Where the Bugs Are

Thomas J. Ostrand
AT&T Labs - Research
180 Park Avenue
Florham Park, NJ 07932
ostrand@research.att.com

Elaine J. Weyuker
AT&T Labs - Research
180 Park Avenue
Florham Park, NJ 07932
weyuker@research.att.com

Robert M. Bell
AT&T Labs - Research
180 Park Avenue
Florham Park, NJ 07932
rbell@research.att.com

Abstract

The ability to predict which files in a large software system are most likely to contain the largest numbers of faults in the next release can be a very valuable asset. To accomplish this, a negative binomial regression model using information from previous releases has been developed and used to predict the numbers of faults for a large industrial inventory system. The files of each release were sorted in descending order based on the predicted number of faults and then the first 20% of the files were selected. This was done for each of fifteen consecutive releases, representing more than four years of field usage. The predictions were extremely accurate, correctly selecting files that contained between 71% and 92% of the faults, with the overall average being 83%. In addition, the same model was used on data for the same system's releases, but with all fault data prior to integration testing removed. The prediction was again very accurate, ranging from 71% to 93%, with the average being 84%. Predictions were made for a second system, and again the first 20% of files accounted for 83% of the identified faults. Finally, a highly simplified predictor was considered which correctly predicted 73% and 74% of the faults for the two systems.

Keywords: Software Faults, Fault-prone, Prediction, Regression Model, Empirical Study, Software Testing.

1. Introduction and Previous Work

Software testing activities play a critical role in the production of dependable systems, and account for a significant amount of resources including time, money, and personnel. If testing effort can be more precisely focused on the places in a software system where faults are likely to be, then available resources will be used more effectively and efficiently, resulting in more reliable systems produced at decreased cost.

Testing effort can be focused by evaluating files for fault-proneness prior to testing them. Our goal is to provide testers with a practical, reasonably accurate measure of which files are most likely to contain the largest numbers of faults, so they can adjust their testing efforts to target these files. The model described in this paper assigns a predicted fault count to each file of a software release, based on the structure of the file and its history in previous releases. Unlike most software testing research which is designed to tell people *how* to test their software or *how* to select test cases, the goal of this research is to tell testers *where* in their software to test.

We have been studying large industrial software systems to determine how faults are distributed over the systems' files, and to identify the characteristics of those source files that have the highest fault densities. Summaries of our preliminary findings can be found in [11], which based the identification of characteristics on twelve releases of an industrial inventory system that had been in the field in continuous use for about three years.

Several other research groups have looked at similar characteristics of software. Results along these lines can be found in [1, 2, 3, 4, 5, 9, 10, 13], among others. The types of characteristics investigated by our group and many of the others include such things as the file size, the file's age, whether or not the file is new to the current release, and if it is not new, whether it was changed during the prior release. Other characteristics considered were the number and magnitude of changes made to a file, the number of detected faults during early releases and the number of faults detected during early development stages.

One of the things that differentiates our research from most of the other work is its magnitude and duration. While most of the published studies have considered only a small number of releases, we have now studied seventeen successive releases of one large industrial software system representing four years of large-scale field usage. We have also collected data from nine releases of a second software system corresponding to two years in the field. For the first

system, our data include faults identified at all stages of development starting from the requirements phase and continuing through field release, while for the second system faults were included beginning with integration testing. By examining such a large number of releases, we have been able to assess whether or not relevant characteristics change as the system matures and stablizes, with new files comprising an increasingly smaller percentage of the system.

Having accumulated information in our earlier studies about which software characteristics are most likely to have a significant impact on the number of faults in a file, the next natural step is to use that information to *predict* which particular files in a software system are likely to contain the highest numbers of faults in the next release. This is another point of departure from much of the earlier research, whose goal was simply to identify the characteristics that made files particularly prone to having faults rather than identifying particular files that should be especially subject to scrutiny. The two papers most closely related to this research are References [4] and [6]. In Section 4.1 we present our results on prediction and in Section 4.2 contrast them with the work done by these other research groups.

The ability to predict where the largest concentrations of faults are likely to reside represents a very valuable tool for organizations that develop software systems, provided it is feasible to make such a prediction with reasonably high accuracy. This information will allow testers to target their testing efforts to the files most likely to contain the highest concentrations of faults. This should enable testers to identify faults more quickly and therefore have additional time to test the remainder of the system. The net result should be systems that are of higher quality, containing fewer faults, and projects that stay more closely on schedule than would otherwise be possible.

To do this sort of prediction, we developed a negative binomial regression model to predict an expected number of faults for each file of a release. The predictions rely on the factors determined to be most relevant, including the file size in terms of the number of lines of code, whether the file was new to the release, or changed or unchanged from the previous release, the age of the file in terms of the number of previous releases it was in, the number of faults in the previous release, and the programming language used.

We first applied the model prospectively to predict the number of faults associated with each file in each of Releases 3 through 12 of the inventory system, covering a period of roughly three years of field release. For each release, we based the predictions on a model fit to data from prior releases only. When the files of each release were ranked in descending order of the predicted number of faults, the first 20% of the files contained from 71% to 85% of the release's faults.

Of course there is nothing magical about the 20% figure. When we looked at the graphs corresponding to many of the releases, we saw that the "knee" of the curve, i.e. the place where the curve flattened out, often occurred around the 20% mark. Also, we are *not* suggesting that a project should *only* test the files that are highly ranked by the prediction. What we are suggesting is that we are likely to get the greatest payoff by testing these files, and therefore a good strategy is to test these files first and with greatest emphasis.

This paper makes four main contributions. The first is the application of our prediction model to five additional releases of the inventory system representing more than a year of additional field experience. This lets us assess the prediction's accuracy as the system matures and stabilizes. The second contribution evaluates the prediction model when data are only available from testing stages later than unit testing. In many production environments, data collection for a system does not begin until after it leaves development, and moves on to integration or system testing. Since the model was developed using data from a system that included faults detected at all stages of development, it was important to see whether the prediction based on post-unit testing data would be comparably accurate. The third contribution provides additional evidence of the prediction's effectiveness by applying the model to a different system, written by different personnel, using different primary programming languages, and performing different functions. The fourth contribution proposes a highly simplified predictive model, and assesses its application to the two systems to which we applied the full model. We discuss the tradeoffs observed in terms of the decreased effort to apply the simplified model versus the decreased accuracy of that model.

The remainder of the paper is organized as follows: Section 2 briefly describes the inventory system that is the primary subject of the case study. In Section 3 we describe the negative binomial regression model and discuss the variables selected to do the prediction. Section 4 presents the results of applying the predictor to all seventeen releases of the system, and compares its success at predicting which files have the highest numbers of faults for early releases with the results on the later, more mature and stable releases. In Section 5, we compare the effectiveness of the predictor when applied to the fault data collected from all stages of development, including unit testing, to its effectiveness when restricted to faults detected only from integration testing and beyond for all seventeen releases. In Section 6 we examine the model's applicability to a second system, to determine whether the key file characteristics for the inventory system are also significant for code written by a different group, for an entirely different purpose. Section 7 discusses a simplification of the predictor and provides information about how much predictive power is lost

Rel	Number of Files	Lines of Code	Mean LOC	Faults Detected	Fault Density	Files Containing Any Faults	Pct Containing Any Faults
1	584	145,967	250	990	6.78	233	39.9
2	567	154,381	272	201	1.30	88	15.5
3	706	190,596	270	487	2.56	140	19.8
4	743	203,233	274	328	1.61	114	15.3
5	804	231,968	289	340	1.47	131	16.3
6	867	253,870	293	339	1.34	115	13.3
7	993	291,719	294	207	0.71	106	10.7
8	1197	338,774	283	490	1.45	148	12.4
9	1321	377,198	286	436	1.16	151	11.4
10	1372	396,209	289	246	0.62	112	8.2
11	1607	426,878	266	281	0.66	114	7.1
12	1740	476,215	274	273	0.57	120	6.9
13	1772	460,437	260	127	0.28	71	4.0
14	1877	482,435	257	235	0.49	95	5.1
15	1728	479,818	278	305	0.64	120	6.9
16	1847	510,561	276	274	0.54	116	6.3
17	1950	538,487	276	253	0.47	122	6.3

Table 1. Inventory System Information

by this simplification. Conclusions and plans for extending this work are discussed in Section 8.

2. The System Under Study

For this study, we use the same inventory system that was analyzed during the earlier studies [11, 12] mentioned in Section 1. While the earlier studies used the first twelve releases of the system, there are now a total of seventeen successive releases, representing more than four years of continuous field use. We used these seventeen releases for the current study.

Many AT&T software projects use a combined version control/change tracking system throughout their life cycles. The basic entity of this system is a *modification request* (MR) that is entered by a developer or tester when a change is deemed necessary. An MR contains a written description of the reason for the proposed change and a severity rating of 1 through 4 that characterizes the importance of the proposed change. If the request results in an actual change, the MR records the file(s) that are changed or added to the system, and the specific lines of code that are added, deleted, or modified. It also includes such information as the date of the change and the development stage at which it was made.

We had initially hoped to use the severity rating as a variable for the fault location predictor, however, we learned that these ratings were highly subjective and also sometimes inaccurate because of political considerations not related to the importance of the change to be made. We also learned that they could be inaccurate in inconsistent ways. We were told that a change could be rated as a Severity 2 modification, for example, when in fact the person writing the modification request rated it as either a Severity 1 or Severity 3 fault. It might be downplayed so that friends or colleagues in the development organization "looked better", provided they agreed to fix it with the speed and effort normally reserved for Severity 1 faults, or it might be "upgraded" so that developers who actually make the change focus on it quickly. For these reasons, we did not include fault severity in our predictor or propose that it be used in any way to influence or weight test case selection.

The inventory system used in the earlier case studies contained changes including faults found at each of nine distinct stages of development: requirements, design, development, unit test, integration test, system test, beta release, controlled release, and general release. Faults were reported for each of these stages, and, surprisingly, almost three quarters of the faults (3407 of 4618 for the first twelve releases), were reported during unit testing. When other pre-unit testing phases were added, roughly 80% of the faults reported in the database occurred during these very early stages.

Table 1 contains information about the system. Not surprisingly, there is a general increase in the size of the system as new functionality is added at most releases, particularly the earlier ones. The average number of lines of code per file remained fairly constant for all releases. With a few exceptions, the fault density, computed as the number of faults observed in a file, divided by the file's size in terms of the number of thousands of commented lines of code (KLOCs),

tended to decrease as the system matured.

The next to last column in the table shows the number of files which contained at least one fault that was detected at any stage of development, while the final column shows the percentage of files that were found to contain at least one fault in the release. At each release after the first, faults occurred in 20 percent or fewer of the files, with those files typically averaging two to three faults apiece. This concentration of faults suggested that testing effort could be reduced greatly if most of the faulty files could be identified prospectively. Of course it is possible, and even likely, that other files in the system contain faults that have gone undetected; however, we can only discuss faults that have been observed.

Notice that from Release 1 to Release 12, the last release used in the earlier studies, the system roughly tripled in size from 584 executable files containing almost 146,000 lines of code to 1,740 executable files, containing a total of more than 476,000 lines of code. The system is primarily written in java, (approximately 70% of the files) but there are files in a number of other languages including shell scripts, makefiles, xml, html, perl, c, sql, awk, and other specialized languages. In each of our studies, non-executable files were excluded. Although the system has continued to grow steadily over Releases 13 to 17, the rate of growth has definitely begun to slow. This is not surprising as new files frequently represent new features, while modified files often represent fault repairs, and as the system matures one would expect fewer new features to be added.

One of the problems we had to address at the outset of our studies was how to identify faults and how to count them. For both systems, there was generally no identification in the database of whether a change was initiated because of a fault, an enhancement, or some other reason such as a change in the specifications. The inventory system recorded more than 5,800 faults over the seventeen releases plus many times that number of other changes, far too many to read all of the reports and make a determination of exactly which were faults and which were not. A rule of thumb suggested by the test team of the project was that if only one or two files were changed by the modification request, then it was likely a fault, while if more than two files were affected, it was likely not a fault. The rationale was that if many files were touched by the change, it was generally a change in the interface caused by a change in the specifications. For the inventory system we used that rule of thumb to identify fault-caused changes.

In Section 6 we consider applying our prediction to a second system. In this case, substantially fewer changes were included in the database, since data was not entered until system test began. We were able to read every modification request, and determine whether or not the change was due to a fault. For most entries it was apparent from the write-up whether or not the change was a fault. For a few entries, the description was unclear, and we asked the test team to make the determination.

In all of the work described in this paper, we use the same fault counting convention as was used in [9] and [3], and our earlier studies [11, 12]. If a problem causes *n* files to be modified, then this is counted as *n* distinct faults. Therefore each fault is associated with exactly one file.

3. Multivariate Analysis of the Number of Faults

We now describe the negative binomial regression models we designed to predict the number of faults in a file during a release. These models are a function of various file characteristics identified in our earlier studies. Modeling serves three primary purposes.

- It provides information regarding the association between the number of faults and individual file characteristics while holding other file characteristics constant.
- Parameter estimation accounts for high variation in fault counts due to fault concentration after file characteristics have been accounted for.
- The model makes predictions of which files will contain the largest numbers of faults in a release, allowing testing resources to be targeted more effectively.

The third of these objectives is the primary goal of the research described in this paper. We briefly outline the model in Section 3.1, while in Section 3.2, we provide more detail, including the specific variables selected, and mention others that we found did not enhance the predictive abilities of the model.

3.1 The Negative Binomial Regression Model

Negative binomial regression is an extension of linear regression which is designed to handle outcomes like the number of faults [8]. It explicitly models outcomes that are nonnegative integers. It is assumed that the expected number of faults varies as a function of file characteristics in a multiplicative rather than in an additive relationship. Unlike the related Poisson regression model, the negative binomial model allows for the type of concentration of faults we observed in this system by adjusting inference for the additional uncertainty in the estimated regression coefficients caused by overdispersion.

Let y_i equal the number of faults observed in file i and x_i be a vector of characteristics for that file. The negative binomial regression model specifies that y_i, given x_i, has a

Poisson distribution with mean λ_i. This conditional mean is given by $\lambda_i = \gamma_i e^{\beta' x_i}$, where γ_i is itself a random variable drawn from a gamma distribution with mean 1 and unknown variance $\sigma^2 \geq 0$. The variance σ^2 is known as the *dispersion parameter*, and it allows for the type of concentration observed for faults. The larger the dispersion parameter, the greater the unexplained concentration of faults. However, to the extent that this concentration is explained by file characteristics x_i that are included in the model, the dispersion parameter will decline.

3.2 Model Specifics

In [12], we presented the results of a negative binomial regression model fit to Releases 1 to 12. The unit of analysis was a file-release combination, leading to a total of 12,501 observations, with the dependent variable being the number of faults associated with the file at the given release. The procedure Genmod in SAS/STAT Release 8.01 [14] was used to fit all models.

Predictor variables for the model included: the logarithm of the number of lines of code; whether the file was new, changed or unchanged, which we refer to as the file's *change status*; the file's age as measured by the number of previous releases in which it appeared; the square root of the number of faults identified during the previous release; the programming language used; and the release number. Each factor included in the model was easily statistically significant at the 0.001 level.

The log of the number of lines of code, file age, and the square root of the number of prior faults were treated as continuous variables. Change status, programming language type, and release number were treated as categorical variables, each fit by a series of dummy (0-1) variables, with one omitted category that served as the reference. For change status, the reference category was unchanged files. In this way, the new and changed coefficients represented contrasts with existing, unchanged files.

We found that the number of lines of code and the file's change status were the strongest individual predictors in the model. The estimated coefficient for the logarithm of lines of code was 1.05. Because the 95 percent confidence interval included 1.00, this result is consistent with a finding that after controlling for all other factors, the number of faults is proportional to the number of lines of code–in other words, that fault density does not change with LOC. The estimated coefficient for the new file variable was 1.86. This implies that new files had about $e^{1.86}$ = 6.4 times as many faults as existing, unchanged files that were similar in all other respects. The estimated coefficient for the changed file variable implies 2.9 times as many faults as an existing unchanged file. The full model, which combined these and the other factors listed above, produced more accurate predictions than models based on any of the individual factors (see [12] and Section 7 of this paper for additional details).

Other possible predictor variables were tried, but not included in the model, as they did little to improve the predictive power. Among the variables that we decided not to include were the number of changes made to a file for those files that were changed since the previous release, whether a file had been changed prior to the previous release, and the logarithm of the cyclomatic number for java files. Since the cyclomatic number [7] has been found to be very highly correlated with the number of lines of code, it is not surprising that this did not appreciably enhance the predictive power of the model.

4. Predictions for Releases 3 Through 17

4.1 Our Prediction Results

The negative binomial regression model assigns a predicted fault count to each file of a release. The higher the predicted fault count, the more important it is to test the file early and carefully. In [12] we computed these predictions for the files of Releases 3-12, in each case using data from Releases 1 through (n-1) to make the predictions for Release n. Sorting the files in descending order of their predicted fault counts creates an ordered list of the files from most to least likely to be problematic. Although the individual fault counts predicted for each file do not generally match their actual fault counts, the great majority of the actual faults in the system occur in the set of files at the top of the listing. Tables 2 and 3 show the percentage of actual faults that occurred in the first 20% of files, as predicted by the model. The results in Table 2, for Releases 3 through 12, were described in [12]. For these releases, the model identified files that contained between 71% and 85% of the faults identified in the system, with an average of 80% over all releases through Release 12.

The accuracy of these predictions was very encouraging, but raised the question of whether the accuracy would diminish for later releases, as the system stabilized and matured. This is particularly important since, as can be seen in Table 1, by Release 10, all of the identified faults are concentrated in less than 10% of the files, making the potential payoff of accurate prediction extremely valuable.

Table 3 shows the prediction results for Releases 13 through 17. The accuracy of the prediction actually improved as the system matured. The average percentage of faults contained in the 20% of the files selected by the model was 89% for the five most recent releases, bringing the overall average for all of the releases through Release 17 to almost 83%.

These five additional releases represent more than a year of additional system field usage, for a total of more than

Release	3	4	5	6	7	8	9	10	11	12	Avg 3-12
% Faults Identified	77	74	71	85	77	81	85	78	84	84	80

Table 2. Percentage of Faults Included in the 20% of the Files Selected by the Model - Releases 3-12

Release	13	14	15	16	17	Avg 13-17
% Faults Identified	82	91	92	92	88	89

Table 3. Percentage of Faults Included in the 20% of the Files Selected by the Model - Releases 13-17

four consecutive years. The results for these additional releases provide a completely independent validation of the model's predictive accuracy for this system. The form of the model and the variables used were selected before looking at the data for Releases 13-17. In addition, the estimated coefficients used to order files in Releases 13 through 17 were those resulting from fitting the model to data from Releases 1 through 12. Consequently, all aspects of the model were independent of the five additional releases.

The accuracy of the overall fault predictions can be evaluated by comparing how close the predicted ordering comes to the ordering according to the actual number of faults discovered. The graphs of Figure 1 show this comparison, both for the full prediction model based on all the significant variables, and for a simplified model that is based only on the lines of code in each file. This simplified model and our assessment of its predictive ability will be discussed in Section 7 for the two systems.

The curves plot the cumulative percent of faults (on the vertical axis) found in a given percentage of the files (on the horizontal axis) in a release. On the Actual Faults curve represented by the heavy solid line, the files are sorted in decreasing order of the number of actual faults found in each file. This is the optimal ordering given the goal of finding all the faults in as few files as possible; it represents perfect prediction. On the Full Model curve represented by the short dashed line segments, the files are sorted according to the full model's prediction of the number of faults contained in each file. The LOC curve, represented by the long dashed line segments, sorts the files solely in decreasing order of their size.

The "goodness" of each of the model predictions can be measured in terms of how close the prediction curve comes to the Actual Faults curve. While the LOC-based predictions for some of the early releases are quite close to the Full Model, by Release 14 the full model predictions have become substantially better, capturing well over 85% of the faults with 20% of the files. The LOC-based predictions will be further discussed in Section 7 .

4.2 Comparison with Other Prediction Approaches

The research most closely related to ours is described in [6] and [4]. Both groups attempt to make predictions relating faults to specific parts of the software system.

Khoshgoftaar et al. [6] developed two classification models based on discriminant analysis that predict whether or not a module will be fault-prone, rather than attempting to predict the actual number of faults that will occur in the module as we do. In their study, they define a module to be a collection of files, and say that a module is fault-prone if it contains 5 or more faults. The models use data from a single release of a large telecommunications system, containing about 1.3 million LOC, with approximately 2000 modules and 25,000 files. For the entire sample set of the study, approximately 12% of the modules were by their definition, fault-prone.

Their predictions are based on software design metrics and reuse information. The design metrics include call graph metrics such as the number of calls from one module to other modules, and the number of distinct modules that are referenced, as well as control-flow graph metrics such as the number of loops, cyclomatic complexity, and nesting level. Reuse information classifies whether a module was new in the current release, and if new, whether it was changed from the previous release.

Model 1 was based solely on the software design metrics, while Model 2 used both the design metrics and reuse information. The discriminant analysis was performed on a randomly chosen set of two thirds of the system's modules. The two models were evaluated according to their ability to correctly predict the fault-proneness of the system modules.

The authors point out that from a testing point of view, it is most important to avoid predicting that a module that is actually fault-prone is not fault-prone, a so-called Type 2 fault. To evaluate the efficacy of the models, they were applied to the remaining one third of the system's modules. Model 1 predicted 21.3% of the actual fault-prone modules to be non-fault-prone, while Model 2 predicted 13.8% of the fault-prone modules to be non-fault-prone. The obvious conclusion is that reuse information contributes significant

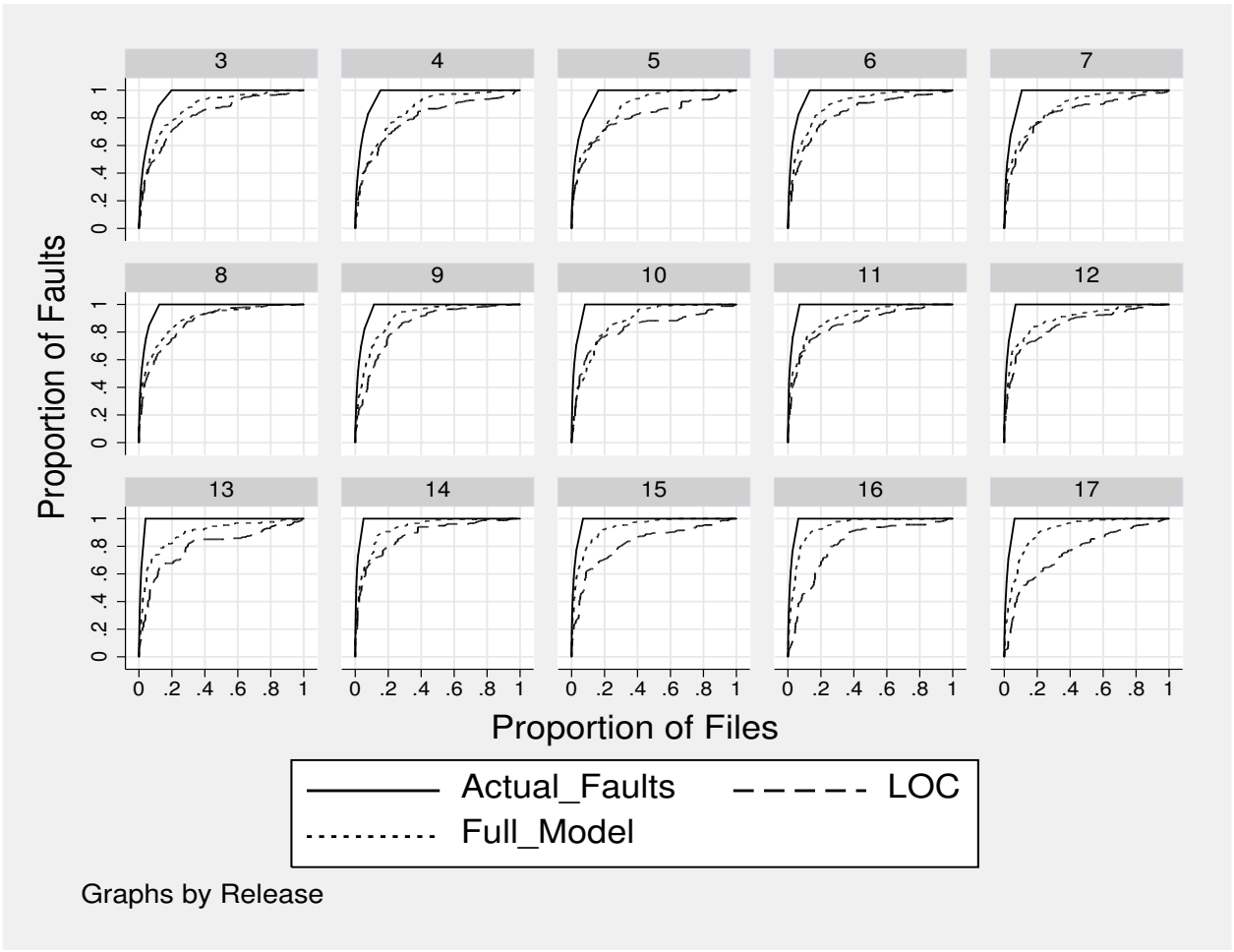

Figure 1. Lorenz Curves for Prospective Predictions at Releases 3-17

predictive power. The study did not evaluate the effectiveness of a model based solely on the reuse information.

There are several significant differences between the research described in [6] and ours. The most important one is the primary goals. The predictions in [6] classify all modules into two categories: fault-prone or not fault-prone, where fault-proneness is defined based on an arbitrary and fixed number of faults in a module. In contrast, our model ranks the files from most to least fault-prone, without choosing an arbitrary cutoff point for fault-proneness. Another important difference is the level of granularity at which the predictions are being done. Khoshgoftaar et al. are working at the module level which is much coarser than the file level at which we are working. Using this coarser-grained prediction, will increase the difficulty of pinpointing where faults are likely to be.

Another difference is the extent of the studies. While Khoshgoftaar et al. applied their models to a portion of a single release of a system, we have applied ours to seventeen successive releases. In addition, in the next section we will describe the application of our model to systems that begin recording fault information only after unit testing, and therefore have significantly less data. Under these circumstances, we've found that our model's predictive accuracy remains comparable to the results using the full dataset containing faults detected at all development stages.

In [4], Graves et al. report a study using the fault history for the modules of a large telecommunications system. Their subject system contained roughly 1.5 million lines of code, divided into 80 modules, each a collection of files. They first considered different file characteristics, and found that module size and other standard software complexity metrics were generally poor predictors of fault likelihood. Their best predictors were based on combinations of a module's age, the changes made to the module, and the ages of the changes. Although they did use the models to predict faults in files, the primary focus of their work was the identification of the most relevant module charac-

Release	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	Avg 3-17
% Faults Identified - ALL	77	74	71	85	77	81	85	78	84	84	82	91	92	92	88	83
% Faults Identified - PUT	80	71	73	90	81	83	81	81	83	90	90	93	85	88	86	84

Table 4. Percentage of Post-Unit Test Faults Included in the 20% of the Files Selected by the Model

teristics and the comparison of models. They applied their model to make predictions using a single two year time interval, but did not discuss the extent to which the predictions were effective. At the conclusion of their study they described applying their model to a one year interval in the middle of the original two year interval and were troubled to discover that certain parameters values differed by an order of magnitude during the two time periods.

There are also significant differences here between their work and ours. They focused on identifying the most relevant module characteristics or groups of characteristics and comparing the apparent effectiveness of the proposed models, rather than predicting faults and using them to guide testing. As with the work described in [6], they worked at the module rather than the file level, and only looked at a single release, albeit one of long duration.

5. Prediction for Releases Based on Integration Testing and Beyond

Having seen the success of our model in making accurate predictions for the first seventeen releases of the inventory system, we are very encouraged because we believe we have identified an approach that is potentially applicable to many production software systems. However, since recording early faults found prior to integration or system testing is not the norm for many software systems we have encountered, we wondered whether the same model could be used for systems for which the database did not include faults discovered during unit testing or earlier. Not only does the change management database for such a system have a different population of faults, but it will also have significantly less data. In this section we discuss our findings using the negative binomial regression model restricted to faults discovered *after* unit testing.

For the inventory system, the change management/version control system contained data for any modifications made for any reason during any phase of development beginning at the requirements stage. Almost 80% of the faults in Releases 1 through 12 were detected prior to integration testing, primarily during unit testing.

In practice, many projects do not actually begin to use the change tracking portion of the system until the project enters integration testing or the system test phase. In particular, it is unusual to find unit test changes of any sort (including faults) recorded for many software systems. The rationale might be that just as an author recognizes that a first draft of an article might well contain unpolished wordings, or even typographical or grammatical errors, the programmer recognizes that while he or she is developing the code there may be missing parts or faults that they will address before releasing the file to others to test. As such, the code is a work in progress, and until it is sent for integration or system testing, any changes made are not really faults, and hence not included in the fault database.

Therefore, we repeated our empirical study predicting which files would contain the highest numbers of faults using the same negative binomial regression model with the same variables. In this case we used only faults detected during the integration testing phase or later, both as the dependent variable and as the measure of faults in the prior release. The findings, are shown in the bottom row of Table 4. To facilitate comparison, we repeat the data for the results using the full fault dataset in the row above.

While the average percentage of faults included in the files identified by the model when all faults are considered is 83%, the average percentage of values for post-unit test faults in Table 4 is 84%. For the full fault dataset the range was from 71% to 92%, while for the post-unit test dataset the range was from 71% to 93%. The fact that our model appears to be applicable to a system for which significantly less data is available, and which does not include faults identified during unit testing or earlier is extremely promising as many systems that we encounter begin data collection (or retention) at this later stage. The prediction approach should therefore be applicable to a far wider field of projects than if fault data from all development phases were required.

6. Applicability of the Model to Other Systems

The next issue we considered was whether or not the model we developed and the particular variables used would be applicable for other software systems. Our goal is to develop a method that can be used to help any software project order their testing efforts. We therefore needed to identify a new software system, with a substantial number of releases and years of field usage, preferably one that was developed by a different organization and hence different personnel. Another requirement was that the system be a different type of system from the one used in the earlier studies. We did identify such a potential study subject. This system is used

Aggregated Rel Name	Rel No.	Number of Files	Lines of Code	Mean LOC	Faults Detected	Fault Density	Files Containing Any Faults	Pct Containing Any Faults
A	1	2008	381,973	190	24	0.06	19	0.9
B	2	2085	397,683	191	85	0.21	63	3.0
	3	2137	412,621	193	52	0.13	41	1.9
	4	2104	406,674	193	10	0.02	9	0.4
	5	2119	407,724	192	6	0.01	6	0.3
C	6	2213	423,895	192	15	0.04	14	0.6
	7	2250	434,772	193	74	0.17	64	2.8
	8	2230	434,781	195	34	0.08	30	1.3
	9	2241	437,578	195	7	0.02	6	0.3

Table 5. Provisioning System Information

to establish or activate a service for a customer by configuring hardware and software. Such a system is often known as a *provisioning system* in the telecommunications industry.

We collected data for nine releases of this system, providing two years of field experience. While the inventory system is primarily written in java, the programming language with the largest number of files in this system is sql, accounting for roughly one quarter of the files. The other two major programming languages used are java, and shell scripts, which together account for another quarter of the system's files.

The two systems are similar in size. Release 9 of the provisioning system has 2,271 files containing a total of 439 KLOCs. Table 5 provides more details about the system. However, while the inventory system grew substantially over its lifetime, with many new files and features being added at each release, the provisioning system was already fairly mature at the point when our data collection began. Consequently, it remained much more stable in size with roughly 250 new files being added over the entire course of the two year period that was studied. Another difference between the systems is that systematic fault reporting for the provisioning system did not begin until unit testing was complete and integration testing began.

For this system, substantially fewer faults were identified than were observed in the inventory system, even when only post-unit testing faults were considered as in Section 5. For the nine releases for which we had data, there was only a total of 307 faults. Upon reflection we decided that this was not surprising since the system did not grow substantially over time and unit testing faults were not included. One of our observations in [11] was that new files generally had substantially higher numbers of faults and fault densities than files that had existed in earlier releases, and the provisioning system generally had relatively few new files at each release. Additionally, the system's releases were often more frequent than once a quarter.

Because there were so few faults associated with this system, with four of the releases having fifteen or fewer faults identified, we decided that there was insufficient data to apply the negative binomial regression model separately for the nine releases.

In order to see whether we could actually use the model to make predictions for this system in spite of the paucity of data, we decided to treat Releases 2 through 5 as a single release, and similarly treat Releases 6 through 9 as a single release. This gave us three releases with the first release (Release A) having 24 faults, the second release (Release B) having a total of 153 faults, and the third release (Release C) having a total of 130 faults.

We used data for Release B to develop predictions of the number of faults and evaluated the results on Release C. No attempt was made to model faults for Release A because we could not determine the change status or fault history for those files. Data from Release A were used to determine values of those factors for Release B. The model was a slight revision of the one developed for the inventory system presented in Section 4.1. Because data for only Release B was used, the model excluded release number and age of the files, which were not known.

As with the inventory system, lines of code and whether a file was new, changed, or unchanged were strong, statistically significant factors for predicting faults in the provisioning system. The estimated coefficient for the logarithm of LOC was 0.73, significantly lower than 1.00, suggesting that in this system, fault density may decrease for longer files, holding all else equal. The estimated coefficient for new files was 1.92, very similar to the value for the other system (1.86). The estimated coefficient for changed files was 1.77, compared with 1.06. Programming language was statistically significant overall, but not strikingly so. That may simply be due to the large drop in the amount of information (few faults) in the new data. The number of prior faults was not significant, perhaps due to very few faults in Release A, but was retained for purposes of making predictions at Release C.

Release	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	Avg 3-17
% Faults Identified	71	68	71	75	76	76	77	77	80	75	68	80	71	67	58	73

Table 6. Percentage of Faults Included in the 20% of the Files Selected by LOC

Using the model estimated with data from Release B to predict which files would likely contain the most faults in Release C, the percentage of faults contained in the 20% of the files selected by the model was 83%, very close to the overall predictive accuracy for the seventeen releases of the inventory system.

7. Simplifying the Model

We have now provided evidence that our proposed model is useful for predicting which files are likely to contain the highest numbers of faults. However, it does require relatively sophisticated use of statistical modeling, and so the next question we consider is whether the model can be significantly simplified so that a development team can make similar sorts of predictions without needing to rely on someone with a strong background in statistics.

Since file size was the most potent predictor by far, we considered using file size alone to predict which files would contain the largest numbers of faults. Table 6 shows the percentage of faults contained in the 20% of the files that were the largest in terms of the number of lines of code, for the inventory system. While the average percentage of faults contained in the 20% of the files selected by the full model over the seventeen releases is 83%, for the simplified model that value is 73%. Thus although there is a definite difference in the accuracy of the prediction when using the full model to target testing effort as compared to the simplified one, there is evidence from this study that using size alone as a predictor is still likely to be very useful to testers.

The graphs of Figure 1 include the curves for the simplified LOC model. While the LOC model approaches the full model in some of the early releases (especially Releases 6-8), in the later more mature releases, it performs significantly poorer. In particular, note that for the last four releases, the LOC model requires nearly 100% of the files to capture all the release's faults, while the full model reaches 100% of the faults with 40 to 50% of the files. This reflects the fact that some very short files contain faults, and characteristics other than size are needed to raise their ranking.

We also examined the LOC model for Release C of the modified provisioning system. Again, although using file size alone as a predictor is less accurate than using the full model, there is value in using the size-alone predictor. For this system, while the full model selects files containing 83% of the faults, the model based solely on lines of code selects files containing close to 74% of the faults. Thus for both systems we see evidence of a tradeoff between the complexity of the model and required expertise to apply the model, versus the degree of accuracy.

One other difference should be noted. Even though size is the most important factor in the full model, several other factors do affect the ordering of files for testing. That means that when the full model is used, small or medium-sized files that have many faults might receive a high fault prediction because, for example, they are new or changed in the previous release. When the simplified model based solely on file size is used, however, this cannot happen and those files cannot be singled out for particular scrutiny.

8. Conclusions and Future Work

We have continued our investigation of software fault behavior, furthering our goal of being able to predict which particular files in a software system are most likely to contain faults in a new release. Using these predictions, testers can focus their efforts and by so doing, produce reliable software systems more quickly than would otherwise be possible.

The predictions are based on a negative binomial regression model whose variables were selected by using the characteristics we identified as being associated with high fault files. The predictions that we were able to make using this model were in fact very accurate whether they were based on faults found at all stages of development, or restricted to just those faults identified after unit testing was completed. Overall, in both cases, the average percentage of faults contained in the 20% of the files identified by the model as likely to be most problematic, was at least 83% for the inventory system. We also applied our model to another software system, and again the top 20% of the files contained 83% of the faults.

Because using our negative binomial regression model requires a certain level of statistical expertise, we considered a simplified model that involved simply sorting files by length and selecting the 20% of the files that are the largest. This was tried because in the case study we found that size was the most significant factor influencing the number of faults, and it is easy to do, requiring no particular statistics knowledge. We found that for both the inventory system and the provisioning system, this highly simplified model, though not providing the same level of predictive accuracy as the full model, still did moderately well. In particular, for both systems, the predictive accuracy was, on average,

roughly 10 percentage points lower using the size model as compared to the full model.

We have now identified another very large software system with a significant number of releases, and have begun to prepare their data for analysis and to try to apply our prediction model to this system. Since this is a relatively new system that is actively being developed, we are optimistic that our prediction can help their testers use this information to positively impact the quality of the system. Thus we expect to both continue our investigation of the use of our model for targeting testing through prediction, and to help the test team apply it to improve the efficiency and reliability of their system.

If, in fact, we do begin using our prediction as systems are being developed and tested, we may find that its use affects the faults identified and therefore the prediction method may require tuning as development progresses. We plan to see if this is an issue in future work.

Acknowledgments

This work could not have been carried out without the cooperation and assistance of developers and testers at AT&T. Jainag Vallabhaneni, Jim Laur, Joe Pisano, Chaya Schneider, Steve Prisco, and Henry Gurbisz were all generous with their time and willingness to answer our questions about their systems.

Raju Pericherla helped greatly with data acquisition and converting raw data into structured formats, as well as finding and adapting a metrics tool for Java code.

Our colleagues Ken Church, Dave Korn, and John Linderman provided helpful assistance with Unix and Microsoft utilities.

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] 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.

[3] 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.

[4] 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.

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

[6] 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.

[7] T.J. McCabe. A Complexity Measure. *IEEE Trans. on Software Engineering*, Vol 2, 1976, pp.308-320.

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

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

[10] 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.

[11] 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.

[12] T. Ostrand, E.J. Weyuker, and R. Bell. Predicting Fault-Proneness in Large Software Systems. submitted for publication.

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

[14] SAS Institute Inc. SAS/STAT User's Guide, Version 8, SAS Institute, Cary, NC, 1999.