Looking For Bugs in All the Right Places

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

ABSTRACT

We continue investigating the use of a negative binomial regression model to predict which files in a large industrial software system are most likely to contain many faults in the next release. A new empirical study is described whose subject is an automated voice response system. Not only is this system's functionality substantially different from that of the earlier systems we studied (an inventory system and a service provisioning system), it also uses a significantly different software development process. Instead of having regularly scheduled releases as both of the earlier systems did, this system has what are referred to as "continuous releases." We explore the use of three versions of the negative binomial regression model, as well as a simple lines-of-code based model, to make predictions for this system and discuss the differences observed from the earlier studies. Despite the different development process, the best version of the prediction model was able to identify, over the lifetime of the project, 20% of the system's files that contained, on average, nearly three quarters of the faults that were detected in the system's next releases.

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 AND PRIOR WORK

Since substantial resources are expended in efforts to make large industrial software systems highly dependable, we have been developing techniques to accurately predict which files in a software system are most likely to be problematic in the next release. That information should help alert software developers to files that may need to be rewritten or to which particular care should be paid and help testers to prioritize and focus their efforts.

ISSTA'06, July 17-20, 2006, Portland, Maine, USA.

In earlier papers [18, 19] we introduced a negative binomial regression model that we used to accurately predict which files are likely to contain the largest numbers of faults in the next release, and which files are likely to account for a pre-specified percentage of the faults in the next release [20].

These predictions were based on file characteristics that could be objectively assessed, including the size of the file in terms of the number of lines of code (LOC), whether this was the first release in which the file appeared, whether files that occurred in earlier releases had been changed or remained unchanged from the previous release, how many previous releases the file occurred in, how many faults were detected in the file during the previous release, and the programming language in which the file was written. We selected these characteristics because our initial studies indicated that they were the most relevant ones [16].

We performed two case studies using two different large industrial software systems: an inventory system and a service provisioning system. Each of these systems ran continuously, and had been in use for multiple years. For both of the systems we found our model was able to make very accurate predictions by associating a predicted number of faults with each file, and sorting the files in decreasing order of the number of predicted faults. For both systems, when we selected the 20% of the files predicted to contain the largest numbers of faults, they contained, on average, 83% of the actual faults that were detected in the next release.

We also considered a highly simplified prediction model for both the inventory and provisioning systems that was based solely on the number of lines of code in files. We found that for the inventory system, the 20% of the files identified by the LOC Model contained, on average, 73% of the faults, while for the provisioning system those files contained 74% of the faults. Thus, we found that even a very highly simplified prediction model could provide value, although the full model always identified a larger percentage of the faults.

Details of these studies can be found in [18, 19]. In those studies, as well as in this paper, the phrase *actual faults* always refers to the faults that have been detected. Obviously, a system may contain undetected faults, but there is no way for us to be aware of them.

In this paper we continue our investigation using a substantially different type of system with very different characteristics. This is an automated voice response system which is used by many companies to provide customer service while limiting reliance on human operators. The most significant differentiating characteristic of this system is the lack of a

formal release schedule, which made the straightforward application of our earlier models problematic.

All three systems used an integrated version control/change management system that required a *modification request* or MR to be written any time a change was to be made to the system. An MR, which is most commonly written by a developer or tester, may identify either (1) a problem or issue found during internal project testing or reported by a customer or (2) a required or requested change, such as a system enhancement or maintenance update.

MRs contain a great deal of information, including a written description of the reason for the proposed change and a severity rating of 1 through 4 characterizing 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 the change was made.

Most projects begin MR data collection at the time that the system test phase begins for the first release, and this was the case for the provisioning system used in our case studies. The inventory system began data collection far earlier, at the requirements stage, and almost three quarters of the reported faults were identified during unit testing. Unlike system testing, which is typically done by professional testers whose sole job function is to develop and run test cases once the system has been fully integrated, unit testing is generally performed by developers while they are creating individual files. In addition, the system test process tends to be far more carefully controlled than the unit testing phase.

Prior to the start of system testing, the software is typically viewed as a "work in progress," and therefore it is very unusual for a project to begin entering code and MRs into the formal version control/change management system prior to system test. For that reason, we wanted to verify that the prediction results observed for the inventory system were not affected by this unusual behavior, and so we also used the model to make predictions after removing all MRs that were associated with faults found during unit test and earlier. When we restricted the data included and analyzed in this way, the results, averaged over the seventeen releases representing four years of field exposure, were almost the same. In particular, the 20% of the files identified by the model contained, on average, 84% of the faults.

Section 2 of this paper describes related research performed by other groups. In Section 3, we provide details of the voice response system, and point out some of the characteristics that make it particularly different, and therefore, interesting to study. In Section 4 we discuss a solution that we used to deal with the lack of a formal release schedule so that we could build a model and make predictions. Section 5 describes the negative binomial regression methodology and the predictor variables we used. Section 6 describes the variable selection process and presents the resulting models. In Section 7, we assess prospective predictions from four alternative models and compare them with our earlier results. Section 8 summarizes the main findings and presents our conclusions.

2. RELATED WORK BY OTHER GROUPS

Researchers have attempted to identify software properties that correlate with fault-prone code for many years. Some of this work is reported in [1, 2, 4, 5, 7, 8, 13, 14, 16, 21]. Attempts to predict the probability, the locations or the quantity of future faults in code are more recent, and include [3, 6, 9, 10, 12, 15] as well as our research described in [18, 19, 20].

Mockus and Weiss [12] studied maintenance requests made for a large telephone switching system, with the goal of constructing a model to predict the probability that the software changes made in response to a request will cause a failure in the modified system. They based their model on a history of about 15,000 maintenance requests over a period of ten years. Their model's predictors are characteristics of the change that are available immediately after the coding for the change has been completed. These include the size of the change; the number of distinct files, modules, and subsystems that are 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.

Their model is constructed using logistic regression, and does not predict which parts of the code are most likely to contain faults. Instead it predicts either failure or non-failure for a given maintenance request. The model, implemented as a web-based tool available to project management, is used to help schedule the implementation of a given maintenance request, and to determine the level of testing resources to apply to validate the implementation. The paper presents the regression formulas used to create the prediction model and describes the use of the tool in a software maintenance environment, but does not report a success rate for the model's predictions.

Graves et al. [6] performed a study to determine characteristics of modules that are associated with faults, and constructed and evaluated several models for predicting the number of faults that would appear in a future version of the modules. Their study used the fault history of a large telecommunications system containing approximately 1.5 million LOC, organized into 80 modules, containing a total of about 2500 files. The prediction models were used to make fault predictions for a single two year time interval, based on the system's history for the preceding two years. They found that module size was a poor predictor of fault likelihood, while the most accurate predictors included combinations of the module's age, the number of changes made, and the ages of the changes. The authors also described the application of their models to a one year interval in the middle of the original two year interval and found that certain parameter values differed by an order of magnitude between the two time periods. Our results in [19] and in the present paper partly agree and partly conflict with [6]. File age and previous changes were positive indicators of fault-proneness in our studies as well as that of [6], but in contrast to [6], we have consistently found file size to be a strong predictor of fault-proneness.

Denaro and Pezze [3] use logistic regression to construct software fault prediction models based on sets of software metrics. Their method constructs a very large number of candidate models (over 500,000 based on sets of up to five metrics chosen from a list of 38) and then chooses the best models by evaluating how close each model comes to correctly identifying all the fault-prone and non-fault-prone modules in the system. A module is defined to be *fault-prone* if it belongs to the smallest set of modules that are collec-

tively responsible for 80% of all the faults in the system. Their experiment uses the publicly available code and fault databases of the Apache web server. The models are constructed using data from Apache version 1.3 and evaluated for their prediction ability on Apache version 2.0. When the modules of Apache 2.0 are ordered according to decreasing fault occurrence, approximately 36% of the modules account for 80% of the actual faults. The best models of [3] require the first 50% of the predicted fault-prone modules to capture 80% of the actual faults.

Work by Khoshgoftaar, Allen, and Deng [10] presents a method to construct a binary decision tree to classify modules of a system as fault-prone or not fault-prone. The tree's decision nodes are based on 24 static software metrics and four execution time metrics. The approach is demonstrated with four releases of a large legacy telecommunications system. The regression-tree construction algorithm of the S-Plus statistics package is applied to training data from the first release, producing the tree that is then evaluated on data of the next three releases.

The authors evaluate the success of the method according to the misclassification rates of fault-prone modules as non-fault-prone, and non-fault-prone modules as fault-prone. Although it is desirable for both rates to be as low as possible, there is generally an inverse relation between the frequencies of the two types of misclassifications. The method's user can select the value of a parameter to control the rates and then choose the value that comes closest to the software project's needs. For instance, if run-time failures are considered intolerable, then misclassifications of fault-prone modules could be lowered, at the expense of predicting more fault-free modules to be fault-prone, and hence increasing the time required for thorough testing. Choosing a parameter value that attempts to keep the two misclassification rates roughly the same resulted in rates of 32.2% and 21.7% on the last of the four releases evaluated. When applied to a real system, the technique will be valuable to the extent that the set of predicted fault-prone modules contains a higher number of actual faulty modules than an equal-size random selection of modules. The authors calculate that applying their model to a hypothetical future release with the same percentage of faulty modules as release 4 would more than double the number of actually faulty modules selected, in comparison to a random set.

3. THE SYSTEM UNDER STUDY

Both the inventory system and the provisioning system used in our earlier empirical studies had been relatively mature when we first began studying them. The inventory system had been in the field for roughly three years (twelve quarterly releases) when we began data collection and analysis, while the provisioning system had had nine releases and roughly two years of field use. We continued to study the inventory system for an additional five releases, yielding a total of seventeen releases done over a period of roughly four years. We were now particularly interested in finding an appropriate system that was earlier in its lifetime, with the hope of actually influencing the project's behavior by providing predictions that could help to prioritize the project's testing effort.

The voice response system was a prime candidate to investigate. It had been under development for approximately 18 months, was already being used by a number of companies in different industries, exposing it to a substantial amount of traffic and a variety of usage modes, and was still undergoing active enhancement and modification. The project also used the same version control/change management system as the earlier projects, allowing us to make use of the data mining experience we had gained with this tool while doing the earlier studies.

Because one of our biggest previous problems with data collection had been the accurate identification of MRs that represented faults, we requested that this project modify the MR form to include an explicit bug identification field. The modified MR form became available during Month 21 of the data collection phase, and gave users three choices to identify an MR: *bug*, *not a bug*, and *don't know*.

To classify MRs as either fault or non-fault, we used this field in combination with an existing MR field called *MR-Category* that describes the reason for creating the MR. This field can take on values such as developer-found, system-test-found, customer-found, new-code, and new-feature. For our analysis, we restricted attention to only those MRs whose categories were either system-test-found or customer-found, because the system tester's job is to find pre-release faults, and customer-reported problems are field faults that have escaped system testing. Any MR in this restricted set whose Bug-id field either said *bug* or was left blank was considered a fault. If the Bug-id field said *not a bug*, the MR was of course considered not to be a fault. If the Bug-id field's value was *don't know*, we based the decision on a careful reading of the MR's detailed description. There were only a few of these, and all turned out to be faults. All system-test-found or customer-found MRs that were created before the bug-identification field was available were considered faults.

As with the earlier studies, the goal of the present study is to predict the number of faults that executable files of the system will contain in the next release. Applying the original model to a software project requires the existence of a regular sequence of releases, since the model's predictions are based on the fault and change history of files in those releases. This was appropriate for the inventory and provisioning systems, which used a *discrete release* development process, releasing versions at regularly scheduled intervals. In contrast, the voice response project used a *continuous release* process, in which changes are continuously made and tested, and the changed system is passed on to customers immediately after it has been tested. Therefore we had to come up with a way to measure the fault and change history and to create a definition of "faults for the next release" that would be appropriate to this type of continuous release schedule. Our solution to the lack of explicit discrete releases is described in the next section.

In light of the voice response system's different development process, we were not sure whether the original prediction model, even if it could be applied, would produce useful results for this system. There are several reasons for this. In the discrete release model, developers are generally given a date significantly before the scheduled release date to complete their modifications and initialization of code, providing sufficient time for system testers to thoroughly test the code, as well as time for the developers to fix any faults identified during testing and then to retest the code. Also, in the discrete release model, substantial amounts of new and modified code are typically inserted into the system at the same time. Therefore, many development organiza-

tions maintain large *regression test suites*, which are rerun to make sure that all of the changes made to the software have not caused it to regress, in the sense that things that worked properly before changes and/or enhancements, no longer work properly now. Regression testing is typically done just prior to a new release.

The voice response project also took testing very seriously, but it was done on a continuous basis as the software was changed, and hence was more integrated into the development process.

4. DEVELOPING SYNTHETIC RELEASES

Our first attempt at dealing with the lack of regular project releases was to designate successive three month periods as synthetic quarterly "releases". Since these periods are not really releases, we will refer to them in the sequel as *quarters*. Under this definition, files that were initialized any time during a given three month period would be associated with that synthetic quarter. However, this would mean that new files could be in their first quarter for anywhere between one day (if they were initialized on the last day of the quarter) and 92 days (if they were initialized on the first day of the quarter). Obviously, a file that was in a quarter for only a few days would be far less likely to have a failure detected in that quarter than one which had been in the system from the beginning of the quarter, all else being equal. Further, the second quarter history would be very different for those two files, since one would be nearly new during its second quarter, while the other would already have undergone nearly three full months of testing and usage.

Based on these considerations, we modified the synthetic releases in two ways. First, we reduced the time period of the synthetic releases from three months to one month. Monthly releases offer the advantage of allowing us to test for finer temporal patterns in terms of when faults occur relative to file creation and either changes or detection of prior faults.

Second, we modified the synthetic releases to avoid initial release periods that were just a few days long. Because a file may be added to the system on any day of the calendar month, the duration of that file in the system for its initial release period is variable. If a file existed at least one half of the days in its initial calendar month (e.g., 14 days in a non-leap year February or 16 days in March), that month is considered the file's initial month, and its change and fault data are included in the data for that month. However, if a file existed for less than one half of the calendar month, its data for that month are combined with data for the subsequent month, and this combined entity is treated as the file's initial synthetic release period. Thus the change and fault data for Month 1 (June 2002) is for files that entered the system on or before June 16. If a file entered on June 17, it is considered a new file in Month 2 (July 2002), and all of its changes and faults from June are included in the Month 2 data. Consequently, the initial synthetic release period for a file can contain anywhere from 14 to 46 days. This period of time, called the *exposure* of a new file, is expressed as a fraction of a month (varying from about 0.5 to 1.5), when it is used as a variable in the prediction model. In the rest of the paper, we refer to all the periods of a file's existence as *months*, including the initial variable-length period.

Table 1 provides information on the voice response system for each of the 29 calendar months that were tracked. A total of 1928 files entered the system at some point over the 29 months, 1926 of which remained through the last month (one file was dropped after each of months 12 and 20). Over the 29 months studied, a total of 1482 faults were detected, mostly during system testing. With 329,070 lines in its latest version, the voice response system is comparable in size to the systems considered in our earlier studies [18, 19]. The rightmost column of Table 1 shows the monthly fault density (faults detected per thousand LOC) of the system. Although we do not analyze fault density in this paper, we provide these figures here to permit comparisons with the fault densities of the earlier systems studied. To maintain comparability with fault densities reported in [18, 19] and elsewhere, we adjust fault densities to correspond to rates per quarter (three months), that is, fault density = faults per KLOC-quarter. For example, the fault density shown for Month 2 is

$$\frac{3 \times 116}{59,371/1000} = 5.86$$

5. MODELING THE NUMBER OF FAULTS IN A MONTH

5.1 The Negative Binomial Regression Model

Negative binomial regression is an extension of linear regression designed to handle outcomes that are nonnegative integers–such as the number of faults in a file during a specified period [11]. In contrast to Poisson regression, which assumes that faults occur "at random" at a rate explained by a set of predictor variables, the negative binomial model allows for some degree of additional variability in fault counts that is not explained by any of the available predictor variables, technically referred to as *over dispersion*.

Because we want to predict the number of faults discovered for an individual file in a specific month, the unit of analysis is the combination of a file and a month. Consequently, a single file may contribute many observations, corresponding to the number of months it is in the system. Let observation i refer to a specific combination of file and month. We let y_i denote the number of faults observed and x_i be the vector of file characteristics for that month. Like Poisson regression, negative binomial regression models the logarithm of the expected number of faults as a linear combination of the explanatory variables. That is, the model assumes that the expected number of faults varies in a multiplicative way with file characteristics, rather than in an additive relationship.

The negative binomial regression model specifies that y_i has a Poisson distribution with mean $\lambda_i = \gamma_i e^{\beta' x_i}$. A random variable γ_i, drawn from a gamma distribution with mean 1 and unknown variance $\sigma^2 \geq 0$, is included to account for the additional dispersion (variation in the number of faults) observed in the data. Its variance σ^2, known as the *dispersion parameter*, measures the magnitude of the unexplained concentration of faults.

The regression coefficients β and the dispersion parameter σ^2 are typically estimated by maximum likelihood [11]. All estimation was performed using Version 9.1 of SAS [22]. Once a model has been estimated, it is simple to compute the predicted numbers of faults for files in subsequent months, as long as the same set of explanatory variables is measured for all files in the new month. These predictions may be

Month	Number of Files	Lines of Code	Mean LOC	Changes Made	Faults Detected	Fault Density
1	61	15,706	257.5	19	3	0.57
2	325	59,371	182.7	257	116	5.86
3	433	80,619	186.2	237	29	1.08
4	611	107,131	175.3	310	119	3.33
5	758	131,742	173.8	398	119	2.71
6	873	151,232	173.2	247	100	1.98
7	959	161,892	168.8	327	82	1.52
8	1060	174,267	164.4	302	91	1.57
9	1203	202,700	168.5	185	32	0.47
10	1281	210,123	164.0	289	121	1.73
11	1312	217,451	165.7	159	97	1.34
12	1334	229,152	171.8	177	89	1.17
13	1404	241,127	171.7	252	81	1.01
14	1468	252,414	171.9	174	71	0.84
15	1597	265,380	166.2	174	40	0.45
16	1756	297,325	169.3	271	62	0.63
17	1788	301,189	168.5	160	23	0.23
18	1807	304,435	168.5	54	9	0.09
19	1809	304,702	168.4	26	22	0.22
20	1821	306,019	168.0	36	4	0.04
21	1825	306,688	168.0	107	52	0.51
22	1825	306,765	168.1	41	12	0.12
23	1846	313,276	169.7	129	29	0.28
24	1866	316,422	169.6	89	32	0.30
25	1887	319,854	169.5	85	25	0.23
26	1900	321,782	169.4	96	11	0.10
27	1906	324,533	170.3	56	2	0.02
28	1921	326,518	170.0	42	7	0.06
29	1926	329,070	170.9	71	2	0.02

Table 1: Voice Response System Data By Month

used to prioritize files in terms of their expected number of faults, which can then be used for purposes of testing.

5.2 Predictor Variables that were Evaluated

Our analysis of fault-proneness relies on a set of simple, readily-accessible variables that do not depend on the set of programming languages in use. We utilized a set of dynamic and some static file characteristics: the age (time in system) of a file in a given month; the recent history of faults and changes; programming language; and the current size of a file, measured simply as lines of code. In addition, we control for month, or equivalently, for the age of the system.

Graves et al. found that the frequency of faults for a specific file decreased with the age of the file [6]. We explore a variety of relationships with file age. How much more fault-prone were new files (i.e., files that were introduced to the system in the current month) compared with those that also existed in previous releases? After a file is introduced, does any decrease in faults occur primarily in the first few months, or is there a gradual decrease over many following months?

For existing files, the time at risk for fault discovery is always one calendar month. However, because files can be added to the system on any date, the duration of exposure for new files varies from roughly 0.5 to 1.5 months. Because the negative binomial model is multiplicative in the predictors, we parameterize exposure for new files as the logarithm of the fraction of a month at risk (positive if more than a month, negative if less than a full month). For existing files, this predictor variable always equals zero.

We found in [18] that existing files in the inventory system that were changed in the immediately previous release (three months earlier) had about three times as many faults as unchanged files (holding all else equal). In addition to comparing changed and unchanged files, we explore whether predictions are improved by counting the number of times that a file was changed. Because some files were changed a large number of times in the voice response system (sometimes up to 18 times in a single month), we also assessed the square root of the number of changes.

In addition, we look at the history of changes to assess whether changes in the prior month are more important than changes two or three or up to six months before. We utilize similar variables to address whether information about a file's recent fault history improves predictions alone or in combination with information about change history.

We include code mass in the model by using the logarithm of LOC (including comments). If faults occur proportional to LOC (i.e., fault density is constant with respect to LOC), then the coefficient for log(LOC) should be approximately 1.0.

As in [18] and [19], we restrict the fault analysis and predictions to files that are normally written and maintained by programmers (i.e., source files), and that are executable.

Files are selected on the basis of their extension. Thus, for example, .java, .xml, and .jsp files are included in the study, while .doc files are not included since they are not executable, and .jpg and .wav files are not included because they are not produced by a programmer. In [19], we found that fault-proneness differed significantly among programming languages (e.g., java, sql, etc.) even after controlling for lines of code and other file characteristics. We test for similar differences using dummy variables for the most common languages in the voice response system. Because there were 34 distinct languages in use by the last months under study, we group less common languages based on average code mass.

As Table 1 illustrates, fault density falls dramatically, though unevenly, from the early months to the later months of the system. While much of that decline may be explained by other variables such as the age and recent change history of individual files, there may still be systematic differences associated with different months. To control for any such differences, we include dummy variables for each month in our primary models.

6. VARIABLE SELECTION FOR THE FAULT MODELS

We now describe the procedures for selecting the models whose predictions are evaluated in the next section. Selection of predictor variables involves both decisions about what factors to include in the negative binomial regression model and, for some factors, the appropriate time frame(s) and transformations.

Variable selection seldom produces an obvious best set of predictors. Rather than use rote optimization criteria, we used our judgment and past experience to guide the process. However, we did take two steps to ensure that assessment of the model predictions (in Section 7) is based on data that were independent of those data used for variable selection. First, we fixed the specification of the predictor variables based on analysis through Month 9 only. Consequently, results from Months 10 onward could not affect the set of predictors used for those months.

Second, we based variable selection on a training sample consisting of a random sample of 1327 files (each file was independently selected at random with probability 2/3), resulting in 4342 observations (combinations of file and month). This allowed predictions to be assessed on a validation sample of 601 files that were not used at all for either the variable selection or for coefficient estimation. If over fitting to the training sample led to including too many variables in the model, that should show up in poor predictive performance for the validation sample.

6.1 Results for Months 1 to 9

Table 2 presents results for the main model (referred to as the Basic Model) that came out of the variable selection process with data for Months 1 to 9. The sign of an estimated coefficient indicates the direction of the relationship between the value of the corresponding predictor variable and the expected number of faults, holding all other predictor variables fixed. For continuous predictors, the coefficient estimate indicates the change in the logarithm of the expected number of faults associated with a unit change in the value of the predictor variable. This can be translated into a multiplicative change in the expected number of faults by exponentiating the coefficient. For example, a unit increase in the square root of changes in the past month is associated with a multiplicative increase of $e^{0.654} = 1.92$ times more faults. For categorical predictor variables, each coefficient estimates the relative difference in the logarithm of the expected number of faults for the corresponding category versus a reference category where all dummy variables are zero.

One of the most important single predictors was the logarithm of lines of code. For the inventory system, the estimated coefficient of 1.05 did not differ significantly from 1.00, so that the model for that system was consistent with the number of faults being proportional to LOC (holding all else equal) [19]. For the provisioning system, the estimated coefficient was significantly lower than 1.00, at 0.73 [19]. However, for the voice response system the estimated coefficient for log(KLOC) was much lower yet, about 0.5. This result implies that the expected number of faults only grows something like the square root of LOC (LOC raised to the 0.5 power), or equivalently, that fault density declines proportional to the square root of LOC (LOC raised to the (1-0.5)th power).

Exploratory analysis indicated that fault-proneness decreases with the age of a file, even after controlling for other file characteristics such as recent changes and faults. However, this relationship was not explained fully by a single linear term for a file's age (in months). For the inventory system, adding a dummy variable for new files improved the fit, by allowing for a relatively larger decline in faults from the first to the second release that a file existed [19]. For the voice response system, there appeared to be nonlinearity beyond the first month. However, because file age is highly correlated with month number during the first nine months of the system, those data did not allow us to reliably determine the pattern of this relationship. To allow for additional nonlinearity over the first few months after file initialization, we decided to include dummy variables for a file being in its second, third, fourth, or fifth months in the Basic Model, in addition to a dummy variable for new files.

Data on prior changes includes counts for all previous months that a file existed in the system (for any month before a file entered the system, a value of 0 is imputed). Because counts of changes for a given month are very skewed (mostly zero or one, but a few large values), we considered three alternative forms: the raw count, the square root, and a binary indicator of whether there were any changes. Based on improvements in the log likelihood of the model, the square root transformation consistently fit better than either of the other two versions. In addition, we investigated whether only changes from the prior month mattered or whether a longer history improved predictions. This exploration suggested that the most recent month was the best predictor, but that incorporating changes in prior months also improved the log likelihood of the fit, with no clear pattern distinguishing among those prior months. Consequently, we selected two measures of prior changes: the square root of the number of changes in the most recent month and the square root of the total number of changes in the five months prior to the most recent month.

The positive estimated coefficients for these two measures of prior changes each indicate a positive relationship between

Predictor	Coef.	Std. Error	t	95 Percent Conf. Interval
LOC and Exposure				
log(KLOC)	.495	.060	8.30	(.378, .612)
log(exposure)	.620	.367	1.69	(-.099, 1.340)
Age of File				
Age (in months)	-.518	.280	-1.85	(-1.066, .030)
New	-.641	1.555	-.41	(-3.688, 2.406)
Second Month	-1.399	1.290	-1.08	(-3.928, 1.131)
Third Month	-2.112	1.018	-2.07	(-4.108, -.116)
Fourth Month	-1.144	.740	-1.54	(-2.595, .307)
Fifth Month	-1.457	.532	-2.74	(-2.500, -.414)
Square Root of Prior Changes				
Past Month	.654	.104	6.26	(.449, .858)
Months 2-6	.467	.096	4.86	(.278, .655)
Program Type				
Very Small	1.208	.301	4.02	(.618, 1.797)
Small	.567	.195	2.91	(.186, .949)
js	$-\infty$	—	—	—
sh	1.093	.234	4.67	(.634, 1.552)
Month Number				
1	-1.074	1.075	-1.00	(-3.180, 1.033)
2	1.484	.346	4.29	(.806, 2.161)
3	.497	.371	1.34	(-.229, 1.224)
4	1.677	.306	5.49	(1.078, 2.276)
5	1.381	.309	4.47	(.775, 1.986)
6	1.147	.309	3.71	(.541, 1.753)
7	1.199	.299	4.02	(.614, 1.784)
8	1.180	.294	4.02	(.605, 1.756)
Dispersion Parameter				
σ^2	2.278	.356	6.40	(1.580, 2.976)

Table 2: Negative Binomial Regression Results for the Basic Model, Months 1 to 9

the number of changes (on the square root scale) and the expected number of faults, while all other variables are fixed. Furthermore, the sizes of the estimated coefficients provide us estimates of the magnitude of those relationships. For example, the estimated coefficient for the most recent month implies that even a single change during that month is associated with a 92% increase ($e^{0.654} = 1.92$) in the expected number of faults relative to a similar file with no changes in the most recent month.

We conducted a similar analysis to investigate the predictive power of the recent history of faults. For the inventory system, we had found that including the square root of the number of faults during the prior release significantly improved predictions [19]. Although fault history was useful in the absence of change history for the voice response system, it was much less useful than the change history. In addition, once change history was included in the model, no combination of fault history variables proved statistically significant. Consequently, we do not include any measures of prior faults in the model.

Simple comparison of fault densities for the various programming languages in the system highlights large differences among languages. Unfortunately, of 34 languages represented in the system, only a handful occurred often enough to support accurate fitting of language-specific dummy variables. Of those, javascript (js) and shell (sh) were clearly significant. For javascript, there were no faults in 203 file-month combinations and more than 43 KLOC during Months 1 to 9, resulting in a maximum likelihood estimate of $-\infty$ for the js coefficient. In contrast, the coefficient for sh indicates approximately three times as many faults ($e^{1.093} = 2.98$), holding all else equal, for sh files relative to other program types (excluding js and languages whose average file size typically contained very few lines of code). very small groupings).

Exploratory analysis also showed that fault densities by programming language tended to be inversely related to the average length of files of that type. Consequently, we tested dummy variables for groups of languages where the average file size was very small (less than 40 LOC) or small (40 to 65 LOC). In comparison with larger file types (excluding js and sh), each group had significantly more faults than otherwise predicted (after controlling for log(KLOC)).

Finally, the Basic Model includes a series of dummy variables contrasting fault rates in Months 1 to 8 against Month 9. Positive coefficients for Months 2 through 8 reflect the relatively higher fault densities in those months compared with that for Month 9 (see Table 1). The only negative coefficient occurred for Month 1, which had only a slightly higher fault density than Month 9, despite consisting of all new files.

The final row of Table 2 shows the estimated dispersion parameter for the model. Coincidentally, this estimate is almost identical to the estimated value of 2.27 reported for

the "Full Model" for the first 12 releases of the inventory system [17].

In addition to the variables identified above as statistically significant, our model development work identified some other program type dummy variables and interactions that seemed to improve the fit for Months 1 to 9. However, because these terms were generally estimated from small slices of the data, we did not feel confident enough to include them in the Basic Model. The first set includes dummy variables for six additional program types: conf, html, java, jsp, xml, and xsl. Each program type had moderately large and positive estimated coefficients, although standard errors were also large. The second set included interactions of log(KLOC) with dummy variables for conf, html, sh, xml, and the grouping of very small program types. In essence, these interactions allow the coefficient for log(KLOC) to vary by program type. For each of the five types listed above (including very small), the end result was an estimated log(KLOC) coefficient near zero. In other words, the model detected little or no relationship between LOC and faults for those program types. Once those interactions were included, the estimated coefficient for all other program types rose to 0.71.

6.2 Models to be Evaluated

We now define the models whose predictions we will assess in the next section.

6.2.1 The Basic Model

Based on the analyses just described for Months 1 to 9, we specified the variables for inclusion in our primary prediction model, which we call the Basic Model. The Basic Model uses the following predictor variables: the size of the file in terms of the log of KLOC; the logarithm of exposure (the proportion of the month the file was in the system); the age of the file (the number of full months the file was in the system); indicators for whether the file was new, one, two, three, or four months old; the square root of the number of changes made during the prior month; the square root of the total number of changes made during the five months preceding the prior month; indicators for files written in javascript (js) or shell (sh); and indicators of whether the file was written in a language for which the average file size was either very small or small, as defined above. The very small category includes file types cls, pl, and cron, among others, while the small category includes sql, xml, and vxml. In addition, the model included a series of dummy variables for all but one month represented in the analysis.

Although similar to the model we used in [19], the set of predictor variables used in this model differs in important ways. Because [19] analyzed quarterly releases for the inventory system, it did not use as detailed a profile of file age or as detailed a history of changes (only whether there were any changes during the prior release). On the other hand, [19] incorporated faults in the prior release, which we found unnecessary for the voice response system. Also, because the mixture of programming languages used in the voice response system was much more extensive than that for the inventory system, we needed to aggregate many of the less common languages (we chose to group by average LOC) in order to avoid overfitting.

To investigate how well this model, which is based on only the first nine months of data, applies to later months, we refit it on data from Months 10 to 28. We excluded Month 29 from the analysis because it only contained a single fault in the training data. Results might change either if the variable selection involved over fitting (incorrect inclusion of variables based on spurious association) or if the "correct" model systematically changed as the system aged. Table 3 presents regression results for the Basic Model fit to data from the training sample files for Months 10 to 28 (22,298 observations). Although this model included a series of 18 dummy variables for Months 10 to 27 (with Month 28 serving as the reference category), we omit those coefficients from the table because they have little intrinsic interest.

The results for Months 10 to 28 are generally similar to those observed for Months 1 to 9, except for the Age-of-File variables, which could not be stably estimated from nine months of data. The coefficient for log(KLOC) fell even further, to 0.38, but the values for the two periods did not differ significantly. Although the estimated coefficient for log(exposure) tripled in the later period, both estimates were consistent with a value of 1.00 (recall that this explanatory variable varies only for new files, so there was little information for estimating this coefficient precisely in Months 10 to 28).

The coefficients associated with the ages of files are all estimated much more precisely (i.e., with much smaller standard errors) for the later set of months and consequently are easier to interpret. The linear coefficient for file age is -0.094, indicating that the expected number of faults decays by an estimated 9% each month (after the file's fifth month). The coefficient for new files indicates a very strong increase in faults associated with files in their first month (analogous to a similar finding in [19] for files in their first three-month release). In addition, the coefficient for the second month suggests that this phenomenon lasts, in part, for two months. On the other hand, the coefficient for the third month shows a surprising association in the opposite direction.

Coefficients for the prior change variables look very similar in both periods. The same is generally true for the program type coefficients, although faults did occur for javascript files during the later periods, so that the js coefficient is only moderately negative and not statistically significant.

6.2.2 The Enhanced Model

Exploratory analysis of the data for Months 1 to 9 suggested some other relationships that we were not as confident of: dummy variables for a few other common programming languages and interactions between log(KLOC) and selected dummy variables for programming language. Rather than include those predictors in the Basic Model, we designated a so-called Enhanced Model to include in the model assessments in Section 7. The Enhanced Model includes all of the factors in the Basic Model plus: dummy variables for conf, html, java, jsp, xml, and xsl; and interactions of log(KLOC) with each of conf, html, sh, xsl, and the very small grouping.

When corresponding models were fit to the data from Months 10 to 28, coefficients for all six new dummy variables fell in size, suggesting that inclusions of those factors may involve over fitting. In contrast, results for four of the five interactions with LOC (all but for xsl) followed the same pattern as in Months 1 to 9.

Predictor	Coef.	Std. Error	t	95 Percent Conf. Interval
LOC and Exposure				
log(KLOC)	.378	.055	6.86	(.270, .486)
log(exposure)	1.936	.610	3.18	(.741, 3.131)
Age of File				
Age (in months)	-.094	.020	-4.82	(-.133, -.056)
New	1.965	.302	6.51	(1.374, 2.557)
Second Month	.725	.303	2.39	(.132, 1.318)
Third Month	-.902	.374	-2.41	(-1.634, -.169)
Fourth Month	.121	.281	.43	(-.430, .672)
Fifth Month	-.190	.278	-.68	(-.736, .355)
Square Root of Prior Changes				
Past Month	1.037	.123	8.46	(.797, 1.277)
Months 2-6	.642	.073	8.80	(.499, .785)
Program Type				
Very Small	.474	.313	1.52	(-.139, 1.087)
Small	.442	.184	2.41	(.082, .802)
js	-.907	.753	-1.20	(-2.383, .569)
sh	.834	.204	4.09	(.435, 1.233)
Dispersion Parameter				
σ^2	6.454	.836	7.72	(4.816, 8.092)

Table 3: Negative Binomial Regression Results for the Basic Model, Months 10 to 28

6.2.3 The Simplified Model

We also designate a Simplified Model, which uses the following information: the size of the file in terms of the log of KLOC; whether the file was initialized during this month; if not, whether it was changed during the prior month; the month as a reflection of the system's age; and the log(exposure) variable. Assessment of predictions from the Simplified Model helps to tell us whether the additional complexities of the Basic and Enhanced Models improve predictions sufficiently to be worthwhile.

6.2.4 The LOC Model

The LOC Model uses only a count of the number of lines of code in a file. The files are simply ordered by decreasing number of lines, and the appropriate number of files are selected to reach a pre-determined percentage of the files. As with our earlier studies, we consider the number of faults contained in 20% of the files predicted to contain the largest numbers of faults. This simple model proved surprisingly effective in predicting fault-prone files for the inventory and provisioning systems in our earlier studies. We therefore include this model in our current study for completeness.

7. PREDICTION RESULTS

In this section, we assess the prediction models described above, by quantifying how well each model is able to predict prospectively where faults will occur. Specifically, for Month N, we use training data from Months 1 to (N-1) to estimate coefficients for predictive models, which are used to predict the number of faults for every file in the system at Month N. After sorting the files from most to least fault-prone, based on each model's predictions, we compute the percentage of faults contained in the top 20% of files for each ordering.

Although estimated coefficients for each model are based on data that do not overlap the assessment data, there may be concern about bias in the assessment results if the assessment uses the same files that variable selection was based on. Assessment of predictions for the validation sample avoids that bias because none of those files overlap the training sample files used for variable selection. It turns out that prediction results for the validation sample were generally better than those for the training sample, so that bias does not seem to have been a problem. Consequently, we report results in this section for the complete set of files.

Table 4 shows prediction results for the four models. To reduce variability resulting from small numbers of faults in some months, especially later months, we aggregate the findings for sets of three consecutive months. The first row of the table shows results from Months 7, 8, and 9. We continue aggregating every 3 successive months through Months 25 to 27. Months 28 and 29 are not included in the analysis because there are insufficient faults for meaningful analysis.

Values were computed by adding together the total number of faults included in the 20% of the files identified by a given model during the three month period and dividing the results by the total number of faults actually detected by testers and users during the three months. So, for example, during Months 7, 8 and, 9, there were 82, 91, and 32 faults, respectively for a total of 205 faults during the three months (see Table 1). Of those faults, the Basic Model identified files containing 55 in Month 7, 47 in Month 8, and 23 in Month 9, for a total of 125. Therefore, for Months 7 to 9, Table 4 shows that the 20% of the files identified by the Basic Model contained $125/205 = 61.0\%$ of the faults.

The first row of Table 4 shows that for Months 7 to 9, all four models included about the same percentage of faults (60% to 63%) in the top 20% of files. However, results diverged in later months. For the LOC Model, performance trended fairly flat, but with a dip in Months 13 to 18. In contrast, performance for the Basic and Enhanced Models generally improved with time, with each model surpassing 70% for most three-month aggregates. The Simplified Model almost always fell somewhere between these two extremes.

Months	Faults Detected	LOC Model	Simplified Model	Basic Model	Enhanced Model
7-9	205	60.0	61.5	61.0	62.9
10-12	307	58.6	63.2	65.5	69.1
13-15	192	46.9	68.2	79.2	75.0
16-18	94	37.2	53.2	70.2	67.0
19-21	78	62.8	69.2	73.1	70.5
22-24	73	67.1	87.7	97.3	95.9
25-27	38	57.9	68.4	78.9	84.2
Weighted	987	55.5	65.3	71.1	71.4
Unweighted	—	55.8	67.3	75.0	74.9

Table 4: Percentage of Faults Detected in Top 20 Percent of Files, By Three Month Periods

	Inventory System		**Provisioning System**		**Voice System (Basic Model)**	
% of Faults Targeted	% Files	% of Faults Included	% Files	% of Faults Included	% Files	% of Faults Included
60	6	54	8	62	12	60
70	9	65	12	71	20	70
80	15	76	18	81	32	81
90	27	89	31	89	52	89
95	41	94	43	95	67	92

Table 5: Targeted and Included Faults for Inventory, Provisioning, and Voice Response Systems

The final two rows of Table 4 show averages over time, computed in two distinct ways. The row labeled "Weighted" shows averages for each model weighted by the numbers of faults in each three-month period. Overall, the Basic and Enhanced Models performed best and were nearly indistinguishable. The Simplified Model performed less well, but was in the ballpark, while the LOC model lagged badly. The "Unweighted" row shows simple averages of the percentages in each column to allow equitable comparison with numbers reported in [19] for the inventory system. This row shows that both the Basic and Enhanced Models identified 75% of faults on average. By comparison, the most comparable model for the inventory system had averaged 83%. A much larger gap of 18 percentage points occurred for the LOC Model, which averaged only 55% for the voice response system versus 73% for the inventory system [19].

In [20], we also viewed the data from a different perspective: comparing the percentage of faults actually detected in a set of files versus the percentage of faults predicted to be in those files. Table 5 summarizes prior results of this sort and new findings for the voice response system, averaged across all releases or periods. For the inventory system, for example, when files were selected with the goal of containing 70% of all faults, it required 9% of files, which actually contained 65% of faults. Using the same target for the provisioning system, the model required 12% of the files, which included 71% of the faults. When viewed this way, predictions for the voice response system are again very accurate, although a substantially larger fraction of the files is needed to reach a given percentage of faults than for the earlier systems. For instance, if the goal is to capture at least 80% of the faults in the selected files, fewer than 20% of the files are needed, on average, for both the inventory system and the provisioning system. For the voice response system, roughly a third of the files are needed, making the prediction somewhat less valuable for practitioners.

8. CONCLUSIONS

We have performed an empirical study of software fault prediction for an industrial voice response system, following the system over a two and one half year period. Because this project did not release new versions of the system at regularly scheduled intervals, it was unclear whether our previously developed prediction models would be applicable.

To overcome the project's lack of regularly scheduled software releases, we aggregated detected faults into time periods defined by calendar months and constructed a model that makes predictions of faults for a future time period based on information from the previous periods. These periods, called *synthetic releases*, are variable-length months ranging from 14 to 46 days for the first time period that a file enters the system, and calendar months thereafter. With this scheme, we were able to make fault predictions that should be of value to practitioners.

We looked at several different models and found that the two we called the Basic and Enhanced Models performed best. When the months were aggregated into quarters, covering the period from June 2002 through August 2004, the 20% of the files selected by the Basic Model as being most likely to contain the largest numbers of faults contained, on average, 75% of the actual faults.

At a high level, the findings of this study confirm those from our two previous software systems. Similar sets of variable are statistically significant in negative binomial regression models to predict software faults. Although the averages for both of the systems investigated during earlier studies averaged 83% for the top 20% of the files, we consider the current results very encouraging. In particular, they imply that we are able to help developers and testers identify files that need to be targeted and to help prioritize testing activities even if their project does not follow a process that uses regularly scheduled releases.

As expected, notable distinctions arise between systems. The most effective way to account for prior changes differed for the new system. In contrast with the inventory system, information on prior faults was no longer significant once the model had accounted for counts of changes in prior months. Another new finding for this system was that fault densities were much higher for program types that are generally small (less than 65 LOC) than for types that generally exceed that size.

Perhaps the most substantial difference was that lines of code was a much poorer predictor of faults than it had been for the other systems. Rather than faults occurring roughly proportional to size (measured as LOC) as they did for the inventory system, the expected number of faults grew more like the square root of LOC. This difference appears to be behind the poorer predictive performance of the best models for the voice response system, compared with that for the inventory and provisioning systems. Given the various differences among systems, it is clear that one should not blindly apply the "best" model from one system to a new system without adequate validation.

For the inventory system, we concluded [19] that a LOC-only model might be a viable alternative to more complex models because it sacrificed only 10 percentage points in terms of faults included in the top 20% of files. However, the gap for the voice response system is about twice that large on average: 55% versus 75%. Consequently, ignoring other factors would not be a viable strategy for the voice response system. Among the more complete models, the performances for the Basic and Enhanced Models were nearly indistinguishable, so that there seems little reason to use the more complex Enhanced Model. Although the Simplified Model does sacrifice some degree of accuracy when compared with the Basic and Enhanced Models, it may be viewed as a viable alternative given its relative simplicity.

It is, of course, risky to attribute different findings from earlier studies to particular characteristics of the voice response system. However, the findings of this study are valuable because they add to the fairly short list of similar studies for large industrial software systems. In addition, it does show that this sort of prediction modeling can be useful for systems that do not use regularly scheduled releases.

Acknowledgments

We are very grateful to all our colleagues at AT&T without whose cooperation and assistance this research could not have been completed. Teresa Irizarry and Dave Morehead allowed us access to developers, testers, and system administrators, who in turn helped us understand the system structure and software development process for the voice response system. Marcy Braunstein and Jim Wilson were especially helpful in providing details of the system's testing and development processes. Mike Carr provided crucial assistance in understanding the version management system and fault repository and also helped the authors construct scripts to extract system data. Linda Halperin and Dan Doheny were also very helpful with these systems and always very generous with their time. When the version management system was itself replaced with a new version, Andy Nocera helped guide us through the transition. Filip Vokolos of Drexel University was instrumental in creating an accurate inventory of the system's files and faults.

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

[4] 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.

[5] 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.

[6] 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.

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

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

[9] 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.

[10] 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.

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

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

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] T. Ostrand, E.J. Weyuker, and R.M. Bell. Using Static Analysis to Determine Where to Focus Dynamic Testing Effort. *Proc. IEE/Workshop Dynamic Analysis (WODA 04)*, May 2004.

[18] 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.

[19] 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.

[20] T.J. Ostrand, E.J. Weyuker, and R.M. Bell. A Different View of Fault Prediction. *Proc. 29th IEEE Annual International Computer Software and Applications Conference (COMPSAC 2005)*, Edinburgh, Scotland, July, 2005.

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

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