Automating Algorithms for the Identification of Fault-Prone Files

Thomas J. Ostrand, Elaine J. Weyuker, Robert M. Bell
AT&T Labs - Research
180 Park Avenue
Florham Park, NJ 07932
(ostrand,weyuker,rbell)@research.att.com

ABSTRACT

This research investigates ways of predicting which files would be most likely to contain large numbers of faults in the next release of a large industrial software system. Previous work involved making predictions using several different models ranging from a simple, fully-automatable model (the LOC model) to several different variants of a negative binomial regression model that were customized for the particular software system under study. Not surprisingly, the custom models invariably predicted faults more accurately than the simple model. However, development of customized models requires substantial time and analytic effort, as well as statistical expertise. We now introduce new, more sophisticated models that yield more accurate predictions than the earlier LOC model, but which nonetheless can be fully automated. We also extend our earlier research by presenting another large-scale empirical study of the value of these prediction models, using a new industrial software system over a nine year period.

Categories and Subject Descriptors: D.2.5 [Software Engineering]: Testing and Debugging – *Debugging aids*

General Terms: Experimentation

Keywords: software faults, fault-prone, prediction, regression
model, empirical study, software testing

1. INTRODUCTION

In earlier research, we developed and assessed prediction models designed to identify which executable files of a large multi-release industrial software system were likely to contain the largest numbers of faults in the next release. This information could then be used by testers to help prioritize testing efforts and by developers to help identify files that might need to be re-architected. We based our first prediction models on an initial study [20] that looked at which characteristics of a large software system were most closely associated with files that turned out to be particularly problematic, and refined the models after studying two additional large production systems [22, 4].

This paper reports new research progress towards our ultimate goal: an automated tool for testers and developers to identify the files most likely to be problematic in the future. This paper makes several important contributions.

The primary new contribution follows from our observation that there is a commonality among the customized models that we built for each of three earlier systems, and that we can exploit this commonality to fully automate our prediction process. The case studies of these three systems are outlined in Section 2. Although the studies provide evidence that our prediction methodology could potentially be of great value to developers and testers, they provide insufficient guidance for how a development project would take advantage of the methodology. This is because it is very time consuming to do the required data extraction and analysis needed to build the models, and few projects have the luxury of extra personnel to do these tasks or the extra time in their schedules that will be needed. In addition, statistical expertise was needed to actually build the models, and that is rare to find on most development projects.

For these reasons, we have developed and assessed in this paper four different models derived from our earlier models. Three of the models are completely automatable, while one is customized for a fourth large industrial system that is the subject of a new case study, intended to evaluate both the usability and effectiveness of the models. Models 1 and 2 are completely pre-specified, and only require information about the size of the system and changes made to its files. For Model 3, the equations are pre-specified, while the coefficients used to predict faults in a future release are based on fault information from previous releases. These three models are easily adaptable to any large system with a reasonably regular release schedule; as a result, they can be incorporated into an automated fault prediction tool. For Model 4, both the equations and the coefficients are customized for the system on the basis of its first five releases.

Using these observations, we provide evidence that a prediction model can be incorporated into a tool that requires no particular expertise to use, and yet can accurately predict which files are likely to contain the largest numbers of faults in the next release. The data extraction portion of the tool has already been built, and we have selected the statistical package that will be integrated with the back-end data extraction function to provide a fully automated tool.

ISSTA'07, July 9–12, 2007, London, England, United Kingdom.

The second contribution is the case study of the fourth system, which is substantially different in two important ways from the previous three. First, it has been in the field for more than nine years, much longer than any of the other systems. Consequently, we can investigate whether the prediction models are appropriate for very mature systems that are quite stable and undergoing little modification. Second, this system was produced and maintained by a different corporation from the one responsible for the three earlier systems. Thus, the development environment, the prevailing corporate culture, and the development paradigms are all likely to differ from those of the earlier systems, providing a good test for the adaptability of the prediction models to a wider variety of systems, and helping to convince potential users of their value. In our experience, practitioners won't even consider using a new technology without evidence that it has worked on a substantial number of real systems of varying types. It is very unlikely that practitioners will be convinced that a new tool is worth learning and evaluating merely on the basis of its demonstration on toy systems or on systems much smaller than the ones they normally develop and maintain.

The paper's third contribution is a different way of assessing the cost effectiveness of each of the four models described. In our earlier papers, we generally compared the percentage of faults contained in identified files to the percentage of files selected. Although we believe that this is often the best assessment criterion, we also present our findings in terms of the percentage of lines of code included in the identified files and discuss the conditions under which these alternatives might be appropriate.

The remainder of the paper is organized as follows. Section 2 summarizes our earlier findings, to provide a perspective on our current findings. Section 3 describes the requirements for applying the sorts of prediction model we have developed. Section 4 provides information about the maintenance support system that is the subject of the current case study. In Section 5 we describe each of the models assessed in this paper. Section 6 compares the effectiveness of the alternative models and considers ways to evaluate whether the effort invested in fault-prediction is cost-effective. Section 7 discusses related work performed by other research groups, and Section 8 presents our conclusions.

2. BACKGROUND

Our original negative binomial regression model was developed for an inventory system with more than 1900 files, 500,000 LOC and 5797 faults in 17 successive releases over four years of field history. A model based on file characteristics and the fault and change history of the early releases was used to predict which files were likely to contain the largest numbers of faults in later releases. By identifying the 20% of the files predicted to contain the largest numbers of faults, this model identified, on average, the location of 83% of the faults over the 17 releases studied. When the first 12 releases were used to make those predictions for releases 13-17, the 20% of the files selected contained, on average, 89% of the system's faults.

We applied essentially the same model to a service provisioning system with more than two years in the field. Because that system had many fewer faults, a total of 307 faults over nine releases, we did not attempt the same type of model development and variable selection. Instead, we simply estimated coefficients for a minor variation of the model developed for the inventory system. We observed remarkably similar results: the 20% of the files selected by the model contained 83% of the faults. This system had 2,241 files during the ninth and final release that we followed, and roughly 438,000 LOC.

Very few of the faults observed in these two systems occurred after release to the field. Over 80% of the inventory system faults were found during unit testing, and most of the remaining faults were caught during system testing. Defect reporting for the service provisioning system began only during system test, and almost all faults were observed during that phase. Details of these results were described in [22].

Following the successful prediction results on these two systems, we considered three different variants of the negative binomial model applied to the 2.5 year history of an automated voice response system, which grew to about 329,000 LOC in its latest version. Despite this system's lack of regularly scheduled releases, the most accurate model selected 20% of the system's files that contained, on average, nearly 75% of the faults that were actually detected, again mostly during the system test phase [4].

While all these findings were very promising, model development for these systems required extensive expertise and effort to specify the best form for predictors (e.g., transformations) and to select among predictors. This could discourage utilization of the methodology by practitioners for their systems. Furthermore, it may take a year or more to collect data on enough faults to support model development, thereby delaying the benefits.

Despite these potential obstacles to model specification, there were promising findings. First, the model developed for the inventory system transferred to the provisioning system with no drop in performance. Second, although model details differed over the systems we had studied, the most important predictors tended to be common across systems. This commonality suggests investigating whether the benefits can be attained using a model with a fixed, pre-specified set of predictors, where only the coefficients need be determined for the current system, or even a completely prespecified model determined by results from previous systems. In this paper we investigate both of these strategies.

3. MODEL AND TOOL REQUIREMENTS

In order for a fault prediction tool to be widely usable by software developers and testers, it should require little or no user expertise in data mining, data analysis, or statistics, and should require minimal human intervention and manual overhead. Our current research is aimed at devising a standard prediction model that can be integrated with automated data extraction tools, resulting in a tool that can be easily applied to predict fault-proneness for many different types of large software systems.

Each of the subject systems we have studied to date has used a variant of the same commercially available, integrated version control/change management system. Any change made to the system must be documented in a *modification request* (MR).

Our approach has been to base the predictions on objectively assessable factors obtainable from the MRs and the version control system. These include file size; whether the file was new to the system in the sense that this is the first re-

lease in which it appeared; whether the file contained faults in earlier releases, and if so, how many; how many changes were made to the file in previous releases; and the programming language used to implement the file. The models developed for the previous systems were customized based on observations of these factors made during the first few releases.

The data extraction part of the fault prediction tool should perform its tasks without human intervention. For each of the previously analyzed systems, we wrote custom-designed scripts that extract the data needed from the MR database. Because all of the previous systems, as well as the current one, use the same MR system, we have now implemented a common table-driven script that can be parameterized for each system. A generalized C tool to do the data extraction has been implemented and validated. This will serve as the back-end of the automated fault-proneness prediction system.

Of course, it is quite possible that other development projects will use different modification request systems, with different data storage formats. For each such change management system encountered, therefore, we plan to build a different back-end to the data extraction tool so that practitioners will only have to select the MR system used and the appropriate databases will be accessed to extract the needed data. In each case, the extracted data will provide the information needed by the prediction portion of the tool in a uniform format.

The simplest implementation of the prediction portion would be a fully pre-specified model, whose coefficients are standard and independent of the software system for which the predictions are being made. The coefficients would be based on results determined during our earlier studies. This would allow us to completely bypass model development and to speed implementation to the greatest extent possible. If the accuracy of the predictions were acceptably high, even if not as high as those produced by the custom-built prediction models, this might be a reasonable tradeoff: slightly reduced accuracy for automation.

If the system-independent model does not produce acceptable results, we could use a model with a fixed, pre-specified set of predictors, where only the coefficients need be determined for the current system. Although this approach would require software to fit negative binomial regression models, it would avoid the need for expert judgment about statistical modeling and would result in quicker implementation than for a customized model. Essentially in this case we would be adapting the model based on the new data extracted for a given new software system in some automatic or semi-automatic way.

Both of these streamlined approaches have been applied to the fourth large system, which we designate the "maintenance support system"; it is a major component of a larger system that supports the maintenance activities for a number of products sold by a large international company. As will be seen, the model developed using the pre-specified set of predictors performed as well as the customized model, supporting our goal of a fully automated tool that can carry out both the data extraction and fault prediction processes.

4. DATA FOR THE CASE STUDY

We analyzed data from 35 releases covering approximately nine years of field experience. During that period, the number of executable files grew from 354 at the end of Release 1 to 668 at the end of Release 35, and the LOC grew to approximately 500,000. Intervals between releases generally ranged from three to five months, with the main exceptions being after Releases 1 (8.5 months), 2 (56 days) and 3 (40 days).

Each release after the first one included a combination of files that were new in the current release and others that existed in the previous release. New files might be added at any point during the lifetime of a release. Because files added early in a release had more chance to be tested and therefore more opportunity for faults to be detected, we define a variable called *Exposure*, which is the fraction of the release for which a new file existed. The Exposure variable was set to 1 for all files that existed in the previous release.

Ten different languages contributed executable files to this system. At the last release, the most frequently used languages included C++ (28%), C++ with embedded SQL (28%), pure SQL (12%), CTL (data-loading scripts for SQL, 12%), and SH (6%).

MRs can be submitted for various reasons, including to report a system failure or problem, to request a maintenance update, or to request the addition of new functionality. Because the MR form does not have a field that explicitly states the reason for its creation, we define a *fault MR* to be one that is submitted during any of the following stages of the software process: system test, end-to-end test, operational readiness test, user acceptance test, or customer usage. This simple rule captures those MRs that are investigated and must be resolved following system integration, and provides a fairly close approximation to the MRs that are created because of an actual fault in the system. A *fault* is defined as a change to an individual executable file in response to a fault MR. Therefore, if a given fault MR causes N different files to be changed, then we count that as N distinct faults.

Counted in this way, a total of 1545 faults was spread across the 35 releases. With the exceptions of four releases with fewer than 8 faults each, there were at least 18 faults during each release. The vast majority of all faults were encountered during an internal testing phase, prior to the software's release to customers.

MRs also provided information about the total number of changes to executable files during each release for reasons other than to fix faults. These might include planned enhancements such as the addition of new features required for specific releases in the original requirements or specification document. Unplanned enhancements are also distinguished from faults and counted as changes but not faults. These might represent new features or functionalities added to the system that were not originally envisioned by the requirements writers. Changes of this sort are especially common in systems like the maintenance support system that have very long lifetimes, as new technology becomes feasible, and competitors' software systems provide new features.

5. MODELS ASSESSED IN THIS PAPER

As in [22, 4], we use negative binomial regression [15] to model the number of faults in a file during a specific release. The unit of analysis is the combination of a file and release. That means that each file contributes as many observations to a regression as the number of eligible releases for which it existed.

Negative binomial regression models the logarithm of the

Variable	Coefficient
log(KLOC)	0.7
New file	2.0
Changed file	1.0
log(exposure)	0.6

Table 1: Coefficients for Model 2

expected number of faults as a linear combination of the explanatory variables. Specifically, the expected number of faults equals $\gamma_i e^{\beta' x_i}$, where the random variable γ_i has a gamma distribution with mean 1 and variance $\sigma^2 \geq 0$. The variance σ^2, known as the *dispersion parameter*, is included to account for greater variance in the faults counts than would be expected from a Poisson distribution.

We estimated model parameters by maximum likelihood, with all computations performed using Version 9.1 of SAS [25]. A fuller description of how this methodology was applied to a different system during an earlier case study can be found in [4]. Once a model has been estimated, we compute predicted numbers of faults for every file in the next release, to prioritize files for purposes of testing and to alert developers that it might be desirable to rearchitect certain files.

For the maintenance support system, we excluded Release 1 from our modeling because it appeared to differ systematically from later releases in ways that might make its results unrepresentative of what followed. As noted above, Release 1 lasted twice the norm. In addition, it had a very high rate of file changes, but only 22 changes were identified as faults because almost no system testing occurred during the release. Consequently, we were concerned that any inferences drawn from Release 1 about fault rates or the characteristics predicting faults could be misleading.

We assess four distinct models/algorithms for the maintenance support system. The four alternatives differ in the degree of pre-specification. The first two are fully specified formulas, including coefficients. These models are obviously fully automatable, requiring only a process for collecting the needed data. The third is a pre-specified model, whose coefficients are to be estimated from data for the current system. Although this alternative requires fitting negative binomial regression models at each release, it can also be automated. The fourth, an open-ended process for development and estimation of a model, serves as an attempted best-case alternative, for comparison. These are the four alternatives we considered:

1. *Lines of code (LOC) only.* In this model, the files are simply ordered from longest to shortest, and file length is the sole factor determining the fault-proneness prediction. This option was included because we found in our earlier studies that the size of the file was generally the most important characteristic for determining fault-proneness. For these earlier systems, although the predictions made using this very simple model were far better than might be expected, we found that the predictions made by the more complicated models were definitely more accurate.

2. *Pre-specified formula, including coefficients.* The files are ordered by a linear combination of four variables. These variables and their coefficients are shown in Table 1.

   For this model, both *New file* and *Changed file* are dummy variables. For *New files*, exposure is the proportion of time during the release that the file existed. Exposure is always 1 for *Old files*. The coefficients are set at approximately the average of values estimated from the inventory system and automated voice response systems studied in earlier research as described in [22, 4]. This model was limited to predictors judged likely to be essential, and for which coefficients could be pre-specified with moderate confidence. We do not specify an intercept term because this would not affect the ordering of files that is needed for our assessment.

3. *Pre-specified model, coefficients estimated from data.* In this case the coefficients to predict Release N are based on a model fit to Releases 2 to (N-1). The set of variables used are: log(KLOC); dummy variables for New files (Age = 0), Age = 1, and Age = 2-4 (where applicable), with Age > 4 as the reference set; the logarithm of Exposure; square roots of Changes in the prior release, Changes two releases ago, and Faults in the prior release; dummy variables for selected programming languages; and dummy variables for all but one release. To reduce the potential of overfitting for less prevalent programming languages, we included in the model only those languages with a cumulative total of 20,000 LOC (20 KLOC) across releases, with an average of at least 2 KLOC per release in recent releases. This allowed dummy variables for files written in C or C++ immediately and for four additional languages after a few releases.

4. *Customized model based on Releases 2 to 5.* The customized model was more of a process than an actual model. As for the other case studies, this process of exploratory data analysis allowed evaluation of alternative transformations of explanatory variables and variable selection, including interactions. The final form of the customized model was based on a combination of empirical results such as statistical significance tests and expert judgment informed in part by the previous case studies.

Besides any differences in terms of predictive accuracy and ease of use, the four models also differ in terms of when predictions become available. Because Models 1 and 2 are completely pre-specified, we are able to start predictions for those models with the first normal release, Release 2 for this system. In contrast, because the coefficients for Model 3 need to be estimated from data for at least one prior release, we begin predictions with Release 3. For Model 4, the first predictions are delayed until Release 6 because model development called for additional training data (Releases 2 to 5).

6. FINDINGS

6.1 Regression Results for Models 3 and 4

Table 2 displays results of negative binomial regressions for Models 3 and 4, fit to data through Release 20 (i.e., for predicting Release 21). The table omits results for the intercept and the dummy variables for releases, because those specific coefficients provide little, if any, insight about how

Predictors	DF	Chi Square
Releases	18	165.9
Prior changes	2	134.7
Language types	5	121.0
LOC	1	117.6
File age	3	44.1
Prior faults	1	14.0
Exposure	1	5.9

Table 3: Likelihood Ratio Chi Squares Associated with Groups of Predictor Variables in Model 3

the models would perform for other systems. Across releases, the estimated coefficients go up and down without any clear pattern. Excluding the programming language dummy variables, all the coefficients in Model 3 were statistically significant in the anticipated direction. The values of "$-\infty$" in the row for C reflect the fact that no faults occurred through Release 20 for any files written in that language. "m4" is a special-purpose scripting language used in the system.

Table 3 shows likelihood ratio chi square statistics associated with the sets of predictor variables in Model 3, as a measure of the marginal impact of each set on the model fit. The greatest overall impact was for release number, which was fit by a series of 18 dummy variables. Of more interest, three other groups based on many fewer degrees of freedom each proved nearly as powerful: prior changes, language, and lines of code. The other predictive factors were substantially less important for this system.

As noted above, we excluded data for Release 1 from our models due to concern about that release being unrepresentative of later releases. It turns out that our concern was unwarranted. For the models that we fit, including data for Release 1 would have had negligible impact on either the estimated regression coefficients or the resulting predictions. For example, two sets of predictions for Release 21, one based on including and the other on not including Release 1, had a Spearman rank correlation of 0.999.

Our development of a completely customized model (Model 4) for the maintenance support system produced a model similar to the pre-specified Model 3. This is not surprising, because our experience from earlier studies [22, 4] indicated which factors were likely to be of greatest importance, and although this had to be validated, it provided a starting point for the customized model. Modeling faults for Releases 2 to 5 confirmed the importance of log(KLOC), log(Exposure), and dummy variables for release number. This analysis also confirmed much higher fault rates for new files and for files with faults in the prior release. In addition, comparison of log likelihoods between models using alternative transformations for prior changes and prior faults suggested that square root transformations were preferable to either raw counts or indicators of "any" versus "none" for both variables.

However, the development of Model 4 did lead to several noteworthy differences. The analysis suggested that a dummy variable for files with Age (number of previous releases in the system) equal to either 1 or 2 was appropriate. We note, however, that it is difficult to infer the best way to handle file age with only a few releases worth of data. Indeed, the pre-specified formulation of Model 3 appears to have worked better. For Model 4, exploratory analysis led to the exclusion of the count of changes two releases ago in any form. This contrasted with our finding from the case study of the automated voice response system [4] and therefore our definition of Model 3 for the maintenance support system. The t-statistic for (Prior Prior Changes)$^{1/2}$ appears to confirm its inclusion in Model 3. The same exploratory analysis led to a substantially reduced set of dummy variables for programming language. Finally, certain unexpected findings for Release 2 (e.g., there were no faults for new files) suggested that Release 2 may not be predictive of what follows. Consequently, for Model 4, we chose to use Releases 3 to (N-1) as the training set for Release N.

System	LOC model	Customized model
Inventory	73%	83%
Provisioning	74%	83%
Voice Response	56%	75%

Table 4: Percentage of Faults in Top 20% of Files for Previously Studied Systems

6.2 Predictive Accuracy

For comparison purposes, we provide a summary of our prediction results from the earlier systems studied. For each of those systems, Table 4 shows the average percentages of faults appearing in the 20% of files predicted to have the most faults based on either LOC or a model customized for the particular system.

Table 5 summarizes prediction results for the maintenance support system studied in this paper. Using groups of five releases, we report the percentage of faults contained in the top 20% of files as ordered by the four models. Summary Rows A and B show mean values for each of the models over all releases through Release 35, as well as the mean values for each of the models over Releases 6 through 35. The reason we considered that restricted range is that data through Release 5 were used to train Model 4.

The results for Model 3 (the pre-specified model) and Model 4 (the customized model) are consistently at or near the best of the four models–with each exceeding 80% of faults on average. Both models tended to improve over time as the amount of training data increased. Agreement between the two models was high, but far from perfect. For example, for Release 21, the Spearman correlation between the two sets of predictions was 0.87. For the top 120 files (20% of 602) identified by each model, there were 94 files in common.

Overall, for the releases in common, Model 3 did slightly better. This is a very encouraging observation since Model 3 is one which we expect to be able to fully automate. We also see from this table that results for Model 3 are comparable to those obtained using a customized model for the inventory system and better than the results observed for the voice response system (Table 4).

Model 2 trails Models 3 and 4, but still reaches 75% when averaged across all releases. Model 2 differs from Model 3 in two respects: it uses a much shorter list of predictor variables, and the coefficients of the predictors are pre-specified and fixed for all releases. We also evaluated a model intermediate between 2 and 3, with the same short set of predictor variables as Model 2, but using coefficients fit to data

Variable	Model 3		Model 4	
	Coefficient	t-statistic	Coefficient	t-statistic
Log(KLOC)	.519	10.75	.528	12.68
New file	1.865	6.16	1.676	5.66
Age = 1	.918	4.05	—	—
Age = 1-2	—	—	.172	1.00
Age = 2-4	.355	2.28	—	—
Log(Exposure)	1.117	2.16	1.143	2.21
(Prior Changes)$^{1/2}$	.441	7.14	.716	11.22
(Prior Prior Changes)$^{1/2}$	.323	5.67	—	—
(Prior Faults)$^{1/2}$	.387	3.78	.383	3.61
C++	-1.260	-4.72	—	—
C++ with SQL	-.555	-2.10	—	—
m4	.389	1.49	1.163	7.64
SQL	-.162	-.53	—	—
C	$-\infty$	NA	$-\infty$	NA
CTL	.207	.78	.863	4.51

Table 2: Coefficients for Models 3 and 4 after Release 20

Release Number	Model 1	Model 2	Model 3	Model 4
2-5	50	58	55*	NA
6-10	63	66	78	77
11-15	56	66	71	68
16-20	68	80	84	81
21-25	77	91	89	91
26-30	72	83	90	87
31-35	63	79	92	92
A: Mean % of Faults, all Releases	64.4	75.2	81.3*	NA
B: Mean % of Faults, Rel 6-35	66.3	77.4	83.9	82.6
C: Mean LOC in 20% of Files, all Releases	69.5	66.1	59.4*	NA
D: Mean LOC in 20% of Files, Rel 6-35	69.9	66.8	60.2	62.9

* Starting with Release 3.

Table 5: Percentage of Faults in Top 20% of Files, by Model for the Maintenance Support System

for Releases 2 through N-1, as in Model 3. This intermediate model bridged 60% of the gap between Models 2 and 3 over the full set of common releases. The intermediate model averaged finding 79.0% of faults over Releases 3 to 35, compared with 75.6% and 81.3% for Models 2 and 3, respectively. These results suggest that the lower performance of the original Model 2 is mainly, but not completely, due to the pre-specification of its coefficients. In all cases, these results represent the percentage of faults included in the 20% of the files identified by the model as likely to contain the largest numbers of faults.

We had originally introduced the Lines of Code Model because size was observed to be the most important factor in preliminary studies and the model requires no expertise to compute; one only needs to sort the files based on their size and select the biggest files for consideration. We saw that although it was less successful than the fully custom prediction model, it nonetheless did far better than would be expected if one randomly selected the order of files to be tested. However, the LOC model consistently falls short of the other options, especially after about Release 15.

6.3 Assessment of Cost Effectiveness

Predicting fault-prone files and concentrating testing effort on them is only cost-effective if the prediction-guided testing can locate more bugs with the same effort as testing without the predictions, or at least the same number of bugs with reduced effort. To do this analysis, it is necessary to quantify the effort required to test a given piece of software, as well as the number of bugs expected to be found in the software without using prediction to guide testing.

Accurate estimation of test effort is an extremely complex task that must be based on many factors [9]. Effort can be related to measures of various different syntactic or semantic aspects of the software, some of which are the number of lines of code, the number of modules, the number of files, the number of functional units, and the number of use cases. Effort also depends on the software's intended use, its expected reliability, and the criticality of the application. Depending on the particular software, these measures may be highly correlated, or quite unrelated.

Arisholm and Briand [2] studied a medium size Java system with about 110,000 LOC and found that LOC, number of functions, and control-flow complexity were closely correlated. Based on these observations, they made the simplifying assumption that test effort for this particular system is proportional to the number of lines of code being tested, and in addition assumed that faults are distributed uniformly across the code. They therefore concluded that accurate prediction of fault-prone files is cost-effective if the

total percentage of the lines of code included in the identified fault-prone files is smaller than the percentage of faults identified in those files. Their prediction model calculates a probability that each class contains a fault and uses a cutoff value of 0.5 to characterize a class as fault-prone. The model identifies 30% of the classes, containing 50% of the LOC, to be fault-prone. Those classes turned out to contain 70% of the actual discovered faults. Given the assumption that faults are uniformly distributed, a randomly selected set of files containing 50% of the LOC would be expected to contain 50% of the faults. They therefore conclude that in their case study, fault prediction is cost-effective, with a potential savings of approximately (70-50)/70 = 29% of effort.

For the maintenance support system, Rows C and D of Table 5 show the average percentage of LOC in the top 20% of files for each of the four models we studied. For Models 2-4, the average percent of identified faults exceeds the average percent of lines of code in the files. In particular, there is potentially a 12% saving for Model 2, a 27% saving for Model 3, and a 24% savings for Model 4 when averaged over all available releases. When cost effectiveness is assessed in this way, Model 3 provides the best results for this system. We note that these figures compare closely to the 29% reported by Arisholm and Briand [2].

However, measuring test effort in terms of the number of LOC is most appropriate for the unit testing phase, when white-box strategies based on code coverage are most likely to be used. For the integration and system testing phases, strategies are much more likely to be based on such factors as the number of interfaces and number of functions, and it seems reasonable that test effort will be more closely related to the number of files in the system. In that case, it seems more reasonable to assess cost effectiveness of the prediction models by comparing the percent of files predicted to be fault-prone to the percent of faults actually located in those files. This has been the basis for our assessments.

7. RELATED WORK BY OTHER GROUPS

A substantial body of research has attempted to identify software properties that are associated with fault-prone code. Papers describing this type of effort include [1, 3, 7, 8, 11, 12, 17, 18, 20, 23]. More recently, a number of research groups have developed models aimed at predicting the locations, the probability, or the quantity of faults that will occur in the future in a software system. The most closely-related work can be found in [2, 6, 10, 13, 14, 16, 19, 24]. To our knowledge, none of these research papers describe findings that have the same goal that we have, namely to identify files that will contain the largest numbers of faults in the next release of a production software system, or to identify those files that are likely to account for a prespecified percentage of the faults in the next release [21, 22, 4]. We outline below some of the most relevant research and describe the points of departure.

Perhaps the most closely related work is the study by Arisholm and Briand mentioned in the previous section. Their goal is also to build a prediction model to help focus attention during testing. Their model was explicitly designed for a single moderate-size Java system containing roughly 110 KLOC, with 17 releases during a 7 year period. They used stepwise logistic regression and data from three releases to make a prediction for one future release. In contrast, for three of the four systems we studied, we made predictions for each of a substantial number of successive releases and observed how the accuracy of the predictions evolved with the system. Arisholm and Briand assessed their results by using probability predictions to identify which classes will and will not be faulty, rather than identifying the most faulty files. They then checked for false positives and false negatives, and found that with a balanced probability value for classifications, they had approximately 20% false positives and negatives.

In a recent paper, Succi et al. [24] use software size plus metrics for object-oriented systems [5] to identify fault-prone classes. They consider Poisson regression, negative binomial regression, and zero inflated models to make predictions. The subjects of their study were two small projects written in C++ (23 and 25 KLOCS). For the first project, they report that 80% of the total faults were contained in less than 30% of the classes, while for the second project 80% of the faults were concentrated in just 2% of the classes. For each of the models they considered, predicted percentages of classes containing 80% of the faults varied between 43% and 48%. Although the goal of this paper was very similar to ours, there were significant differences. First, they looked at very small systems which were less than 10% the size of the systems we studied. In addition, they did not seem to use their predictions on successive releases, but rather looked at a single interval for each of the systems. Finally, their predictions appeared to be far less accurate than ours have been.

Mockus and Weiss [16] used logistic regression to construct a model based on the change history of roughly 15,000 maintenance requests for a large telephone switching system, over a ten year period. Rather than predicting which parts of the code are most likely to contain faults as ours do, their model's purpose is to predict the probability that the software changes made in response to a maintenance request will cause a failure in the modified system. The model's inputs include the size of each change; the number of distinct files, modules, and subsystems that were touched by the change; the time duration needed to complete the change; the number of developers involved in making the change; and the experience with the system of the programmers who made the change. They implemented their model as a web-based tool to be used by project managers to guide the scheduling of the implementation of maintenance requests. Although the model has apparently been used in practice, the paper provides no information on the success rate of its predictions for the system analyzed in their study.

Graves et al. [10] carried out a study similar to ours on a large telecommunications system over a two year period. Their first aim was to determine module characteristics that are associated with faults, a similar goal to our early work described in [20]. They then constructed and evaluated several models for predicting the number of faults likely to appear in the next version of the modules. They relied on the fault history of the system that contained approximately 1.5 million LOC, divided into 80 modules, containing approximately 2500 files. Rather than considering a series of releases as we did, the authors used the prediction models to make fault predictions for a single two year time interval. All predictions were based on the system's behavior during the preceding two years. They also applied their prediction models to a one year period that occurred within the two

year interval they studied and found that certain parameter values were significantly different between the entire two year period and the internal one year period. In particular, these values sometimes differed by as much as an order of magnitude.

There are several important differences between our work and theirs, both from the point of view of findings and process. One important difference is the level of granularity of the study. They worked at the module level, while we worked at the much smaller file level. In addition, we followed systems for a minimum of 9 releases, and up to 35 releases for the maintenance support system. In general, these releases occurred at roughly quarterly intervals. In contrast, they looked at a single two year interval. Finally we evaluated the success of our predictions, and consider that a major contribution of our research, while Graves et al. did not provide information on assessment of the effectiveness of their predictions.

Denaro and Pezze [6] constructed software fault prediction models using logistic regression, based on sets of static software metrics. They constructed a very large number of potential models (reportedly over 500,000) by using sets of up to five metrics selected from among a list of 38. They then selected the best models based on how close each model came to correctly determining all of the system's fault-prone and non-fault-prone modules.

Their definition of a *fault-prone module* states that it belongs to the smallest set of modules that are collectively responsible for 80% of faults identified in the system. The authors used the code and fault databases for the publicly available Apache web server as the basis for their experiment. They constructed models based on data from Apache version 1.3 and evaluated their predictive ability on Apache version 2.0. Ordering the modules of Apache 2.0 based on a decreasing fault count, they found that approximately 36% of the modules account for 80% of the observed faults. In their experiment, their best models required the first 50% of the modules predicted to be fault-prone in order to include 80% of the actual faults. In contrast, our model was able to identify 83% of the actual faults in only 20% of the files for both the inventory and provisioning systems as described in [22], as well as 81% of the faults when averaged over all 35 releases, and 84% of the faults when averaged over releases 6 through 35 for the maintenance support system described in this paper.

Khoshgoftaar et al. [14] described a method for constructing a binary decision tree that could be used to classify modules of a system as either fault-prone or not fault-prone, where fault-prone is defined as containing at least 5 faults. Decisions were made based on a set of 24 static software metrics plus four execution time metrics. They applied this method to four releases of a large legacy telecommunications system. They used training data from the first release to produce the tree and then evaluated it on the remaining three releases. Misclassification rates of fault-prone modules as non-fault-prone ones, and non-fault-prone modules as fault-prone modules were used as a measure of the method's success. Of course ideally, both of these rates should be very low. However, in practice there is an inverse relationship between the occurrence of the two misclassifications types. The user of the method can often study a parameter that trades off between the two events and then select the value that is closest to the project's needs. For example, if a project cannot tolerate run-time failures, then the number of fault-prone modules that are misclassified could be lowered by predicting more fault-free modules to be fault-prone. This would likely lead to an increase in the time needed for thorough testing. The authors selected a parameter value designed to keep the two misclassification rates roughly equal. This resulted in rates of 32.2% and 21.7% on the last of the four releases that they evaluated.

They argue that the technique should be able to more accurately predict fault-prone modules than an equal number of randomly selected modules, and use that as a measure of their technique's success. The authors claim that by applying their model to a hypothetical future release containing the same percentage of faulty modules as release 4, their technique should be able to select more than twice the number of faulty modules than a randomly selected set would have identified.

8. CONCLUSIONS

The results for the case study using the maintenance support system generally confirm those from the three systems studied previously. For the best models, 20 percent of files yield more than 80 percent of faults, averaged across releases. Also, the same predictor variables generally led the way.

We were able to avoid new model development (definition of predictors and variable selection) and still estimate effective prediction models. Indeed, Model 3 slightly outperformed Model 4, developed specifically for this case study. In other words, several years of data for other systems was as least as informative as 10 months of data for the current system. Furthermore, the even simpler pre-specified formula (Model 2) – which bypasses model estimation as well as model development – performed surprisingly well, losing only about six percentage points across releases (relative to Model 3).

In contrast, the LOC model loses quite a bit relative to both Model 2 (11 percentage points) and to Model 3 (17 percentage points), especially at later releases when the main discriminators are the status of files as either *new* or *changed*. Given the improvement possible with very little marginal effort, we recommend against the LOC-only model.

It is risky to generalize too much from a small number of systems. How well any particular model can predict faults may depend on many system attributes, including size, complexity, the mix of programming languages, the rate of additions and modifications, and testing style. Such factors may play an even larger role in whether models developed on one set of systems translate well to others.

Nonetheless, the strong consistency across the systems we have studied provides promising evidence that our methodology can be implemented in the real world without extensive statistical expertise or modeling effort. Results for Models 2 and 3 indicate that a prediction model can be both simple and highly accurate, and therefore easily incorporated into an automated tool.

The back-end portion of the tool that extracts data from the change management/version control database used by the four systems we have studied is complete and validated. We have also selected a public-domain statistical package to implement the negative binomial regression model predictions. We plan to begin integration and testing of these two portions shortly.

Acknowledgments

This research could not have been carried out without the help of Andrew Gauld, Gabe Gurman, Alfred Marzouk, Kathy Daley, and Vern Lazaro. We are very grateful for their support.

9. REFERENCES

[1] E.N. Adams. Optimizing Preventive Service of Software Products. *IBM J. Res. Develop.*, Vol 28, No 1, Jan 1984, pp. 2-14.

[2] E. Arisholm and L.C. Briand. Predicting Fault-prone Components in a Java Legacy System. *Proc. ACM/IEEE ISESE*, Rio de Janeiro, 2006.

[3] V.R. Basili and B.T. Perricone. Software Errors and Complexity: An Empirical Investigation. *Communications of the ACM*, Vol 27, No 1, Jan 1984, pp. 42-52.

[4] R.M. Bell, T.J. Ostrand, and E.J. Weyuker. Looking for Bugs in All the Right Places. *Proc. ACM/International Symposium on Software Testing and Analysis (ISSTA2006)*, Portland, Maine, July 2006, pp. 61-71.

[5] S. R. Chidamber and C. F. Kemerer. A Metrics Suite for Object Oriented Design. *IEEE Trans. on Software Engineering*, vol 20 no 6, June 1994, pp.476-493.

[6] G. Denaro and M. Pezze. An Empirical Evaluation of Fault-Proneness Models. *Proc. International Conf on Software Engineering (ICSE2002)*, Miami, USA, May 2002.

[7] S.G. Eick, T.L. Graves, A.F. Karr, J.S. Marron, A. Mockus. Does Code Decay? Assessing the Evidence from Change Management Data. *IEEE Trans. on Software Engineering*, Vol 27, No. 1, Jan 2001, pp. 1-12.

[8] N.E. Fenton and N. Ohlsson. Quantitative Analysis of Faults and Failures in a Complex Software System. *IEEE Trans. on Software Engineering*, Vol 26, No 8, Aug 2000, pp. 797-814.

[9] D.R. Graham. "The Cost of Testing". *Encyclopedia of Software Engineering*, J.J. Marciniak, ed., John Wiley, 1994, p. 1335.

[10] T.L. Graves, A.F. Karr, J.S. Marron, and H. Siy. Predicting Fault Incidence Using Software Change History. *IEEE Trans. on Software Engineering*, Vol 26, No. 7, July 2000, pp. 653-661.

[11] L. Guo, Y. Ma, B. Cukic, H. Singh. Robust Prediction of Fault-Proneness by Random Forests. *Proc. ISSRE 2004*, Saint-Malo, France, Nov. 2004.

[12] L. Hatton. Reexamining the Fault Density - Component Size Connection. *IEEE Software*, March/April 1997, pp. 89-97.

[13] T.M. Khoshgoftaar, E.B. Allen, K.S. Kalaichelvan, N. Goel. Early Quality Prediction: A Case Study in Telecommunications. *IEEE Software*, Jan 1996, pp. 65-71.

[14] T.M. Khoshgoftaar, E.B. Allen, J. Deng. Using Regression Trees to Classify Fault-Prone Software Modules. *IEEE Trans. on Reliability*, Vol 51, No. 4, Dec 2002, pp. 455-462.

[15] P. McCullagh and J.A. Nelder. *Generalized Linear Models*, Second Edition, Chapman and Hall, London, 1989.

[16] A. Mockus and D.M. Weiss. Predicting Risk of Software Changes. *Bell Labs Technical Journal*, April-June 2000, pp. 169-180.

[17] K-H. Moller and D.J. Paulish. An Empirical Investigation of Software Fault Distribution. *Proc. IEEE First International Software Metrics Symposium*, Baltimore, Md., May 21-22, 1993, pp. 82-90.

[18] J.C. Munson and T.M. Khoshgoftaar. The Detection of Fault-Prone Programs. *IEEE Trans. on Software Engineering*, Vol 18, No 5, May 1992, pp. 423-433.

[19] N. Ohlsson and H. Alberg. Predicting Fault-Prone Software Modules in Telephone Switches. *IEEE Trans. on Software Engineering*, Vol 22, No 12, December 1996, pp. 886-894.

[20] T. Ostrand and E.J. Weyuker. The Distribution of Faults in a Large Industrial Software System. *Proc. ACM/International Symposium on Software Testing and Analysis (ISSTA2002)*, Rome, Italy, July 2002, pp. 55-64.

[21] T.J. Ostrand, E.J. Weyuker, and R.M. Bell. Where the Bugs Are. *Proc. ACM/International Symposium on Software Testing and Analysis (ISSTA2004)*, Boston, MA, July 2004.

[22] T.J. Ostrand, E.J. Weyuker, and R.M. Bell. Predicting the Location and Number of Faults in Large Software Systems. *IEEE Trans. on Software Engineering*, Vol 31, No 4, April 2005.

[23] M. Pighin and A. Marzona. An Empirical Analysis of Fault Persistence Through Software Releases. *Proc. IEEE/ACM ISESE 2003*, pp. 206-212.

[24] G. Succi, W. Pedrycz, M. Stefanovic, and J. Miller. Practical Assessment of the Models for Identification of Defect-prone Classes in Object-oriented Commercial Systems Using Design Metrics. *Journal of Systems and Software*, Vol 65, No 1, Jan 2003, pp. 1 - 12.

[25] SAS Institute Inc. *SAS/STAT 9.1 User's Guide*, SAS Institute, Cary, NC, 2004.