Automatic bug triage using text categorization

Davor Čubranić
Department of Computer Science
University of British Columbia
201–2366 Main Mall
Vancouver, BC, V6T 1Z4
cubranic@cs.ubc.ca

Gail C. Murphy
Department of Computer Science
University of British Columbia
201–2366 Main Mall
Vancouver, BC, V6T 1Z4
murphy@cs.ubc.ca

Abstract

Bug triage, deciding what to do with an incoming bug report, is taking up increasing amount of developer resources in large open-source projects. In this paper, we propose to apply machine learning techniques to assist in bug triage by using text categorization to predict the developer that should work on the bug based on the bug's description. We demonstrate our approach on a collection of 15,859 bug reports from a large open-source project. Our evaluation shows that our prototype, using supervised Bayesian learning, can correctly predict 30% of the report assignments to developers.

1 Introduction

Large software development projects require a bug tracking system to manage bug reports and developers who work on fixing them. A ubiquitous example of such a system is Bugzilla,[1] an open-source system first introduced in the development of the Mozilla web browser, but now used in numerous other projects.

Bug tracking systems are particularly important in open-source software development, where the team members can be dispersed around the world. In such widely-distributed projects, the developers and other project contributors may rarely, if ever, see each other. Consequently, the bug tracking system is used not only to keep track track of problem reports and feature requests, but also to coordinate work among the developers.[2]

Most bug-tracking systems allow posting of additional comments in bug reports. With communication channels between open source team members limited by their geographical and time separation, this feature has evolved to fill a niche for focused, issue-specific discussion. The comments on the bug report serve as forum for discussion of implementation details or feature design alternatives. Developers who can help in design deliberations because of their expertise and insight, and stakeholders whose code will be impacted by the proposed modifications, or who will have to implement and integrate them, are quickly brought into the discussion by "CC-ing" them on the bug report.[3] Other members of the project with interest in the issue, often users who urgently need the feature or the bug fix, also join in. More contentious issues—usually requests for new features—can take months to resolve and can involve over a hundred comments from dozens of people.

In many ways, the bug tracking system is the public face that an open source development team presents to its user community. Therefore, it is important that new bug reports be dealt with as quickly as possible. Few things will turn the users away—and kill the project's community—faster than the perception that the developers are not responsive and ignore the users' bug reports and feature requests.

However, successful large open source projects are faced with the challenge of managing the incoming deluge of new reports.[4] Effectively deciding what to do with a new report—*bug triage* in Mozilla parlance—can be a problem: it takes time to figure out whether the report is a real bug or a feature worth considering, to check that it is not a duplicate of an existing report, and to decide which developer should work on it. Past a certain rate of new bug reports, the time commitment for triage becomes too much of a burden for an experienced developer, whose attention is more valuable elsewhere. Projects such as Mozilla and Eclipse[5]

[1] http://www.mozilla.org/projects/bugzilla.

[2] The bug tracking system therefore serves to track more than just bugs, and it may be more appropriate to call it "issue tracking system". We use the terms "bug tracking system" and "bug report" for historical reasons, but in their wider, all-inclusive, sense.

[3] All developers on the CC list for a given bug report are automatically emailed notifications of changes to the report's status and new comments.

[4] The Mozilla project has received an average of 168 new bug reports per day in the week of 9 February 2004, for example.

[5] An extensible integrated development environment developed by IBM

have therefore been forced to introduce team members who are dedicated to bug triage [2]. This solution is not ideal, however, because it requires an additional step before the developer can start working on a bug. It also introduces potential errors, and more delays, if the triager makes a wrong decision to which developer to assign the report.

In this paper we present our investigation of using machine learning, and in particular text categorization, to "cut out the triageman" and automatically assign bugs to developers based on the description of the bug as entered by the bug's submitter. The method would require no changes to the way bugs are currently submitted to Bugzilla, or to the way developers handle them once the bugs are assigned. The benefit to software development teams would be to free up developer resources currently devoted to bug triage, while assigning each bug report to the developer with appropriate expertise to deal with the bug.

We begin this paper with a brief overview of related work, followed by an introduction to the classification framework used and the theory behind it. We then present an experiment in which we applied these techniques to a selection of bug reports from the Eclipse project and tested their accuracy in assigning reports to developers. We conclude the paper with a discussion of results and possible avenues for future work.

2 Related work

We are aware of no other work on computer-assisted bug report triage, although there are some key insights on the interrelationship between bug reports, source code, and the developers that we share with the following two projects:

Fischer et al. mapped program features to the source code where they were implemented, and then tracked the code changes against problem reports involving those features [4]. They then visualize the established relationships to search for feature overlap and dependencies. Such visualization of the evolution of features across time can then be used to find locations in the code where there may be erosion in the software architecture of the system, indicating future problem spots for software maintentance.

Bowman and Holt have analyzed which developers worked on each file in a software system to determine its *ownership architecture* [1]. The ownership architecture complements other types of architectural documentation. It identifies experts for system components, and can be used to infer the project's internal organization into sub-teams. The ownership archtecture can also show non-functional dependencies: in their example device drivers for a given architecture could be easily seen, even if they otherwise shared no code and resided in separate portions of the filesystem hierarchy, because they were "owned" by the same small group of developers.

Although our purpose is different, our approach bridges Bowman and Holt's idea that there is a correspondence between a system's components and individual developers with that of Fischer et al. on the link between bug reports and program features. We also note that all three projects are for support of *managing* software development, even if they mine the source code for the relevant information.

Machine learning and data mining techniques have already been applied to source code and program failure reports, although so far only to support the code-writing/debugging component of the software development effort. For instance, Zimmermann et al. mined source version histories to determine association rules which can then be used to predict files (or smaller program elements, such as functions and variables) that usually change together [11]. Such predictions can help prevent errors due to incomplete changes or show program couplings that wouldn't be visible to methods such as program dependency analysis.

Also, Podgurski et al. use machine learning to cluster software failure reports to automatically determine which ones are likely to be manifestations of the same error [9]. The failure reports in this case are automatically generated, unlike the bug reports we deal with, and consist of stack traces at the moment of program crashes.

3 Classification framework

We treat the problem of assigning developers to bug reports as an instance of text classification, or "the problem of assigning a text document into one or more topic categories or classes" [6]. More specifically, it is a *multi-class, single-label classification* problem: each developer corresponds to a single class, and each document (that is, a bug report) is assigned to only one class (that is, a developer working on the project). Furthermore, it is a *supervised learning* problem, since we can view the correspondences of developers with the bugs that they fixed in the past as the training data.

A variety of techniques for supervised learning have been applied to text classification in recent years, for example: regression models, k-nearest neighbour, Bayes belief networks, decision trees, support vector machines, and rule-learning algorithms. (See Yang [10] for an overview of these approaches and a comparative evaluation of their performance.) In this paper, we report on the use of Bayesian learning approach for this project, because it is conceptually elegant, is easily adapted to multi-class classification, and performs well. The algorithm used, introduced by Kalt [5] and further developed by Nigam et al. [8] is presented in the following section, followed by the explanation of how we applied it in the bug triage domain.

as open source software, http://www.eclipse.org.

3.1 Naive Bayes classifier for multinomial word-document model

The following are the framework's assumptions: the data set is represented as a collection of documents, $\mathcal{D} = \{d_1, \ldots, d_{|\mathcal{D}|}\}$, and each document has a class label $c \in \mathcal{C} = \{c_1, \ldots, c_{|\mathcal{C}|}\}$. Documents in $\mathcal{D}$ are generated by a mixture model and there is a one-to-one correspondence between mixture components and classes in $\mathcal{C}$. Mixture components, in turn, are parametrized on θ. Therefore,

$$P(d_i|\theta) = \sum_{j=1}^{|\mathcal{C}|} P(d_i|c_j, \theta) P(c_j|\theta) \quad (1)$$

Furthermore, the documents are represented as "bags of words": each document d_i consists of words w_t drawn from vocabulary $\mathcal{V} = \{w_1, \ldots, w_{|\mathcal{V}|}\}$. The *naive Bayes* assumption is that the words are independently and identically distributed (i.i.d.): the probability of each word is independent of its context and position in the document. Thus, the documents are drawn from a multinomial distribution:

$$P(d_i|c_j, \theta) = \prod_{t=1}^{|\mathcal{V}|} \delta_{tj}^{N_{ti}} \quad (2)$$

where $\delta_{tj} = P(w_t|c_j, \theta)$ and N_{ti} is the number of times word w_t occurs in the document d_i. While the naive Bayes assumption is clearly false in many real-world situations, classifiers based on it perform surprisingly well, and it turns out that it is not a mathematically unreasonable assumption to make in classification tasks [3].

Using the Bayes rule, a previously unseen document d_i can then be assigned label c_j which maximizes:

$$P(c_j|d_i, \theta) = \frac{P(c_j|\theta) P(d_i|c_j, \theta)}{P(d_i|\theta)} \quad (3)$$

$$\propto P(c_j|\theta) \prod_{t=1}^{|\mathcal{V}|} \delta_{tj}^{N_{ti}} \quad (4)$$

The priors are estimated from the training data:

$$P(c_j|\theta) = \frac{\sum_{i=1}^{|\mathcal{D}|} P(c_j|d_i)}{|\mathcal{D}|} \quad (5)$$

where $P(c_j|d_i) = \{0, 1\}$ as given by the training set labels (that is, $P(c_j|\theta)$ equals the number of times c_j occurred in the test set divided by the size of the test set); and

$$P(w_t|c_j, \theta) = \frac{\sum_{i=1}^{|\mathcal{D}|} N_{ti} P(c_j|d_i)}{\sum_{m=1}^{|\mathcal{V}|} \sum_{i=1}^{|\mathcal{D}|} N_{ti} P(c_j|d_i)} \quad (6)$$

(That is, $P(w_t|c_j, \theta)$ equals the number of times word w_t occurs in class c_j divided by the total number of all word occurrences in that class.) In practice, a Laplace prior is often used to avoid zero probabilities of words occurring infrequently in $\mathcal{V}$ and was used here as well:

$$P(w_t|c_j, \theta) = \frac{1 + \sum_{i=1}^{|\mathcal{D}|} N_{ti} P(c_j|d_i)}{|\mathcal{V}| + \sum_{m=1}^{|\mathcal{V}|} \sum_{i=1}^{|\mathcal{D}|} N_{ti} P(c_j|d_i)} \quad (7)$$

3.2 Bug triage as a Naive Bayes classifier

Our dataset $\mathcal{D}$ is a collection of bug reports $\{d_1, \ldots, d_{|\mathcal{D}|}\}$ entered into the bug tracking database. When a new bug report is submitted, it is given a one-line summary and a longer description. The bug report d_i thus consists of a set of words w_t that appear in its summary and description. The order of words does not matter, but we do keep track of multiple occurrences of a word in a single bug report, N_{ti}.

The developers working on the project form our set of classes $\mathcal{C} = \{c_1, \ldots, c_{|\mathcal{C}|}\}$. Although in real world a bug report d_i may be handled by a number of people, only one of them, c_j ultimately resolves it—implements a bug fix or a requested feature, rejects a proposed enhancement, determines that the report is not really a bug, etc.—and therefore we assign to d_i the class label c_j.

Once we have built our model θ using the existing bug reports as training data, bug triage of a new bug report $d_{|\mathcal{D}|+1}$ simply follows from Equation 3: we assign it to the developer $c \in \mathcal{C}$ for whom $P(c|d_{|\mathcal{D}|+1}, \theta)$ is maximized.

4 Experimental results

To test the approach, we applied it to a selection of bug reports from the Eclipse project and tested its accuracy in assigning reports to developers.

4.1 Data set

We selected all reports entered into Eclipse's bug tracking system[6] between January 1, 2002 and September 1, 2002. A total of 15,859 reports were selected. The system records for each bug the id of the user[7] who submitted it (the *submitter*), a one-line summary accompanied with a longer free-text description of the problem (which may include steps to reproduce it, or information from the error logs, core dumps, and stack traces), and various attributes such as its status (*new, resolved*, etc.), who it is assigned to, and the list of users on the "CC" list who are automatically notified of any changes to the report. The report can also

[6] Available online at https://bugs.eclipse.org

[7] "User" in this section denotes the user of the bug tracking system, who may be a developer actively working on the project, and occasional contributor, or simply a user of the software with interest in certain issues or features

contain a list of free-text comments which can be made by any user, and which include the author's id and time of posting. Finally, each report stores a timestamped history of all the changes to its attributes, including the assigned-to.

To determine a document's class (that is, the developer to whom it should be assigned), a straightforward approach would be to choose whoever was the report assigned to in the bug tracking system. However, this obvious approach is misleading for two reasons: first, in many cases this *assigned-to* "user is actually an email alias for a whole subteam that deals with the module in question; second, just as often, the developer who actually implements the fix for the bug or requested feature—or who makes the decision to remove it from further consideration—is not the developer to whom the bug was nominally assigned in the bug tracking system.

Instead, we used our observations and experiences with the bug tracking and development procedures in the Eclipse project and devised the following heuristic to determine a report's class (developer who should handle it from the outset):

1. If the report was *resolved* by the assigned-to developer, the report is labelled by his or her class regardless of who the submitter was or what the report's *resolution* was (e.g., *fixed*, *duplicate*, *invalid*, *later*, etc.). This is clearly the case of a developer who was in charge of the report and who has completed processing it.

2. If the report was *resolved* by someone other than the assigned-to developer, but not by the person who submitted it, we label the report with the class of the developer who marked it resolved. The reasoning is that whoever made the decision to resolve the report is the person to whom it should have been assigned all along.

3. If the report was *resolved* as *fixed*, regardless of who the resolver was, we assume that this is the developer who implemented the fix and label the report with the class of that developer, as this is probably the person who had done the real work on the report. This rule covers the frequent case where an Eclipse developer files a report, which is then assigned to somebody else or a sub-team alias by default, and then later implements the fix himself.

4. If the report was *resolved* as non-fixed (i.e., with resolution *duplicate*, *invalid*, etc.) by the person who submitted it, and who was not also assigned to it, the report is labelled with the class of the first person who responded to the reporter. This handles the many cases of a submitter throwing the report away after being informed that it is a feature and not a bug, or after being prompted by a developer for details of his or her setup and discovering that the bug does not exist any more. We choose the first responder to the report rather than the assigned-to person for reasons outlined above.

5. If the report was *resolved* as non-fixed by the submitter who was not the assigned-to developer, and nobody responded, we assume that the report was submitted in error—for example, not knowing the proper operation of Eclipse—and that the mistake was caught by the submitter before anyone could react. These reports are removed from the training set, as they cannot be reliably labelled.

6. If the report was not *resolved*, we label it with the class of the most recent assigned-to developer.

These heuristics are not perfect, and we have noticed three or four examples where they are definitely not correct. However, based on a non-exhaustive examination of its results, they perform much better than always simply labelling a report with the class of the assigned-to developer. The labelling heuristics are obviously tailored to the development practices of the Eclipse project, and may need various amounts of modification before they could be applied to a different project. Mozilla, for example, uses a strict code review practice in which two senior developers need to check off on a proposed implementation (usually a patch that's attached in the Bugzilla database), and so it is usually the reviewers who close the bug and not the developer responsible for the implementation.

During the labelling, we threw away 189 reports as described in step 4, for a total of 15,670 labelled documents (reports) and 162 classes (developers). We then extracted the summary and description of each report, tokenized all alphabetic sequences of characters (lower-cased and disregarding words in the standard SMART system stoplist of 524 common words such as "the", "a", etc.), and used that as the content of the document in classification. No stemming of words was done, except where so noted in the results section.

4.2 Methodology and measures

The data set was divided into a test set and a train set by randomly selecting a percentage of the documents from the data set for placing into the train set, with the remainder going to the test set. The model was learned using the train set, and then tested for label predictions of documents from the test set. We used the Bow toolkit [7], and configured it with the parameters as described above.

The classification accuracy was calculated as the percentage of documents for which the algorithm predicted the correct label. The predictions of course exist for all classes (that is, we calculate all the $P(c_j|d_i, \theta)$, where $\sum_j P(c_j|d_i, \theta) = 1$), but only the top prediction counts

when determining accuracy. The results reported below are the average over multiple runs, where each run used a new randomly built training and test sets (three runs per data point).

4.3 Results

In our experiments, we varied the size of the test set, the size of the vocabulary, and the criterion used to truncate the vocabulary. Figure 1 shows the classification accuracy as a function of the train/test set split, when the full vocabulary $\mathcal{V}$ of words found in bug reports is used.

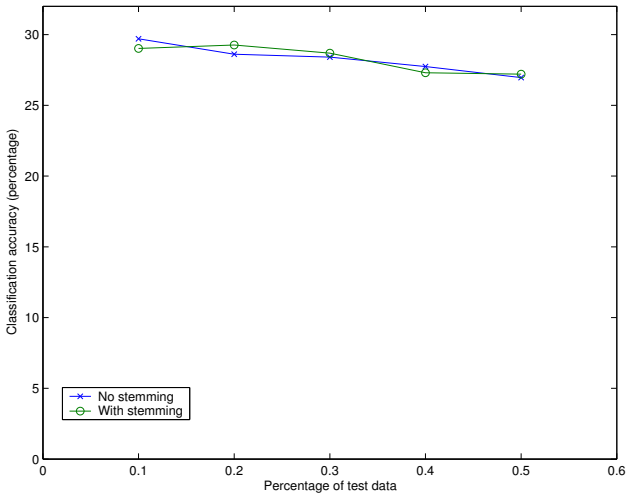

Figure 1. Classification accuracy without vocabulary truncation.

As we can see, the algorithm corectly assigns just under 30% of the bugs, when 90% of the document corpus is used as training and 10% as the test set. The accuracy slowly declines to 27% as the test set's size is increased to 50% of the corpus.

Figure 1 also shows the results when the vocabulary was created using stemming, which identifies most grammatical variations of a word—such as "see," "sees," "seen," for example—and treats them as a single term.[8] The results are virtually unchanged, and any differences between the two conditions are within about one standard deviation at each data point.

Figure 2 shows the classification accuracy when the vocabulary was truncated to eliminate words that do not occur in at least d documents, for $d = \{1, 2, 5, 10, 20\}$. The accuracy is slightly lower than when the full vocabulary was used for $d = 1$, and almost a third worse (just above 20%) for $d = 20$. There is a slight downward trend as the size of the test set increases, but not in all cases.

[8]The standard Porter stemming algorithm was used.

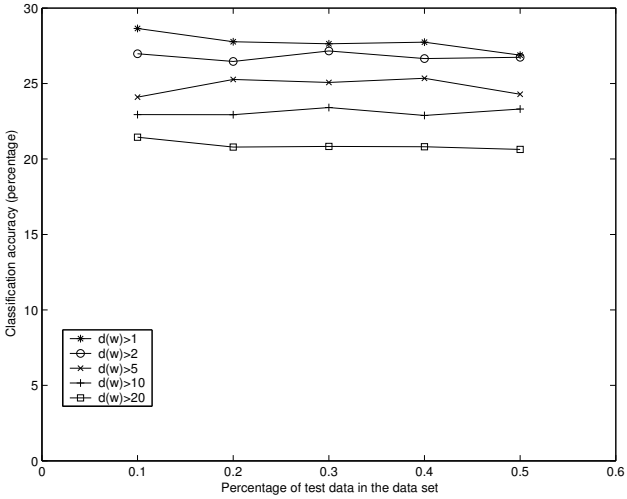

Figure 2. Classification accuracy when words occurring in fewer than d documents are removed from the vocabulary.

Figure 3 also shows the classification accuracy of a truncated vocabulary, but using a different truncating criterion. In this case, we eliminated all words that occur fewer than T times in the entire collection, for $T = \{5, 10, 20, 40, 80\}$. Again, the classification accuracy is slightly lower for $T = 5$ than when the full vocabulary is used, and falls to just over 20% for $T = 40$ and to around 18% for $T = 80$. Interestingly, for higher values of T, the accuracy improves with smaller training set, indicating some overfitting was occurring otherwise.

5 Discussion and future work

Overall, the performance of the algorithm was lower than expected, although the results are sufficiently promising to warrant further investigation. For example, we expected that truncating the vocabulary would have helped, by reducing the danger of overfitting, but that was clearly not the case, although smaller vocabulary speeds up the classification.

Also, we would like to involve the developers from the Eclipse project to evaluate the classification results based on their own subjective experience. For example, in those cases when a document is mis-classified, is the classification still "reasonable"—such as to a colleague on the same sub-team, who could handle the bug himself.

Our heuristics for deducing the developer-bug assignment in the data set could be improved further. It is currently based on our own observations of the bug-handling process, and could benefit from insight gained by directly involving the project's developers. Also, we build our set of

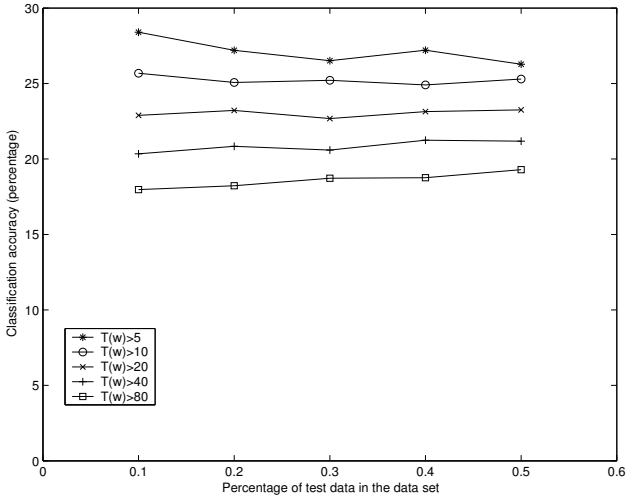

Figure 3. Classification accuracy when words occuring in the collection fewer than T times are removed from the vocabulary.

possible labels automatically from user ids in the bug tracking system. We could limit this set only to those users that we know are real developers on the team. This would eliminate labels corresponding to people who were perhaps only the bug submitters or interested bystanders, and who were falsely determined to be the bug "owners" by our heuristic.

Another weakness and a potential avenue of improvement is that in the process of creating the data set used in training and testing of the classifier, we either force a bug to a developer's class, or throw it away. This is the consequence of the naive Bayes classifier algorithm that we use, which cannot deal with unlabelled documents in the corpus. However, there are extensions to this algorithm that combine it with Expectation Maximization (EM) methods to achive very good results classifying a document corpus that contained a high proportion of unlabelled documents [8]. An interesting variation would be to label the documents with a range of probabilities, rather than just 1 or 0 we currently use, which would allow us to reflect our degree of certainty in the classification during the learning phase.

6 Summary

In this paper, we described an application of supervised machine learning using a naive Bayes classifier to automatically assign bug reports to developers. We evaluated our approach on bug reports from a large open-source project, Eclipse.org, achieving 30% classification accuracy with current prototype. We believe that the system could be easily incorporated into current bug-handling procedures to decrease the resources currently devoted to bug triage. New bug reports would be automatically assigned to the developer predicted to be the most appropriate to the content. Mispredictions could be handled in a light-weight fashion by their assignee, "bouncing" them to a dedicated triager for human inspection and classification. Clearly, even the classification accuracy we can currently achieve, would significantly lighten the load that the triagers face under the present conditions.

References

[1] I. T. Bowman and R. C. Holt. Reconstructing ownership architectures to help understand software systems. In *Proc. of IWPC 1999*, pp 28–37. IEEE Press.

[2] T. Creasey. Personal communication, 14 July 2003. Eclipse developer, IBM Canada.

[3] P. Domingos and M. Pazzani. Beyond independence: Conditions for the optimality of the simple Bayesian classifier. In *Proc. of ICML 1996*, pp 105–112. Morgan Kaufmann.

[4] M. Fischer, M. Pinzger, and H. Gall. Analyzing and relating bug report data for feature tracking. In *Proc. of WCRE 2003*, pp 90–99. IEEE Press.

[5] T. Kalt. A new probabilistic method of text classification and retrieval. Technical Report IR-78, Center for Intelligent Information Retrieval, University of Massachussetts, Amherst, MA, 1996.

[6] A. McCallum. Multi-label text classification with a mixture model trained by EM. In *AAAI Workshop on Text Learning*, 1999.

[7] A. K. McCallum. Bow: A toolkit for statistical language modeling, text retrieval, classification and clustering, 1996.

[8] K. Nigam, A. K. McCallum, S. Thrun, and T. M. Mitchell. Learning to classify text from labeled and unlabeled documents. In *Proc. of AAAI 1998*, pp 792–799. AAAI Press.

[9] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun, and B. Wang. Automated support for classifying software failure reports. In *Proc. of ICSE 2003*, pp 465–475. IEEE.

[10] Y. Yang. An evaluation of statistical approaches to text categorization. *Information Retrieval*, 1(1–2):69–90, 1999.

[11] T. Zimmermann, P. Weißgerber, S. Diel, and A. Zeller. Mining version histories to guide software changes. In *Proc. of ICSE 2004*, to appear. IEEE Press.