Assigning Bug Reports using a Vocabulary-Based Expertise Model of Developers*

Dominique Matter, Adrian Kuhn, Oscar Nierstrasz

Software Composition Group
University of Bern, Switzerland
http://scg.unibe.ch

Abstract

For popular software systems, the number of daily submitted bug reports is high. Triaging these incoming reports is a time consuming task. Part of the bug triage is the assignment of a report to a developer with the appropriate expertise. In this paper, we present an approach to automatically suggest developers who have the appropriate expertise for handling a bug report. We model developer expertise using the vocabulary found in their source code contributions and compare this vocabulary to the vocabulary of bug reports. We evaluate our approach by comparing the suggested experts to the persons who eventually worked on the bug. Using eight years of Eclipse development as a case study, we achieve 33.6% top-1 precision and 71.0% top-10 recall.

1. Introduction

Software repositories of large projects are typically accompanied by a bug report tracking system. In the case of popular open source software systems, the bug tracking systems receive an increasing number of reports daily. The task of triaging the incoming reports therefore consumes an increasing amount of time [3]. One part of the triage is the assignment of a report to a developer with the appropriate expertise. Since in large open source software systems the developers typically are numerous and distributed, finding a developer with a specific expertise can be a difficult task.

Expertise models of developers can be used to support the assignment of developers to bug reports. It has been proposed that tasks such as bug triage can be improved if an externalized model of each programmer's expertise of the code base is available [12]. Even though approaches for expertise models based on software repository contributions are available, existing recommendation systems for bug assignment typically use expertise models based on previous bug reports only [4, 8, 29, 25, 20]. Typically a classifier is trained with previously assigned bug reports, and is then used to classify and assign new, incoming bug reports.

In this paper, we propose an expertise model based on source code contributions and apply in it a recommendation system that assigns developers to bug reports. We compare vocabulary found in the `diff`s of a developer's contributions with the vocabulary found in the description of a bug report. We then recommend developers whose contribution vocabulary is lexically similar to the vocabulary of the bug report.

We implemented our approach as a prototype called DEVELECT[1], and evaluate the recommendation system using the Eclipse project as a case study. We develop and calibrate our approach on a training set of bug reports. Then we report the results of evaluating it on a set of reports of the remaining case study.

The contributions of this paper are as follows:

- We propose a novel expertise model of developers. The approach is based on the vocabulary found in the source code contributions of developers.
- We propose a recommendation system that applies the above expertise model to assign developers to bug reports. We evaluate the system using eight years of Eclipse development as a case study.
- We report on the decay of developer expertise, observed when calibrating our approach. We apply

* In Proceedings of the 6th IEEE Working Conference on Mining Software Repositories (MSR 2009). pp. 131-140. DOI: 10.1109/MSR.2009.5069491

1 DEVELECT is open source, written in Smalltalk, and available at `http://smallwiki.unibe.ch/develect`. The name *develect* is a portmanteau word of *developer* and *dialect*.

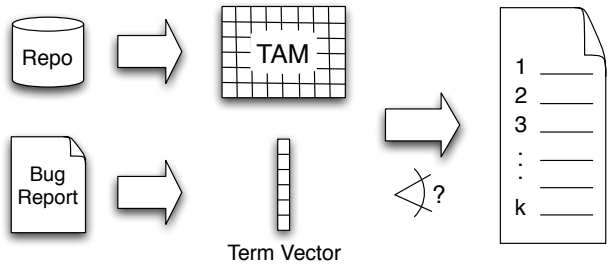

Figure 1. Architecture of the Develect recommendation system: (left) bug reports and versioning repositories are processed to produce term vectors and term-author-matrices; (right) the cosine angle between vector and matrix columns is taken to rank developers in the suggested list of experts.

two weighting schemes to counter this effect.

The remainder of this paper is structured as follows: In Section 2 we present our expertise model and its application for bug assignment. In Section 3 we provide the algorithm to build our expertise model. In Section 4 we evaluate our expertise model using Eclipse as a case study. In Section 5 we discuss calibration of our approach and threats to validity. In Section 6 we discuss related work. In Section 7 we conclude with remarks on further applications of our expertise model.

2. Our Approach in a Nutshell

In this paper we present i) the construction of an expertise model of developers and ii) the application of a recommendation system that uses this expertise model to automatically assign developers to bug reports. Our approach requires a versioning repository to construct the expertise model as well as a bug tracking facility to assign developers to bug reports. We realized the approach as a prototype implementation, called DEVELECT.

In his work on automatic bug report assignment, John Anvik proposes eight types of information sources to be used by a recommendation system [3]. Our recommendation system focuses on three of these types of information: the information of the textual description of the bug (type 1), the one of the developer who owns the associated code (type 6), and the information of the list of developers actively contributing to the project (type 8). We refine information type 6 to take into account the textual content of the code owned by a developer. Our recommendation system is based on an expertise model of the developer's source code contributions. For each developer, we count the textual word frequencies in their change sets. This includes deleted code and context lines, assuming that any kind of change (even deletions) requires developer knowledge and thus familiarity with the vocabulary.

Our system currently does not consider the component the bug is reported for (type 2), the operation system that the bug occurs on (type 3), the hardware that the bug occurs on (type 4), the version of the software the bug was observed for (type 5), or the current workload of the developers (type 7). Information types 2–4 are indirectly covered, since textual references to the component, operation system, or hardware are taken into account when found in bug reports or source code. Information of type 7 is typically not publicly available for open source projects and thus excluded from our studies. Furthermore, we deliberately disregard information of type 5, since developer knowledge acquired in any version pre-dating the bug report might be of use.

Given a software system with a versioning repository, the creation of the expertise model works as illustrated in Figure 1:

1. For each contributor to the versioning repository, we create an empty bag of words.
2. For each contribution to the versioning repository, we create a `diff` of all changed files and count the word frequencies in the diff files. We assign the word frequencies to the contributor's bag of words.
3. We create a term-author-matrix $M_{n \times m}$, where n is the global number of words and m the number of contributors. Each entry $m_{i,j}$ equals the frequency of the word t_i in the contributions of the contributor a_j.

Given the above term-author-matrix and a bug tracking facility, the assignment of developers works as follows:

1. We count the word frequencies in the bug report and create a query vector of length n where v_i equals the frequency of word t_i in the bug report.
2. For each developer in the term-author-matrix, we take the cosine of the angle between the query vector and the developer's column of the matrix.
3. We rank all developers by their lexical similarity and suggest the top k developers.

To evaluate our approach we use Eclipse as a case study. We train our system with weekly summaries of

all CVS commits from 2001 to 2008, and use the resulting expertise-model to assign developers to bug reports. We evaluate precision and recall of our approach by comparing the suggested developers to the persons who eventually worked on the bug and its report.

For example, for a report that was submitted in May 2005, we would train our system with all commits up to April. Then we would evaluate the suggested developers against those persons who worked on the bug and its report in May or later on to see if they match.

3. The Develect Expertise Model

Given the natural language description of a bug report, we aim to find the developer with the best expertise regarding the content of the bug report. For this purpose, we model developer expertise using their source code contributions.

Developers gain expertise by either writing new source code or working on existing source code. Therefore we use the vocabulary of source code contributions to quantify the expertise of developers. Whenever a developer writes new source code or works on existing source code, the vocabulary of mutated lines (and surrounding lines) are added to the expertise model. These lines are extracted from the version control system using the `diff` command.

Natural language documents differ from source code in their structure and grammar, thus we treat both kind of documents as unstructured bags of words. We use Information Retrieval techniques to match the word frequencies in bug reports to the word frequencies in source code. This requires that developers use meaningful names *e.g.* for variables and methods, which is the case given modern naming conventions [17].

Given a bug report, we rank the developers by their expertise regarding the bug report. The expertise of a developer is given by the lexical similarity of his vocabulary to the content of the bug report.

3.1. Extracting the Vocabulary of Developers from Version Control

To extract the vocabulary of a specific developer we must know which parts of the source code have been authored by which developer.

We extract the vocabulary in two steps, first building a Chronia model of the entire repository [13] and then collecting word frequencies from the `diff` of each revisions. The `diff` command provides a line-by-line summary of the changes between two versions of the same file. The diff of a revision summarizes the changes made to all files that changed in that revision of the repository. The identifier names and comments that appear on these lines give us evidence of the contributor's expertise in the system. Thus, we add the word frequencies in a revision's `diff` to the expertise model of the contributing developer.

Word frequencies are extracted as follows: the lines are split into sequences of letters, which are further split at adjacent lower- and uppercase letters to accommodate the common *camel case* naming convention. Next stopwords are removed (*i.e.* common words such as *the*, *and*, etc). Eventually stemming is applied to remove the grammatical suffix of words.

The comment message associated with a revision is processed in the same way, and added to the expertise model of the contributing developer as well.

3.2. Modeling the Expertise of Developers as a Term-Author-Matrix

We store the expertise model in a matrix that correlates word frequencies with developers. We refer to this matrix as a term-author-matrix. Although, technically it is a term-document-matrix where the documents are developers. (It is common in Information Retrieval to describe documents as bags of words, thus our model is essentially the same, with developers standing in for documents.)

The term-author-matrix has dimension $n \times m$, where n is the global number of words and m the number of contributors, that is developers. Each entry $m_{i,j}$ equals the frequencies of the word t_i summed up over all source code contributions provided by developer a_j.

We have found that results improve if the term-author-matrix is weighted as follows:

- *Decay of Vocabulary.* For each revision, the word frequencies are weighted by a decay factor that is proportional to the age of the revision. In the Eclipse case study, best results are obtained with a weighting of 3% per week (which accumulates to 50% per half year and 80% per annum). Please refer to Section 5 for a detailed discussion.

3.3. Assign Developers to Bug Reports regarding their Expertise

To assign developers to bug reports, we use the bug report's textual content as a search query to the term-author-matrix. Given a Bugzilla[2] bug report, we count the word frequencies in its textual content. In particular we process both short and all long descriptions

2 http://www.bugzilla.org

(threats to validity see Section 5). We disregard attachments that are Base-64 encoded, such as attached images, as well as fields that refer to persons. From the extracted word frequencies, we create a *term vector* that uses the same word indices as the term-author-matrix. We then compute the lexical similarity between two term vectors by taking the cosine of the angle between them. The similarity values range from 1.0 for identical vectors to 0.0 for vectors without shared terms. (Negative similarity values are not possible, since word-frequencies cannot be negative either.)

We compare the term vector of the bug report with the term vectors of all developers (*i.e.* the columns of the term-author-matrix) and create a ranking of developers. For the assignment of bug reports to developers, a suggestion list of the top-k developers with the highest lexical similarities is then provided.

We have found that the results improve if the term-author-matrix is further weighted as follows:

- *Inactive Developer Penalty.* If a developer has been inactive for more than three months, the lexical similarity is decreased by a penalty proportional to the time since his latest contribution. In the Eclipse case study, best results are obtained with a penalty of 0.2 per annum. Please refer to Section 5 for a detailed discussion.

4. Case Study: Eclipse platform

To evaluate our approach we take Eclipse[3] as a case study. Eclipse is a large open source software project with numerous active developers. Eclipse has been developed over several years now. Therefore, its version repository contains a great deal of source code developed by many different authors. Furthermore, Eclipse uses Bugzilla as its bug tracking system, storing bug reports dating back to nearly the beginning of the project. We evaluate our results by comparing the top-k developers with the persons who eventually worked on the bug report.

Our case study covers the Eclipse project between April 22, 2001, and November 9, 2008. The source code contributions of Eclipse are stored in a CVS repository[4], the bug reports in a Bugzilla database[5]. This represents almost eight years of development, including 130,769 bug reports and 162,942 global revisions (obtained from CVS's file versions using a sliding time-window of 2 minutes [30]). During this time, 210 developers contributed to the project.

3 http://www.eclipse.org/eclipse

4 :pserver:anonymous@dev.eclipse.org/cvsroot/eclipse

5 https://bugs.eclipse.org/bugs

4.1. Setup of the Case Study

The setup of the Eclipse case study consists of two different parts. The first part is about the change database, here we use all changes before the actual bug report. The second part is about the bug database, here we make 10 partitions of which two are used in this case study. We process and evaluate both parts in weekly iterations as follows:

- We create a DEVELECT expertise model based on contributions between April 22, 2001, and the last day of the previous week.
- We generate a suggested list of the top-10 experts for all bug reports submitted in the current week.
- We evaluate precision and recall by comparing the suggestion list with the developers who, between the first day of the next week and November 9, 2008, eventually worked on the bug report.

For example, for a bug report submitted on May 21, 2005, we would train our system with all commits between April 22, 2001 and May 15, 2005, and then evaluate the list of suggested experts against the set of developers who, between May 23, 2005, and November 9, 2008, eventually handled the bug report.

We use systematic sampling to create 10 partitions of 13,077 bug reports (ordered by time) that span the entire time of the project. One partition is used as training set for the development of our approach, and another partition is used as validation set to validate our approach. We applied the approach to the validation set only after all implementation details and all calibration parameters had been finally decided on. The other partitions remain untouched for use as validation set in future work.

In this section, we report on our results obtained on the validation partition #2. In Section 5 we report on results obtained from the training partition #1 while calibrating the approach.

4.2. Precision and Recall

We evaluate our approach by comparing the suggested list of experts with the developers who eventually worked on the bug report. We report on precision and recall for different sizes of suggested lists, between $k = 1$ and $k = 10$. Comparing our results to the persons who eventually worked on the bug is not optimal. For example, the person could have been assigned to the bug report by some factor other than expertise. Obtaining a better list of experts requires manual interrogation of the development team.

Precision is the percentage of suggested developers who actually worked on the bug report. Recall is the percentage of developers who worked on the bug who were actually suggested. It is typical for Information Retrieval approaches that there is a trade-off between precision and recall.

Getting the list of persons who eventually worked on a bug report is tricky. The *assigned-to* field does not always denote a person who eventually solved the bug report [29, 4, 5]. Therefore we compare our results against three configurations (C1–C3) of bug-related persons:

1. Developers who committed an actual bug fix to the software repository. For Eclipse, this information is not stored in the Bugzilla database, therefore we must rely on information from CVS commit messages. In the validation set, this information is provided for 14.3% of the bug reports only. This configuration evaluates how well we perform in suggesting experts who provide actual bug fixes.
2. Persons given by the *assigned-to* field or a *who* field of the bug report. That is, the eventual assignee (if this is a developer) and all developers who ever discussed the bug in the comment section of the report. This configuration evaluates how well we perform in suggesting experts who are capable of understanding and discussing the bug. Note that resolving a bug is not limited to providing code fixes; often the discussion is just as important to the resolution of the bug.
3. As in configuration #2, but additionally including the person identified by the *reporter* field, if the reporter is a developer, *i.e.* has a CVS login. This reflects the fact that bugs are sometimes resolved by the same people who find and track them.

Please refer to Section 5 for further discussion of the above configurations and their threats to validity.

4.3. Results

Figure 2 illustrates the results of the Eclipse case study. We compare lists of suggested persons of list size 1 to 10 with set of "bug related persons" as given by the three configurations (C1-3) above.

The figure illustrates that recall and precision of configuration C1 are better than C2 and C3. When comparing the suggested list to the bug fixing person (C1) we achieved the best score with 33.6% top-1 precision only and 71.0% top-10 recall. Comparing of the suggested list to all related persons (C3) we score 26.0% top-1 precision and 54.2% top-10 recall. When excluding the reporter of the bug report (C2) from the suggested list scores are at 22.3% top-1 precision and 57.4% top-10 recall.

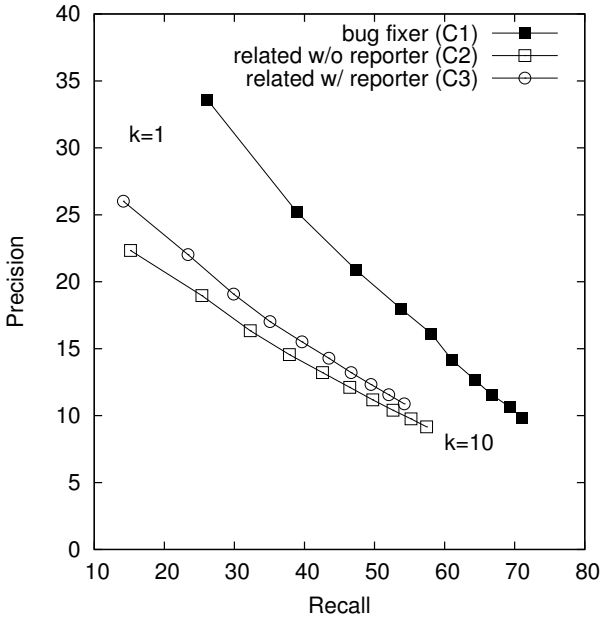

Figure 2. Recall and precision of the Eclipse case study: configuration C1 scores best with 33.6% **top-1 precision and** 71.0% **top-10 recall.**

The fact that configuration C3 scores slightly better then C2 indicates that bug reporters are sometimes indeed experts regarding the reported bug and thus should be considered when triaging bug reports. We can thus conclude that an automatic assignment system should provide to the triaging person a suggested list of people that may include the reporter.

5. Discussion

In this section, we first discuss the calibration of our approach and then cover threats to validity.

Compared to related work, an advantage of our approach is that we do not require a record of previous bug reports. We are able to recommend developers who did not work on bugs previously. For example, we do not require that developers have worked on at least 10 resolved bug reports. On the other hand, our approach requires at least a half to one year of versioning history in order to suggest developers.

One obvious threat to validity is the quality of our evaluation benchmark. We compare our suggested list against the developers who eventually worked on the bug report and assume that these are the top experts. For example, the bug report could have been assigned to a developer by some factor other than expertise.

Settings	Precision	Recall
reference	19.7	42.6
weighted `diff`	18.5	41.1
desc. fields only	16.7	37.7
with LSI	16.5	35.1
decay 0.03	20.5	43.2
decay 0.05	20.4	42.4
decay 0.10	20.5	41.5
decay 0.20	18.0	38.0
penalty 0.1	24.5	50.9
penalty 0.2	24.8	51.6
penalty 0.3	24.8	51.9
final calibration	26.2	54.6

Table 1. Summary of calibration of training set, for each settings top-1 precision and top-10 recall are given.

This threat is hard to counter. A better list of experts can be obtained by manual interrogation of the development team, but even this is not a golden oracle.

Another threat to validity is that we use all long descriptions, including comments, of a bug report as information source. This may include discussions that happened after the bug has been eventually assigned or fixed, information which is not actually available when doing initial bug triage. This might impact the performance of our approach.

5.1. On the Calibration of Develect

We used 1/10th of the Eclipse case study as a training set to calibrate our approach. The calibration results are summarized in Table 1.

The table lists top-1 precision and top-10 recall. On the first row, we list the results before calibration ($p = 19.7\%, r = 42.6\%$), and on the last row the results of the final calibration. Please note that the final results on the training set are slightly better than the results reported in subsection 4.3 for the validation set.

The output of the `diff` command consists of added, removed, and context lines. We experimented with different weightings for these lines (weighted `diff` in Table 1). However, we found that weighting all lines the same yields best results.

As Bugzilla bug reports consist of many fields, we experimented with different selections of fields. We found that taking short and long descriptions ("desc. fields only" in Table 1) yields worse results than selecting all fields except those that refer to persons, or Base64 encoded attachments.

We also experimented with Latent Semantic Indexing (LSI), an Information Retrieval technique typically used in search engines that detects polysemy and synonymy by statistical means [10]. However, we found that LSI yields poor results ("with LSI" in Table 1).

5.2. On the Decay of Vocabulary

In our experiments, we found that developer expertise decays over time. In our approach we introduced two weighting schemes to counter this effect:

- *Decay of Vocabulary.* For each revision, the word frequencies are weighted by a decay factor that is proportional to the age of the revision. Developers change their interests and by doing so change their expertise and vocabulary. To take such a shift into account, we fade the old vocabulary out bit by bit every week, so that the newest words are weighted slightly more than older ones. With time, the old words eventually fade out completely.
- *Inactive Developer Penalty.* If a developer has been inactive for more than three months, the lexical similarity is decreased by a penalty proportional to the time since his latest contribution to the software system. Inactive developers will most likely not resolve bugs anymore. In order to only recommend currently active developers (we assign bug reports during a period of eight years), developers who did not recently make a change to the software system receive a penalty.

Figure 3 illustrates the effect of these settings. On the left, the unbroken curve illustrates the decreasing quality of unweighted results, whereas the dotted curve shows the results obtained with weighting. Even though results improved significantly, the quality of the weighted results still slightly decreases over time. We cannot fully explain this effect; it may be due to the increasing complexity of Eclipse as a project, or perhaps the lack of mappings from CVS logins to persons (see subsection 5.4) in the early years of the project impacts the results. Another cause for the trend in the most recent year, *i.e.* 2008, might be that the list of persons that worked on a bug is not yet completely known to us, which may impact the evaluation.

In the middle of Figure 3, precision and recall for different *decay of vocabulary* settings are given. On the right, precision and recall for different *inactive developer penalty* settings are given.

Decay of vocabulary scores best results with a weighting of 3% per week (which accumulates to 50% per half year and 80% per annum). This shows that implementation expertise acquired one year ago or earlier does not help in assigning developers to bug reports.

The inactive developer setting scores best results with a penalty of 0.2 per annum. As a result of the penalty, the matching score of a developer who has been inactive for a year is decreased. The matching

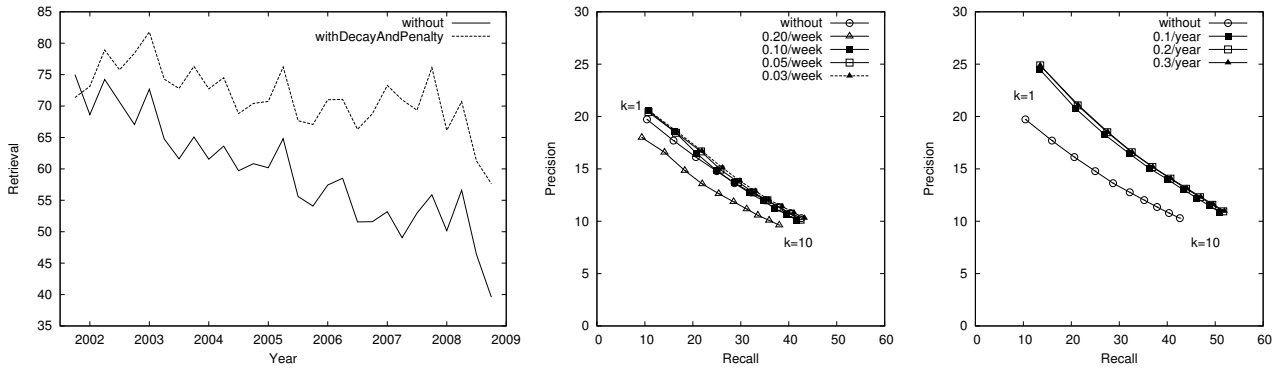

Figure 3. Decay of vocabulary: (left) decreasing quality of unweighted results, compared to results with decay of vocabulary and inactive developer penalty settings, (middle) precision and recall for different decay of vocabulary settings, (right) precision and recall for different inactive developer penalty settings.

scores are the lexical similarity values (between 1.0 and 0.0). Decreasing this value by 0.1 or more is typically enough to exclude a result from the top-10 list.

Interestingly, any penalty above 0.1 is better than none. The results obtained with different penalty values are almost the same. Please note, that even though the penalty removes inactive developers from the top of the suggested list, their vocabulary is not lost. The results reported for the calibration of the penalty do not make use of vocabulary decay. If a developer becomes active again, all his past expertise is reactivated as well. Thus, we use a moderate penalty of 0.2 in combination with a decay of 3% as the final calibration settings.

5.3. On Grouping Diffs by Week

To cope with the size of our case study, we decided to run weekly iterations rather than fine-grained iterations per bug report and revision. This reduced the time complexity from over 160,000 iterations down to 394 weekly iterations.

Grouping diffs by both author *and* week introduces the following threats to validity: If vocabulary is added and removed within the same week, it does not add to the developer's expertise. In the same way, if a file is added and removed within the same week, it is not taken into account at all. If bug reports are submitted late in the week, we might miss developers who acquired novel expertise early in the week.

If several authors worked on the same file, we cannot tell their weekly contributions apart. In this case, we weight the word frequencies by $\frac{1}{\sqrt{n}}$, where n is the number of co-developers, and assign the weighted frequencies to all co-developers. For the Eclipse case study, this applies to 3.6% of weekly file changes.

5.4. On other Threats to Validity

Establishing an identity relationship between CVS logins and people mentioned in bug reports is not trivial. The developer information in the CVS log is provided as a mere login name. People mentioned in a Bugzilla bug report are listed with their email address and sometimes additionally with their first and last name. For Eclipse, the mapping between logins and active developers can be found on the Eclipse website[6]. However, the list of names of the former Eclipse developers does not include their corresponding logins[7]. We could not map 17.1% of the CVS logins and had thus to exclude 2.7% of the bug reports from our evaluation.

Information about copy patterns is not available in CVS. Bulk renaming of files appears in the change history of CVS as bulk removal of files followed by bulk addition of files. Given our current implementation of DEVELECT, this may lead to an incorrect acquisition of developer knowledge, since the entire vocabulary of the moved files is assigned to the developer who moved the files. We are thus in good shape to further improve our results by using a copy pattern detection approach [9].

6. Related Work

There was previous work done on mining developer expertise from source code. However, if such an expertise model was applied to assist in the triage of bug reports, it required an explicit linkage between bug reports and source files [5]. Most existing recommendation systems for the assignment of developers to bug reports use other expertise models, as *e.g.* mod-

6 http://www.eclipse.org/projects/lists.php

7 http://www.eclipse.org/projects/committers-alumni.php

els based on previous reports. Thus, the main differences between our approach and other recommendation systems are: First, our system is trained on the source code repository rather than on previous bug reports. Second, for the recommendation of a developer suitable to handle a bug report, we do not need a linkage between the bug tracking system and the software repository. As a collorary, our approach is a) not dependent on the quality of previous bug reports, and b) can also be used to assign developers that have not worked on any bug report previously. The main requirement for a developer to be assignable is that he has a contribution history of approx. half a year or more.

Mockus and Herbsleb [25] compute the experience of a developer as a function of the number of changes he has made to a software system so far. Additionally, they compute recent experience by weighting recent changes more than older ones. The experience is then used to model the expertise of a developer. Furthermore, they examine the time that a developer needs to find additional people to work on a given modification request. Based on the results, they report that finding experts is a difficult task.

Fritz et al. [12] report on an empirical study that investigates whether a programmer's activity indicates knowledge of code. They found that the frequency and recency of interaction indicates the parts of the code for which the developer is an expert. They also report on a number of indicators that may improve the expertise model, such as authorship, role of elements, and the task being performed. In our work, we use the vocabulary of frequently and recently changed code to build an expertise model of developers. By using the vocabulary of software changes, lexical information about the role of elements and the kind of tasks are included in our expertise model.

Siy et al. [28] present a way to summarize developer work history in terms of the files they have modified over time by segmenting the CVS change data of individual Eclipse developers. They show that the files modified by developers tend to change significantly over time. Although, most of the developers tend to work within the same directories. This accords with our observation regarding the decay of vocabulary.

Gousios et al. [14] present an approach for evaluating developer contributions to the software development process based on data acquired from software repositories and collaboration infrastructures. However, their expertise model does not include the vocabulary of software changes and is thus not queryable using the content of bug reports.

Alonso et al. [1] describe an approach using classification of the file paths of contributed source code files to derive the expertise of developers.

Schuler and Zimmerman [27] introduce the concept of usage expertise, which manifests itself whenever developers are using functionality, *e.g.* by calling API methods. They present preliminary results for the Eclipse project indicating that usage expertise is a promising complement to implementation expertise (such as our expertise model). Given these results, we consider as future work to extend our expertise model with usage expertise as well.

Hindle et al. [15] perform a case study that includes the manual classification of large commits. They show that large commits tend to be perfective while small commits are more likely to be corrective. Commits are not normalized in our expertise model, thus the size of a commit may affect our model. However, since large commits are rare and small commits are common, we can expect our expertise model to equally take perfective and corrective contributions into account.

Bettenburg et al. [7] present an approach to split bug reports into natural text parts and structured parts, *i.e.* source code fragments or stack traces. Our approach treats both the same, since counting word frequencies is applicable for natural-language text as well as source code in the same way.

Anvik et al. [4] build developers' expertise from previous bug reports and try to assign current reports based on this expertise. They label the reports. If a report cannot be labeled, it is not considered for training. Additionally, reports involving developers with a too low bug fixing frequency or involving developers not working on the project anymore are filtered. They then assign bug reports from a period of lower than half a year. To find possible experts for their recall calculation, they look for the developers who fixed the bug in the source code (by looking for the corresponding bug ID in the change comments). They reach precision levels of 57% and 64% on the Eclipse and Firefox development projects respectively. However, they only achieve around 6% precision on the Gnu C Compiler project. The highest recall they achieve is on average 10% (Eclipse), 3% (Firefox) and 8% (gcc). Please note that the recall results are not directly comparable to ours, since they use different configurations of bug-related persons to compute recall.

Cubranic and Murphy [29] propose to use machine learning techniques to assist in bug triage. Their prototype uses supervised Bayesian learning to train a classifier with the textual content of resolved bug reports. This is then used to classify newly incoming bug reports. They can correctly predict 30% of the report assignments, considering Eclipse as a case study.

Canfora and Cerulo [8] propose an Information Re-

trieval technique to assign developers to bug reports and to predict the files impacted by the bug's fix. They use the lexical content of bug reports to index source files as well as developers. They do not use vocabulary found in source files, rather they assign to source files the vocabulary of related bug reports. The same is done for developers. For the assignments of developers, they achieve 30%–50% top-1 recall for KDE and 10% to 20% top-1 recall for the Mozilla case study.

Similar to our work, Di Lucca et al. use Information Retrieval approaches to classify maintenance requests [11]. They train a classifier on previously assigned bug reports, which is then used to classify incoming bug reports. They evaluate different classifiers, one of them being a term-documents matrix using cosine similarity. However, this matrix is used to model the vocabulary of previous bug reports and not the vocabulary of developers.

Anvik and Murphy evaluate approaches that mine implementation expertise from a software repository or from bug reports [5]. Both approaches are used to recommend experts for a bug report. For the approach that gathers information from the software repository, a linkage between similar reports and source code elements is required. For the approach that mines the reports itself, amongst others, the commenters of the reports (if they are developers) are estimated as possible experts. Both approaches disregard inactive developers. Both recommendation sets are then compared to human generated expert sets for the bug report.

Minto and Murphy's Emergent Expertise Locator (EEL) [24] recommends a ranked list of experts for a set of files of interest. The expertise is calculated based on how many times which files have been changed together and how many times which author has changed what file. They validate their approach by comparing recommended experts for files changed for a bug fix with the developers commenting on the bug report, assuming that they "either have expertise in this area or gain expertise through the discussion" [24].

We are not the first to apply an author-topic model [19, 6]. Linstead et al. showed promising results in applying author-topic models on Eclipse and Baldi et al. applied topic models to mine aspects.

Lexical information of source code has previously been proven useful for other tasks in software engineering, such as: identifying high-level conceptual clones [21], recovering traceability links between external documentation and source code [2], automatically categorizing software projects in open-source repositories [16], visualizing conceptual correlations among software artifacts [17, 18], and defining cohesion and coupling metrics [22, 26].

7. Conclusion

We presented a novel expertise model of developers. The model is based on the source code vocabulary of developers. The vocabulary of developers is obtained from the `diff` of their source code contributions. We applied the model in a recommendation system that assigns developers to bug reports. We evaluated the recommendation system using eight years of Eclipse development as a case study, and achieved 33.6% top-1 precision and 71.0% top-10 recall.

When calibrating our approach, we found that developer expertise decays over time. To counter this effect we applied two weighting schemes: i) *decay of vocabulary* weights expertise by a decay factor that is proportional to the time since the developer acquired that expertise, ii) *inactive developer penalty* downgrades developers that had been inactive for a certain time.

In the future, we would like to extend our expertise model with developer knowledge from other sources, *e.g.* mailing lists. Furthermore we would like to include additional Information Retrieval techniques, as well as combine our approach with approaches that are trained on previous bug reports (*e.g.* [4, 8, 29, 20]).

For more information on DEVELECT and the case study presented in this paper, please refer to the Master's thesis of Dominique Matter [23].

Acknowledgments

We thank Ann Katrin Hergert for proof reading the final draft of this paper. We thank Lukas Renggli and Jorge Ressia for their feedback on this paper. We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Bringing Models Closer to Code" (SNF Project No. 200020-121594, Oct. 2008 - Sept. 2010).

References

[1] O. Alonso, P. T. Devanbu, and M. Gertz. Expertise identification and visualization from cvs. In *MSR '08: Proceedings of the 2008 Intl. working conference on Mining software repositories*, pages 125–128, New York, NY, USA, 2008. ACM.

[2] G. Antoniol, U. Villano, E. Merlo, and M. D. Penta. Analyzing cloning evolution in the linux kernel. *Information and Software Technology*, 44(13):755–765, 2002.

[3] J. Anvik. Automating bug report assignment. In *ICSE '06: Proceedings of the 28th Intl. conference on Software engineering*, pages 937–940, New York, NY, USA, 2006. ACM.

[4] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *Proceedings of the 2006 ACM Conference on Software Engineering*, 2006.

[5] J. Anvik and G. C. Murphy. Determining implementation expertise from bug reports. In *MSR '07: Proceedings of the 4th Intl. Workshop on Mining Software Repositories*, page 2, Washington, DC, USA, 2007. IEEE Computer Society.

[6] P. F. Baldi, C. V. Lopes, E. J. Linstead, and S. K. Bajracharya. A theory of aspects as latent topics. In *OOPSLA '08: Proceedings of the 23rd conference on Object-oriented programming systems languages and applications*, pages 543–562, New York, NY, USA, 2008. ACM.

[7] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim. Extracting structural information from bug reports. In *MSR '08: Proceedings of the 2008 Intl. working conference on Mining software repositories*, pages 27–30, New York, NY, USA, 2008. ACM.

[8] G. Canfora and L. Cerulo. How software repositories can help in resolving a new change request. In *Proceedings of the Workshop on Empirical Studies in Reverse Engineering*, September 2005.

[9] H.-F. Chang and A. Mockus. Evaluation of source code copy detection methods on freebsd. In *MSR '08: Proceedings of the 2008 Intl. working conference on Mining software repositories*, pages 61–66, New York, NY, USA, 2008. ACM.

[10] S. C. Deerwester, S. T. Dumais, T. K. Landauer, G. W. Furnas, and R. A. Harshman. Indexing by latent semantic analysis. *Journal of the American Society of Information Science*, 41(6):391–407, 1990.

[11] G. A. Di Lucca, M. Di Penta, and S. Gradara. An approach to classify software maintenance requests. In *Processings of 18th IEEE International Conference on Software Maintenance (ICSM 2002)*, pages 93–102, 2002.

[12] T. Fritz, G. C. Murphy, and E. Hill. Does a programmer's activity indicate knowledge of code? In *ESEC-FSE '07: Proceedings of the 6th European software engineering conference*, pages 341–350, New York, NY, USA, 2007. ACM.

[13] T. Gîrba, A. Kuhn, M. Seeberger, and S. Ducasse. How developers drive software evolution. In *Proceedings of International Workshop on Principles of Software Evolution (IWPSE 2005)*, pages 113–122. IEEE Computer Society Press, 2005.

[14] G. Gousios, E. Kalliamvakou, and D. Spinellis. Measuring developer contribution from software repository data. In *MSR '08: Proceedings of the 2008 Intl. working conference on Mining software repositories*, pages 129–132, New York, NY, USA, 2008. ACM.

[15] A. Hindle, D. M. German, and R. Holt. What do large commits tell us?: a taxonomical study of large commits. In *MSR '08: Proceedings of the 2008 Intl. working conference on Mining software repositories*, pages 99–108, New York, NY, USA, 2008. ACM.

[16] S. Kawaguchi, P. K. Garg, M. Matsushita, and K. Inoue. Mudablue: An automatic categorization system for open source repositories. In *Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC 2004)*, pages 184–193, 2004.

[17] A. Kuhn, S. Ducasse, and T. Gîrba. Semantic clustering: Identifying topics in source code. *Information and Software Technology*, 49(3):230–243, Mar. 2007.

[18] A. Kuhn, P. Loretan, and O. Nierstrasz. Consistent layout for thematic software maps. In *Proceedings of 15th Working Conference on Reverse Engineering (WCRE'08)*, pages 209–218, Los Alamitos CA, Oct. 2008. IEEE Computer Society Press.

[19] E. Linstead, P. Rigor, S. Bajracharya, C. Lopes, and P. Baldi. Mining eclipse developer contributions via author-topic models. In *MSR '07: Proceedings of the 4th Intl. Workshop on Mining Software Repositories*, page 30, Washington, DC, USA, 2007. IEEE Computer Society.

[20] G. d. Lucca. An approach to classify software maintenance requests. In *ICSM '02: Proceedings of the Intl. Conference on Software Maintenance (ICSM'02)*, page 93, Washington, DC, USA, 2002. IEEE Computer Society.

[21] A. Marcus and J. I. Maletic. Identification of high-level concept clones in source code. In *Proceedings of the 16th International Conference on Automated Software Engineering (ASE 2001)*, pages 107–114, Nov. 2001.

[22] A. Marcus and D. Poshyvanyk. The conceptual cohesion of classes. In *Proceedings International Conference on Software Maintenance (ICSM 2005)*, pages 133–142, Los Alamitos CA, 2005. IEEE Computer Society Press.

[23] D. Matter. Preliminary title: Automatic bug report assignment with vocabulary-based developer expertise model. Master's thesis, University of Bern, 2009 (To appear).

[24] S. Minto and G. C. Murphy. Recommending emergent teams. In *MSR '07: Proceedings of the 4th Intl. Workshop on Mining Software Repositories*, page 5, Washington, DC, USA, 2007. IEEE Computer Society.

[25] A. Mockus and J. D. Herbsleb. Expertise browser: a quantitative approach to identifying expertise. In *ICSE '02: Proceedings of the 24th Intl. Conference on Software Engineering*, pages 503–512, New York, NY, USA, 2002. ACM.

[26] D. Poshyvanyk, A. Marcus, R. Ferenc, and T. Gyimthy. Using information retrieval based coupling measures for impact analysis. *Empirical Software Engineering*, 14(1):5–32, Feb. 2009.

[27] D. Schuler and T. Zimmermann. Mining usage expertise from version archives. In *MSR '08: Proceedings of the 2008 Intl. working conference on Mining software repositories*, pages 121–124, New York, NY, USA, 2008. ACM.

[28] H. Siy, P. Chundi, and M. Subramaniam. Summarizing developer work history using time series segmentation: challenge report. In *MSR '08: Proceedings of the 2008 Intl. working conference on Mining software repositories*, pages 137–140, New York, NY, USA, 2008. ACM.

[29] D. Čubranić and G. C. Murphy. Automatic bug triage using text categorization. In F. Maurer and G. Ruhe, editors, *SEKE*, pages 92–97, 2004.

[30] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *26th International Conference on Software Engineering (ICSE 2004)*, pages 563–572, Los Alamitos CA, 2004. IEEE Computer Society Press.