: LSI and VSM. LSI and VSM return a list of documents ordered by relevance. We used Apache Lucene's implementation for VSM and Semantic Vectors[3] for LSI. The default value for k was used in Semantic Vectors which was 200 dimensions. They were set to output the top-5 and top-10 relevant SCEs and were run on both class level and method level granularity. For class level granularity (in Java), there is a 1:1 correspondence between documents and classes. For method level granularity, each method was put into a separate document. A simple script based on regular expressions was used to split the methods into separate documents. Note that the IR techniques were not dependent on eye gaze data so they could have been run prior to the data collection phase.

6.7 Generating Traceability Links Using Eye Gaze

After the data collection phase of the study, traceability links were generated by the gaze-link algorithm. The gaze data collected from the tasks was fed into the gaze-link algorithm to obtain a list of SCE's related to features corresponding to the bug report task. Data collected from multiple respondents on the same task were merged using the composite approach defined in the algorithm to generate a single list of SCE's for each task. In addition, it was noted that all of the VSM matches to the oracle occured in the top four VSM results. We use only VSM results in the *usefulness context* of the evaluation phase.

6.8 Metrics Used in Evaluation Phase and Benchmarking Phase

For the *precision and recall* context of the evaluation phase, a class-level oracle was produced that assigned a value of 1 to any class related to the bug and a value of 0 to all other classes. This oracle was generated based on the way the JabRef developers fixed the bug in the Sourceforge repository. Comparing the results from each IR method using the oracle, we determined that VSM had both the best precision and best recall. However, we report the precision and recall for both VSM and LSI. We calculate the following for each bug. The first four equations state how precision and recall is calculated for IR and gaze-link methods with respect to commits. IR refers to the information retrieval method used i.e., LSI and VSM. ET refers to the eye tracking method (gaze-link traceability approach) proposed in this paper.

$$IRprecision_{\text{w.r.t commits}} = |(IR \cap Commits)|/|IR| \tag{2}$$

$$IRrecall_{\text{w.r.t commits}} = |(IR \cap Commits)|/|Commits| \tag{3}$$

$$ETprecision^{\text{w.r.t commits}} = |(ET \cap Commits)|/|ET| \tag{4}$$

$$ETrecall_{\text{w.r.t commits}} = |(ET \cap Commits)|/|Commits| \tag{5}$$

For the benchmarking phase, we need the following two equations to compare IR (LSI and VSM) precision and recall with respect to eye gaze with the IR (LSI and VSM) precision

[3] https://code.google.com/p/semanticvectors/

and recall with respect to commits (the first two equations above). In the benchmarking phase, we conduct the analysis at the class and method level.

$$IRprecision_{\text{w.r.t ET}} = |(IR \cap ET)|/|IR| \quad (6)$$
$$IRrecall_{\text{w.r.t ET}} = |(IR \cap ET)|/|ET| \quad (7)$$

where IR represents the set of entities generated by LSI or VSM set at a threshold of top-5 or top-10, ET represents the entities found by eye gazes, namely $\text{ET}_{\text{weighted}}$ or ET_{raw}, and commits represents the set of entities committed into the SourceForge repository as a result of the bug fix. The precision and recall is calculated at two levels - class and method granularity. This was done to determine if there are any significant differences in the results.

In the benchmarking phase, we present two versions of eye tracking generated entities. The first is based on all the SCEs the developers saw without filtering, i.e., without running the algorithm. We refer to this as ET_{raw} derived directly from the gaze files as output by *iTrace*. The list of SCEs generated by the algorithm after filtering and scoring is referred to as $\text{ET}_{\text{weighted}}$. We use both these sets to show the importance of having an algorithm sweep through the data since a lot of the things a developer looks at early in the bug localization session are abandoned as they get closer to the solution.

6.9 Hypotheses for Benchmarking Phase

Based on our research questions RQ3 and RQ4, we present the following null hypotheses.

H_1: *There is no difference between the the commit and eye tracking benchmarks.*

H_2: *There is no difference in precision and recall between the IR methods: VSM and LSI, when calculated with respect to the two benchmarks based on commits and eye tracking.*

The purpose of H_1 is to determine if there really is a difference in the proposed benchmark based on eye tracking data when compared to commits. If they are indeed different, we can reject this hypothesis and claim that eye tracking is another means of comparing IR method results and that eye tracking data is truly different because it considers not only what changed at the end but also what developers studied/observed for the change.

The purpose of H_2 is to determine how different IR methods compare with the two benchmarks. We discuss several versions of this hypothesis in the results. For instance, we look at the top-5 vs. the top-10 results of the IR method, two variants of the ET benchmark ($\text{ET}_{\text{weighted}}$ and ET_{raw}) at two levels of granularity: method and class. The goal is to determine if we see any trends or flip results between the IR techniques when compared to the two benchmarks.

6.10 Evaluation Phase

To evaluate the traceability links generated from our gaze-link algorithm, we compare our links with those generated using standard IR techniques (LSI and VSM in our case).

We perform the evaluation phase in two contexts. The *precision and recall* context is calculated at the class-level for both IR and gaze-link with respect to the oracle generated from developer commits of the fixed bugs. The gaze-link algorithm gives traces at the method/line level however, in this study we map that to the class it belongs to in order to compare it to IR methods. The IR methods fair significantly poorly when it comes to method level analysis and hence we chose a class level approach in this study. We acknowledge there are many

other possibilities to test within IR methods as well but we leave that as future work. The *usefulness* context is with respect to how useful another set developers found the links generated by IR and gaze-link approaches. The usefulness rating is based on whether the link would help fix the bug.

When calculating precision and recall, we compare the SCE lists with the list of actual SCE's that were modified by the JabRef developers when they were fixing the bug. One problem with this approach is that there is sometimes more than one way to fix a bug. For example, one of the bugs in our study required that a particular value be added to a list in a preference menu. The preference menu list was populated from a global array. One valid approach to solving the bug would have been to add the needed value directly to the global array. However, the developer chose not to solve the bug that way. Instead, he/she appended the value to the global array in the method that created the preference menu. In this case, the global array is relevant to the bug even though it was not modified by the developer. Also, the boolean nature of the oracle fails to take into consideration that there are different degrees of relevance to different SCEs.

For the second *usefulness context* in the evaluation phase, we recruit a different group of developers to rate the usefulness of the SCE's (links) generated by both IR and gaze approaches for fixing the bug. For each task, SCE's from IR and our gaze algorithm were mixed randomly, and developers were asked to rank each SCE on a five-point scale (where 1 = not useful, 5 = very useful). This rank denotes the usefulness score of the link. The developers were asked to rate each SCE based on its usefulness in helping to locate, understand and eventually fix the bug. The *usefulness context* part of the evaluation phase was conducted online. The developers were given a copy of the JabRef 1.8.1 source code and a working copy of the JabRef 1.8.1 software. For each bug, they were also given the full text of the bug report and a description of the solution as implemented by the developer on the real project. In addition, for each task, the two SCE lists (from VSM and our gaze-link algorithm) were combined into a single list and ordered randomly. For the usefulness context, we chose to show links for only VSM as it performed the best. Developers were then asked to rank each SCE on a five-point scale based on its usefulness in helping to locate and understand the bug with 5 being the most useful and 1 being the least useful. The developers did not know which SCE came from which link prediction method, or even how the SCE list was generated. Hence the main task in this phase was to rank the list of SCE's based on their usefulness to solve the task. Each developer was asked to rank all the eight bugs shown in Table 2. For this phase, we had eight participants with only one participant completing 5 out of 8 tasks. The participants were told to try and fix the bug before they rank the entities.

7 Study Results

We first present an example comparing IR and gaze-based approaches. Next, results on the bug task results are presented. This is followed by results of the precision and recall context and the usefulness context of the evaluation phase. Finally, results of the benchmarking phase are presented. We follow the evaluation method as stated in Section 6.10.

7.1 An Example Comparing IR and Gaze Approaches

We present an example of the IR results, the gaze-link results, and the commits for one specific bug to provide the reader with a concrete indication of how the SCEs differ between approaches. The relevant SCEs generated via IR, the gaze algorithm, and the commits are

shown in Table 5 for one specific bug. We consider the Bug ID 1489454 entitled "Acrobat Launch fails on Win98". The top-5 SCEs are shown. The SCEs for IR and eye tracking ($ET_{weighted}$) are shown in order of rank (in case of IR) and weighted score (in case of eye gaze) with the top most result indicating highest rank.

This was a simple bug that required a change to one single method. In the `openExternalViewer` method of the class `Util`, a method-level string variable `cmd` is hard-coded with a command for launching Acrobat sent to "cmd.exe". Then the `cmd` string is passed to a `Process` object called `child` which executes the command in Windows. In Win98 and earlier, "cmd.exe" needs to be changed to "command.com." The solution implemented by the developer checks the version of Windows using a conditional statement to create the appropriate string for `cmd`.

The relevant SCEs to this bug generated by the gaze algorithm are the `Util` class, the `openExternalViewer` method and the `child` and `cmd` variables. At the class level, LSI listed `Util` as the fifth in the list but it was not listed at the method level. The classes identified by VSM were not closely related to the bug. The gaze algorithm gave the relevant method as its first-ranked result, and the two relevant variables as the second and third ranked entities.

In this particular case, there is no overlap between the VSM, and ET datasets. However, we do see an overlap with the commit and eye tracking data. The $ET_{weighted}$ list was generated by the gaze algorithm. If we compare the $ET_{weighted}$ with the ET_{raw}, we can see how the algorithm excluded irrelevant entities and narrowed it down to three main related entities. The gaze-link algorithm gave the relevant method as its first-ranked result, and the two relevant variables as the second and third ranked results. In our analysis we only present findings at either the class level or the class and method level. We exclude eye gazes on

Table 5 An example of the class and method level SCEs for Bug ID 1489454 entitled "Acrobat Launch fails on Win98"

Technique	*Class-level*	*Method-level*
Commit	`Util`	`Util.openExternalViewer`
ET_{raw}	`BasePanel`	`RightClickMenu.actionPerformed`
	`JabRefFrame`	`Util.bool`
	`Util`	`Util.checkLegalKey`
	`JabRef`	`Util.checkName`
	`RightClickMenu`	`Util.createNeutralId`
		`Util.delimToStringArray`
		`Util.openExternalViewer`
		`[Util.openExternalViewer.child]`
		`[Util.openExternalViewer.cmd]`
		`Util.pr`
		`Util.replaceSpecialCharacters`
$ET_{weighted}$	`Util`	`Util.openExternalViewer`
		`[Util.openExternalViewer.child]`
		`[Util.openExternalViewer.cmd]`
LSI	`ExternalTab`	`GUIGlobals.isStandardField`
	`BrowserLauncher`	`GUIGlobals.isWriteableField`
	`IntegrityCheck`	`JabRefFrame.getTabIndex`
	`ImportMenuItem`	`ImportFormatReader.import_File`
	`Util`	`GeneralTab.readyToClose`
VSM	`JabRefFrame`	`FieldContentParser.format`
	`ImportInspectionDialog`	`EntryTable.TableClickListener.mousePressed`
	`Globals`	`LatexFieldFormatter.format`
	`KeyBindingsDialog`	`JabRefFrame.quit`
	`ImportUnknownMenuItem`	`CiteSeerFetcherPanel.actionPerformed`

Variables are shown in square brackets

variable names which is out of scope of the current paper. This is why they are shown in square brackets in Table 5. They are not included in the precision and recall calculations.

7.2 Bug Task Results

For each task, we calculated the IR performance score by averaging the SCE relevance scores for the SCE's predicted by IR. Gaze-link algorithm performance scores were calculated in the same way. Refer to Table 6 for developers' accuracy in the study tasks and the averaged scores that the gaze link algorithm generated for each task. One observation is that even though the developers did not complete the task for Bug ID: 1436014, the gaze link algorithm reported a high score of 2.58. In order to interpret this number we decided that it was necessary to determine the usefulness of the SCEs generated by the gaze algorithm. The gaze-link algorithm had a higher (better) score for 6 of the 8 tasks, and IR had a higher score for the other two. For the two bugs on which our algorithm performed worse, none of the developers working on that bug had found a correct solution. This observation makes sense because if they did not find a solution, there is a high chance that they did not look at the relevant methods, classes or variables.

Another interesting observation is bug number 2 (bug id: 1436014). Even though neither of the developers solved this bug correctly in the data collection phase, the gaze-link algorithm gives a higher score to the SCEs that make up their results. This indicates that the developer did look at the correct places in the code but did not state out his/her answer for that particular bug. This type of behavior has many possible causes and cannot be found without fine grained line-level tracking of the user's eyes. This is how our approach to deriving traceability links is unique and can add additional insight to existing approaches.

7.3 Evaluation Phase Results - Precision and Recall Context

The results of the evaluation phase for the *precision and recall context* is given in Table 7. We find that the average precision for the gaze-link approach is 55 % with recall averaging at 67 %. This is higher than both LSI and VSM at the top 4 and top 10 level. In the gaze-link approach the recall was 100 % for five of the tasks. The precision and recall were

Table 6 Results for individual bugs

ID	Bug ID	Difficulty Rating	Scope of Solution	Solved by at least 1 developer	IR mean score	Gaze-link mean score
1	1548875	moderate	single method	no	2.50	1.91
2	1436014	difficult	multiple classes	no	2.28	2.58
3	1594123	easy	single method	yes	2.56	4.56
4	1489454	easy	single method	yes	2.16	3.75
5	1594169	difficult	multiple classes	no	3.07	2.43
6	1540646	moderate	single method	yes	2.25	2.96
7	1631548	easy	single method	yes	2.50	2.75
8	1542552	difficult	multiple methods	yes	2.72	3.69

All SCE scores for each task were averaged to create an IR mean score and a gaze-link mean score. A higher number indicates more relevance

Table 7 Precision and Recall for Gaze-link, LSI and VSM (top 4 and top 10) with respect to the commits done by JabRef developers (oracle)

		LSI top 10			VSM top 10			LSI top 4			VSM top 4			Gaze-Link algorithm			
ID	O	M	P	R	M	P	R	M	P	R	M	P	R	C	M	P	R
1	3	1	10 %	33 %	1	10 %	33 %	0	0 %	0 %	1	25 %	33 %	5	1	20 %	33 %
2	1	1	10 %	100 %	0	0 %	0 %	0	0 %	0 %	0	0 %	0 %	1	1	100 %	100 %
3	2	1	10 %	50 %	0	0 %	0 %	1	25 %	50 %	0	0 %	0 %	3	2	67 %	100 %
4	2	2	20 %	100 %	2	20 %	100 %	2	50 %	100 %	2	50 %	100 %	2	2	100 %	100 %
5	1	1	10 %	100 %	1	10 %	100 %	1	25 %	100 %	1	25 %	100 %	4	0	0 %	0 %
6	1	0	0 %	0 %	1	10 %	100 %	0	0 %	0 %	1	25 %	100 %	1	1	100 %	100 %
7	2	1	10 %	50 %	1	10 %	50 %	1	25 %	50 %	1	25 %	50 %	2	0	0 %	0 %
8	1	0	0 %	0 %	1	10 %	100 %	0	0 %	0 %	1	25 %	100 %	2	1	50 %	100 %
	Avg		9 %	54 %		9 %	60 %		16 %	38 %		22 %	60 %			55 %	67 %

O refers to the count of items in the oracle. M refers to the number of matches with the oracle, P refers to precision, R refers to recall, C refers to the count (number) of links generated by gaze-link

both 100 % for three of the tasks. For the other two tasks that had 100 % recall, the precision was 50 % or higher. The *count* column indicates how many links the eye tracking algorithm returned. Since the links it returns depends on what developers look at and the gaze-link algorithm, we cannot increase this list as can be done with IR methods.

Comparing LSI and VSM, we can see that VSM had higher recall compared to LSI. We do see the recall for LSI increase from 30 % from the top 4 list to 54 % in the top 10 list indicating that relevant entities were lower down in the ranked list when compared to VSM. Depending on what the developer looks at, the links generated from the gaze data could be less than 4 (which they are in six of the eight cases as seen in Table 7). These results are very encouraging and show that it is indeed feasible to achieve continuous traceability while a developer is working on bug fixing tasks.

7.4 Evaluation Phase Results - Usefulness Context

The resulting list of SCEs from the data collection phase along with the top four SCEs for each bug from VSM analysis were independently evaluated by a second group of developers. We chose VSM results because they were better than LSI. In the usefulness ranking part of the evaluation phase, the total number of links in VSM was four for each task and between one and six for eye gaze. There were cases where the counts were lower than four for eye gaze because of how many things the developer looked at and how the gaze-link algorithm works.

In the evaluation phase, SCEs were rated on their usefulness in helping to locate and understand the bug. For each SCE, we had ratings from seven to eight developers. The ratings were averaged to give an SCE usefulness score for both IR and gaze-link results.

SCE scores are generated when the link generation methods (VSM and gaze-link) are run after the data collection phase. SCE rankings are generated by developers in the evaluation phase of the study. We wanted to determine if there was some correlation between the SCE scores and the SCE rankings. See Fig. 6 for a scatter plot of these two sets of data. The x-axis shows the ranking from the prediction method (IR or gaze) and the y-axis average usefulness of that SCE as ranked by the developers. We would expect to see a downward

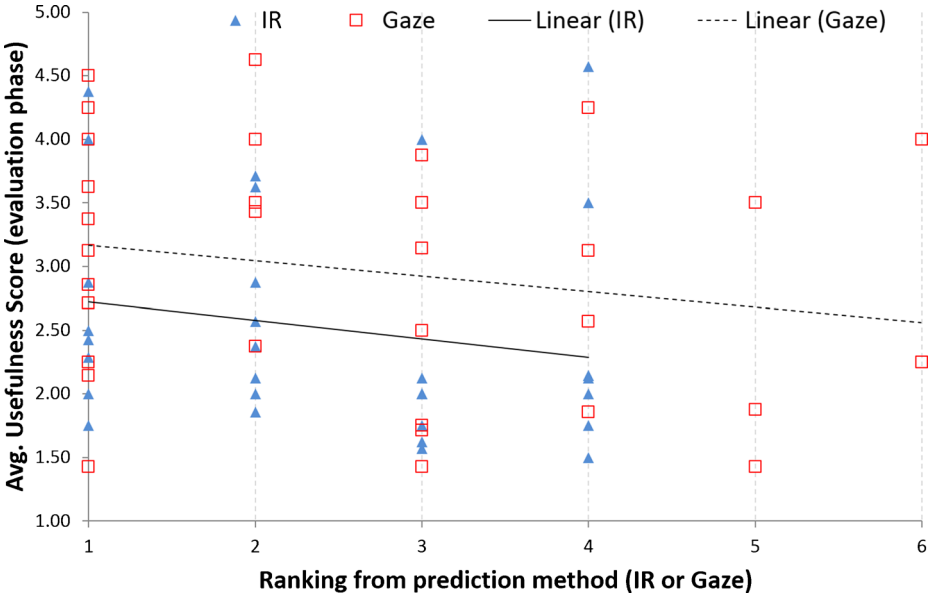


Fig. 6 Comparing usefulness scores and rankings. The y-axis represents the average usefulness score for the SCE's as ranked in the evaluation phase. The x-axis represents the ranking order from the method used to recover traceability links. Lines show least squares fit. Actual SCEs are not shown on the scatter plot. Refer to companion website for SCE information. If the usefulness of the ranking prediction goes down as the rank increases, we get a downward slope

slope. If the rank is low and usefulness is high, this means that the initial links (links found first in the list) were ranked very useful to the developers. If the rank is high and usefulness is low, it means that the found links were much lower in the list and the developers did not find them very useful. The least squares fit lines in the figure show that we do in fact see a downward trend for both VSM and gaze-link analysis techniques. Furthermore, the slopes of the lines are similar for both techniques. We can conclude that both techniques perform similarly in correctly ranking the results they return.

Both VSM and the gaze-link algorithm provide a ranked list of results (SCEs) from most relevant to least relevant denoted by the SCE score.

In all of the cases, this second set of developers rated the usefulness of the gaze algorithm output as very useful. This indicates that the number generated (2.58, see Table 6) by the gaze link algorithm in case of Bug ID: 1436014 did indeed include SCEs that will help the developer to eventually fix the bug. This indicates that the developer did look at the correct places in the code but did not state out his/her answer for that particular bug. This type of behavior has many possible causes and cannot be found without fine-grained line-level tracking of the user's eyes. This is where our approach is unique and can add additional insight to existing approaches. For the two bugs on which our algorithm performed the worst (Bugs 5 and 7), none of the developers working on those bugs had found a correct solution. This observation puts forth the proposition that if they did not find a solution, there is a high chance that they did not look at all possible relevant methods, classes or variables.

We compare scores over all the bug tasks of the best IR method (VSM) with the gaze-link method. The first row in Table 8 shows rankings using SCEs from all eight tasks. The gaze algorithm SCEs had an average ranking of 2.97 out of 5, and the VSM SCEs had an

Table 8 Overall resulting rankings from the evaluation phase

	SCE's from IR method		SCE's from gaze link algorithm		SCE Comparison			
	Mean	Median	Mean	Mean Median	Diff between averages (Gaze - IR)	Diff between medians (Gaze - IR)	Wilcoxon Rank Sum; p-value for H0: Gaze <= IR	95 % Significance
All bugs	2.51	2.13	2.97	3.13	0.46	0.99	0.032	YES
Bugs with at least 1 relevant SCE identified by at least 1 developer	2.44	2.13	3.48	3.50	1.04	1.38	< 0.001	YES
Bugs with no relevant SCE's identified by any developer	2.62	2.26	2.31	2.20	−0.31	−0.06	0.795	NO
All bugs as ranked by novices	2.47	2.33	3.05	3.00	0.58	0.67	0.019	YES
All bugs as ranked by non-novices	2.46	2.33	2.81	2.67	0.35	0.33	0.086	NO
Bugs that can be fixed in a single method	2.51	2.13	2.89	2.86	0.38	0.72	0.197	NO
Bugs that require multiple method or class changes to fix	2.50	2.25	3.06	3.13	0.56	0.88	0.045	YES

Gaze-link SCE ranks were significantly higher than those from VSM with 95 % confidence. The scale is from 1 to 5 with 5 being the most useful

average ranking of 2.51 out of 5. To test for statistical significance, the Wilcoxon Rank Sum test was used in place of the t-test because the data are not normally distributed. We rejected a null hypothesis that the gaze algorithm results are less than or equal to the IR results with greater than 95 % confidence in four of the bug categories listed.

In three out of the eight tasks, no developer was able to identify a relevant source code entity (i.e., they were unable to solve the bug-localization task correctly). For this subset of tasks, IR had a higher score than our gaze algorithm. Finally, we also looked at only the subset of tasks for which at least one developer found at least one relevant SCE, and found that the gaze results were significantly better. Another interesting note is that the bugs that required multiple method or class changes were handled much better by the gaze-link method of traceability link retrieval.

Next, we also split our participants into novices and non-novices in the evaluation phase based on their programming background. There were three novices (1-2 years experience in Java) and three non-novices (more than 5 years experience in Java). It is interesting to see that more novice developers ranked the gaze algorithm SCEs higher compared to non-novice developers. For VSM, the usefulness scores were unchanged, with a score of 2.47 for novices and 2.46 for non-novices. For the gaze algorithm, however, the novice rankings were higher with a average score of 3.05 verus 2.81 when ranked by non-novices. Further investigation is needed as to why this is the case. We plan to see if this trend holds in current studies we are conducting.

While we do not a definitive answer as to why novices might like the gaze algorithm results better that non-novices, one possible explanation is the specificity of the results. While VSM always returns class or method-level results, the gaze algorithm returns any SCE, including classes, methods and variables. Because of this, the gaze algorithm results are often much more specific. Perhaps non-novice developers have an easier time navigating through the code in a large class and are therefore fairly satisfied with a prediction that gets them to the correct class. Novices might feel overwhelmed having to look through too many lines of code. For that reason, novices may benefit more from the specificity of the gaze algorithm results than non-novices.

7.5 Benchmarking Phase Results

We show the overall precision and recall for all eight bugs at the class level with IR setting capped at top-5 in Fig. 7. We provide all the results for precision and recall for the two benchmarks in Fig. 8. Overall, VSM and LSI both have higher recall with respect to the commit. We also observe that VSM precision and recall with respect to the $ET_{weighted}$ is higher than LSI.

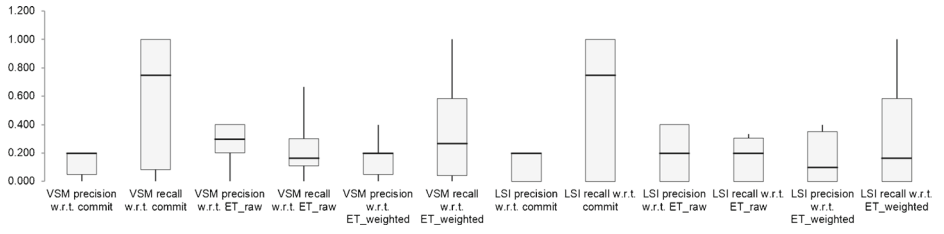


Fig. 7 Descriptive statistics for precision and recall (top-5) of the two IR techniques w.r.t. the two benchmarks at the class level granularity

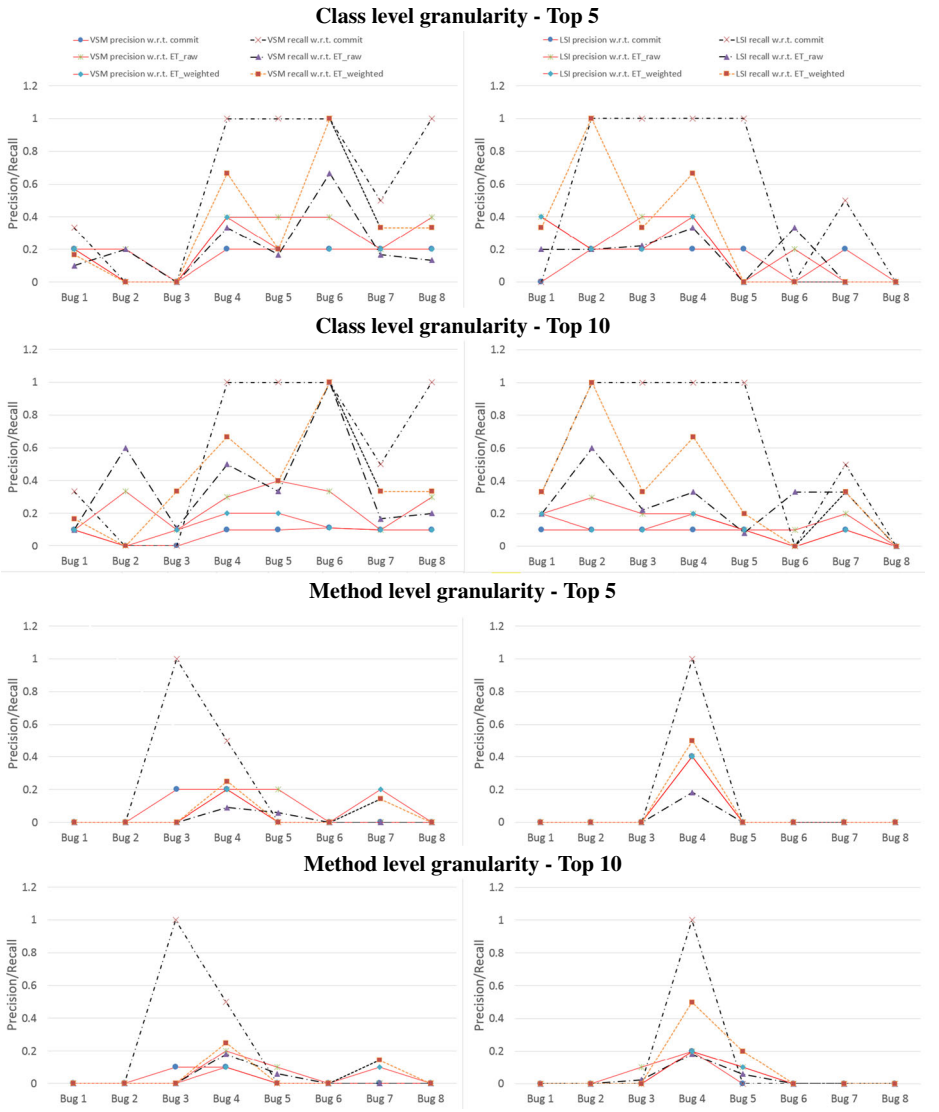


Fig. 8 Precision and recall for each of the eight bugs. The first two rows indicates class level granularity for the top-5 and top-10 sets respectively. The bottom two rows indicates method level granularity for the top-5 and top-10 sets respectively

Consider the first row of graphs indicating class level top-5 category. The VSM and LSI recall w.r.t. commits got the highest scores. This was closely followed by the VSM and LSI recall w.r.t. $ET_{weighted}$ and lastly by VSM recall w.r.t. ET_{raw}. The second row of graphs represents the class level top-10 category. A similar trend is observed for the top-10 category with even more support for LSI and VSM recall w.r.t. ET_{raw} for Bug 2. The third row of graphs represents the method level top-5 category. We did not see very good results at the method level. The only two good results here are for Bug 3 (VSM recall w.r.t. commits was 1) and Bug 4 (LSI recall w.r.t. commits was 1). The next best results were for VSM recall

w.r.t. $ET_{weighted}$ followed by ET_{raw}. The final set of graphs are at the method-level top-10 category. Bug 4 was the the only one that performed fairly better than any other bug. Once again VSM recall w.r.t. commits was the highest followed by VSM recall w.r.t. $ET_{weighted}$. LSI worked better at the method top-10 level. In particular the recall w.r.t. commits had the best score followed by the LSI recall w.r.t. $ET_{weighted}$.

Refer to Fig. 9 for a precision and recall plot using all the different combinations of variables for Bug 5. This is another view of the data for one specific bug. Bug 5 was not solved correctly by any of the five developers. In this figure for Bug 5, we observe that LSI top-10 performs much better than the top-5 list. The commit benchmark had the highest recall but low precision. The $ET_{weighted}$ benchmark had a higher recall for VSM than the ET_{raw} at both top-10 and top-5 levels at the class level granularity. At the method level however, the results are flipped, where the ET_{raw} has a higher recall for the VSM technique.

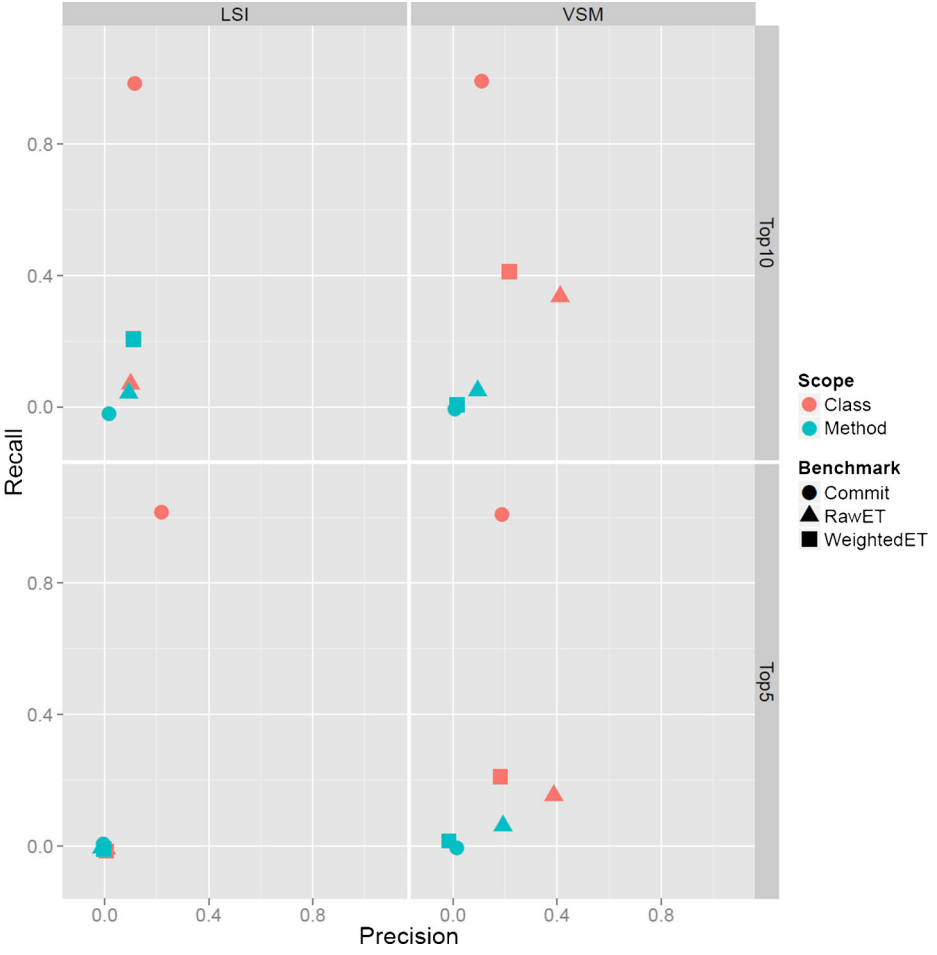


Fig. 9 Bug 5: Precision and recall for LSI and VSM w.r.t. the two benchmarks on top-5 and top-10 at the class and method level. The class scope is shown in red and method scope is shown in blue. The commits are shown as circles, raw eye tracking data as triangles, and weighted eye tracking data as squares. For e.g., a red triangle in the LSI and Top 10 quadrant indicates class scope for the raw eye tracking data set

Establishing a more definitive reasoning behind this flipped behavior would require a study on a much larger sample, which we plan for future work.

Hypothesis H_2 stated that there is no difference in precision and recall between the IR methods, VSM and LSI, when calculated with respect to the two benchmarks based on commits and eye tracking. We can clearly reject this hypothesis because we see a broad difference and do not spot a unique trend between the different IR categories.

We present in Table 9 the number of SCEs for each bug at the method and class level for both benchmarks and IR methods as well as the number of overlaps between the commit and eye tracking benchmarks. Hypothesis H_1 stated that there is no difference between the the commit and eye tracking benchmarks. Based on the first five rows, we can see that this is clearly not the case. There are a few common SCEs in commits and ET data (both raw and weighted) but for the most part there is very little overlap (Bug 5 and Bug 7 have no overlap and Bug 1 and Bug 3 have no overlap at the method level). This indicates that the eye tracking benchmark is truly different from the commit benchmark. We are therefore able to reject Hypothesis H_1.

We also show the overlap between LSI and VSM for the top-5 and top-10 sets. There is no overlap between LSI and VSM for Bug 2 and Bug 3 and for Bug 7 and Bug 8 at the method level. See Table 9 (last four rows).

8 Discussion

We revisit each of the research questions in the following sections. The first two questions (RQ1 and RQ2) are discussed together followed by a discussion on questions (RQ3 and RQ4) related to investigating eye tracking as another avenue for benchmarking in addition to commits.

8.1 Discussion on RQ1 and RQ2 - IR vs. Gaze-Link Approaches for Traceability Links

Recall that RQ1 seeks to determine if eye gaze can be considered as a viable and feasible source to recover traceability links. Based on the results of the gaze-link algorithm and comparing it with the oracle (the way JabRef developers actually fixed the bugs), we find that the gaze-link algorithm does find the method and variable level links which narrows down the specific place to look for the solution. This indicates that it is a viable and feasible approach to recovering traceability links for true-to-life tasks. We get an average precision of 55 % and recall of 67 % for all tasks. Based on the results, we see that the gaze-link method outperforms both LSI and VSM in terms of precision and recall with respect to the commit oracle.

The second research question (RQ2) was to compare IR methods to the gaze-link method for recovering traceability links. After the comparisons above, we find that the gaze-link approach worked better in 6 out of the 8 tasks and developers found the links generated by the gaze-link algorithm to be significantly more useful than IR links in a majority of the tasks. All of the above indicate that this approach can be used to silently capture traceability links while the developer is at work moving towards the goal of continuous traceability.

When asked about the overall difficulty of the tasks, one developer rated them as easy, two as average and three as difficult. Some of the variation in these answers might be attributable to the fact that developers were not all given the same set of bugs.

Table 9 The number of SCEs generated by commits and eye tracking (ET) both raw and weighted. The intersect between commits and ET is also presented

	Bug 1-C	Bug 1-M	Bug 2-C	Bug 2-M	Bug 3-C	Bug 3-M	Bug 4-C	Bug 4-M	Bug 5-C	Bug 5-M	Bug 6-C	Bug 6-M	Bug 7-C	Bug 7-M	Bug 8-C	Bug 8-M
Commits	3	3	1	1	1	1	1	2	1	1	1	1	2	1	1	1
Et_{raw}	10	61	5	9	9	44	6	11	12	17	3	13	6	31	15	52
$ET_{weighted}$	6	14	1	1	3	3	3	4	5	5	1	4	3	7	3	2
Commits $\cap$ ET_{raw}	2	0	1	1	1	0	1	2	0	0	1	1	0	0	1	1
Commits $\cap$ $ET_{weighted}$	1	0	1	1	1	0	1	2	0	0	1	1	0	0	1	1
$ET_{raw} \cap$ $ET_{weighted}$	6	11	1	1	3	1	3	4	4	3	1	4	3	6	3	2
LSI-Top5 $\cap$ VSM-Top5	1	0	0	0	0	0	3	2	1	1	1	1	1	0	2	0
LSI-Top10 $\cap$ VSM-Top10	3	1	2	0	3	0	4	2	2	1	2	1	2	0	2	0
LSI-Top5 $\cap$ VSM-Top10	1	0	0	0	0	0	3	2	1	1	2	1	1	0	2	0
LSI-Top10 $\cap$ VSM-Top5	2	0	1	0	2	0	4	2	1	1	1	1	2	0	2	0

All numbers represent cardinalities. The M represents method level, C represents class level

Next, we present some discussion on bugs where developers did not find a starting point at which they could possibly start to solve the bug. For three of the bugs, none of the developers that participated in the data collection phase were able to identify a single relevant SCE (based on the oracle i.e., the way the JabRef developer fixed the bug). We were particularly interested in seeing if our gaze analysis for these three bugs produced any relevant SCE's despite the fact that the developers could not list a solution. The average usefulness scores for those three bugs were the lowest out of the eight bugs.

There is clear evidence that our gaze analysis does not perform as well when the developer who is generating the gazes does not understand the bug or know how to solve it. However, the bug with ID 2 see Table 6) , which had a usefulness score of 2.58 (for gaze-link), at least performed better than VSM, which had a usefulness score of 2.28 even though it was not solved by a single developer in the data collection phase. Moreover, our gaze algorithm successfully identified the class FieldContentSelector for this bug. This was not only the SCE that was given the highest usefulness score (by developers in the evaluation phase) for that bug, but was also the primary class that was modified by the JabRef developers when they fixed the bug.

The results from Bug 2 (Bug ID: 1436014) suggest it is possible to extract useful gaze results even from a session where the developer fails to solve the task at hand. This suggests that gaze data could be a useful tool not only for traceability but also for assisting with eventually fixing the bug. After a failed session, if that same developer or another developer were to return to the same task at a later date, they could use the gaze results from the previous session as a starting point.

The following discusses variability in bug results. There is much wider variability in the performance of the gaze algorithm versus VSM. Looking at average usefulness scores (see Table 6) for the individual bugs, for VSM the lowest score was 2.16 and the highest score was 3.07. For the gaze algorithm, the lowest score was 1.91 and the highest score was 4.56. Some of this variability can be attributed to variability in the performance of the developers when the gaze data were collected. As mentioned previously, developers were not always successful in identifying the relevant SCE(s) during the data collection phase, and the scores were negatively impacted when no developer was successful. There could be other causes for this variability as well. Our algorithm might handle certain types of bugs better than others. In Table 8, we divided the bugs into those that could be solved using a single method versus those that required changes to multiple methods or classes. The VSM results were unchanged, with a score of 2.51 for single-methods bugs and 2.50 for multi-method bugs. However, the gaze algorithm had a score of 2.89 for single-method bugs and 3.06 for multi-method bugs. Given the small sample size in this study, it is difficult to say whether this difference is significant, but it at least suggests that our algorithm may work better on bugs that have a broader scope.

8.2 Discussion on RQ3 and RQ4 - Benchmarks based on Eye Gaze and Commits

Research questions RQ3 and RQ4 seek to investigate if eye tracking is a viable benchmark in comparison to commits. To do this, we investigate how LSI and VSM perform when compared to commits vs. eye gaze. The data/oracle for eye gaze was generated as a result of RQ1 using the gaze-link algorithm described in this paper on the same tasks. In other words, we seek to determine how commits and eye gaze perform as oracles when compared to IR technique results. To do this, we calculated precision and recall and also conducted an overlap analysis presented in Section 7.5.

Techniques for feature location are commonly evaluated on benchmarks formed from commits (Zhou et al. 2012; Dit et al. 2013a). Although, commits are a reasonable source, they only capture entities that were eventually changed to fix a bug or resolve a feature. We are interested in investigating another type of benchmark based on eye tracking data, namely links generated from the bug-localization tasks given to the developers in the data collection phase. Eye tracking data provide insights into activities that go beyond entities that were changed. To establish a benchmark, we analyzed eye-gaze information for the same five developers from the data collection phase of the study who were required to perform bug-localization tasks on the open source subject system JabRef. Latent Semantic Indexing (LSI) and Vector Space Model (VSM) are commonly used for bug localization/feature location. The source code entities relevant to subjected bugs recommended from each of these IR techniques were evaluated on both commit and eye-tracking benchmarks. An overlap analysis of the IR methods on commits and eye tracking data show that these two benchmarks are indeed different with only a small overlap which is expected. The gaze algorithm presented here is crucial to weeding out irrelevant and stray glances. Results from the gaze-link algorithm are more specific than the rankings from current IR methods. The gaze algorithm also exhibited the notable property of finding useful SCEs (based on an informal survey on another set of developers) given unsuccessful attempts to fix a bug by developers. This indicates that a developer might have found a starting point but not the exact solution yet. With commits, the IR techniques can only be evaluated if the solution is committed. The eye tracking benchmark can be used even if the solution is incomplete. The transparency and minimal effort required by developers makes gaze tracking as a benchmark an attractive possibility. It can certainly complement commits and in the process provide more information about how a developer fixes a bug. Our preliminary analysis shows that eye tracking also provides an opportunity to empirically assess the results from different IR tools.

9 Threats to Validity

This section describes the threats to validity to our experiment and measures taken to minimize these threats.

9.1 Internal Validity

Internal Validity deals with the extent to which we can establish a causal relationship between the variables we are testing and the results obtained. As mentioned earlier, we modified the algorithm since the previous experiment by Walters et al. (2014) was published. Because of this, we had to set new parameter values in our algorithm. These values were set based on trial and error of what values gave the best results, and there is a possible risk of overfitting. It is not clear that the same algorithm using the same parameters would be as successful on a different set of bug reports or a different software system. In order to test for overfitting, we ran our current algorithm on the gaze data generated in our previous experiment (Walters et al. 2014) and compared the results against those from the previous algorithm. Indeed, we did find that the new algorithm performed worse on the previous gaze data. It should be noted that this difference may be caused in part by the different nature of the two experiments. In the previous experiment, developers were given a specific starting point in a jsp file when searching for the bug. The “relevant” SCEs were the ones along the logical path between the starting point and the bug. In this experiment, developers could start wherever they chose. This often meant that the first part of the session was spent

looking through completely irrelevant code. Because of this, our new algorithm is heavily biased towards code that is examined later in the session. It is likely that our new algorithm punished some relevant SCEs in the previous experiment because they were encountered early in the session.

In terms of the history threat to internal validity, we do not use the same developers in the data collection and evaluation phases. In the evaluation phase, all of the eight participants ranked all the bug tasks for both VSM and *iTrace*. The VSM and *iTrace* links were randomized and they did not know where the links came from. Each bug came from a different part of the system and this helped to minimize learning effects. In terms of the selection threat to internal validity, the developers all had Java experience. We did a separate analysis on the novices and non-novices to determine if any effects exist in order to overcome this threat. Another possible threat is that we used only eight bug tasks with only two of the tasks being done by all five developers in the data collection phase. The reason for this decision was to keep the study time around 60 minutes to avoid developer fatigue. The grouping of developers to tasks could affect the validity of results. The grouping of developers to tasks was done randomly with the goal of trying to maintain a balance of easy and difficult tasks. For the usefulness ranking, we use VSM (best IR method in terms of precision and recall) and eye gaze links randomly in a blind ranking survey. According to Gethers et al. (2011) where they compare IR methods for traceability, it is shown that IR methods do vary in results. We chose VSM as it performed the best when compared to LSI. We did not want to give the developers to rank too many links due to fatigue issues.

9.2 Construct Validity

In reduce the mono-operation bias threat, we use eight different bugs in three different levels of difficulty. In addition, each bug was part of a different section of the system. To reduce the social threat of hypothesis guessing, we did not make our developers aware of our hypotheses or our research questions. They were simply told to try and find the places that the bug needs to be fixed.

It should also be noted that tracking of developer gaze is not the only possible method for collecting information about a developer's activity within an IDE. Information from a developer's interaction with the IDE such as the opening of a file or the editing a file could also be collected. This information may also be a useful resource for developing benchmarks and finding traceability links. One example of a program that collects such data is Eclipse Mylyn, formerly known as Mylar (Kersten and Murphy 2005). Mylyn collects data from a developer's interaction with the Eclipse IDE and uses a degree-of-interest (DOI) model to identify the files and code most relevant to the current task. It then provides highlighting and other modifications to the Eclipse display that makes it easier for the user to find relevant code. While Mylyn is designed more for productivity than for development of traceability links, their DOI model is essentially a set of predictions about which SCE's are relevant to a particular task. It is easy to imagine that the data Mylyn collects might also be useful for determining traceability links or creating a benchmark.

Since data can already be collected from a developer's actions within an IDE, one might question whether the use of an eye-tracker is necessary for determining traceability links or benchmarks. In this paper, we do not attempt to prove the superiority of eye-tracking data to other developer activity data for determining traceability links. However, we can say that some of information collected from an eye-tracker does not overlap with other data that can be collected. For example, while it's easy to determine which file is visible in an IDE, without an eye tracker it is not possible to know the exact line a developer is reading. We

have demonstrated that eye-tracking data can be succesfully used to determine traceability links in Walters et al. (2014). Programs such as Mylyn show that data from developer's interactions with an IDE can also have some success toward the same purpose. In fact, we have shown in Kevic et al. (2015) (where we compared eye tracking data to Mylyn data), that eye gaze data is indeed much more richer and definitely different than Mylyn data. The ideal algorithm would likely use both streams of data in order to make use of all information available.

9.3 Conclusion Validity

Conclusion validity asks if there is a relationship between what we are testing and the actual result and if this relationship is reasonable. We use standard precision and recall in two different contexts to determine the validity of our method. The precision and recall context uses current state of the art methods to compare the results of our method and IR methods to an oracle from commits. There are two problems with this. First, different developers have different ways of fixing/adding code when given a feature to implement. Second, just considering commits is again a very boolean oracle as it is possible that only method A and method B was changed but for a developer to actually change method A and B, they had to have investigated method C and D in detail as well. The commit history will not state that method C and D are linked to the feature even though the developer used them quite a bit to get to his final answer. Eye tracking using our approach will. In order to test this, we also wanted to evaluate the usefulness of results from IR methods and from eye gaze methods. To do this, we asked developers to rank the results of the best IR approach (i.e. VSM) and the gaze link approach. The developers did not know which method produced the results. We believe this comparison is fair.

9.4 External Validity

The main issue concerning external validity is the generalization of the results to another set of developers. In this study, we used only one subject system, namely JabRef. Hence we cannot claim that these results would work for all systems. To mitigate this threat, we made sure we used an open source system and real bug fixing tasks. More studies need to be done to assess the generalizability of the results to other types of systems (both in terms of size and different domains). Another external threat to validity is caused by using developers that are unfamiliar with the source code they are debugging. However, in a real-world scenario, debugging tasks are performed by developers who are not as familiar with the source code and the software application as the original developer. It could be that the original developers approach debugging tasks differently. All developers were made aware of the system and source code at the beginning of the study. Even though they were sent a small description before the study began, none of the developers actually read it which makes this threat non-existent. To avoid experimenter effects, the moderator did not interact with the developers while they were solving the tasks.

10 Related Work

We focus our related work on IR methods used in feature location and bug localization including traceability of features from requirements to code. Next, we present work on the use of eye tracking as a means to enhance existing software tasks and tools.

10.1 Feature Location and Bug Localization

A thorough literature review on feature location is given by Dit et al. (2013b). They also present a taxonomy to classify the literature into nine key dimensions. Such a taxonomy is helpful in comparing existing feature location techniques including the benchmark we propose in this paper. Binkley and Lawrie also provide a survey of how IR techniques have been used in feature location and impact analysis (Binkley and Lawrie 2011). Xing et al. (2013) call for more widespread use of benchmarks and provide a large scale Linux-based benchmark for feature location. Next, we present selected work in feature location (not an exhaustive list). Alhindawi et al. (2013) introduce a new method based on stereotypes to improve feature location. They compare their approach using LSI both with and without stereotypes showing an improved performance using stereotypes.

Hill et al. (2013) investigate which feature location technique is the most effective. They state how existing evaluations use the location of the bug fix (commit) but not at what code needs to be understood to fix the bug. This is precisely what our paper addresses with the eye tracking benchmark. It adds a new dimension into validating feature location techniques. Dit et al. (2012) provide a mechanism in Tracelab (Keenan et al. 2012) to compare and contrast feature location techniques. Our algorithm is compatible with Tracelab's architecture and would be ideal as a source for more comparative studies in feature location. They also provide several other benchmarks (Dit et al. 2013).

Wang et al. found distinct phases, recurring patterns, and elementary actions in the way developers performed six feature location tasks (Jinshui Wang Xin Peng and Zhao 2011). Our study is different in that it uses eye tracking to determine the precise location where the developer is looking while solving a feature location task and we use it to drive IR benchmarks. Later on, Wang et al. also propose a multi-faceted interactive program exploration approach to concept location (Wang et al. 2013). Their new approach allows developers to interact and filter the results, increasing their performance.

Scanniello et al. use both textual information in source code and structural dependencies between SCEs to help with feature location (Scanniello and Marcus 2011). They show that their approach outperforms the baseline. Our method would help such studies to further evaluate their methods on a different type of data set i.e., developers eye gaze. Scanniello et al.'s most recent work makes use of dependency information using a link analysis algorithm in order to improve feature location techniques (Scanniello et al. 2015). Their algorithm can be considered analogous to our gaze algorithm that finds all related SCEs in gaze data. They also evaluate bug descriptions for feature location as we do.

Several researchers have evaluated their bug localization techniques on commits from source code repositories (Rao and Kak 2011; Dit et al. 2013b; Zhou et al. 2012; Lukins et al. 2010). These approaches have not looked at *how* the bugs get fixed just that certain files were committed and hence relevant to the fix. Feature location is an important pre-requisite to fixing bugs. A developer is typically faced with fixing bugs and he/she needs to be able to find the relevant feature that the bug might be related to. The eye tracking benchmarks provide a richer dataset than just commits because they record the process used to get to the bug fix (important for feature location). Since commits do not store the *how* aspect of bug localization. Bug localization techniques (Rao and Kak 2011; Lukins et al. 2010; Zhou et al. 2012) rely mainly on commits to evaluate their results (Dit et al. 2013b). Commits, however, only capture relevant entities that were eventually changed after a new feature was added or a bug was fixed (Hill et al. 2013). They state how existing evaluations use the location of the bug fix (commit) but not at what code needs to be understood to fix the bug. This is precisely what our paper addresses with the eye tracking benchmark.

10.2 Retrieving Traceability Links

Marcus and Maletic (2003) were the first researchers to use Latent Semantic Indexing (LSI) to recover source code to documentation traceability links. They replicated the study presented by Antoniol et al. (2002) that used the probabilistic and vector space models to trace source code onto documentation. The results show that there is no difference between the two methods.

Gethers et al. (2011) combine the VSM, probabilistic JS model, and relational topic modeling (RTM) to recover traceability links. Their results indicate that the integrated method outperforms a single IR method with a statistically significant margin. They also look at the orthogonality of different IR methods.

DeLucia et al. present ADAMS, an advanced artifact management system (De Lucia et al. 2006). An empirical study is presented by Oliveto et al. (2010) to determine the equivalence of different traceability link recovery techniques based on IR methods show Jensen-Shannon (JS), VSM, and LSI to be equivalent with LDA, which although less accurate, is able to capture a unique dimension.

Asuncion et al. (2010) use a machine learning probabilistic topic modeling approach to record traceability links between artifacts. The operations performed by the developers are monitored to identify a list of potentially related artifacts that are used to extract a set of topics, which in turn lead to relationships between requirements and design documents.

Bavota et al. (2013) conduct a study to determine if the size and vocabulary of the repository affects the results of IR techniques such as VSM and LSI.

Other methods based on clustering support for concept location (Scanniello and Marcus 2011; Scanniello et al. 2015), heuristics (Capobianco et al. 2009) as well as alternative IR methods (Qusef et al. 2010) are proposed to improve link accuracy.

10.3 Eye Tracking and Software Traceability

A comprehensive survey on the use of eye tracking in software engineering is given in Sharafi et al. (2015). We first introduced the notion of using eye tracking for software traceability tasks in Sharif and Kagdi (2011) and further refined our *iTrace* framework in Walters et al. (2013). We then conducted a pilot study (Walters et al. 2014) to determine if eye tracking was a feasible approach to derive traceability links. The system we used was *iTrust* where we injected 1-line bugs to test our approach. We also introduced the first version of the gaze-link algorithm in Walters et al. (2014). In this paper, we take the work done in Walters et al. (2014) one step further by determining how the gaze-link algorithm compares to state-of-the-art IR approaches such as LSI and VSM.

The most related work to using eye tracking in software traceability is Ali et al. (2012) albeit used in a different context. Ali et al. used eye tracking on small snippets of code to determine what a developer is looking at most. Since they find that developers look more at method names they use this information to propose a new weighting scheme for LSI. They found that developers tend to look more at method names than class names. They also ask developers to rank source code entities (class, method, comments, variable names). Based on this finding, they propose two weighting schemes for LSI (IR-based method) to retrieve traceability links from the most frequently used data sets: iTrust (v. 10) and Pooka (v. 2). In contrast, our work presented here relies on eye tracking data of developer sessions to derive links from tasks to code. Ali et al. (2012) do not use eye tracking data to directly derive links nor do they use eye tracking on the source code of subject systems. The code snippets they use were unrelated to the system that was used to derive link. Also, their ranking is on which

SCE is preferred, our ranking in this paper is on validating the usefulness of traceability links generated from our algorithm and VSM. We are not aware of any study that compares IR methods with eye tracking methods to derive traceability links.

Rodeghero et al. (2014) look at the degree at which programmers look at keywords, control structures, method signatures, and method invocations. Similar to Ali et al. (2012), they then use this information to adapt the weights that VSM tf/idf technique extracts for a summarization task. The task they focus on is to summarize small Java code snippets and is not applied to software traceability. Fritz et al. (2014) use psycho-physiological measures including eye tracking to predict, using machine learning algorithms, software task difficulty while the developer is at work.

We are not aware of any work that has used eye tracking as a source of developing a new benchmark. The task we focus on is bug localization. The contributions of this paper are to show that eye tracking is indeed a viable source of traceability links that can also be used as a benchmark (in addition to commits) for comparing IR approaches in software engineering. In addition, a comparative analysis is presented to show that with commits we have a lower bound but with eye tracking we get a much broader perspective on how developers solve the task. This rich set of data provides for a new paradigm in benchmarking. We call on other researchers to conduct followup experiments to build community support as this is crucial for getting a benchmark accepted.

11 Conclusions and Future Work

This paper presents a comparative study between traceability links generated from IR methods and developer eye gazes. The eye gaze approach outperforms standard LSI and VSM approaches and reports a 55 % precision and 67 % recall on average for all tasks when compared to how the developers actually fixed the bug. The eye gaze method also proves to outperform the IR methods in a majority (six out of eight) of the tasks when we look at how useful a different set of developers find the links from both approaches. This indicates that our approach is feasible and works reasonably well in an open source project and on real bug tasks. It also indicates that traceability under the hood as developers work is possible.

As the cost of eye tracking hardware continues to fall, widespread adoption of gaze tracking techniques may become feasible. The transparency and minimal effort required by developers compared to other traceability methods make gaze tracking an attractive possibility and also targets the important challenge in traceability - making it effortless. The SCE's scores were higher for the gaze-link algorithm for a variety of categories with the exception of bugs which the developers were not able to solve. While our scores were higher, some of the categories did not meet the 95 % confidence threshold. The most important category (all bugs) did cross that threshold, however.

The gaze algorithm presented here is crucial to pruning out irrelevant and stray glances. Results from the gaze tracking algorithm are also more specific than the rankings from current IR methods. The gaze algorithm also exhibited the notable property of finding useful links given unsuccessful attempts to find a starting point towards solving the bug. This indicates that a developer might have found a starting point but not the exact solution yet. With commits, the IR techniques can only be evaluated if the solution is committed. With eye tracking, we do not have this limitation. The eye tracking benchmark can be used even if the solution is incomplete.

Future work will be conducted to replicating this study on a larger sample, both of participants and of source code and number of features analyzed. Another direction is to

develop specialized learning algorithms to learn as the developer is solving a task and tailor the weights accordingly on the fly. This method is very developer-centric and will greatly enhance the validity of using eye gaze as a benchmark. A developer may start and stop the maintenance task in between. If and when this happens, the developer has the option of using the trace data collected with eye gaze to avoid doing the same things over again. We leave this scenario to be tested as a future study that mimics task interruptions.

We also plan to conduct a second phase of this study where we give a different set of developers the same bugs and the gaze-link algorithm's generated links. Then, we can test to see if those links were actually useful to those developers vs. those that were not given the links. Another study in which two groups perform a feature location task manually where only one group's eyes are tracked and not the others could also help in increasing support for using eye tracking as part of continuous traceability. Another direction of inquiry is the usefulness of gaze tracking for novice vs. experienced developers. The rankings obtained in this study show that novices find the results of this algorithm more useful than non-novices. This suggests a possible application in training or education.

Another feature in *iTrace* would be to compare eye-tracking data of different developers and build a confidence-based link recovery model. For example, if the same pattern is found in the eye gaze data of several developers, it is likely to be a link. We will also be able to infer link directionality based on the gaze data while developers navigate through software artifacts. We can also factor in developer expertise to inform the link algorithms.

To conclude, *iTrace* is designed to provide a novel platform that would directly support two key software traceability tasks: traceability link generation/recovery, and traceability link maintenance and evolution (continuous traceability). Additionally, it will provide a new methodology for researchers to conduct empirical studies in the software traceability domain.

References

Alhindawi N, Dragan N, Collard M, Maletic J (2013) Improving feature location by enhancing source code with stereotypes. In: 29Th IEEE international conference on software maintenance, ICSM 2013

Ali N, Sharafi Z, Guéhéneuc Y. G., Antoniol G (2012) An empirical study on requirements traceability using eye-tracking. In: ICSM, pp 191 – 200

Antoniol G, Canfora G, Casazza G, De Lucia A, Merlo E (2002) Recovering traceability links between code and documentation. IEEE Trans Softw Eng 28(10):970–983

Asuncion H, Asuncion A, Taylor RN (2010) Software traceability with topic modeling. In: 32Nd ACM/IEEE international conference on software engineering, vol. 1, pp. 95–104. ACM

Asuncion H, Francois F, Taylor RN (2007) An end-to-end industrial software traceability tool. In: 6Th ESEC/FSE, pp 115–124

Bavota G, Lucia AD, Oliveto R, Panichella A, Ricci F, Tortora G (2013) The role of artefact corpus in lsi-based traceability recovery. 2013 7th International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE) 0, 83–89. doi:10.1109/TEFSE.2013.6620160

Binkley D, Lawrie D (2011) Information retrieval applications in software maintenance and evolution. In: Encyclopedia of software engineering

Capobianco G, De Lucia A, Oliveto R, Panichella A, Panichella S (2009) On the role of nouns in ir-based traceability recovery. In: 17Th IEEE international conference on program comprehension, pp 148–157

Cleland-Huang J, Czauderna A, Dekhtyar A, Gotel O, Hayes JH, Keenan E, Leach G, Maletic JI, Poshyvanyk D, Shin Y, Zisman A, Antoniol G, Berenbach B, Egyed A, Maeder P (2011) Grand challenges, benchmarks, and tracelab: developing infrastructure for the software traceability community. In: 6th TEFSE, pp 17–23. Panel

Cleland-Huang J, Gotel O, Zisman A (2012) Software and systems traceability Spring-Verlag

Coest.org (2007) Center of excellence for software traceability. http://www.coest.org/index.php/grand-challenges9

Davis APH (1993) Software requirements: Objects, Functions and States. Prentice Hall
De Lucia A, Fasano F, Oliveto R, Tortora G (2006) Can information retrieval techniques effectively support traceability link recovery? In: 14Th IEEE international conference on program comprehension (ICPC'06), pp 307–316
De Lucia A, Fasano F, Oliveto R, Tortora G (2007) Recovering traceability links in software artefact management systems using information retrieval methods. ACM (TOSEM) 16(4):13
De Lucia A, Oliveto R, Tortora G (2009) Assessing ir-based traceability recovery tools through controlled experiments. Empirical Softw Engg 14(1):57–92. doi:10.1007/s10664-008-9090-8
Deerwester S, Dumais ST, Furnas GW, Landauer TK, Harshman R (1990) Indexing by latent semantic analysis. J Am Soc Inf Sci 41(6):391–407
Dit B, Holtzhauer A, Poshyvanyk D, Kagdi H (2013) A dataset from change history to support evaluation of software maintenance tasks. In: Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13, pp. 131–134. IEEE Press, Piscataway, NJ, USA. http://dl.acm.org/citation.cfm?id=2487085.2487114
Dit B, Moritz E, Poshyvanyk D (2012) A tracelab-based solution for creating, conducting, and sharing feature location experiments. In: Program Comprehension (ICPC), 2012 IEEE 20th International Conference on, pp 203–208. doi:10.1109/ICPC.2012.6240489
Dit B., Revelle M., Gethers M., Poshyvanyk D. (2013) Feature location in source code: a taxonomy and survey. J Softw: Evol Process 25(1):53–95. doi:10.1002/smr.567
Dit B, Revelle M, Gethers M, Poshyvanyk D (2013) Feature location in source code: a taxonomy and survey. J Soft Maint Evo.: Res Pract 25(1):53–95
Duchowski AT (2003) Eye tracking methodology: Theory and practice. Springer-Verlag, London
Fritz T, Begel A, Müller S. C., Yigit-Elliott S, Züger M. (2014) Using psycho-physiological measures to assess task difficulty in software development. In: Proceedings of the 36th international conference on software engineering, ICSE 2014. ACM, New York, pp 402–413
Gethers M, Oliveto R, Poshyvanyk D, De Lucia A (2011) On integrating orthogonal information retrieval methods to improve traceability link recovery. In: 27Th IEEE international conference on software maintenance (ICSM'11), pp 133–142
Gotel O, Cleland-Huang J, Hayes JH, Zisman A, Egyed A, Grünbacher P., Antoniol G (2012) The quest for ubiquity: A roadmap for software and systems traceability research. In: 2012 20th IEEE International Requirements Engineering Conference (RE), pp 71–80. doi:10.1109/RE.2012.6345841
Hill E, Bacchelli A, Binkley D, Dit B, Lawrie D, Oliveto R (2013) Which feature location technique is better? In: Proceedings of the 2013 IEEE international conference on software maintenance, ICSM'13. IEEE Computer Society, Washington, pp 408–411
Jinshui Wang Xin Peng ZX, Zhao W (2011) An exploratory study of feature location process: Distinct phases, recurring patterns, and elementary actions. In: Proceedings of the 27th IEEE international conference on software maintenance, ICSM'11, pp 213–222
Keenan E, Czauderna A, Leach G, Cleland-Huang J, Shin Y (2012) Tracelab: An experimental workbench for equipping researchers to innovate, synthesize, and comparatively evaluate traceability solutions. In: ICSE, p 4
Kersten M, Murphy G (2005) Mylar: a degree-of-interest model for ides. In: 4th international conference on aspect-oriented software development, pp 159–168
Kevic K, Walters B, Shaffer T, Sharif B, Fritz T, Shepherd DC (2015) Tracing software developers eyes and interactions for change tasks Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering
Lin J, Lin CC, Cleland-Huang J, Settimi R, Amaya J, Bedford G, Berenbach B, Khadra OB, Duan C, Zou X (2006) Poirot: A distributed tool supporting enterprise-wide automated traceability. 2014 IEEE 22nd International Requirements Engineering Conference (RE) 0, 363–364. doi:10.1109/RE.2006.48
Lucia AD, Oliveto R, Sgueglia P (2006) Incremental approach and user feedbacks: a silver bullet for traceability recovery. In: 2006 22nd IEEE International Conference on Software Maintenance, pp 299–309. doi:10.1109/ICSM.2006.32
Lukins SK, Kraft NA, Etzkorn LH (2010) Bug localization using latent dirichlet allocation. Inf Softw Technol 52(9):972–990
Mader P, Gotel O, Philippow I (2009) Motivation matters in the traceability trenches. In: RE 2009, pp 143–148
Manning C, Raghavan P, Schütze H. (2008) Introduction to information retrieval cambridge university press
Marcus A, Maletic JI (2003) Recovering documentation-to-source-code traceability links using latent semantic indexing. In: 25Th IEEE/ACM international conference on software engineering (ICSE'03), pp 125–137

Oliveto R, Gethers M, Poshyvanyk D, De Lucia A (2010) On the equivalence of information retrieval methods for automated traceability link recovery. In: 18Th IEEE international conference on program comprehension (ICPC), pp 68–71

Qusef A, Oliveto R, De Lucia A (2010) Recovering traceability links between unit tests and classes under test: an improved method. In: IEEE International conference on software maintenance, pp 1–10

Ramesh B, Jarke M (2001) Towards reference models for requirements traceability. IEEE TSE 27(1):58–93

Rao S, Kak A (2011) Retrieval from software libraries for bug localization: a comparative study of generic and composite text models. In: Proceedings of the 8th working conference on mining software repositories, MSR '11. ACM, New York, pp 43–52

Rodeghero P, McMillan C, McBurney PW, Bosch N, D'Mello S (2014) Improving automated source code summarization via an eye-tracking study of programmers. In: ICSE 2014. ACM, New York, pp 390–401

Scanniello G, Marcus A (2011) Clustering support for static concept location in source code. In: Program Comprehension (ICPC), 2011 IEEE 19th International Conference on, pp 1–10. doi:10.1109/ICPC.2011.13

Scanniello G, Marcus A, Pascale D (2015) Link analysis algorithms for static concept location: An empirical assessment. Empir Softw Eng 20(6):1666–1720. doi:10.1007/s10664-014-9327-7

Shaffer T, Wise JL, Walters B, Müller S. C., Falcone M, Sharif B (2015) itrace: Enabling eye tracking on software artifacts within the ide to support software engineering tasks. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, pp. 954–957. ACM, New York, NY, USA. doi:10.1145/2786805.2803188

Sharafi Z, Soh Z, Guéhéneuc Y. G. (2015) A systematic literature review on the usage of eye-tracking in software engineering Elsevier Journal of Information and Software Technology (IST)

Sharif B, Kagdi H (2011) On the use of eye tracking in software traceability. In: 6Th TEFSE, pp 67–70

Walters B, Falcone M, Shibble A, Sharif B (2013) Towards an eye-tracking enabled ide for software traceability tasks. In: 7Th TEFSE, pp 51–54

Walters B, Shaffer T, Sharif B, Kagdi H (2014) Capturing software traceability links from developers' eye gazes. In: Proceedings of the 22Nd International Conference on Program Comprehension, ICPC 2014, pp. 201–204. ACM, New York, NY, USA. doi:10.1145/2597008.2597795

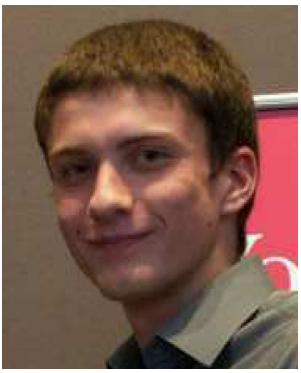
Wang J, Peng X, Xing Z, Zhao W (2013) Improving feature location practice with multi-faceted interactive exploration. In: Proceedings of the 2013 International Conference on Software Engineering, ICSE '13, pp. 762–771. IEEE Press, Piscataway, NJ, USA. http://dl.acm.org/citation.cfm?id=2486788.2486888

Xing Z, Xue Y, Jarzabek S (2013) A large scale linux-kernel based benchmark for feature location research. In: Proceedings of the 2013 International Conference on Software Engineering, ICSE '13, pp. 1311–1314. IEEE Press, Piscataway, NJ, USA. http://dl.acm.org/citation.cfm?id=2486788.2486992

Zhou J, Zhang H, Lo D (2012) Where should the bugs be fixed? - more accurate information retrieval-based bug localization based on bug reports. In: Proceedings of the 34th international conference on software engineering, ICSE '12. IEEE Press, Piscataway, pp 14–24

Bonita Sharif Ph.D. is an Associate Professor in the Department of Computer Science at Youngstown State University, Youngstown, Ohio USA. Prior to this position, she was an adjunct Assistant Professor at Ohio University. She received her Ph.D. in 2010 and MS in 2003 in Computer Science from Kent State University, U.S.A and B.S. in Computer Science from Cyprus College, Nicosia Cyprus. Her research interests are in eye tracking related to software engineering, empirical software engineering, software traceability, and software visualization to support maintenance of large systems. She has authored over 30 refereed publications. She serves on numerous program committees and organizing committees. Sharif is also a recent recipient of the NSF CAREER award on empowering software engineering with eye tracking. She directs the Software Engineering Research and Empirical Studies Lab and the Usability Lab in the Computer Science and Information Systems department at Youngstown State University.

John Meinken is a software application developer at the Center for Health Informatics at the University of Cincinnati, Cincinnati, Ohio USA. He received his M.S. in Computing and Information Systems at Youngstown State University. He received his B.S. in Biochemistry at the University of Cincinnati. His primary focus is designing and building applications that support data analysis and analytics for scientific research.



Timothy Shaffer is a graduate student in the Department of Computer Science and Electrical Engineering at the University of Notre Dame. He received bachelor's degrees in mathematics and chemistry from Youngstown State University in 2015. He is currently working with Notre Dame's Cooperative Computing Lab on harnessing large-scale distributed systems for research in scientific fields such as high energy physics and bioinformatics. His research interests include distributed systems, remote filesystems, and containers/virtualization.

Huzefa Kagdi is an Assistant Professor in the Department of Electrical Engineering and Computer Science at Wichita State University. His research interests are in software engineering with emphasis on software maintenance and evolution, empirical software engineering, program comprehension, and software analytics. He received his Ph.D. in Computer Science from Kent State University, USA.