How Long will it Take to Fix This Bug?

Cathrin Weiß
Saarland University
weiss@st.cs.uni-sb.de

Rahul Premraj
Saarland University
premraj@cs.uni-sb.de

Thomas Zimmermann
Saarland University
tz@acm.org

Andreas Zeller
Saarland University
zeller@acm.org

Abstract

Predicting the time and effort for a software problem has long been a difficult task. We present an approach that automatically predicts the fixing effort*, i.e., the person-hours spent on fixing an issue. Our technique leverages existing issue tracking systems: given a new issue report, we use the Lucene framework to search for similar, earlier reports and use their average time as a prediction. Our approach thus allows for early effort estimation, helping in assigning issues and scheduling stable releases. We evaluated our approach using effort data from the JBoss project. Given a sufficient number of issues reports, our automatic predictions are close to the actual effort; for issues that are bugs, we are off by only one hour, beating naïve predictions by a factor of four.*

1. Introduction

Predicting when a particular software development task will be completed has always been difficult. The time it takes to fix a defect is particularly challenging to predict. Why is that so? In contrast to programming, which is a *construction* process, debugging is a *search* process—a search which can involve all of the program's code, its runs, its states, or even its history. Debugging is particularly nasty because the original assumptions of the program's authors cannot be trusted. Once the defect is identified, fixing it is again a programming activity, but the earlier effort to search typically far outweighs the correction effort.

In this paper, we address the problem of estimating the time it takes to fix an *issue*[1] from a novel perspective. Our approach is based on leveraging the experience from earlier issues—or, more prosaic, to extract issues reports from bug databases and to use their features to make predictions for new, similar problems. We have used this approach to predict the *fixing effort*—that is, the effort (in person-hours) it takes to fix a particular issue. These estimates are central to project managers, because they allow to plan the cost and time of future releases.

Our approach is illustrated in Figure 1. As a new issue report r is entered into the bug database (1), we search for the existing issue reports which have a description that is most similar to r (2). We then combine their reported effort as a prediction for our issue report r (3).

In contrast to previous work (see Section 8), the present paper makes the following original contributions:

1. We leverage *existing bug databases* to automatically estimate effort for new problems.
2. We use *text similarity* techniques to identify those issue reports which are most closely related.
3. Given a sufficient number of issue reports to learn from, our *predictions are close* to the actual effort, especially for issues that are bugs.

The remainder of the paper is organized as follows: In Section 2, we give background information on the role of issue reports in the software process. Section 3 briefly describes how we accessed the data. Section 4 describes our statistical approach, which is then evaluated in a case study (Section 5 and 6) involving JBoss and four of its subprojects. After discussing threats to validity (Section 7) and related work (Section 8), we close with consequences (Section 9).

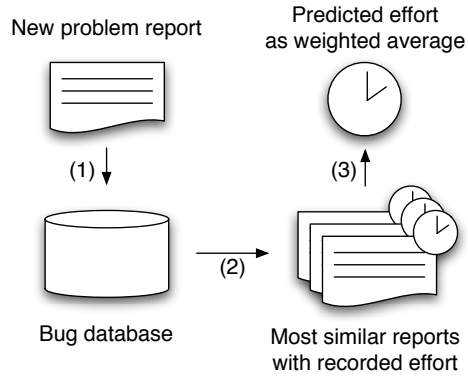

Figure 1. Predicting effort for an issue report

[1] An *issue* is either a bug, feature request, or task. We refer to the database that collects issues as bug database or issue tracking system.

Fourth International Workshop on Mining Software Repositories (MSR'07)
0-7695-2950-X/07 $20.00 © 2007 IEEE

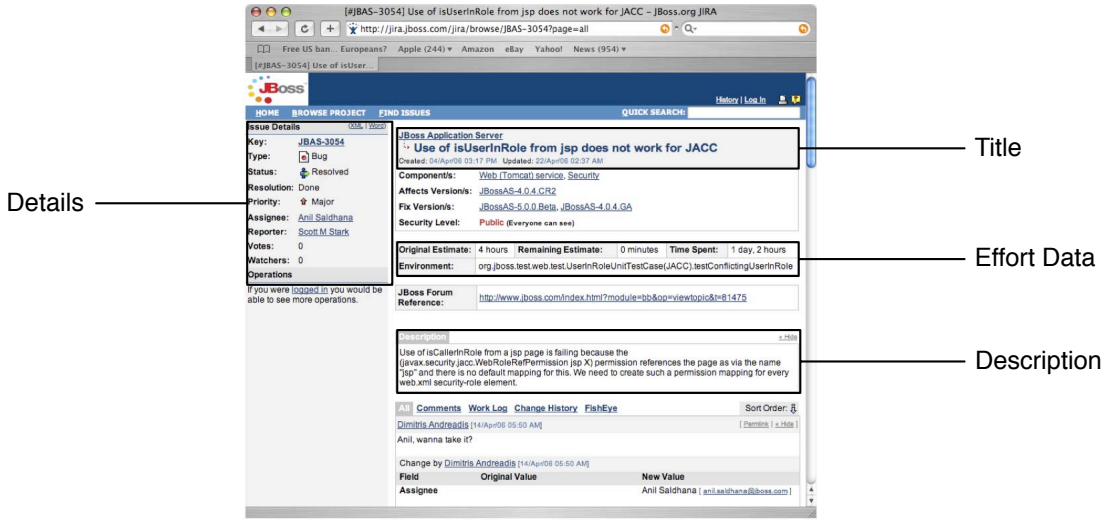

Figure 2. JBoss issue JBAS-3054 in the Jira Web interface.

2. A Bug's Life

Most development teams organize their work around a *bug database*. Essentially, a bug database acts as a big list of issues—keeping track of all the bugs, feature requests, and tasks that have to be addressed during the project. Bug databases scale up to a large number of developers, users—and issues.

An individual record in a bug database is called a *issue report*; it is also known as *problem report* or *ticket*. An issue report provides fields for the *description* (what causes the issue, and how can one reproduce it), a *title* or *summary* (a one-line abstract of the description), as well as a *severity* (how strongly is the user affected by the issue?). The severity can range from "enhancement" (i.e. a feature request) over "normal" and "critical" to "blocker" (an issue that halts further development). These fields are normally provided by the original submitter.

At the moment an issue report is submitted, it gets a unique *identifier* by which it can be referred to in further communication. Figure 2 shows the JBoss issue report JBAS-3054 from the JBAS subproject in the Jira Web interface.[2] At the top, we see the title "Use of isUserInRole from jsp does not work for JACC"; at the bottom, the detailed description.

Let us assume that someone has just entered this very issue report into the bug database. While the issue is being processed, the report runs through a *life cycle* (Figure 3). The position in the life cycle is determined by the *state* of the issue report. Initially, every single issue report has a state of `UNCONFIRMED`. It is then checked for validity and uniqueness; if it passes these checks, it becomes `NEW`. At

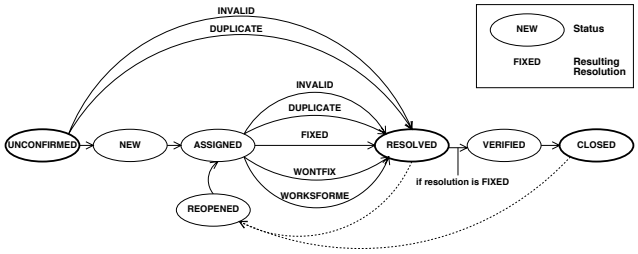

Figure 3. The life cycle of an issue report [14].

this point, the issue report is also assigned a *priority*—the higher the priority, the sooner it is going to be addressed. Typically, the priority reflects the risk and/or damage of the issue/bug. In Figure 2, priority and state are shown in the *details* column on the left.

At the time of the initial assessment, the staff may also include an *estimate* of the time it will take to fix the issue. For JBAS-3054, this original estimate was 4 hours; it is also shown in the Web interface (Figure 2). Both priority and estimate are crucial in scheduling fixes—and in estimating when a stable state will be reached.

Eventually, the issue report is assigned to an individual developer—its state is then changed to `ASSIGNED`. The developer now works on the issue, sometimes resulting in additional comments, questions, and re-assignments, all stored in the bug database. Eventually, the developer comes up with a *resolution.* This resolution can be `FIXED`, meaning that the problem is solved, but also `WONTFIX` (meaning the problem is not considered as such) or `WORKSFORME` (meaning that the problem could not be reproduced). With this resolution, the state becomes `RESOLVED`.

At this stage, the lead developer may record the *effort* it

[2] `http://www.atlassian.com/software/jira/`

0-7695-2950-X/07 $20.00

took to resolve the report. In the case of JBAS-3054, the effort is 10 person-hours (the web interface reports 8 hours as 1 day), which is considerably off the original estimate.[3]

As the problem is now fixed, two more steps remain: the testers must confirm the success of the fix (resulting in `VERIFIED` state), and finally, the fix must be deployed as a patch or a new release, closing the issue report (`CLOSED`)—and thus ending the issue's life, unless one day, it gets `REOPENED`.

The bug database thus is at the center of the development process. Developers query bug databases to find out their tasks, as well as to learn about the project history. Managers use bug databases to query, schedule, and assign the project's tasks. If the bug database is publicly accessible, users check it to see the progress on the bugs they submitted. As the bug database grows, it becomes a *project memory* of the group—listing all the problems as they occurred in the past, and how they were addressed. As we show in this paper, this memory can be a valuable resource when it comes to assess the project's future.

3. Mining Bug Databases

As mentioned above, the JBoss project uses the Jira issue tracking system to organize issue reports. Jira is one of the few issue tracking systems that support effort data. Since Jira provides free licensing for non-profit organizations, it recently became very popular in open source development, slowly supplanting Bugzilla. The most prominent projects using Jira include Hibernate, Apache, and JBoss.

However, only few open source projects collect effort data because measuring effort seems to be tedious and useless. The JBoss project is one exception. Although not all subprojects collect effort data and some do only sporadically, the JBoss issue tracking system has a vast amount of effort data available.

We developed a tool that crawls through the web interface of a Jira database and stores the issues. In this paper, we use Jira's the title, the description, and effort data of an issue (time spent). Furthermore, we only consider issue reports whose resolution is `FIXED`, ignoring duplicates, invalid issues (because of "overbilling", i.e., more time spent than the issue's lifetime, which may be a result of developers stating the effort), and all other "non-resolutions". In Table 1, we list the prerequisites for issues to qualify for our study. In total, 567 issues met these conditions and finally became the input to our statistical models.

[3]The web interface also reports the *remaining estimate*—the effort still needed to resolve the bug. It is constantly updated (reduced) while a developer is spending time on a bug: *remaining-estimate* = $max(0, \textit{original-estimate} - \textit{effort})$ When a bug is closed, this estimate can be non-zero (i.e., the issue needed less effort) or zero (the issue needed the estimated effort or more).

Table 1. Prerequisites for issues.

	Count
Issues reported until 2006-05-05	11,185
Issues with	
– effort data (*timespent_sec* is available)	786
– valid effort data (*timespent_sec*$\leq$*lifetime_sec*)	676
– *type* in ('Bug', 'Feature Request', 'Task', 'Sub-task')	666
– *status* in ('Closed, 'Resolved')	601
– *resolution* is 'Done'	575
– *priority* is not 'Trivial'	574
Issues indexable by Lucene	**567**

4. Predicting Effort for Issue Reports

In order to predict the effort for a new issue report, we use the nearest neighbor approach (Section 4.1) to query the database of resolved issues for textually similar reports (Section 4.2). We also increase the reliability of our predictions by extending the nearest neighbor approach to explicitly state when there are no similar issues (Section 4.3).

4.1. Nearest Neighbor Approach (kNN)

The nearest neighbor approach (kNN) has been widely used for effort and cost estimation for software projects early in their life-cycle [12]. The advantage of using the nearest neighbor approach lies in its ease and flexibility of use, ability to deal with limited data that may even belong to different data types. Moreover, kNN has been shown to outperform other traditional cost estimation models such as linear regression and COCOMO [12]. Since we borrow our reasoning for this research from software cost estimation (i.e., similar issues are likely to require similar fixing times), we chose to use the nearest neighbor approach for predicting effort for issue reports.

More formally, we use kNN as follows: a *target* issue (i.e., the one that needs to be predicted) is compared to *previously* solved issues that exist in a repository. In order to identify similar issues, we define a *distance* function that combines the distances between individual features of two issues reports into a single distance. Then, the k most similar issues (= the ones with the closest distance), referred to as *candidates*, are selected to derive a prediction for the target. In addition to the prediction, we report the candidates to the developer which supports him to comprehend how our prediction came to be.

To apply kNN, we first need to identify the features that can be used to measure similarity between issues. At the time of reporting an issue, two crucial fields of information pertaining to the issue are available to us: the *title* (a one-line summary) and the *description*—both of them are known *a priori* and we use them to compute the similarity

between two issues reports. Since these two features are in the form of free text, they require a rather sophisticated mechanism to measure similarity, which we describes in the following section.

4.2. Text Similarity

Most information pertaining to an issue is entered in the title and description fields of the issue report. Hence, their inclusion when measuring similarity between issues is crucial for making estimations based on previously fixed, similar issues. For this purpose, we applied a *text similarity measuring engine*—Lucene, developed by the Apache Foundation [7]. Our choice was motivated by Lucene's competence demonstrated by its deployment in systems at high profile places including FedEx, New Scientist Magazine, MIT, and many more for text comparisons.

We use Lucene, which measures similarity between two texts with a vector-based approach, for indexing existing issue reports in a repository. Before indexing, we filter all common English stop words such as *a*, *an*, and *the*; and symbols such as $+$, $-$, and $()$. When we query a new issue report, we compare it to all other issue reports to generate a similarity score. Scores closer to 1 indicate issues with very similar titles and descriptions, while scores close to 0 indicate marginal similarity. More technically, we use the *multi field query* feature of Lucene: the titles and descriptions of issues are compared separately and combined into a single score by using a *boost factor.* For our experiments we used 1:1 as the boost factor such that similarity for titles and descriptions are weighted the same.

4.3. Nearest Neighbor with Thresholds (α-kNN)

Let us demonstrate the nearest neighbor approach with a small example for $k = 3$. Table 2 lists the three issues that are most similar to JBAS-3054, the issue presented in Section 2. All three issues needed the same amount of effort, namely 1 day. To predict the effort for JBAS-3054, the nearest neighbor approach takes the average of these efforts, which is again 1 day. This prediction is only two hours off the actual effort for JBOSS-3054 (1 day, 2 hours)—this difference is also called *residual*.

However, the appropriateness of the prediction for JBAS-3054 is questionable since the most similar issue shows only a similarity of 0.04800. Therefore speaking of similar issues is not always justified. In order to avoid making unjustified predictions, we introduce the concept of *thresholds* to kNN. The nearest neighbors with thresholds approach (α-kNN), considers only issues with a similarity of at least α and takes at most k issues. When there are no issues with enough similarity, our approach returns *Unknown*; for JBAS-3054, already a threshold of $\alpha = 0.1$ would have yielded *Unknown*. In our opinion, explicit ignorance is preferable over random predictions that might mislead the developer.

Table 2. Issues similar to JBAS-3054 "Use of isUserInRole from jsp does not work for JACC" (Effort: 1 day, 2 hours).

Issue	Title	Effort	Similarity
JBAS-1449	Update the ServerInfo memory ops to use the jdk 5 mbeans	1 day	0.04800
JBAS-2814	Use retroweaved jsr166 tests to validate backport-concurrent integration	1 day	0.04251
JBAS-1448	Update the ServerInfo listThreadDump to use the jdk 5 stack traces	1 day	0.04221

In this paper, we evaluate both nearest neighbor approaches, with and without thresholds. The measures we used for this, are described in the next section.

5. Evaluation Method

In order to evaluate the predictions by kNN and α-kNN we replay the history of the issue tracking system. For each issue we make predictions by using all previously submitted issues as training set in which we search for nearest neighbors. In particular, this means that for the first submitted issue, we cannot make any predictions as the training set is empty. Over time, the training set is increasing as sketched in Figure 4: for the newer issue Y the training set is larger than for issue X. We use the following measures in our evaluation:

Average absolute residual. The difference between the predicted effort p_i and actual effort e_i reported for an issue i is referred to as the *residual* r_i.

$$r_i = |e_i - p_i| = |\text{Actual effort} - \text{Estimated effort}|$$

Of course, the lower the residual, the more precise is the estimate. We use the *average absolute residual AAR* to measure the prediction quality of our model.

$$AAR = \frac{\sum_{i=1}^{n} r_i}{n}$$

This measure is closely related to the sum of absolute residuals which is regarded as an unbiased statistic [8]. It is noteworthy that we assume the direction of error to be immaterial, i.e., both over and under estimation are equally undesirable, although this may not be the case in all situations.

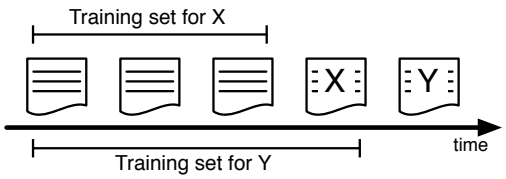

Figure 4. The training set for newer issue Y is larger than for issue X.

Percentage of predictions within $\pm$x%. The value of the *Pred(x)* measure gives the *percentage of predictions* that lie within $\pm x\%$ of the actual effort values e_i.

$$Pred(x) = \frac{|\{i \mid r_i/e_i < x/100\}|}{n}$$

Again, both over and under estimation are equally undesirable. For our experiments we used *Pred(25)* and *Pred(50)*.

Feedback. In case there are no similar issues, α-kNN reports *Unknown* and does not make a prediction. With *Feedback*, we measure the percentage of issues for which α-kNN makes predictions. Note that for kNN and α-kNN with $\alpha = 0$, the value of *Feedback* is 1.

The interpretation of *AAR* is fairly straightforward. A large number of good estimates would result in a low value of *AAR*, while many poor estimates would increase its value. One can infer that models delivering lower values of *AAR* fair better when compared to those that deliver larger errors. The problem with *AAR* is that it is greatly influenced by outliers, i.e., extremely large residuals might lead to misinterpretations. In order to appropriately describe the distribution of residuals, we additionally report the values for *Pred(25)* and *Pred(50)*. The larger the values for *Pred(x)*, the better the quality of predictions.

6. Experiments

We evaluated our approach on 567 issues from the JBoss data set. For our experiments we used the entire set of issues (referred to as JBOSS) and additional subsets based on *type* (bugs, feature requests, tasks) and *project* (Application Server, Labs project, Portal project, QA project). Table 3 lists the short names of these subsets, their issue counts, and average effort in hours (including standard deviation). Note that we only included projects with the four highest number of data points.

Initially, we planned to benchmark our predictions against estimates of experts (*timeoriginalestimate_sec*); however, there were not enough issue reports with these estimates to allow significant comparisons. Instead, we compared our results against a *naïve* approach that predicts the average effort of past issues without using text similarity, since this would be the simplest method to estimate effort. In our experimental setup, this approach corresponds to α-kNN with $\alpha = 0.0$ and $k = \infty$.

Table 3. Datasets (average effort in hours).

Name	Description	Count	Avg. Effort
JBOSS	issues for *JBoss* (see Table 1)	567	15.9±32.0
BUGS	*bugs* for JBoss	125	4.8± 6.3
FEATURES	*feature requests* for JBoss	149	21.0±44.7
TASKS	*tasks* and *sub-tasks* for JBoss	293	18.2±29.9
JBAS	issues for *Application Server*	51	12.5±22.8
JBLAB	issues for *Labs* project	125	11.7±25.5
JBPORTAL	issues for *Portal* project	82	14.0±23.7
JBQA	issues for *QA* project	71	16.1±17.5

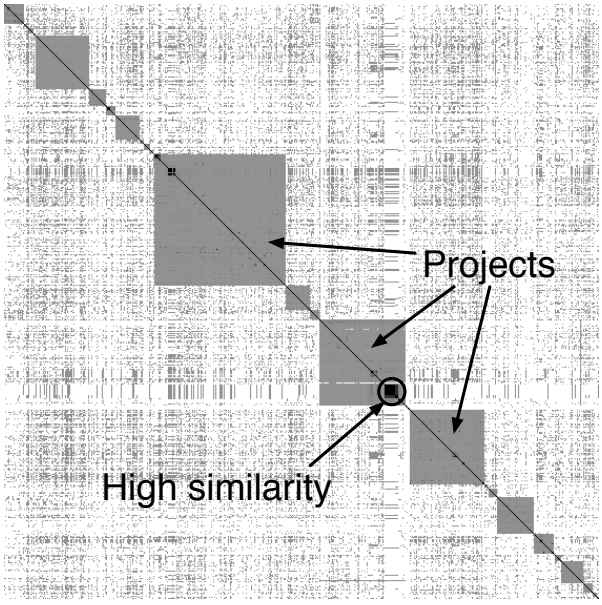

Figure 5. Similarity between issue reports.

In this section, we first present experiments for text similarity (Section 6.1), kNN vs. α-kNN (Section 6.2) and for the breakdown to project and issue type (Section 6.3).

6.1. Similarity of Issues

Our approach relies on text similarity to identify nearest neighbors. Therefore, we first visualize how similar the issues are to each other. In Figure 5, each pixel represents the similarity between two issue reports of the JBOSS dataset; white indicates no similarity, light colors indicate weaker similarity, and dark colors indicate stronger similarity.

The issues are sorted by *key* (e.g., JBAS-3054) such that issues of the same project are grouped together. Since Jira includes the key (and thus the project) in the title of an issue

0-7695-2950-X/07 $20.00 © 2007 IEEE

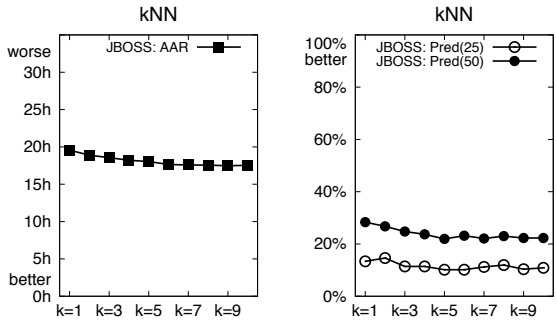

Figure 6. Accuracy values for kNN.

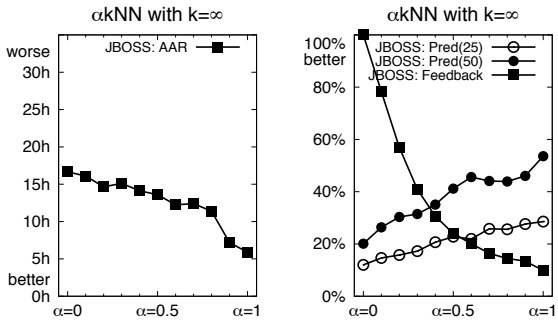

Figure 7. Accuracy for α-kNN with k=∞.

report, issues that are within the same project have a guaranteed non-zero similarity (because of the shared project identifier). As a consequence, projects can be spotted in Figure 5 as the gray blocks around the diagonal.

Overall, the similarity values are low (indicated by the rareness of dark pixels), only few issues have high similarity values. This observation supported our decision to evaluate α-kNN in addition to kNN.

6.2. Nearest Neighbor without/with Thresholds

Figure 6 shows the *AAR*, *Pred(25)* and *Pred(50)* values for when varying the k parameter from 1 to 10. The *AAR* values improve with higher k values, i.e., the average error decreases. Since, the *Pred(25)* and *Pred(50)* values worsen (i.e., decrease), there is no optimal k in our case. Overall the accuracy for kNN is poor. On average, the predictions are off by 20 hours; only 30% of predictions lie within a $\pm$50% range of the actual effort. We explain this poor performance by the diversity of issue reports (see Section 6.1).

The α-kNN approach takes only similar nearest neighbors into account and therefore should not suffer as much from diversity as kNN. In Figure 7, we shows the accuracy values for α-kNN when varying the α parameter from 0 to 1 in 0.1 steps. We used $k = \infty$ for this experiment to eliminate any effects from the restriction to k neighbors.

The combination of $k = \infty$ and $\alpha = 0$ uses *all previous* issues to predict effort for a new issue (naïve approach without text similarity). It comes as no surprise that accuracy is at its lowest, being off by nearly 35 hours on average.

However, for higher α values, the accuracy improves: for $\alpha = 0.9$, the average prediction is off by only 7 hours and almost every second prediction lies with $\pm$50% of the actual effort value. Keep in mind that higher α values increase the accuracy at the cost of applicability; for $\alpha = 0.9$, our approach makes only predictions for 13% of all issues. Our future work will focus on increasing the *Feedback* values by using additional data, such as discussions on issues.

6.3. Breakdown to Project and Issue Type

In Figure 8, we compare the accuracy for different projects (upper row) and different types of issues (lower row). The plots show the values for *AAR* (left column), *Pred(50)* (middle column), and *Feedback* (right column) when varying the α parameter from 0 to 1 in 0.1 steps. In all cases, textual similarity beats the naïve approach ($\alpha = 0.0$).

For JBAS we face a high diversity: no two issues in this project have a similarity of more than 0.5; as a consequence α-kNN stops making predictions for α parameters $\geq$0.5. The results for JBLAB and JBQA are comparable to the results for JBOSS in Figure 7. For JBPORTAL, we achieve a relatively high accuracy: for $\alpha = 0.8$, the α-kNN approach is off by only two hours on average, and more than 60% of all predictions lie within $\pm$50% of the observed effort.

Surprisingly, the accuracy values vary greatly for different types of issues such as bugs, feature requests, and tasks (lower row of Figure 8). For BUGS, the average error *AAR* is constantly below 4 hours (for $\alpha \geq 0.4$ even below 1 hour). Since also the *Pred(50)* values are promising (above 50% in most cases), we believe that α-kNN is an appropriate tool to provide developers with early estimates on effort once a new bug report is submitted.

7. Threats to Validity

As any empirical study, this study has limitations that must be considered when interpreting its results.

Threats to external validity. These threats concern our ability to generalize from this work to industrial practice. In our study, we have examined a total of 567 issue reports and four JBoss subprojects. All reports researched are real reports that occurred in a large industrial framework, and which were mostly addressed and fixed by paid developers.

For training and evaluation, we only examined those issue reports for which effort data was available. These issue reports may not necessarily be representative for all issue reports, and therefore cannot be used to generalize for the entire JBoss framework.

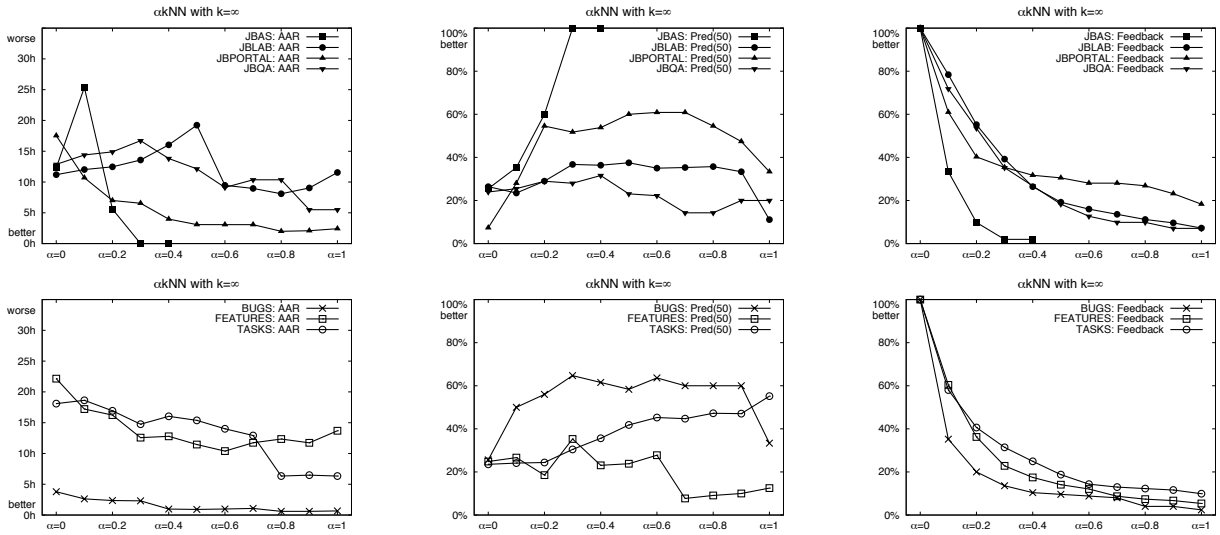

Figure 8. Accuracy values for α-kNN with k=∞ for different projects and issue types.

Although the researched projects themselves are quite different, we cannot claim that their bug databases and effort data would be representative for all kinds of software projects. Different domains, product standards, and software processes may call for alternate processing of bug reports as well as efforts—and come up with different results. Due to these threats, we consider our experiments as a *proof of concept,* but advise users to run an evaluation as described in this work before putting the technique to use.

Threats to internal validity. These threats concern our ability to draw conclusions about the connections between independent and dependent variables. The chief threat here concerns the quality of the data. The effort, as entered into the bug database, is not tracked during the task, but rather estimated *post facto* by developers, i.e., after the fix.

We asked the JBoss developers whether they found their effort was accurate. One of the JBoss managers explicitly warned us against putting too much faith in the effort data, as he generally considered post facto estimates to be too imprecise. On the other hand, one should note that the effort data is entered voluntarily, and one of the JBoss developers confirmed that *"For the cases were **[sic]** the time spent is reported, this should be relatively accurate."* Our communication with JBoss developers emphasizes that contact with people directly involved in collecting and recording data during development cannot be neglected. Such contact is vital for resolution of issues, clarifications regarding the data and on the whole, better understanding [11].

Another source of threats is that our implementation could contain errors that affect the outcome. To control for these threats, we ensured that the *diagnosis tools had no access to the corrected versions* or any derivative thereof. We also did careful cross-checks of the data and the results to eliminate errors in the best possible way.

Threats to construct validity. These threats concern the appropriateness of our measures for capturing our dependent variables, i.e., the effort spent for an issue. This was justified at length when discussing the statistical approach.

8. Related Work

While much emphasis has been laid on estimating software cost or effort over the last three decades [3], to the best of our knowledge, little has been done to predict effort to fix software bugs. Previously, Song *et al.* [13] used association rule mining to classify effort in intervals using NASA's SEL defect data. They found this technique to outperform other methods such as PART, C4.5 and Naïve Bayes.

A self-organizing neural network approach for estimating effort to fix defects, using NASA's KC1 data set, was applied by Zeng and Rine [15]. After clustering defects from a training set, they computed the probability distributions of effort from the clusters and compared it to individual defects from the test set to derive a prediction error. While their technique seemed to have performed favorably, unfortunately, they used magnitude of relative error for evaluation which is asymmetric [8, 6] casting doubt on the validity of the results.

Manzoora [9] created a list of pointers for experts to keep in mind when estimating effort for defect correction. They were grouped on the basis of the type of defect and programming environment in use, such as programming practices, application architecture, and object-oriented design.

Another contextually relevant work includes automated bug triaging by Anvik et al. [2], which in part motivated this research. They developed a technique that analyzed bug descriptions to automatically assign them to developers. The underlying conception behind their work was that new bugs

should be assigned to those developers who have previously fixed similar bugs. Their approach could incorporate our estimations, since the choice of who should fix a bug also depends on the workload of developers and the estimated effort and not only technical skills.

Other bug triaging approaches include a Naïve Bayes approach by Čubranić and Murphy [5] and a probabilistic text similarity approach by Canfora and Cerulo [4]. Anvik et al. observed that the quality of publicly available bug databases is mixed [1], which emphasizes the need to replicate our study on projects other than JBoss; unfortunately, only few open source projects record effort data.

9. Conclusion and Consequences

Given a sufficient number of earlier issue reports, our automatic effort predictors beat the naïve approach; in particular, our predictions are very close for bug reports. As a consequence, it is possible to predict effort at the very moment a new bug is reported. This should relieve managers who have a long queue of bug reports waiting to be estimated, and generally allow for better allocation of resources, as well for scheduling future stable releases.

The performance of our automated model is the more surprising if one considers that our effort predictor relies only on two data points: the title, and the description. Our future work aims at leveraging further information:

- Issue reports contain *additional fields*—such as version information, stack traces, or attachments—which can be specifically exploited. This may require alternative feature-based models integrating text similarity.
- To estimate effort, a project expert can exploit all her knowledge about the problem domain, the project history, and of course, about the software itself. How can we leverage this information? We are currently focusing on techniques which map the issue reports to the related code fixes, such that we can leverage *features of fixes and code* as well.
- On the statistical side, the nearest neighbor approach has some shortcomings when applied to cost estimation [10]. One proven way to improve estimation accuracy could be by *filtering outliers* in the data base, i.e., issues with abnormal effort values.
- Last but not least, we would like to extend our research to further projects and see how the results generalize.

However, one should not forget that the current estimates are not yet perfect. The big question looming behind this work is: What is it that makes software tedious to fix? Are there any universal features that contribute to make debugging hard? Or is it that debugging efforts are inherently unpredictable? With the advent of open source bug data, and sometimes even actual effort and estimated effort data, we finally have a chance to address these questions in an open, scientific, and competitive way—and to come up with solutions that address the very core questions of software engineering.

Acknowledgments. We are grateful to the JBoss team members who responded to our questions regarding their data and to the reviewers for their valuable comments.

References

[1] J. Anvik, L. Hiew, and G. C. Murphy. Coping with an open bug repository. In *Proc. of the OOPSLA workshop on Eclipse technology eXchange*, pages 35–39, 2005.

[2] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *Proc. of the International Conference on Software Engineering*, pages 361–370, Shanghai, China, May 2006.

[3] B. Boehm, C. Abts, and S. Chulani. Software development cost estimation approaches — A survey. Technical Report 2000-505, Uni. of California and IBM Research, Los Angeles, USA, 2000.

[4] G. Canfora and L. Cerulo. Supporting change request assignment in open source development. In *Proc. of ACM Symposium on Applied Computing*, pages 1767– 1772, Dijon, France, April 2006.

[5] D. Čubranić and G. C. Murphy. Automatic bug triage using text categorization. In *Proc. International Conference on Software Engineering & Knowledge Engineering (SEKE)*, pages 92–97, 2004.

[6] T. Foss, E. Strensrud, B. Kitchenham, and I. Myrtveit. A simulation study of the model evaluation criterion MMRE. *IEEE Trans. on Software Engineering*, 29(11):985–995, November 2003.

[7] E. Hatcher and O. Gospodnetic. *Lucene in Action*. Manning Publications, December 2004.

[8] B. Kitchenham, L. M. Pickard, S. G. MacDonell, and M. Shepperd. What accuracy statistics really measure. *IEE Proceedings - Software*, 148(3):81–85, 2001.

[9] K. Manzoora. A practical approach to estimate defect-fix time. http://homepages.com.pk/kashman/, 2002.

[10] R. Premraj. *Meta-Data to Enhance Case-Based Prediction*. PhD thesis, Bournemouth University, UK, 2006.

[11] R. Premraj, M. Shepperd, B. Kitchenham, and P. Forselius. An empirical analysis of software productivity over time. In *In Proc. of the 11th IEEE International Software Metrics Symposium*, Como, Italy, September 2005. IEEE.

[12] M. Shepperd and C. Schofield. Estimating software project effort using analogies. *IEEE Trans. on Software Engineering*, 23(12):736–743, November 1997.

[13] Q. Song, M. Shepperd, M. Cartwright, and C. Mair. Software defect association mining and defect correction effort prediction. *IEEE Trans. on Software Engineering*, 32(2):69–82, February 2006.

[14] A. Zeller. *Why Programs Fail*. Morgan Kaufmann, 2005.

[15] H. Zeng and D. Rine. Estimation of software defects fix effort using neural networks. In *Proc. of the Annual International Computer Software And Applications Conference (COMPSAC '04)*, Hong Kong, Sept. 2004. IEEE.