Chapter 1
Changes, Evolution and Bugs
Recommendation Systems for Issue Management

Markus Borg and Per Runeson

Abstract Changes in evolving software systems are often managed using an issue repository. This repository may contribute to information overload in an organization, but it may also help navigating the software system. Software developers spend much effort on issue triage, a task in which the mere number of issue reports becomes a significant challenge. One specific difficulty is to determine whether a newly submitted issue report is a duplicate of an issue previously reported, if it contains complementary information related to a known issue, or if the issue report addresses something that has not been observed before. However, the large number of issue reports may also be used to help a developer to navigate the software development project to find related software artifacts, required both to understand the issue itself, and to analyze the impact of a possible issue resolution. This chapter presents recommendation systems that use information in issue repositories to support these two challenges, by supporting either duplicate detection of issue reports or navigation of artifacts in evolving software systems.

1.1 Introduction

As software systems evolve, modifications due to discovered defects or new feature requests are inevitable. Typically, projects manage change requests and defect reports in **issue repositories** [21, 42, 49]. In large software engineering projects, the number of **issue reports** reaches several thousands and challenges engineers' abilities to overview the content [4, 23]. Also, distributed development, in terms of both geographical and organizational distances, intensifies the need for efficient management of archived issue reports. Further, issue reports can constitute junctures for

Markus Borg
Dept. of Computer Science, Lund University, Sweden e-mail: markus.borg@cs.lth.se

Per Runeson
Dept. of Computer Science, Lund University, Sweden e-mail: per.runeson@cs.lth.se

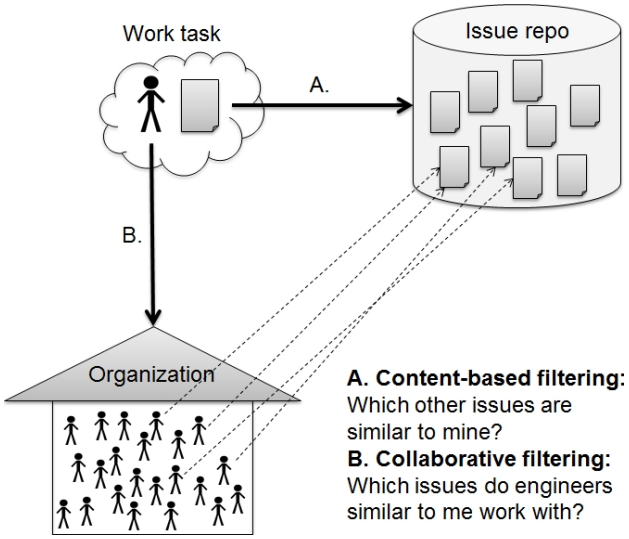

Fig. 1.1: Main principles of an RSSE for issues, content-based and collaborative filtering. The two approaches can also be combined in a hybrid system.

several other software artifacts, with pointers to e.g. requirements, test cases and code components that are involved in the resolution of the issue.

The relationships between issue reports and other software artifacts implies challenges in managing the large amount of information. On the other hand, it also brings opportunities in using the link information to support software developers in their tasks. Networks of software artifacts can be actionable input to a system recommending related information for the task at hand. With proper tool support, archived issue reports can be harnessed to support developers in tasks such as issue triage and change impact analysis.

Issue management in software engineering is similar to task management in general, for example, in a service organization. Issue reports are similar to the baton in a relay race; different actors (e.g., developers, testers, quality assurance, customers) contribute to solving the task, and the **issue management system** is the central node which dispatches subtasks to the actors. Issue reports may originate from several sources, within the development organization or from outside customers or subcontractors. Issues may be pure defect reports, but may also contain change requests and proposals.

An issue repository is typically a database where issue reports (i.e., defect reports and change requests) are stored and maintained over time [21, 42]. The Bugzilla open source issue repository [37] is probably the best known, although several open source and proprietary alternatives exist. Existing issue repositories have features for storing and dispatching issue reports to actors as well as statistics functionality for management reporting. In Chapter **??**, **?**] elaborate further on issue management.

To support the management and resolution of issues in software development, RSSEs have been proposed. Fig. 1.1 shows the two basic approaches to RSSEs, *content-based filtering* and *collaborative filtering*, in the context of issue reports. In an RSSE based on **content-based filtering**, each issue is represented by a set of features. In previous work, issues have typically been represented by textual features, i.e., the terms in their descriptions. Apart from the textual content of the issue reports, issues can be represented by features such as severity, submission date, responsible developer, impacted source code etc. [34]. The RSSE then compares the features of the given issue to all other issues in the issue repository to recommend the most similar issues. Section 1.2 presents several examples of how RSSEs have been used to recommend duplicate issue reports, as well as results from empirical evaluations.

Collaborative filtering on the other hand, relies on a crowd of developers in the organization. In a narrow sense, algorithms for collaborative filtering identify users with similar preferences to produce recommendations for the information seeker [47]. In an RSSE for issues, this would mean matching the profile of the information seeker with the other developers. When the peers most similar to the information seeking developer have been identified, the RSSE can recommend the issues that these peers most often interact with. The user profiles could be based on either previous interaction with issue reports or by features such as role, team, location etc (further discussed in Chapter **??**).

In a wider sense, collaborative filtering can be used to refer to all social recommendation systems. One approach is to reuse the "trails" in the software engineering information landscape, i.e., following in the footsteps of previous work. This type of collaborative RSSEs identifies patterns in data, produced by developers as part of their normal work tasks. This approach can also be referred to as social data mining [47]. The idea is to aggregate the decisions from previous work and make it explicit, with the purpose to support future decision making. Section 1.3 presents two applications of this approach in related to navigation from issue reports to other artifacts during software evolution.

1.2 Supporting issue triage using RSSEs

Issue triage, analyzing a new issue report and deciding how to react to it, requires a lot of effort in large software projects [10]. Questions that are typically asked include: Have this issue been reported before? Should this issue be fixed? Who should fix this issue? Where should this issue be fixed? When debugging large software systems, answering the last question is easier if the developer is aware of all relevant reported pieces of information, thus also similar issue reports are of interest.

Software engineering research has addressed several aspects of issue triaging. Examples include work by Guo *et al.* on predicting which reported issues get fixed at Microsoft [19]. Based on this information, developers could easier prioritize issues during triage, e.g., to decide which bugs should be closed or migrated to future

product versions. A related question, however more specific, is how long it will take to resolve a given issue. Both Weiss *et al.* [49] and Raja [36] report that using the average resolution time of textually similar issue reports can be used as an early estimate for newly submitted issues.

Several researchers focused on assessing the severity of issue reports. Menzies and Marcus developed a tool that alerts developers when the manually assigned severity is anomalous [30]. Lamkanfi *et al.* report promising results from automated severity assignments in a study on three issue repositories used in development of OSS [27]. Another approach to identify severe issue reports was presented by Gegick *et al.*. With a research focus on security-critical software development, they successfully identified about 80% of the reported issues related to security on a large software system from Cisco [17].

Another challenge in large software engineering projects is to assign issue reports to the most appropriate developer [10], to reduce resolution times and minimize reassignment of bug reports also knows as 'bug tossing'. Anvik and Murphy trained a classifier to automatically assign incoming issue reports to developers and reported promising results on five OSS projects [2]. Jonsson *et al.* did similar work and evaluated their prototype on a large proprietary system at Ericsson, reporting performance comparable to manual assignment by human experts [22].

The rest of this section presents work on duplicate detection of issue reports to aid issue triage. When searching for related or duplicate issue reports, part of the problem lies in defining what counts as a duplicate. Duplicates can be categorized as either those that describe the *same failure* and those that describe two *different failures with the same underlying fault* [39]. These two kinds are inherently different in that the former type, which describes the same failure, generally uses similar vocabulary. The latter type on the other hand, which describes two failures stemming from the same fault, may use different vocabulary. RSSEs relying on content-based filtering based on textual features are thus better suited for addressing duplicates of the former type. In this section we refer to the first submitted issue report on a specific fault as the *master report*, and subsequent reports as *duplicate reports*.

1.2.1 Duplicates – Burden or asset?

During the life cycle of large software systems, maintenance activities account for a majority of the development costs [5]. In many software projects, the management of the maintenance work revolves around issue reports in an issue repository. However, the inflow of issue reports often requires significant effort to address them, typically exceeding the available resources [20]. This challenge is further intensified in open source projects, where the software users directly report issues to an open issue repository. Anvik *et al.* highlighted the continuous inflow of new issue reports in the Mozilla community as challenging already in 2005, when the average number of daily submitted issue reports was about 300 [1].

One reason for the daunting inflow of issue reports is that the same issues are reported in multiple reports. Previous studies have shown that the number of duplicate issue reports in issue repositories can be considerable. Sureka and Jalote report that 13% of the issue reports in the Eclipse project (among 205,242 issue reports) were duplicates [46], while Anvik *et al.* studied an earlier stage of the Eclipse project and found that the duplicate fraction in 2005 was 20% (among 18,165 issue reports) [1]. In the same study by Anvik *et al.*, they also studied the issue repository used in development of Mozilla Firefox, observing that it contained 30% duplicate reports (among 2,013 issue reports). Another study on Mozilla software, however not restricted to Firefox, was conducted by Jalbert and Weimer [20]. They observed that 26% of the issue reports were duplicates (among 29,000 issue reports). While a majority of studies on issue management have addressed open source development, the challenge of duplicate issue reports have also been reported from proprietary contexts. Runeson *et al.* showed that the phenomenon exists also at Sony Ericson Mobile Communications (SEMC), where practitioners acknowledged the extra effort caused by duplicates [39]. At SEMC, practitioners approximated 10% of the issue reports to be duplicates.

On the other hand, based on results from a survey on duplicate issue reports among open source developers, Bettenburg *et al.* present another view on the matter. While a majority of the respondents had experienced duplicate reports, only few of them considered it to be a serious problem [4]. On the contrary, the respondents stressed that multiple issue reports related to the same issue often provide additional information, thus decreasing resolution times. Furthermore, Bettenburg *et al.* present empirical evidence confirming that additional information is present in duplicates, in the context of the Eclipse project. Their findings show that duplicates are most often submitted by other users, and that duplicates provide different perspectives and additional information, e.g., additional steps to reproduce the issue and supplementary stacktraces. Consequently, duplicate detection enables merging of issue reports, a feature that can support bug triaging.

As already indicated, providing duplicate recommendations can be meaningful at different points in time in a software development project. First, a tool can support detection of duplicate issue reports on the submitter side. At submission time, only the information entered by the submitter is available, typically limited to basic system information and a natural language description of the observed software behavior. As such, the tool can rely on content-based filtering using text retrieval techniques to recommend the most similar issues reports among the ones already existing in the issue repository. With this type of support, the submitter can decide whether to either 1) submit a new issue report, 2) add additional information to an already open issue report, or 3) skip submitting the issue report, if all information is already available in the issue repository. Making the right decision at submission time has the potential to speed up the issue triage on the developer side. On the other hand, Runeson *et al.* report that it might be hard to make authors of issue reports use the tool in such ways [39]. If someone has taken the time to write a full issue report, he will most likely submit it regardless of the outcome of a duplicate detection, as that action requires the least effort.

Second, a tool can support an engineer on the receiving side of the issue repository. When the developer first receives the issue report, again the only information available is typically a natural language description of the issue and some basic system information. Thus, the same options regarding decision making based on the output of a content-based filtering recommender is available. However, as the developer probably is more knowledgeable than the submitter, the decision might not be the same. Being aware of the duplicate status can be used both to avoid double triaging, and to support issue resolution by aggregating all available information. In an interview with developers on the receiving side, Runeson *et al.* found support for the feasibility of issue duplicate detection in a proprietary context [39].

1.2.2 RSSEs for Duplicate Detection

Current best practice on RSSEs for duplicate detection is based on content-based filtering, and has reached a level of maturity to allow a transition to some well-established software engineering tools. For instance, both HP Quality Center and Bugzilla implement automatic comparisons between newly submitted issue reports and previously reported issues, and this functionality is also used in the marketing of both tools. Also SuggestiMate for JIRA offers this feature. Two general approaches to duplicate detection based on content are used in RSSEs for duplicate detection, either treating it as an *Information Retrieval* (IR) problem, or a *classification* problem. Both approaches have mostly relied on analyzing the textual content in the issue descriptions, thus they share several steps as presented in Fig. 1.2. However, while an indexed document space of issue reports is enough to deploy an IR-based approach, a duplicate detector based on classification requires also a classifier trained on an annotated subset of issue reports. This section presents some experiences from implementations of duplicate detection for issue reports, and summarizes evaluation results and lessons learned.

The accuracy of a tool for duplicate detection can be evaluated in several ways. The most commonly reported measure in the literature is average **recall** when considering the top-k recommendations (recall@k), e.g., how many of the duplicates do I find within the top 1, 2, 3, ... 10 recommendations. As we assume that each duplicate report has exactly one master report, recall@k shows the fraction of duplicate reports with their corresponding master reports presented among the top k recommendations [29]. Another possible perspective is to consider sets of duplicate issue reports, i.e., considering a network of issue reports with undirected edges representing duplicates [8]. Since a majority of previous work apply the first perspective, we use it as the basis for our discussions, more specifically recall@10.

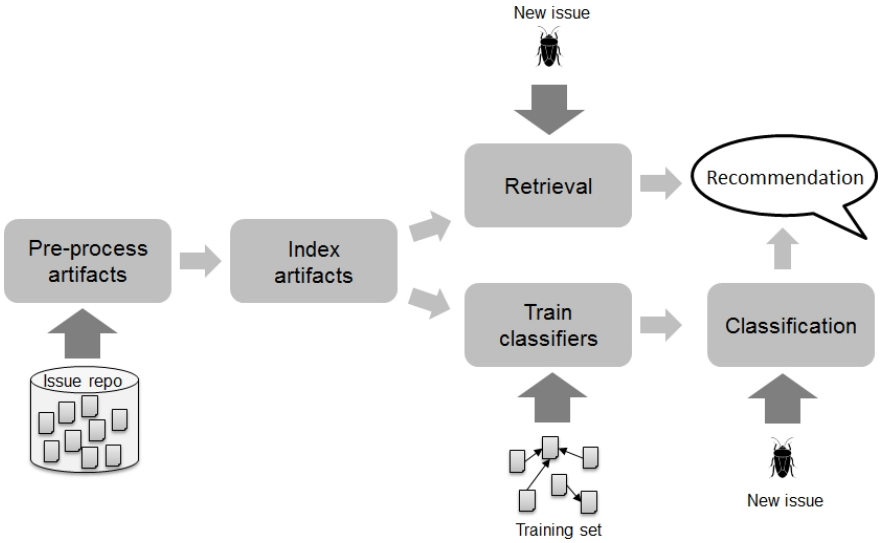

Fig. 1.2: Overview of RSSEs for duplicate detection. The steps in the top track show an RSSE based on the IR approach, while the bottom track displays a classification-based approach.

1.2.3 Duplicate Detection as an IR Problem

The detection of duplicate issue reports can be treated as an IR problem, e.g., implementing classical algebraic IR models. IR is defined as "*finding material (usually documents) of an unstructured nature (usually text) that satisfies an information need from within large collections (usually stored on computers)*" [29]. Thus, IR deals with analyzing large collections to retrieve the most relevant to a given information need. In the context of duplicate detection, this transforms into "given this issue report, which other issue reports are most likely to be duplicates?". The system then returns a ranked list of potential duplicates to the user of the tool.

As depicted in Fig. 1.2, an IR-based RSSE for duplicate detection implements three main steps. The first step, after the issue reports are extracted from the issue repository, is to *pre-process* their textual content. The most common pre-processing steps are:

- **Normalization**. Converting all text to lower case. Removing special characters. Pruning white spaces from the text, keeping only single white spaces between terms.
- **Stop word removal**. Filtering out words that are not suitable as textual features, as they appear in most texts. Examples of such words that capture no semantics of an issue report include the, is, at, and, one, which, etc. Freely available lists of stop words can be found on the Web. Also, stop word functions can be used in

combination with lists, i.e., a function that filters out all words containing fewer characters than a given threshold.
- **Stemming** Reducing inflected words to their stem. This step addresses grammatical variation such as conjugation of verbs and declension of nouns. Stemming should reduce "crash", "crashes" and "crashing" to the same stem, converting them to identical terms. In software engineering, Porter's stemmer [35] is the most commonly applied stemmer for English.

The second step in Fig. 1.2 depicts *indexing* the pre-processed issue reports. RSSEs for duplicate detection typically consider the issue reports as a bag-of-words, a simplifying assumption that represents a document as an unordered collection of words, disregarding word order. The standard technique is to apply the *Vector Space Model* (VSM), representing all issue reports as feature vectors of their contained terms [29]. All terms after pre-processing are stored in a document-term matrix that represents each issue report based on the frequencies of the terms contained in their respective description. Thus, the VSM represents the issue reports in a high-dimensional space where each term constitutes a dimension. An entry in the matrix denotes the weight of a specific term in a given issue report. While term weights can be both binary (i.e., existing or non-existing) and raw (i.e., based on Term Frequency (TF)), usually some variant of *Term Frequency-Inverse Document Frequency* (TF-IDF) weighting is applied. TF-IDF is used to weight a term based on the length of the document and the frequency of the term, both in the document and in the entire document collection. Further details on representing text using the VSM is presented in Chapter **??**.

When VSM is used for IR, document relevance is assumed to be correlated with textual similarity. Thus, when looking for possible duplicates of a given issue report, its similarities to all other indexed issue reports are calculated. When a new issue report arrives, it must first be pre-processed and represented in the same vector-space as was used for indexing the issue repository. The most common similarity measure applied is the cosine similarity, calculated as the cosine of the angle between feature vectors as presented in Fig. 1.3. As no entries in the document-term matrix are negative, the resulting similarity value is bounded in [0,1]. Furthermore, calculating cosine similarity is efficient in sparse high-dimensional spaces as only non-zero dimensions need to be considered. As presented as the final step in the top track in Fig. 1.2, the most similar issue reports are *retrieved* and used as recommendations.

Runeson *et al.* were the first to propose extracting textual features from issue reports and applying the VSM to find duplicates [39]. They considered the textual content in the title and description of the issue reports, and applied the standard pre-processing steps stop word removal and stemming. The resulting textual features were then weighted according to $TF = 1 + log(frequency)$ before the issue reports were represented as feature vectors in the vector space.

Runeson *et al.* evaluated their system on data from Sony Ericsson Mobile Communications, a large company where a software product line is used for developing mobile phones. At the company, about 10% of the issue reports were signaled as duplicates. In this context, Runeson *et al.* evaluated their RSSE for duplicate detection, and explored a number of variations of their system: a) the length of the stop

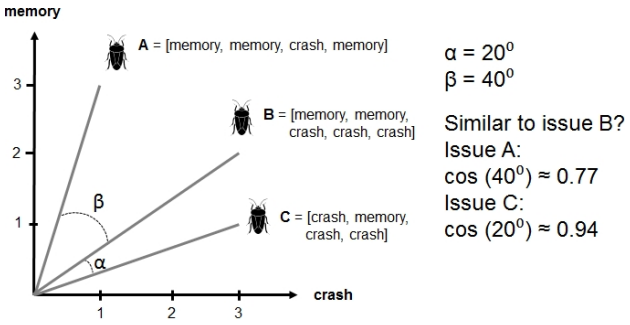

Fig. 1.3: Duplicate detection based on the VSM. In the example, applying raw term weights and cosine similarities, the two most similar issue reports are Issue B and Issue C.

word list, b) adding a thesaurus to deal with synonyms in the issue reports, c) adding a spell checker to auto-correct misspelled terms, d) considering the textual content of an additional field in the issue repository, i.e., "project name" e) up-weighting textual features in the title, to make it more important than the content in the description f) different similarity measures (apart from cosine similarities, also Dice's coefficient and the Jaccard similarity coefficient were evaluated), and g) filtering duplicates according to time frames. However, while some modifications had a small effect on the performance, e.g., adding extra weight to terms in the title, they conclude that little was gained from such fine tuning. About 2/3 of the duplicates could be identified using their system, and they achieved a recall@10 of 40%. This result was well-received among practitioners at the company, who confirmed the potential to save effort.

Wang *et al.* considered in addition to the textual content of the issue report (title and description), the stack traces attached to an issue report [48], and represented the features in two separate VSM models. They pre-processed text using stop word removal and stemming, and then they applied TF-IDF feature weights. Wang *et al.* proposed representing also stack traces in a vector space, and let each invoked method constitute a dimension. As such, each issue report was represented both in a vector space of textual features, and a vector space of invoked methods. Then, the combined similarity was calculated by treating both vector spaces as equally important. If a similarity above a defined threshold was detected in any of the vector spaces, they classified a report as a duplicate. The authors used machine learning to establish suitable values for these thresholds. Wang *et al.* calibrated their system on a small set containing 220 issue reports from the Eclipse project, and then they evaluated their approach on a larger dataset, containing 1,749 issue reports, collected from the issue repository used for the development of Firefox. They conclude that complementing textual descriptions with stack traces improved performance, as did relying on the aforementioned thresholds. Either having two issue reports with highly similar stack traces attached, or two issue descriptions that share a high

degree of its content, was a strong indication of duplicates in the Firefox dataset. Moreover, they confirmed Runeson *et al.*'s finding that up-weighting terms in the title can be beneficial. Wang *et al.* reported that their system reached a recall@10 as high as 93% on the Firefox dataset.

Sun *et al.* proposed a more advanced model for duplicate detection, including both categorical issue features as well as more advanced weighting of textual features [44]. Initially, they performed the same pre-processing operations as in previous work, i.e., they stemmed the content in the title and description and they removed all stop words. However, thereafter they calculated textual similarities, both for unigrams (single terms in isolation) and bigrams (sequences of two terms), using the BM25F model, a state-of-the-art IR model for probabilistic retrieval [38] (an alternative to the algebraic VSM). Furthermore, the authors also considered three nominal features (product, component, and type), and two ordinal features (priority, and version). Finally, to tune the weighting of all parameters in the similarity function, they applied machine learning to tune the feature weighting, i.e., they applied learning-to-rank ranking in their IR-approach [28].

Sun *et al.* evaluated their system using issue reports extracted from three open source contexts: OpenOffice, Eclipse, and the Mozilla community. Moreover, they compared the results with output from their previously implemented RSSE for duplicate detection, implementing a classification-based approach (presented in Section 1.2.4). In all experimental runs they obtained better results (recall@10 consistently between 65% and 70%), also they reported major improvements concerning execution times.Again, their empirical results showed that the textual content of the title was the single most important feature. However, the results showed that also the product and version information were important features when recommending duplicate issue reports.

Sureka *et al.* presented a different approach to textual similarities, focusing on characters rather than terms [46]. They proposed a character n-gram model to calculate textual similarities between issue reports. They did not perform any language specific stemming and stop word removal, and thus their approach allow also cross-language recommendation of duplicates. Sureka *et al.* applied a feature extraction model that extracted all n-grams of sizes 4 to 10 from the titles and descriptions of issue reports. They evaluated their approach on a random sample of 2,270 issue reports, and obtained the best results when computing textual similarities based on titles only (recall@10 of 40%).

1.2.4 Duplicate Detection as a Classification Problem

Duplicate detection can also be considered a **classification** problem. Given a newly submitted issue report, an RSSE can classify it as either a duplicate or a non-duplicate based on the previously submitted issue reports. As presented in the bottom track in Fig. 1.2, a classification-based RSSE for duplicate detection typically involves four steps. The first two steps are shared with the IR-based approach. First,

Study	Retrieval model and features used	Dataset (#issue reports)	Recall@10	Lessons learned
Runeson *et al.* [39]	VSM. Word unigrams from title and description with TF weights. Categorical feature: project.	SEMC (Undisclosed "large")	40%	Fine tuning had little effect. However, best results for a) short stop word list, b) using thesaurus and c) spell checker, d) considering the project field, e) doubling the weight of terms in the title, and f) applying a 50-day filter for issue reports.
Wang *et al.* [48]	VSM. Word unigrams from title and description with TF-IDF weights. Invoked methods from stack traces with binary weights.	OSS project: Firefox (1,749)	93%	Combining textual features and stack traces improve performance, especially when independent thresholds are applied. Doubling the weight of terms in the title beneficial.
Sureka *et al.* [46]	Character n-grams from title and description ($4 \leq n \leq 10$).	OSS project: Eclipse (2,270)	40%	Acceptable results without preprocessing, thus cross-language detection possible. Character n-grams in titles most useful.
Sun *et al.* [44]	BM25F. Word unigrams and bigrams from title and description with TF-IDF weights. Categorical features: product, component, type, priority, version.	OSS projects: OpenOffice (31,138), Eclipse (209,058), Mozilla (75,653)	OpenOffice: 65%, Mozilla: 65%, Eclipse: 70%	Textual content of title most important feature, followed by product and version information. Faster and more accurate than [45].

Table 1.1: Summary of studies on IR-based duplicate detection, sorted chronologically.

the issue reports are *pre-processed*, and then the issue reports are *indexed* by the remaining terms, e.g., as feature vectors in the VSM as presented in Fig. 1.3. Third, machine learning is used to *train classifiers*, either multiple binary classifiers or a multi-class classifier. A major difference between the classification approach and the IR approach to duplicate detection is thus that the former normally requires *supervised learning*, i.e., learning from an annotated training set containing both positive and negative examples. The standard IR approach on the other hand, uses the existing information only, without any learning process. In this section, we focus on describing an approach proposed by Sun *et al.* [45]. They trained an individual classifier for each existing issue report in three large issue repositories, and used the classifiers to predict whether a newly submitted issue report is a duplicate or not. Fourth, when a new issue report is submitted to Sun *et al.*'s prototype, all classifiers answer the question: "How likely is this newly submitted issue report a duplicate of this master issue report?".

A frequently used approach for "off-the-shelf" supervised learning is to apply *Support Vector Machines* (SVM), when training classifiers in a new domain [40]. The SVM model maps all issue reports as points in space, where different terms can constitute the dimensions as in the VSM, and individual issue reports are typically represented as the endpoints of their corresponding feature vectors. SVMs then construct a maximum margin separator, a hyperplane that divides the positive

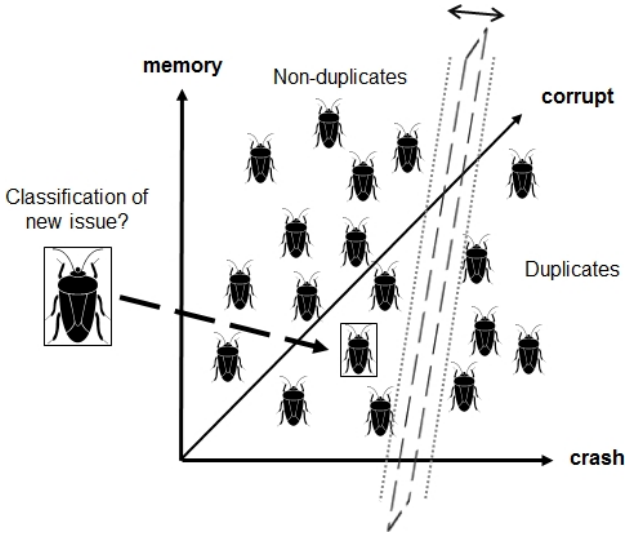

Fig. 1.4: Duplicate detection based on SVM. In this example, the three terms memory, corrupt and crash constitute the dimensions in the vector space. An individual SVM classifier, trained for a specific issue report, predicts that a newly submitted issue report is a non-duplicate. An optimal separating hyperplane, a maximum margin separator, is shown to the right.

and negative examples with a gap as wide as possible as illustrated in Fig. 1.4, thus creating two classes: duplicates and non-duplicates with respect to a given master issue report. As the fourth and final step, newly submitted issue reports are mapped to the same space, and depending on which side of the hyperplane they belong, the classifier predicts whether the issue reports are duplicates or not. When all SVM classifiers have made their predictions, the classification-based RSSE can *recommend* potential duplicates. Further details on SVM is presented in Chapter **??**.

The classification-based RSSE implemented by Sun *et al.* uses a hash-map-like *bucket structure* [45]. Each bucket contains a master report as the key, and all its duplicate reports as values. Then, an SVM classifier is trained for each bucket, using the duplicate reports as positive examples, and the others as negative examples. When a new issue report is submitted, all classifiers report a probability value given by the distance to the separating hyperplane. The probability values are then used to rank the output, used to recommend potential master issues of a submitted issue report. Sun *et al.* used a rich set of textual features to train their classifiers. The high number of features originate from considering three different bags-of-words (title, description, and title+description) independently, allowing three different calculations of inverse document frequencies (as IDF weights are calculated based on frequencies in the entire collection). Moreover, they considered both unigrams and bigrams. In total, 54 different textual similarities were calculated between each pair of issue reports, 27 features based on unigrams and 27 features based on bigrams.

Sun *et al.* [45] evaluated their system on issue reports from three open source projects: OpenOffice, FireFox, and Eclipse. Furthermore, they implemented three previously proposed systems for duplicate detection for comparison (Runeson *et al.* [39], Wang *et al.* [48], and Jalbert *et al.* [20]). On all three datasets, they showed that their system outperforms the others. Sun *et al.* reported recall@10 of 60 % for both Firefox and Eclipse, and recall@10 of 55% for OpenOffice. They also reported that the execution time of their system was longer than what was required in previous work.

Jalbert *et al.* also did work on duplicate detection that they refer to as classification-based [20]. While their approach is based on IR techniques, they extended it by also training a classifier based on linear regression. First they considered textual content in the title and the description as two separate bags-of-words, and pre-processed them using stop word removal and stemming. Through experimentation they showed that considering IDF did not improve performance in their context. Instead, they found it the most useful to consider only term frequencies and weigh the textual features as $TF = 3 + 2 * log_2(frequency)$. All issue reports were represented as feature vectors in the VSM, and they calculated cosine similarities to induce a graph of issue reports, connecting issue reports by undirected edges if two issue reports were more similar than a certain threshold. Jalbert *et al.* then applied a graph clustering algorithm developed for social network analysis [32], to generate a set of possibly overlapping nodes in the graph, i.e., potentially duplicated reports. Furthermore, they considering a set of ordinal and nominal surface features: severity, operating system, and the number of associated patches or screenshots. They used all the features to train a linear regression model, and to find a corresponding output value cut-off that distinguishes between duplicates and non-duplicates. The linear regression model could then be used for newly submitted issue reports, to perform classifications against each existing issue report.

Jalbert *et al.* evaluated their system on 29,000 issue reports from the Mozilla community, containing reports from several development projects. They report that their system is at least as good as Runeson *et al.*'s approach [39], and achieved a recall@10 of 45%. By conducting leave-one-out analysis on their textual features, they concluded that the textual content of the title was the most important, followed by the description. While other features also brought value, they all contributed less to the linear model. Also, Jalbert *et al.* simulated the performance of their system over 16 weeks using the submission dates of the issue reports in their dataset, i.e., they reported how their tool would have performed if it was deployed in the Mozilla context. They used the chronological first half of the dataset as the training set, and evaluated their work on the second half. Their fully automated system correctly filtered 8% of all possible duplicates, while allowing at least one report for each real issue to reach developers. The authors estimate that this could have saved 1.5 developer-weeks of triage effort over sixteen work weeks (assuming that each manual issue triage would on average require 30 minutes).

Study	Classifier and features used	Dataset (#issue reports)	Recall@10	Lessons learned
Jalbert *et al.* [20].	Linear regression model. Word unigrams from title and description with TF weights. Categorical features: severity, operating system, nbr attached patches or screenshots.	OSS projects from Mozilla (29,000)	45%	Performs comparably to [39]. Through simulation they show that their system realistically could save effort. Also, they show that the textual content in the title is the most important feature.
Sun *et al.* [45]	SVM. Word unigrams and bigrams from title and description with several TF-IDF weights (in total 54 textual features).	OSS projects: OpenOffice (12,723), Eclipse (44,652), Firefox (47,704)	About 60%	Outperforms [39], [48], and [20]. While the high number of textual features leads to better results, more execution time is required.

Table 1.2: Summary of studies on duplicate detection based on classification.

1.3 Navigating from Issue Reports in Evolving Software Systems

Software development typically involves managing large amounts of information, i.e., formal and informal software artifacts, that evolve in response to environmental changes and user needs. In traditional software engineering, the software is systematically progressed through analysis, specification, design, implementation, verification, and maintenance. However, the increase of formalized knowledge-intensive activities tend to increase the number of artifacts maintained in a project [50]. When software evolution accumulates changes, made by many different developers over time, possibly from different development sites, it is a challenge to stay on top of all information.

A different development context, also highly challenging in terms of information access, is development of *Open Source Software* (OSS). Mature software such as Android, Eclipse, and Mozilla Firefox have successfully adopted OSS development practices. Development of OSS is often characterized by globally distributed workforces and rapid software evolution. As most collaboration is online, communication within the team must be smooth, and all available information must be easily accessible. Open source projects typically rely on simple techniques such as discussion forums and mailing lists for communication, complemented by advanced version control systems for managing source code and supporting artifacts [12]. While the number of artifact types are typically lower than in traditional software development, quick and concise access to information is essential as teams cannot rely on face-to-face communication.

Thus, both large traditional projects and OSS projects risk being threatened by *information overload*, a state where individuals do not have time or capacity to process all available information [16]. Knowledge workers frequently report the stressing feeling of having to deal with too much information [15], and in general spend a substantial effort on locating relevant information [24]. Thus, an impor-

tant characteristic of a software development context is the *findability* it provides, defined as "the degree to which a system or environment supports navigation and retrieval" [33]. A prerequisite to developing an RSSE for navigation support is to properly understand the context of the work task that is to be supported.

This section presents two separate examples of RSSEs supporting software evolution, where issue reports are used as 'hubs' in generating traces between information items. First, we present *Hipikat*, proposed by Cubranic *et al.* [13], an RSSE targeting software evolution in open source projects, specifically aiming at helping project newcomers. Second, we introduce *ImpRec*, an RSSE supporting safety-critical change impact analysis in a company with rich development processes. Both RSSEs are based on knowledge reuse from previous collaborative effort in projects complemented by textual analysis of artifact content, and rely on artifact usage rather than explicit ratings provided by engineers.

We present both Hipikat and ImpRec using a four step model, shown in Fig. 1.5. The development of the RSSEs starts by *modelling the information space*, to create a schema that can be used to represent the involved software artifacts. Second, the developed model is *populated* by historical data from the corresponding projects, and stored in an actionable data structure. Also as part of this step, textual content of artifacts are pre-processed and indexed. When the historical data has been been processed, the RSSEs are ready to *calculate recommendations*. This can either be initiated by the developer explicitly, or implicitly based on the work task the developer pursues at a given time. The final step in the model covers how the *recommendations are presented* to the developers.

Hipikat and ImpRec are not the only approaches supporting navigation in large software engineering projects. Begel *et al.* developed CodeBook, a framework for connecting engineers and their software artifacts [3]. It was developed to support mining various software repositories, and to capture relations among people and artifacts in a single graph strucutre. Codebook was evaluated by implementing portal solutions at Microsoft, helping developers discover and track both colleagues and work artifacts in a large software project. Seichter *et al.*, also inspired by the social media revolution, addressed the management of artifacts software ecosystems by creating "social networks" with artifacts as first-class citizens [41]. The explicitly visible network of artifacts supported maintaining relations between artifacts and enabled personal "news feeds" for involved developers, containing recommendations for relevant changes and possible dependencies etc. Gethers *et al.* proposed automated impact analysis from textual change requests [18], an approach that is reused in ImpRec. Their tool combined IR techniques, analysis of software evolution using data mining, and execution information via dynamic analysis to recommend an initial set of impacted methods in the source code of four OSS systems.

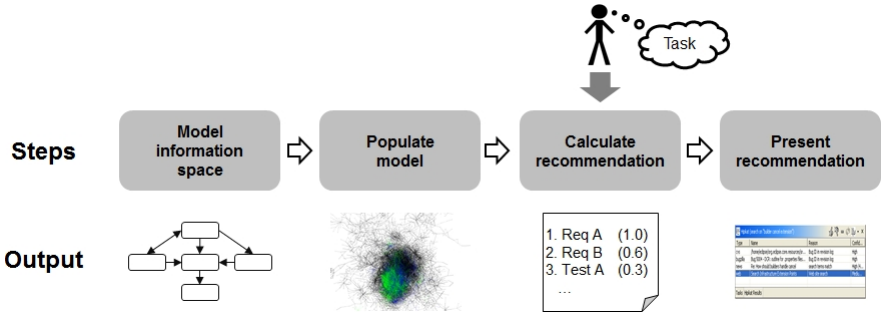

Fig. 1.5: Four step model for development of an RSSE for navigation support.

1.4 Hipikat – Helping a Project Newcomer Come Up-to-Speed

Hipikat was developed by Cubranic *et al.* to build a project memory to support newcomer software developers by using information about past modifications to the project. The aim is to help them perform modification tasks to the system more effectively [11–13]. Hipikat has mainly been studied in the context of the Eclipse OSS community[1]. The main Eclipse project is the Eclipse Platform, a mature IDE written in Java, known for its extensible architecture and many third-party plug-ins.

For software developers joining the Eclipse Platform project, the first contact with the heterogeneous information space of the project can be discouraging: there are several thousands of files, issue reports, documentation, and discussions. In a traditional project, the developer would join a team and gain knowledge through mentoring [43]. An experienced team member would work closely with the newcomer and orally impart the information structure and help him becoming productive. However, in OSS projects such lightweight interaction is typically not possible as the developers are globally distributed. Thus, it is challenging for a project newcomer to come up-to-speed and learn a new software system, e.g., navigating source code and finding issue reports relevant to the work task at hand.

Hipikat is implemented as a client-server architecture, as depicted in Fig. 1.6. The server maintains the *project memory*, a semantic network of artifacts and relations, formed during development and updated as the target system evolved. Developers interact with Hipikat clients, which can be implemented in various ways, such as the Eclipse plug-in developed by Cubranic *et al.* [13]. The server and the clients communicate over a SOAP RPC protocol, and the server provides recommendations to the clients in an XML format [11]. Sections 1.4.1-1.4.2 present Hipikat according to the structure in Fig. 1.5.

[1] www.eclipse.org

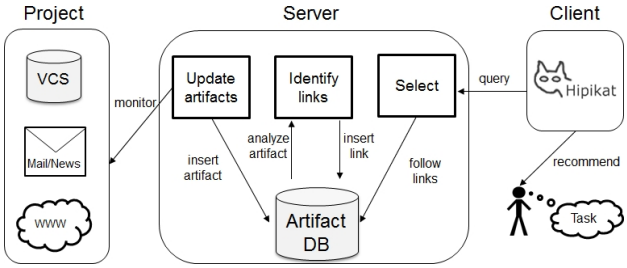

Fig. 1.6: Client-Server architecture of Hipikat. Adapted from [13].

1.4.1 Step 1: Modeling the Hipikat Project Memory

Cubranic *et al.* developed Hipikat with the ambition to provide a project newcomer recommendations of source code, accompanying information and stored developer communication relevant to an issue report. Fig. 1.7 displays the five artifact types represented in the project memory, and the relations among them. Four of the types correspond to artifacts commonly created in OSS projects. The central entity is the *issue report*. Source code *file versions* implement resolutions to issues described in issue reports, and might be related to other file versions in the same **commit**. Project *documents* represent accompanying information posted on the project website, e.g., design documentation, and such artifacts might document aspects relevant to an issue report. Another artifact type in the model is the project *message* (e.g., in discussion forums or mailing lists) that might contain information about an issue report. Messages might also be related to each other as they might be posted as responses, i.e., "reply-to" links. Furthermore, the Hipikat model also covers implicit relations between documents and between issue reports respectively, modeling that these artifacts might have outgoing "similar to" relations if artifacts share much content (represented by dashed edges in Fig. 1.7). Finally, the scheme of the project memory represents *persons*, who might work on issue reports, post messages, and write documents and source code file versions.

1.4.2 Step 2: Populating the Hipikat Project Memory

The Hipikat server is responsible for populating the Hipikat project memory. The server has three functions, implemented in three subsystems as seen in Fig. 1.6. The *update artifacts* subsystem monitors the project information space for additions and changes during the OSS evolution. Cubranic *et. al* distinguish between three categories of artifacts in the Eclipse Platform development project: *immutable* (e.g., source file revisions), *modifiable but not deletable* (e.g., issue reports in Bugzilla), and *changeable* (e.g., web pages). The update subsystem has separate modules for the following project information sources: CVS (the version control system),

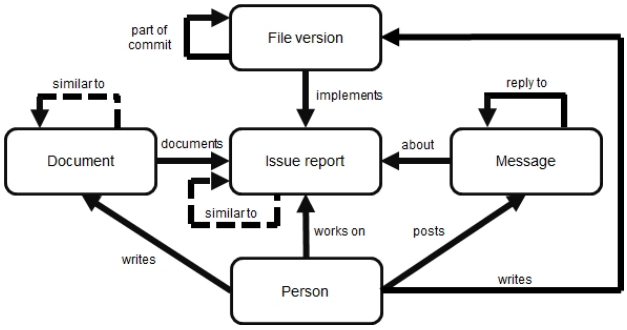

Fig. 1.7: Entity-Relation diagram of the software artifacts in the Hipikat project memory. Adapted from [13].

Bugzilla (the issue repository), www.eclipse.com (the web page), Usenet newsgroups, and Mailman (the archive of email messages). *Change listeners* in the subsystem are notified as the information space changes, and new and modified artifacts are inserted in the artifact database (cf. Fig. 1.6). More specifically, the artifact database stores mostly artifact metadata, e.g., ID, author, path and creation date, but also natural language content is stored such as commit comments, issue descriptions and email body text.

Apart from the artifacts stored in the project memory, a vital aspect of the project memory is the relations among them. Some explicit relations are directly available in the artifact metadata, e.g., authors and dates. However, Hipikat also implements several modules that independently analyze artifact content to deduct additional links. Moreover, Hipikat creates links with different *confidence*, a measure of trustworthiness that is used when presenting recommendations to the developer. The four link types "implements", "part-of-commit", "reply-to", and "similar-to" (cf. Fig. 1.7) are identified in the *identify links* subsystem using the following five modules [11]:

The log analyzer uses regular expressions to identify commit comments containing issue IDs to insert high confidence "implements" links.

The activity matcher searches for commits by a developer that occur shortly before the *same* developer changes the status of an issue report to resolved. Inserts implements links between source code file version and issue reports to the project memory with confidence reflecting the time span.

The CVS commit matcher identifies file versions checked in within a few minutes. "Part-of-commit" links are added if they have the same author and commit comment.

The thread matcher identifies both conversation threads of newsgroup postings and email threads by looking for specific headers in the stored messages. "Reply-to" links are inserted accordingly.

The text similarity matcher predicts relations among documents and among issue reports (cf. Fig. 1.7) based on the similarity of the textual content. This is implemented in the same manner as the IR-based duplicate detection described

Module	Type of link	Confidence	Link source	Link target
Log analyzer	Implements	High	Commit	Issue report
Activity matcher	Implements	Low, Medium, Medium high, High	Commit	Issue report
CVS commit matcher	Part-of-commit	Medium low, Medium, Medium high, High	Commit	Commit
Thread matcher	Reply-to	N/A	Message	Message
Text similarity matcher	Similar-to	Cosine similarity (0-1)	Document	Document
Text similarity matcher	Similar-to	Cosine similarity (0-1)	Issue report	Issue report

Table 1.3: Summary of links followed in the Hipikat project memory.

in Section 1.2.3. All textual content is pre-processed using stop word removal and stemmed using Porter's stemmer, then indexed using the VSM. Hipikat uses the log-entropy model for feature weighting. As an additional step, the vector space is transformed using Latent Semantic Indexing [14] (further described in Chapter **??**), an approach that aims to remove noise and to deal with synonymy. Finally, cosine similarities are calculated and "similar-to" links are added to the project memory.

1.4.3 Step 3: Calculating Recommendations in Hipikat

The third subsystem Cubranic *et al.* developed in the Hipikat server, *Select* (cf. Fig. 1.6), calculates recommendations by using the relationship links in the project memory. Hipikat recommendations are always calculated in response to a query, either explicitly initiated by the user, or implicitly as the user performs work tasks [13]. An implicit query identifies the artifact from which the query originates, and might also contain additional context options. For explicit queries, Hipikat searches for the artifact the developer specifies in the query. The select subsystem locates that artifact in the project memory, and follows relationship links to generate a set of recommended artifacts for the developer to consider in his current work task.

The select subsystem contains modules that correspond to the five identification modules described in Section 1.4.2. As the modules recommend artifacts, they also provide rationales for their choices as well as Hipikat's confidence for the recommendations. Before the final recommendations are presented to the developer, the output from the modules is merged. If multiple modules recommend the same artifact, only the one with the highest confidence will be kept. Table 1.3 lists the link types that are followed by Hipikat, to enable the RSSE to detect recommendation trails in the project memory. Fig. 1.8 shows an example of such trails, originating from a study on Avid visualizer, a Java tool for visualizing the operation of a Java system [11].

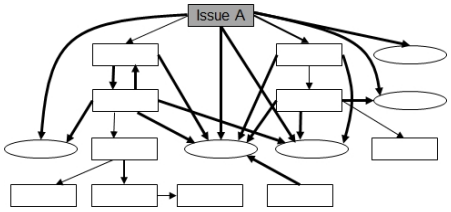

Fig. 1.8: Example of Hipikat recommendation trails, in this case items in the project memory related to Issue A. Issue reports are represented as boxes, and source code file versions as ovals. All edges are directed, and have a confidence value as indicated by the edge weights. Adapted from [11].

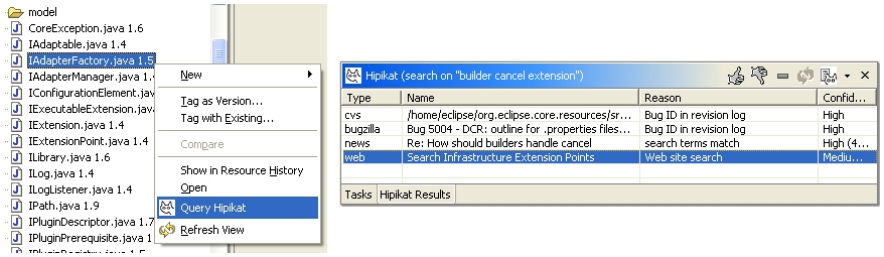

Fig. 1.9: The Hipikat user interface. To the left, querying Hipikat from a context menu. To the right, presentation of Hipikat recommendations. Reproduced with permission from the original developers [11].

1.4.4 Step 4: Presenting Recommendations

The Hipikat client is available as an Eclipse plug-in, which means it is integrated in the IDE. Cubranic *et al.*'s goal was to develop an unobtrusive client, and the user interaction is kept simple. Primarily, the developer explicitly queries Hipikat for recommendations from context menus. "Query Hipikat" is an available menu item in the context menus of several entities in Eclipse, e.g., version controlled files either in the workspace Navigator or opened in the Java editor, files in Repository view, revisions in the Resource History view and Java classes in the Outline or Hierarchy views.

As a response to queries from the Hipikat client, the Hipikat server returns a list of recommended artifacts as presented in Section 1.4.3. The list is presented in a Hipikat *Results view*, where each recommended artifact is displayed together with its type, why it is recommended and the confidence of the recommendation. The recommended articles are grouped by artifact type. Double-clicking on an artifact opens them for viewing, either directly in Eclipse, or in a web browser. Moreover, the developer can also initiate new Hipikat queries from the context menues of the artifacts in the Results view.

Developers using Hipikat can also provide feedback on the recommendations. For each recommended artifact, a developer can select “like” or “dislike”. Dislikes clean the list of recommendations, i.e., disliked artifacts are removed from the list. Liked recommendations on the other hand move to the top of the list. To better use the developer feedback is one out of several improvements Cubranic outlined to further improve Hipikat [11], however the RSSE is not actively developed anymore.

1.5 ImpRec – Supporting Impact Analysis in a Safety Context

The goal of ImpRec is to support artifact navigation in a development organization in a large multinational company, active in the power and automation sector. The development context is safety-critical embedded development in the domain of industrial control systems, governed by IEC 61511[2] and certified to a Safety Integrity Level (SIL) of 2 as defined by IEC 61508[3]. The targeted system has evolved over a long time, the oldest source code was developed in the 1980s. A typical project has a duration of 12-18 months and follows an iterative stage-gate project management model. The number of developers is in the magnitude of hundreds, distributed across sites in Europe, Asia and North America.

As specified in IEC 61511, the impact of proposed software changes should be analyzed before implementation. In the case company, this process is integrated in the issue repository [6]. As part of the analysis, engineers are required to investigate the impact of a change, and document their findings in an *impact analysis report* according to a project specific template. The template is validated by an external certifying agency, and the impact analysis reports are internally reviewed and externally assessed during safety audits.

A slightly modified version of this template is presented in Table 1.4. Several questions explicitly ask for trace links (6 out of 12 questions). The engineer is required to specify source code that will be modified (with a file-level granularity), and also which related software artifacts need to be updated to reflect the changes, e.g., requirement specifications, design documents, test case descriptions, test scripts and user manuals. Furthermore, the impact analysis should specify which high-level system requirements cover the involved features, and which test cases should be executed to verify that the changes are correct once implemented in the system. In the addressed software system, the extensive evolution has created a complex dependency web of software artifacts, thus the impact analysis is a daunting work task.

Fig. 1.10 shows an overview of the ImpRec recommendation approach. To the left, a developer is about to conduct a new impact analysis as part of a defect correction, i.e., answering Q1–Q12 in the impact analysis template in Table 1.4. First, content-based filtering is used to find issue reports with descriptions similar to the current issue report stored in the issue repository. The same techniques as presented

[2] Functional safety - Safety instrumented systems for the process industry sector

[3] Functional safety of Electrical/Electronic/Programmable Electronic safety-related systems

Impact Analysis Questions for Error Corrections	
Q1)	Is the reported problem safety critical?
Q2)	In which versions/revisions does this problem exist?
Q3)	How are general system functions and properties affected by the change?
Q4)	**List modified code files/modules and their SIL classifications, and/or affected safety related hardware modules.**
Q5)	**Which library items are affected by the change? (e.g., library types, firmware functions, HW types, HW libraries)**
Q6)	**Which documents need to be modified? (e.g., product requirements specifications, architecture, functional requirements specifications, design descriptions, schematics, functional test descriptions, design test descriptions)**
Q7)	**Which test cases need to be executed? (e.g., design tests, functional tests, sequence tests, environmental/EMC tests, FPGA simulations)**
Q8)	**Which user documents, including online help, need to be modified?**
Q9)	How long will it take to correct the problem, and verify the correction?
Q10)	What is the root cause of this problem?
Q11)	How could this problem have been avoided?
Q12)	**Which requirements and functions need to be retested by product test/system test organization?**

Table 1.4: Impact analysis template. Questions in bold fonts require explicit trace links to other artifacts. Based on a description by Klevin [26].

in Section 1.2.3 on detection of duplicate issue reports using IR approaches are applied. Then, originating from the most similar issue reports, the collaboratively constructed trace link network, the "trails of previous developers in the information landscape" is used to recommend which trace links the developer should consider specifying in the impact analysis report. As such, the new impact analysis work task is seeded by the pre-existing traceability from past impact analysis reports. The network of collaboratively created trace links is referred to as the *knowledge base*, a concept corresponding to the project memory in Hipikat. Note that the current scope of ImpRec is limited to recommend non-code artifacts relevant to an issue report, as these are considered more challenging in the case company.

1.5.1 Step 1: Modeling the ImpRec Knowledge Base

Fig. 1.11 shows the model of the knowledge base as an entity-relation diagram. The *impact analysis report* that is attached to *issue reports* that cause changes to safety-critical source code is the hub in the model. An impact analysis report can contain trace links to several different artifact types, specifying relationships from individual issue reports (Q4-Q8 and Q12 in Table 1.4). *Requirements* (e.g., system requirements, safety requirements, and functional descriptions) can specify functionality that is impacted by the problem described in an issue report. *Test specifications* can verify functionality that is described in an issue report. Also, making the changes required to resolve an issue report might force updates to test specifications as well.

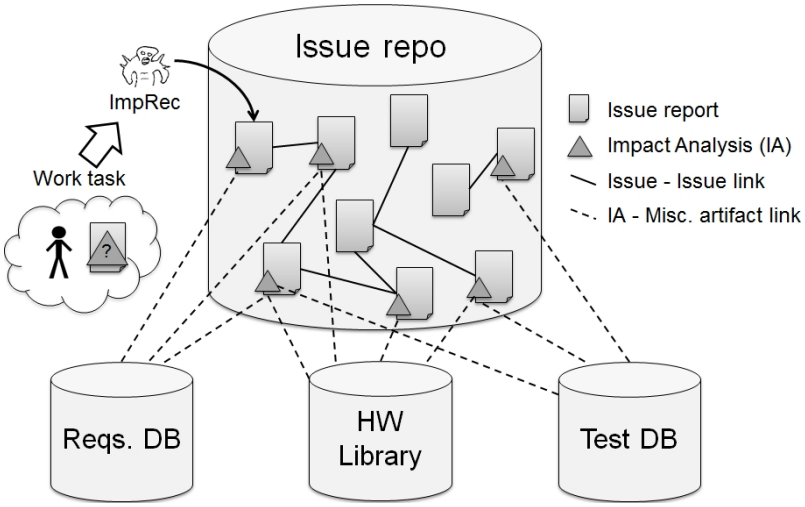

Fig. 1.10: Impact analysis supported by ImpRec. Trace link structure created by collaborative effort. Trace links among defects and from impact analysis reports to requirements, HW descriptions and test cases.

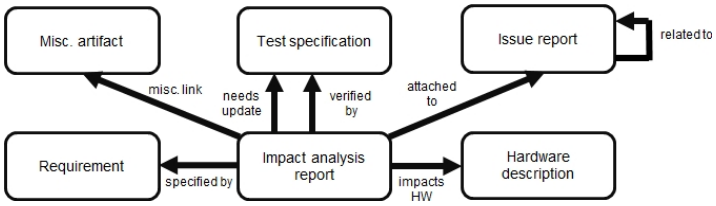

Fig. 1.11: Entity-Relation diagram of the software artifacts in the ImpRec knowledge base.

The changes might also impact *hardware specifications*. Finally, an issue report can be related to other issue reports, a relation that is explicitly specified by developers in the issue repository (presented also in Fig. 1.10). Furthermore, as the type of artifacts cannot always be deduced by ImpRec, also *misc. artifacts* and *misc. links* are included in the model.

1.5.2 Step 2: Populating the ImpRec Knowledge Base

To aggregate the trace links from previous developers, ImpRec mines the issue repository [7]. In the studied case, 4,845 out of the 26,703 issue reports in the issue repository contain impact analysis reports. As a first step, the issue reports in the issue repository were exported to an extended CSV format, a format specified

by the vendor of the issue repository, and transformed to XML. Thus, the overall information of the issue reports were well structured, however the attached impact analysis reports were stored as text elements. On the other hand, the textual information in the text elements were semi-structured according to the structure in the impact analysis template in Table 1.4.

Then, regular expressions were used to extract *trace links from the impact analysis reports*. Due to the fixed format of artifact IDs, this method could extract all correctly formatted trace links. To determine the type of the extracted trace links, two heuristics were used. Thanks to the structure of the impact analysis template, each trace links corresponded to a specific question. As such, in the context of Q8 and Q12 it could be deduced that the meaning of the links was related to verification, Q7 and Q9 deal with document updates, and Q6 refers to impact on hardware descriptions. Second, as requirement IDs have a distinguishable format in the company, also requirements specifications could be identified.

Next, explicit *trace links between issue reports* in the issue repository were extracted [8]. Each defect in the issue repository has a field "Related issues", used by engineers to manually signal other issues as related, by adding their issue IDs. Finally, the two extracted networks were combined into a single network, the ImpRec *knowledge base*. The knowledge base, consisting of 29,000 nodes and 28,230 edges, is represented as a semantic network expressed in GraphML [9]. Tables 1.5-1.6 summarize the different types of extracted trace links and trace artifacts.

Note that this only contains artifacts actually pointed out by the previous impact analysis reports, and that the total number of artifacts in the content management systems in the company is much higher. However, while the extracted traceability is a partial view, this is the traceability associated with the most volatile parts of the system to date, and thus a pragmatic starting point for future impact analyses.

Fig. 1.12 shows an overviews of the largest interconnected component of the knowledge base, comprising 36.2% of the nodes and 81.7% of the edges, created in the graph editor yEd [31] using an *organic layout*. The graph nodes are treated like physical objects with mutually repulsive forces. The edges on the other hand are considered to be metal springs attached to nodes, producing repulsive forces if they are long and attractive forces if they are short. By simulating these physical forces, the organic layout finds a minimum of the sum of the forces emitted by nodes and edges. Output from organic layouts typically show inherent symmetric and clustered graph structures, useful for finding highly connected backbone regions in a graph. Although the primary purpose of the knowledge base is not to enable visual analytics, visual representations of complex information might enable additional insights [25]. Regarding the knowledge base, we can make some general observations. First, Fig. 1.12 shows that there is a highly interconnected central region containing thousands of artifacts, rather than several distinctive clusters. This region displays a high link density, and while "specified by" (i.e., links to requirements) dominate the central region, all link types are present. This implies that changes to an artifact in this region could impact a high number of artifacts. In general, the complex link structure displayed in Fig. 1.12 suggests that much traceability information about artifact relations in the software system has been captured in the knowledge base.

Trace link type	Description	Classification strategy	Count
related to	Link to another issue report that has been signaled by an engineer as related. The link is not bidirectional by default.	Separate field in issue report	18,835
specified by	Link to a specific requirement. Used to signal that a requirement needs to be updated, or requires verification.	Format of requirement IDs	3,996
verified by	Link to a test case description that needs to be executed, or a requirement that requires verification.	IA template, Q8 and Q12	2,297
needs update	Link to a software artifact that needs to be updated.	IA template, Q7 and Q9	1,106
impacts HW	Link from an IA report to a hardware description that is impacted by the issue or its implemented resolution.	IA template, Q6	1,221
misc. link	Trace links from an IA report to an artifact, but the meaning of the link could not be deduced.	Default choice	775

Table 1.5: Types of links extracted from the issue repository. All links have an issue report as source. IA stands for Impact Analysis.

Trace artifact type	Description	Classification strategy	Count
issue report	An individual issue report.	Separate item in issue repository	26,703
impact analysis report	A documented impact analysis.	Attached to an issue report	4,845
requirement	An individual requirement. The requirements are organized in requirement specifications.	Separate ID format	572
test specification	A document that contains test case descriptions.	IA template, Q8 and Q12	243
HW description	An artifact that describes the behavior of hardware	Separate ID format	1,106
misc. artifact	An artifact whose type could not be deduced.	Default choice	376

Table 1.6: Types of nodes extracted from the issue repository. IA stands for Impact Analysis.

1.5.3 Step 3: Calculating Recommendations in ImpRec

When the knowledge base is represented as a semantic **network**, ImpRec calculates recommendations in three steps:

1. Retrieval of likely related issue reports, based on their textual *content*.
2. Search for artifacts that previously were marked as impacted, based on the *collaboratively* created knowledge base.
3. Ranking the identified artifacts based on textual similarities and network structure.

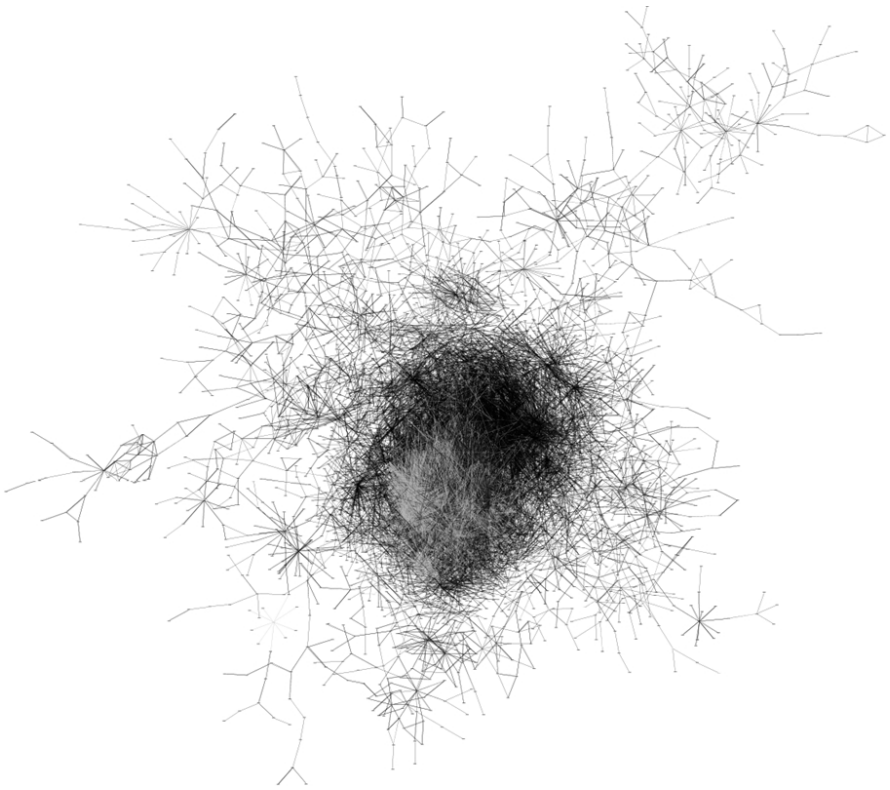

Fig. 1.12: A visualization of the knowledge base, displaying the largest component (10,211 artifacts and 23,078 relations).

First, ImpRec uses content-based filtering, based on the textual content of the issue reports, to identify starting points in the knowledge base. Both terms in the title and description of issue reports are considered, after stemming and stop word removal. Then the remaining textual features are assigned TF-IDF weights before representing them in the VSM. ImpRec then calculates cosine similarities between the given issue report and all others, and finally rank them accordingly. This work is in line with previous work on IR-based duplicate detection presented in Section 1.2.3. ImpRec considers the top five issue reports, corresponding to the five highest non-zero cosine similarity values, as *starting points 1-5* ($START_i$) in the knowledge base. Moreover, the similarity values of the five issue reports are re-normalized between 0 and 1 (SIM_i) for later ranking purposes.

Originating from the starting points, ImpRec performs breadth-first searches in the knowledge database to find artifacts (ART_x) that previously have been pointed out as impacted. First, impacted artifacts linked from starting points are identified, then issue reports connected to the starting points are considered. ImpRec searches for impacted artifacts up to three levels away from starting points ($LEVEL$), i.e.,

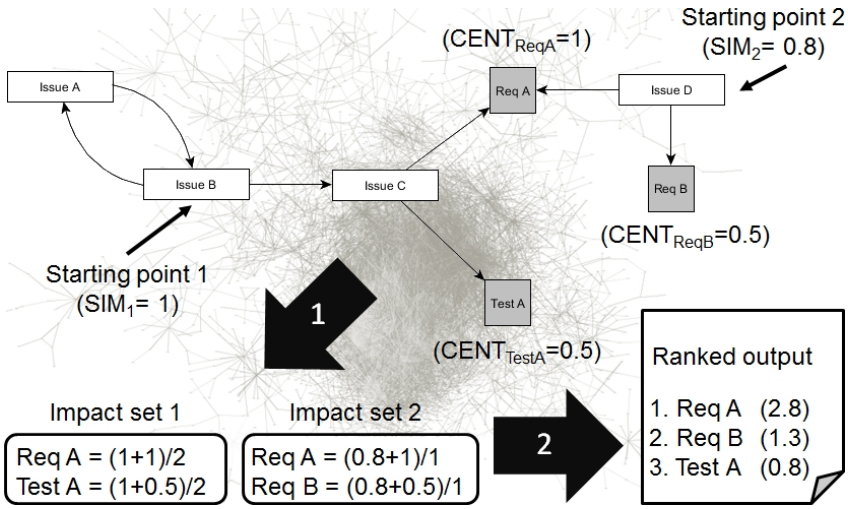

Fig. 1.13: Example calculation of impact recommendation in a simplified knowledge base.

a maximum of three "related issue" links from a starting point are followed. The searches from each starting point results in an *impact set* of possibly impacted artifacts (SET_i), which are then used as input to the ranking engine.

In the knowledge base, ImpRec calculates centrality measures for each artifact ($CENT_x$). As the artifacts ImpRec recommends are only link targets, i.e., they have no outgoing links themselves, only the number of incoming edges are considered for this calculation. The resulting *indegree centralities* are normalized between 0 and 1 and also used as input to the ranking engine.

The ranking of the artifacts in the set of recommendations is based on their individual weights. The general equation for calculating the weight of a recommended artifact, $Weight(ART_x)$, is the following:

$$Weight(ART_x) = \sum_{\substack{ART_x \in Set_i \\ 1 \leq i \leq 5}} \frac{a * SIM_i + b * CENT_x}{c * LEVEL} \tag{1.1}$$

where SIM_i is the similarity of the issue report that was used as starting point when identifying ART_x, $LEVEL$ is the number of related issue links followed from the starting point to identify ART_x, and $CENT_x$ is the centrality measure of ART_x in the knowledge base. a, b, and c are constants that enables tuning for context-specific improvements.

Fig. 1.13 illustrates the calculation of an example recommendation in a simplified knowledge base where $a = b = c = 1$. First, content-based filtering based on textual features finds two starting points in the network, with normalized similarity values equals to 1 and 0.6 respectively. Starting point 1 (Issue B in the figure) does not have any direct links to any impacted artifacts, however a breadth-first search identifies *Req A* and *Test A* through Issue C. Both Req A and Test A are added to *Impact set 1*. Starting point 2 (Issue D in the figure) on the other hand has direct links to impacted artifacts, thus *Req A* and *Req B* are both added to *Impact set2*. The weight of the individual artifacts in each impact set is then calculated according to Equation 1.1, considering textual similarities ($Sim_1 = 1$ and $Sim_2 = 0.8$), node centralities ($Cent_{ReqA} = 1$, $Cent_{ReqB} = 0.5$, and $Cent_{TestA} = 0.5$), and the number of links followed between issue reports (2 for artifacts in impact set 1, 1 for artifacts in impact set 2).

Finally, the weights of the artifacts in the two impact sets are summarized (presented in the lower right part of Fig. 1.13). *ReqA* is included in both impact sets 1 and 2, thus its final weight in the list of ranked recommendations is $1 + 1.8 = 2.8$. The weights of ReqB and TestA are lower, 1.3 and 0.8 respectively, which results in lower rankings.

1.5.4 Step 4: Presenting Recommendations

The developers at the case company all work in an MS Windows environment, thus we have not considered ImpRec for multiple platforms. The current version of ImpRec is written as a light-weight standalone tool in .NET, and supports basic user interaction. Fig. 1.14 shows the ImpRec user interface. The leftmost frame is used to input the description of the issue report for which the developer is conducting an impact analysis. Based on the textual content in the text box, recommendations are calculated when the developer clicks the “Impact?” button.

The center and rightmost frames in ImpRec are used to present the calculated recommendations. In the center frame, the issue reports ImpRec recommends a developer to investigate are presented, i.e., similar to how RSSEs for duplicate detection (see Section 1.2) typically report candidate duplicates. In the rightmost frame, ImpRec lists the most likely impacted artifacts. The content in both the frames are sorted, to ensure the recommendations with the highest confidence are presented first.

For ImpRec to be truly integrated in the impact analysis process, the tool needs to be integrated in the working environment of the developers. As the impact analyses are conducted with issue reports as starting points, and the outcome is stored as attachments to issue reports, the natural approach is to develop an ImpRec plug-in to the issue repository used in the organization.

Early evaluations using 90% of the dataset for training and the last 10% as a test set show that about 40% of the previous impact could be identified by ImpRec. The highest possible recall for this dataset is thus relatively low. However, the ranking

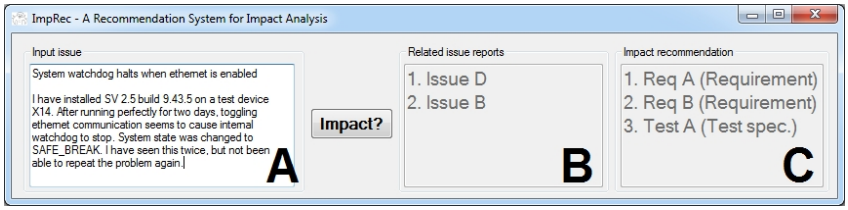

Fig. 1.14: Prototype user interface for ImpRec. A is used for inputting the natural language issue report. Output is presented to the right, possibly related issues in B, and suggestions for impacted software artifacts in C.

function appears promising as ImpRec achieved a recall@5 of 30%. This result was positively acknowledged by developers in the targeted organization as a quick way to find a starting point in the impact analysis work task with a manageable number of **false positives**. Moreover, recommendations regarding related issue reports were appreciated, and considered more practical than the current search function in the issue repository. As a final note, developers stressed the need to keep the knowledge base updated. Test runs on new issue reports using the 18 month old knowledge base in the prototype revealed that ImpRec's recommendations did not reflect the latest work in the organization. While this finding is not surprising due to the dynamics of software engineering, it highlights the need for future work both on how to automatically update the knowledge base and how to identify obsolete content.

1.6 Concluding Remarks

Issue reports are primarily used as 'batons' in the communication between different actors in the software development process, whether in house or open source. This chapter demonstrates that there are several aspects to issue reports, that benefit from recommendation systems. First, we presented different approaches to recommending duplicate issue reports, either for reducing information overload by merging duplicates, or to provide more information of the issue at hand, e.g., for other recommendation systems. Evaluations indicate recall of 40-93% when considering the top 10 recommendations. While the results are promising, further research is needed to understand variations and tailoring to specific contexts. Second, we presented two approaches to recommending traces to software engineering artifacts, using issue reports as an information 'hub', implemented in the tools Hipikat and ImpRec. Both approaches have shown potential of being useful for practitioners to help navigating a continuously expanding software project. Still, they have to be better integrated into development environments, and heuristics for the search methods have to be improved to make them feasible for everyday practice.

Acknowledgements This work was funded by the Industrial Excellence Center EASE – Embedded Applications Software Engineering[4]. Thanks go to the developers of Hipikat for providing related material.

References

1. Anvik, J., Hiew, L., Murphy, G.: Coping with an open bug repository. In: Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange, pp. 35–39 (2005)
2. Anvik, J., Murphy, G.: Reducing the effort of bug report triage: Recommenders for development-oriented decisions. Transactions on Software Engineering and Methodology **20**(3), 10:1–10:35 (2011)
3. Begel, A., Phang, K.Y., Zimmermann, T.: Codebook: discovering and exploiting relationships in software repositories. In: Proceedings of the 32nd International Conference on Software Engineering, pp. 125–134 (2010)
4. Bettenburg, N., Premraj, R., Zimmermann, T., Sunghun, K.: Duplicate bug reports considered harmful... really? In: Proceedings of the International Conference on Software Maintenance, pp. 337–345 (2008)
5. Boehm, B., Basili, V.: Software defect reduction top 10 list. Computer **34**(1), 135–137 (2001)
6. Borg, M.: Findability through traceability: A realistic application of candidate trace links? In: Proceedings of the 7th International Conference on Evaluating Novel Approaches to Software Engineering, pp. 173–181 (2012)
7. Borg, M., Gotel, O., Wnuk, K.: Enabling traceability reuse for impact analyses: A feasibility study in a safety context. In: Proceedings of the 7th International Workshop on Traceability in Emerging Forms of Software Engineering (2013)
8. Borg, M., Pfahl, D., Runeson, P.: Analyzing networks of issue reports. In: Proceedings of the 17th European Conference on Software Maintenance and Reengineering, pp. 79–88 (2013)
9. Brandes, U., Eiglsperger, M., Lerner, J., Pich, C.: Graph markup language (GraphML). In: Tamassia, R. (ed.) Handbook of Graph Drawing and Visualization. CRC Press (2010)
10. Cubranic, D.: Automatic bug triage using text categorization. In: Proceedings of the 16th International Conference on Software Engineering & Knowledge Engineering, pp. 92–97 (2004)
11. Cubranic, D.: Project history as a group memory: Learning from the past. PhD thesis, University of British Columbia, Dept. of Computer Science (2004)
12. Cubranic, D., Murphy, G.: Hipikat: Recommending pertinent software development artifacts. In: Proceedings of the 25th International Conference on Software Engineering, pp. 408–418 (2003)
13. Cubranic, D., Murphy, G., Singer, J., Booth, K.: Hipikat: A project memory for software development. Transaction on Software Engineering **31**(6), 446–465 (2005)
14. Deerwester, S., Dumais, S., Furnas, G., Landauer, T., Harshman, R.: Indexing by latent semantic analysis. Journal of the American Society for Information Science **41**(6), 391–407 (1990)
15. Edmunds, A., Morris, A.: The problem of information overload in business organisations: A review of the literature. International Journal of Information Management **20**(1), 17–28 (2000)
16. Eppler, M., Mengis, J.: The concept of information overload: A review of literature from organization science, accounting, marketing, MIS, and related disciplines. The Information Society **20**(5), 325–344 (2004)
17. Gegick, M., Rotella, P., Xie, T.: Identifying security bug reports via text mining: An industrial case study. In: Proceedings of the 7th Working Conference on Mining Software Repositories, pp. 11–20 (2010)

[4] http://ease.lth.se

18. Gethers, M., Dit, B., Kagdi, H., Poshyvanyk, D.: Integrated impact analysis for managing software changes. In: Proceedings of the 34th International Conference on Software Engineering, pp. 430–440 (2012)
19. Guo, P., Zimmermann, T., Nagappan, N., Murphy, B.: Characterizing and predicting which bugs get fixed: an empirical study of Microsoft Windows. In: Proceedings of the 32nd International Conference on Software Engineering, pp. 495–504 (2010)
20. Jalbert, N., Weimer, W.: Automated duplicate detection for bug tracking systems. In: Proceedings of the International Conference on Dependable Systems and Networks With FTCS and DCC, pp. 52–61 (2008)
21. Johnson, J., Dubois, P.: Issue tracking. Computing in Science Engineering **5**(6), 71–77 (2003)
22. Jonsson, L., Broman, D., Sandahl, K., Eldh, S.: Towards automated anomaly report assignment in large complex systems using stacked generalization. In: Proceedings of the 5th International Conference on Software Testing, Verification and Validation, pp. 437–446 (2012)
23. Just, S., Premraj, R., Zimmermann, T.: Towards the next generation of bug tracking systems. In: Proceedings of the 2008 Symposium on Visual Languages and Human-Centric Computing, pp. 82–85 (2008)
24. Karr-Wisniewski, P., Lu, Y.: When more is too much: Operationalizing technology overload and exploring its impact on knowledge worker productivity. Computers in Human Behavior **26**(5), 1061–1072 (2010)
25. Keim, D., Mansmann, F., Schneidewind, J., Thomas, J., Ziegler, H.: Visual analytics: Scope and challenges. In: Simoff, S., Böhlen, M., Mazeika, A. (eds.) Visual Data Mining, no. 4404 in Lecture Notes in Computer Science, pp. 76–90. Springer Berlin Heidelberg (2008)
26. Klevin, A.: People, process and tools: A study of impact analysis in a change process. Master thesis, Lund University, http://sam.cs.lth.se/ExjobGetFile?id=467 (2012)
27. Lamkanfi, A., Demeyer, S., Giger, E., Goethals, B.: Predicting the severity of a reported bug. In: Proceedings of the 7th Working Conference on Mining Software Repositories, pp. 1–10 (2010)
28. Liu, T.Y.: Learning to Rank for Information Retrieval. Springer (2011)
29. Manning, C., Raghavan, P., Schutze, H.: Introduction to Information Retrieval. Cambridge University Press (2008)
30. Menzies, T., Marcus, A.: Automated severity assessment of software defect reports. In: Proceedings of the 24th International Conference on Software Maintenance, pp. 346–355 (2008)
31. Mierswa, I., Wurst, M., Klinkenberg, R., Scholz, M., Euler, T.: YALE: rapid prototyping for complex data mining tasks. In: Proceedings of the 12th ACM SIGKDD International Conference on Knowledge discovery and data mining, pp. 935–940 (2006)
32. Mishra, N., Schreiber, R., Stanton, I., Tarjan, R.: Clustering social networks. In: Bonato, A., Chung, F. (eds.) Algorithms and Models for the Web-Graph, no. 4863 in Lecture Notes in Computer Science, pp. 56–67. Springer (2007)
33. Morville, P.: Ambient Findability: What We Find Changes Who We Become. O'Reilly Media (2005)
34. Pol, M., Teunissen, R., van Veenendaal, E.: Software testing: A guide to the TMap approach. Pearson Education (2002)
35. Porter, M.F.: An algorithm for suffix stripping. Program: Electronic Library and Information Systems **14**(3), 130–137 (1980)
36. Raja, U.: All complaints are not created equal: text analysis of open source software defect reports. Empirical Software Engineering **18**(1), 117–138 (2013)
37. Reis, C.R., Pontin de Mattos Fortes, R.: An overview of the software engineering process and tools in the Mozilla project. In: Proceedings of the Workshop Open Source Software Development, pp. 155–175 (2002)
38. Robertson, S., Zaragoza, H.: The probabilistic relevance framework: BM25 and beyond. Foundation and Trends in Information Retrieval **3**(4), 333–389 (2009)
39. Runeson, P., Alexandersson, M., Nyholm, O.: Detection of duplicate defect reports using natural language processing. In: Proceedings of the 29th International Conference on Software Engineering, pp. 499–510 (2007)

40. Russell, S., Norvig, P.: Artificial Intelligence: A Modern Approach. 3rd edn. Prentice Hall Press, Upper Saddle River, NJ, USA (2009)
41. Seichter, D., Dhungana, D., Pleuss, A., Hauptmann, B.: Knowledge management in software ecosystems: software artefacts as first-class citizens. In: Proceedings of the 34th International Conference on Software Engineering, pp. 119–126 (2012)
42. Serrano, N., Ciordia, I.: Bugzilla, ITracker, and other bug trackers. IEEE Software **22**(2), 11–13 (2005)
43. Sim, S., Holt, R.: The ramp-up problem in software projects: A case study of how software immigrants naturalize. In: Proceedings of the 20th International Conference on Software Engineering, pp. 361–370 (1998)
44. Sun, C., Lo, D., Khoo, S.C., Jiang, J.: Towards more accurate retrieval of duplicate bug reports. In: Proceedings of the 26th International Conference on Automated Software Engineering, pp. 253–262 (2011)
45. Sun, C., Lo, D., Wang, X., Jiang, J., Khoo, S.C.: A discriminative model approach for accurate duplicate bug report retrieval. In: Proceedings of the 32nd International Conference on Software Engineering, pp. 45–54 (2010)
46. Sureka, A., Jalote, P.: Detecting duplicate bug report using character n-gram-based features. In: Proceedings of the 17th Asia Pacific Software Engineering Conference, pp. 366–374 (2010)
47. Terveen, L., Hill, W.: Beyond recommender systems: Helping people help each other. In: Human-Computer Interaction in the New Millennium, pp. 487–509. Addison-Wesley (2001)
48. Wang, X., Zhang, L., Xie, T., Anvik, J., Sun, J.: An approach to detecting duplicate bug reports using natural language and execution information. In: Proceedings of the 30th International Conference on Software Engineering, pp. 461–470 (2008)
49. Weiss, C., Premraj, R., Zimmermann, T., Zeller, A.: How long will it take to fix this bug? In: Proceedings of the 4th International Workshop on Mining Software Repositories (2007)
50. Zantout, H., Marir, F.: Document management systems from current capabilities towards intelligent information retrieval: An overview. International Journal of Information Management **19**(6), 471–484 (1999)