# Programmer-based Fault Prediction

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

## ABSTRACT

**Background**: Previous research has provided evidence that a combination of static code metrics and software history metrics can be used to predict with surprising success which files in the next release of a large system will have the largest numbers of defects. In contrast, very little research exists to indicate whether information about individual developers can profitably be used to improve predictions.

**Aims**: We investigate whether files in a large system that are modified by an individual developer consistently contain either more or fewer faults than the average of all files in the system. The goal of the investigation is to determine whether information about which particular developer modified a file is able to improve defect predictions. We also continue an earlier study to evaluate the use of counts of the number of developers who modified a file as predictors of the file's future faultiness.

**Method**: We analyzed change reports filed by 107 programmers for 16 releases of a system with 1,400,000 LOC and 3100 files. A "bug ratio" was defined for programmers, measuring the proportion of faulty files in release R out of all files modified by the programmer in release R-1. The study compares the bug ratios of individual programmers to the average bug ratio, and also assesses the consistency of the bug ratio across releases for individual programmers.

**Results**: Bug ratios varied widely among all the programmers, as well as for many individual programmers across all the releases that they participated in. We found a statistically significant correlation between the bug ratios for programmers for the first half of changed files versus the ratios for the second half, indicating a measurable degree of persistence in the bug ratio. However, when the computation was repeated with the bug ratio controlled not only by release, but also by file size, the correlation disappeared. In addition to the bug ratios, we confirmed that counts of the cumulative number of different developers changing a file over its lifetime can help to improve predictions, while other developer counts are not helpful.

**Conclusions**: The results from this preliminary study indicate that adding information to a model about which particular developer modified a file is not likely to improve defect predictions. The study is limited to a single large system, and its results may not hold more widely. The bug ratio is only one way of measuring the "fault-proneness" of an individual programmer's coding, and we intend to investigate other ways of evaluating bug introduction by individuals.

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

**General Terms**: Experimentation

**Keywords**: software faults, bug ratio, fault-prone, prediction, regression model, empirical study

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$5.00.

## 1. INTRODUCTION

It is commonly believed that individual programmers differ widely in their ability to write fault-free code, or their propensity to introduce bugs into software. However, many other factors might affect the productivity and accuracy of any individual programmer, including such things as the difficulty of the problem being programmed, the clarity and completeness of the requirements, the time available to do the coding, whether an earlier version of the code was originally written by a different programmer, and the relation of the code to the rest of a larger system.

In our earlier work, we investigated whether the number of developers who interacted with a file during either the previous release, or at any time during the file's existence, influenced the chances of the file having bugs. In [23] we considered three different developer metrics: the number of different people who changed a file during the prior release, the number of different people who changed a file for the first time during the prior release, and the cumulative number of people who changed the file over all releases up to and including the prior release. For the systems we studied, we found that only the cumulative number of developers who changed the file over all prior releases yielded a modest average improvement.

In this paper, we re-examine this question and address the related question of whether the number of faults that occur in files touched by a particular developer in a release can help to predict the future fault proneness of all files touched by that developer.

These issues are attempts to improve the software fault prediction technology we have been developing for several years. We have created a negative binomial regression (NBR) model using both static code characteristics and system history data, and used the model to make predictions for six different large industrial software systems. Together these systems have been in the field for almost thirty five years and we have made predictions for roughly 130 distinct releases of these systems.

The variables used in the basic model include the size of the files, the number of releases the file has been a part of the system, the number of changes made to previous releases, the number of defects found in previous releases, and the language in which the file was written. Here we consider the addition of developer variables.

We have generally based our assessment of our predictions on the percentage of actual faults contained in the files predicted to be most faulty. Usually we have considered the "worst" 20% of the files. Even though the predictions have generally been quite successful with the identified files containing a large percentage of the defects, we have steadily tried to improve the prediction model.

The primary focus of this paper is whether including information about individual developers can further improve prediction results. In informal conversations about our previous research on prediction models, many people have hypothesized that knowing that programmer P made changes to the code might be a very important clue as to whether or not it is likely to contain defects. However, we are aware of little research that has investigated the role that individual developers play in determining the defect content of files.

In particular, we make the following contributions in this paper:

- Analyze a seventh large industrial system, and confirm the effectiveness of our unaugmented, standard negative binomial regression model applied to that system.
- Investigate whether the developer metrics introduced in [23] improve the predictive power of the standard negative binomial regression model for this new system.
- Investigate whether individual developers' performance can be used to enhance the predictive power of the standard model.

## 2. PREDICTION MODEL

We have generally used negative binomial regression (NBR) [12] to model fault counts for files at specific releases as a function of file characteristics at the corresponding releases.

While NBR shares many similarities with standard linear regression, it differs in two fundamental ways. Rather than model the expected value of the outcome itself, NBR models the *logarithm* of the expected fault count as a linear function of predictor variables. Because counts must be nonnegative integers, NBR assumes that each count comes from a negative binomial distribution rather than from the normal distribution implicitly assumed by standard linear regression.

Let $y_i$ be the observed fault count for a specific combination of file and release, and let $x_i$ be a corresponding vector of predictor variables. Negative binomial regression models $y_i$ as a negative binomial distribution with mean $\lambda_i = e^{\beta' x_i}$ and variance $\lambda_i + k{\lambda_i}^2$ for unknown $k \geq 0$. The inclusion of $k{\lambda_i}^2$ allows for over dispersion that we typically observe for fault counts relative to that implied by Poisson regression.

The list of predictor variables is a variety of file characteristics, some static and some that can change from release to release. The most powerful predictor is often code mass of a file at the beginning of the release, measured by the logarithm of the number of lines of code, log(LOC); large files typically have almost proportionately more faults than small files on average.

Because files tend to exhibit the most problems soon after they are added to a system, we account for how long a file has been in the system by four categories of files: new, in one prior release, in 2-4 prior releases, or in 5 or more prior releases. Because a recent history of changes and faults also tends to foreshadow future problems, we include counts of faults in the prior release and changes in each of the two prior releases (set to zero when a file did not exist); we use the square root of each of these counts to reduce skewness of the predictor variables.

We account for the programming language of files, which is sometimes an important predictor, using a series of dummy variables. Finally, it is important to account for the release, also implemented by a series of dummy variables, because the number of faults often varies greatly from release to release, even after controlling for the ages of files in the system.

## 3. PREVIOUS RESULTS

Table 1 includes our prediction results using the basic model for six large industrial, multi-release systems. These include one system with a medium lifespan of 17 releases over four years, two relatively young systems in the field for roughly two years each, and three very mature systems, in use for nine and seven years.

The predictions are made by an automated tool that determines the predicted number of faults that will be in each file in the next release and then sorts the results in decreasing order of predicted number. The tool allows the user to ask to see the predictions for all files or for any percentage of files.

Since we have repeatedly observed that faults are usually heavily concentrated in a relatively small percentage of files, and since the intent of the tool is to identify those files that are likely to contain the largest numbers of faults to help the tester prioritize testing resources, or help projects decide which files should have a code or design review or similar special attention, it is important to restrict attention to a relatively small percentage of the files. For this reason we have typically assessed the success of the predictions by determining what percentage of the actual faults that were detected during a release were in the X% of files predicted to contain the most faults. We often set X to be 20 and that is what is shown in the last column of Table 1.

Although these results have been consistently good, correctly identifying an average of at least 75% of the actual faults for each of the six systems studied, and often averaging over 80%, we still wanted to assure ourselves that we could not do better, and so we assessed prediction methods using three alternate models with the same variables.

We compared the results obtained when making predictions using recursive partitioning, random forests, and Bayesian additive regression trees (BART) to those obtained using the negative binomial regression model for the three systems Maintenance Support A, B and C [24]. The predictions from the negative binomial regression model were consistently statistically better than those from the recur-

| System | Years in the Field | Releases | LOC | % Faults: Top 20% Files |
|---|---|---|---|---|
| Inventory | 4 | 17 | 538,000 | 83% |
| Provisioning | 2 | 9 | 438,000 | 83% |
| Voice Response | 2.25 | 9 | 329,000 | 75% |
| Maintenance Support A | 9 | 35 | 442,000 | 81% |
| Maintenance Support B | 9 | 35 | 384,000 | 93% |
| Maintenance Support C | 7 | 27 | 329,000 | 76% |

**Table 1: Percentage of Faults in Top 20% of Files for Previously Studied Systems**

| Release | Files | LOC | Faults | % Faulty Files | % Changed Files |
|---|---|---|---|---|---|
| 1 | 1832 | 816,000 | 275 | 7.3% | 12.7% |
| 2 | 1847 | 842,000 | 161 | 4.7% | 9.5% |
| 3 | 1860 | 854,000 | 392 | 6.5% | 10.3% |
| 4 | 1913 | 872,000 | 238 | 6.4% | 18.5% |
| 5 | 1933 | 892,000 | 182 | 5.5% | 12.4% |
| 6 | 2027 | 914,000 | 277 | 6.4% | 10.2% |
| 7 | 2190 | 960,000 | 569 | 8.6% | 19.0% |
| 8 | 2320 | 1,005,000 | 347 | 7.2% | 18.5% |
| 9 | 2460 | 1,044,000 | 438 | 8.5% | 24.8% |
| 10 | 2798 | 1,150,000 | 678 | 11.0% | 27.7% |
| 11 | 2843 | 1,206,000 | 563 | 8.1% | 13.9% |
| 12 | 2809 | 1,212,000 | 161 | 3.1% | 3.7% |
| 13 | 3003 | 1,280,000 | 717 | 7.4% | 22.0% |
| 14 | 3058 | 1,323,000 | 452 | 5.8% | 12.5% |
| 15 | 3084 | 1,357,000 | 389 | 5.6% | 9.9% |
| 16 | 3085 | 1,370,000 | 184 | 3.5% | 6.5% |

**Table 2: Size, Faults, and Changes for System 7 (Provisioning System)**

sive partitioning and BART models. While there was no statistical difference between NBR and random forests, the computation time for NBR was substantially less than the computation time for random forests, and therefore NBR was clearly preferable for making predictions.

As mentioned in the Introduction, we have also studied whether developer counts could improve the predictions [23]. Since we did find statistically significant, but modest improvements using the cumulative number of different developers who changed a file, we now consider whether this and other developer information can be useful. In Section 5 we evaluate these same developer count metrics for our new system, and in Section 6 we evaluate the use of individual developer performance.

## 4. A NEW SYSTEM

This section presents fault predictions made for 16 consecutive quarterly releases of a seventh large industrial system. The system performs provisioning services, similar to the functionality of the second system we analyzed, but on a much larger scale. The new provisioning system is our largest subject, containing a total of roughly 1,400,000 lines of code and 3,100 files in the last release, with most files written in Java.

Table 2 gives basic information about the size of this system, including the percent of files in each release that contained faults, and the percent that were changed during each release.

### 4.1 Evaluation Metrics

Various measures have been proposed to evaluate the success of predictor models. Ohlsson and Alberg [19] defined the *Alberg diagram* as a means of visualizing the cumulative faults that are contained in successively larger sets of the files, when they are put in descending order of predicted number of faults.

If the prediction is in perfect order, with the files in descending order of the true number of faults they contain, the Alberg curve reaches the maximum of 100% of the faults as rapidly as possible. Once all faulty files are identified, the curve is horizontal for the remaining non-faulty files.

If the order is not perfect, then there will be a gap between the curve representing the perfect order, and the curve representing the predicted order. The total area under the Alberg curve is a measure of the quality of the prediction, with the perfect order prediction having the maximum possible area, and the farther the divergence of the perfect and prediction, the smaller the area will be.

As noted above, we have usually assessed the results of the prediction models in terms of the percent of actual faults (the *fault yield*) detected in the first 20% of files in the prediction. The cutoff value of 20% was chosen because of the often-noted 80/20 Pareto distribution of faults, and because it is a relatively small percentage of the files that practitioners can use to target files for particular scrutiny. Additionally, it offers a simple way to assess results and to compare different prediction models and systems.

However, because 20% can still seem a somewhat arbitrary value, we recently proposed an alternate measure called the *fault-percentile average* that represents the average fault yield over all values of the cutoff percent. More precisely, the fault-percentile average is the average, over all values of $m$,

| Release | % Faults: Top 20% Standard Model Without Developers | % Faults: Top 20% Model with Cumulative Developers | Increment |
|---|---|---|---|
| 3 | 85.2 | 89.3 | 4.1 |
| 4 | 83.2 | 85.3 | 2.1 |
| 5 | 89.0 | 89.6 | 0.6 |
| 6 | 67.9 | 70.8 | 2.9 |
| 7 | 86.3 | 89.3 | 3.0 |
| 8 | 86.7 | 89.0 | 2.3 |
| 9 | 84.7 | 83.8 | -0.9 |
| 10 | 80.4 | 81.0 | 0.6 |
| 11 | 89.7 | 90.4 | 0.7 |
| 12 | 87.0 | 90.0 | 3.0 |
| 13 | 91.6 | 93.6 | 2.0 |
| 14 | 91.8 | 93.4 | 1.6 |
| 15 | 94.6 | 96.1 | 1.5 |
| 16 | 94.0 | 96.2 | 2.2 |
| Avg | 86.6 | 88.4 | 1.8 |

**Table 3: Top 20% Metric for Model With and Without Developer Information**

| Release | FPA: Standard Model Without Developers | FPA: Model with Cumulative Developers | Increment |
|---|---|---|---|
| 3 | 90.9 | 91.8 | 0.9 |
| 4 | 90.1 | 89.9 | -0.2 |
| 5 | 92.0 | 91.8 | -0.2 |
| 6 | 79.2 | 79.6 | 0.4 |
| 7 | 91.0 | 92.1 | 1.1 |
| 8 | 90.8 | 91.5 | 0.7 |
| 9 | 90.7 | 90.5 | -0.2 |
| 10 | 87.7 | 87.9 | 0.2 |
| 11 | 93.0 | 92.8 | -0.2 |
| 12 | 92.3 | 93.0 | 0.7 |
| 13 | 93.3 | 94.0 | 0.7 |
| 14 | 93.5 | 94.3 | 0.8 |
| 15 | 95.3 | 95.5 | 0.2 |
| 16 | 95.0 | 95.5 | 0.5 |
| Avg | 91.1 | 91.4 | 0.4 |

**Table 4: Fault-Percentile-Average Metric With and Without Developer Information**

of the percent of faults contained in the top $m$ files, sorted in decreasing order of predicted numbers of faults.

When a model's predictions are good, the files with high fault counts will be clustered at the beginning of the ordering, and will be repeatedly added into the average. A file predicted to contain few or no faults will appear far down the list and will be added into the average very few times. If such a file in fact contains many faults, those faults will be added into the total only a relatively few times, leading to a low fault-percentile average. In [24], we showed that this value is equivalent to the area under the predictor curve in the Alberg diagram.

The fault-percentile average is primarily used to evaluate the overall success of a prediction model that has been applied to multiple releases of a system, and to compare models across several systems that may differ in their size and the number of faults they contain. For a prospective user of the prediction model, both the fault-percentile average and the fault yield from a specific cutoff can provide confidence in the model's accuracy.

We have usually found the 20% cutoff values most useful for practitioners as the results are easy to understand and directly related to their goal of deciding which files to target for particular attention. On the other hand, researchers might find the fault-percentile average metric (or Alberg diagram) more useful for comparing the results observed for competing prediction models.

## 4.2 Prediction Results

Column 2 of Table 3 shows the prediction results for the new provisioning system using the basic negative binomial regression model with a 20% cutoff value, while Column 2 of Table 4 shows the fault-percentile averages achieved for this system using the basic NBR model. What we observe is that the two ways of assessing predictive success yield similar information, with the choice of which metric to use being generally motivated by the intended use of the results.

We observe that for this system the value of the fault-

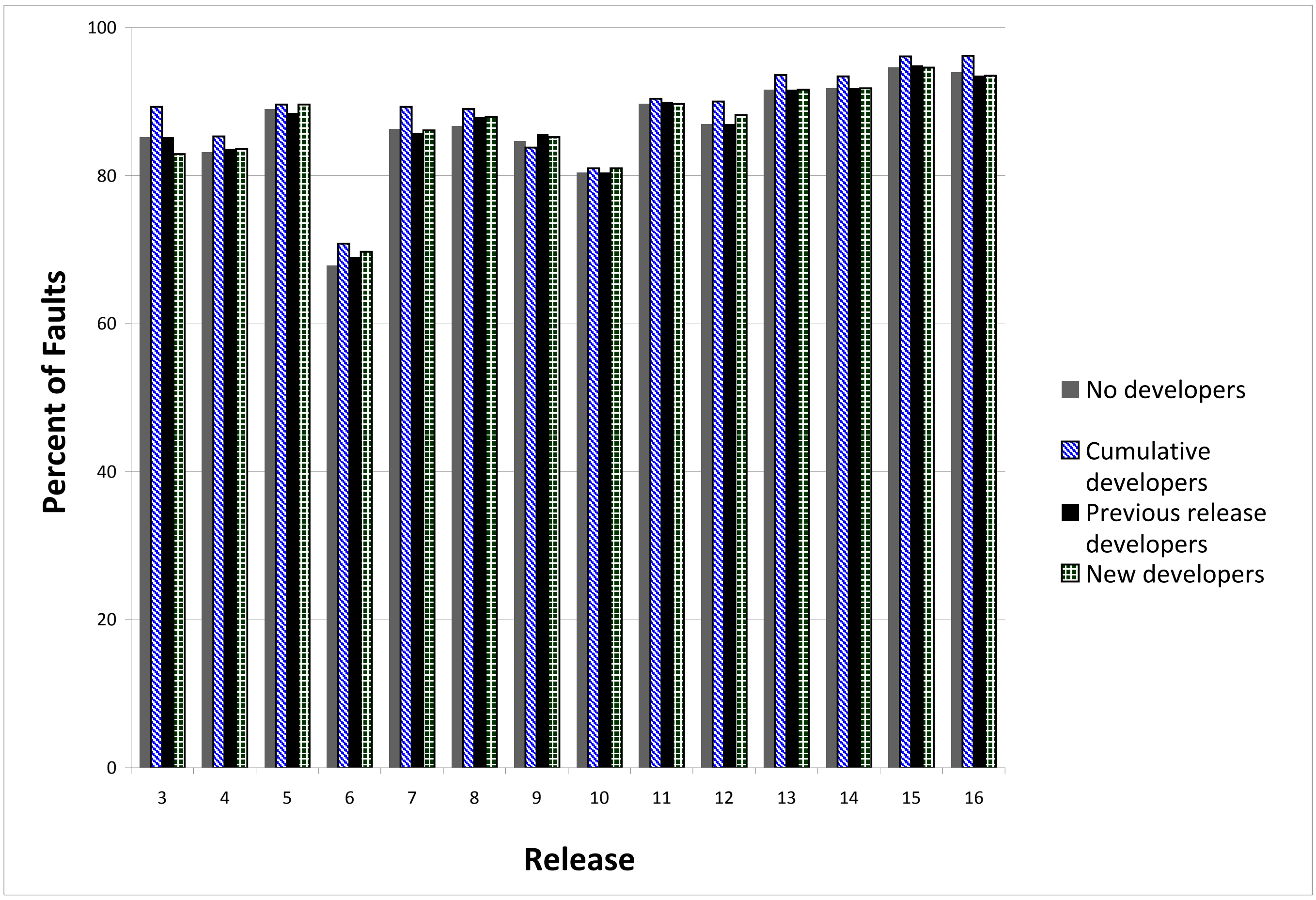

**Figure 1: Standard and Developer-augmented Models for Releases 3-16**

percentile average metric is always greater than the fault yield of the 20% cutoff. As the cutoff percent is increased, it will eventually reach a point where the percent of faults captured will exceed the fault-percentile average. If a faulty file falls outside the cutoff, even if very near it, it gets counted not at all in the cutoff metric, whereas if a file is ranked as less fault-prone than it really is, when using the fault-percentile average metric, it will still be counted, but with a lower weight. Therefore we see a sort of moderating effect when using the fault-percentile average metric as long as the predictions are reasonably accurate.

## 5. DOES DEVELOPER DATA HELP?

As mentioned in the Introduction, we previously considered whether the addition of certain types of developer information improved the percentage of defects detected in the targeted 20% of the files. In [23] we studied three systems and observed that there was a minimal improvement, on average, when we augmented the model with a variable that counted the cumulative number of different developers who had modified the file in all releases up to and including the one prior to the release for which we want to make predictions. For those three systems, Maintenance Support A, B, and C, the improvement was 0.2%, 1.0% and 0%, respectively.

For the other two ways of augmenting the model with developer information, the number of distinct developers who modified files in the previous release, and the number of developers who modified a file for the first time in the previous release, we observed no statistically significant improvement in the prediction results for any of the three systems.

The three developer-augmented prediction models delivered very similar results for System 7. Figure 1 shows the percentage of faults in the top 20% of files for the standard and the three augmented models across Releases 3-16 of System 7. The results from the two models using previous release modifiers and first-time modifiers showed almost no difference from the standard model results. In contrast, the model including cumulative developers again showed modest improvement over the standard model. For all except Release 9, the cumulative developer model improves over the standard model, and it is roughly equal to or better than either of the other two augmented models.

Table 3 shows a precise comparison of results without and with cumulative developers for the 20% cutoff metric, with the increment due to cumulative developers displayed in the last column. Across 14 releases, the increment is positive 13 times, and the average increment is 1.81% with a standard error of 0.35% (based on release-to-release variation). Table 4 shows the same comparison for the fault-percentile average metric. For this metric, the increment is positive for 10 of 14 releases, and the average increment is 0.40% with a standard error of 0.13%. Consequently, the improvement across all releases is statistically significant for both metrics.

Because the fault-percentile average is equivalent to averaging the top X% method for all possible values of X, the large discrepancy between the increments averaged across releases (1.81% verus 0.40%) is surprising. Figure 2 shows increments in the top X% metric for each integer X from 0 to 100 (each displayed value is an average across 14 releases). The graph shows that the biggest increments are all in the range of X between 13% to 32%. Although the value at X = 20% is at the high end of the range, the result would exceed 1.0% for any value of X in the neighborhood of 20%.

## 6. INDIVIDUAL DEVELOPER INFLUENCE

In this section, we investigate whether tracking results for individual developers might improve prediction models. In particular, are the files changed by certain developers systematically more (or less) likely to contain faults at the next release. If so, we might be able to use that information to improve the accuracy of our prediction models.

Before presenting our analysis methods and findings, we want to emphasize that we are not investigating whether developers differ in quality of their coding. Even if some developers' files have systematically more faults than average, that does not imply that they are performing poorly. Perhaps the best developers are assigned the most difficult files to develop and maintain. But even if differences of this sort are unrelated to developer quality, they might still be valuable for predicting faulty files, and thereby improving the efficiency of software testing.

We note that there is a challenge in using individual developer data compared with file characteristics such as lines of code, prior faults or changes, and software language. Gathering sufficient data about individual developers to support reliable estimation of the likelihood that their files contain faults may require many releases. For example, over the first 15 releases of the system under study, most of the 107 developers changed fewer than 35 files even when we count separately changes to the same file in multiple releases.

Our analysis focuses on the occurrence of faults among files that underwent at least one change in the prior release. Thus, we ignore the vast majority of files at any release that are either new to the system or unchanged during the prior release. Although previously changed files typically constitute only 10 to 20 percent of all files for most releases, they constitute 67 percent of faulty files for releases 2 to 16.

Over the first 15 releases of the system, there were 5464 instances of changed files, counting only one change of a file during any particular release. About 28 percent of changed files involved changes by two or more developers, resulting in a total of 8384 combinations of file, release, and developer who made one or more changes.

Figure 3 displays graphs of a release-to-release *bug ratio* for 12 of the 43 developers who changed at least 50 files over the 15 releases. The bug ratio for Developer D at Release R is the proportion of faulty files in Release R among all files changed by D in Release R-1. In the graphs, each point is accompanied by a label showing the number of files changed by that developer during the prior release. For example, the fourth point in the upper left panel of Figure 3 shows that Developer 2 changed 17 files during Release 14. The vertical placement of the point indicates the bug ratio, the proportion of these files that were faulty (i.e., contained at least one fault) at Release 15. For this case the bug ratio is 0.41 (7 out of 17). As a further example, the middle graph in the bottom row shows that all 3 of the files changed by Developer 38 at Release 5 had bugs in Release 6, and 4 of the 5 files changed in Release 12 had bugs in Release 13.

| **Control Variables** | **Correlation** | **P-Value** |
|---|---|---|
| Release | 0.388 | 0.010 |
| Release, log(LOC) | 0.091 | 0.564 |

**Table 5: Pearson Correlations between Excess-Fault Measures in Early and Late Halves**

The solid background line in each graph shows the mean proportion of buggy files across developers for each release, and serves as a benchmark. Note that any files changed by multiple developers during a single release would be represented in the plots for each of those developers.

The 12 developers shown in Figure 3 are a systematic, non-random sample selected to best represent the range of developers in terms of their overall proportion of faulty files relative to the averages for releases where they made changes. Moving from the top row to the bottom, results are shown for developers with relatively few faulty files after accounting for the release when those files were changed to those with relatively many faulty files. While results for some developers almost always fall either below or above the the benchmark line, results for many of the developers move back and forth frequently.

The wide variety of patterns across the plots in Fig 3, as well as in the ones not included, makes it difficult to characterize the overall degree of persistence across developers. For that reason, we devised a simpler way to look at whether results for developers persisted over time. For each developer, we divided the releases into two sets of contiguous releases - early and late - that most closely divided the number of changed files in half. For example, for Developer 2, the closest to equal division places releases 11, 12, and 13 in the "early half", and releases 14 and 15 in the "late half", yielding 56 and 22 files in the early and late halves, respectively; notably, this is the third most unbalanced split for any of the 43 developers. For developers active earlier in the life of this system, the split point would move up accordingly.

For each half for each developer, we computed a measure of excess faulty files among the files changed by the developer in the prior release. This measure was the proportion of faulty files among those changed files in the prior release minus the proportion predicted based on which releases the developer made changes. For example, Developer 2 changed 22 files in his/her late half. Of those files, 0.32 (7 of 22) contained faults at the next release. Based on the overall results for the two relevant releases, we would have expected about 0.47 of the files to be faulty. Consequently, the excess is 0.32 - 0.47 = -0.15. The first row of Table 5 shows the Pearson correlation between the early and late excess-faults measures just described. That correlation, 0.388, is sizable and statistically significant, providing strong evidence that to some extent the fault propensity of files changed by a developer persists over time. However, the correlation alone tells us nothing about the reason for this pattern.

Further exploration reveals code mass as a possible explanation. Developers with above average values on the excess-fault measures (e.g., those in the last row of Figure 3) tend

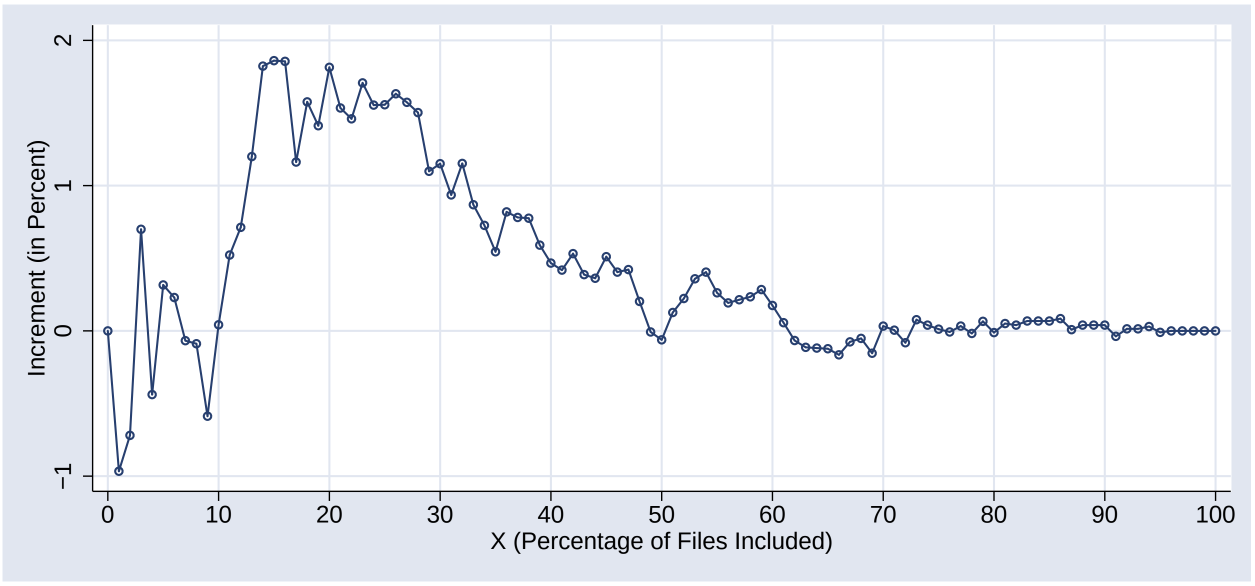

**Figure 2: Increment in Top X% Metric from Using Cumulative Developer Model, Average across Releases 3-16**

to have changed files that are larger than average. Furthermore, there is a high correlation between a developer's average LOC in the early and late halves.

To investigate this connection more formally, we computed new excess-fault measures that controlled for code mass (log(LOC)) as well as release. Specifically, we fit a logistic regression for whether a changed file was faulty as a function of release (treated as a categorical variable) and log(LOC). The new excess-fault measure equaled the observed proportion of faulty files for a developer minus the proportion predicted from the logistic regression.

The correlation between the new excess-fault measures adjusted for log(LOC) dropped to 0.091, far from being statistically significant. Most of the evidence of a persistent pattern associated with developers is explained by variation in the sizes of the files that the developers have changed. Because log(LOC) is already an integral input to our prediction model, there is little reason to expect that using individual developers would improve predictions further.

## 7. THREATS TO VALIDITY

Any time one performs an empirical study to assess the success of a proposed technique such as our negative binomial regression model to predict fault-proneness, it is necessary to determine the extent to which these results can be assumed to hold for systems other than the one explicitly studied.

This is why replication of studies is so critical. We have now applied our standard predictive model to seven different industrial software systems, each of which has run continuously in the field for multiple years.

Collectively the systems we have studied have been in the field for well over 35 years. We have made predictions for over 145 releases. The consistency of our results lend credence to the success of our model, and the diversity of the systems encourage us to believe that this model is in some sense "universal" and applicable to many different types of systems. The more different systems we can apply it to, the more evidence we can present, and the more confident we can be in using our tool to guide prioritized quality efforts. The same is true for replicating the studies about the effects of the cumulative number of developers.

Of course, it can never be proven that the results will be similar for the next system we apply our model to, but we are unaware of any other prediction models that have been so widely applied to real systems containing naturally-occurring bugs, allowing us to draw similar conclusions.

Of course, our negative finding about the potential value of individual developer information to improve predictions is less solid because it is based on a limited number of developers working on a single system. Relationships for another system might be quite different depending on characteristics of the programming tasks and how developers are assigned work.

## 8. RELATED WORK

Fault prediction researchers have investigated a wide variety of predictor variables. Many authors have utilized code mass, various complexity metrics, design information and variations on the history variables that make up our basic model. Work of this type is described in papers such as [2, 3, 6, 14, 15, 17, 18, 20, 23, 24, 25]

A number of research groups have also considered whether the addition of calling structure information improved the success of identifying faulty code entities over models that did not include that information. Some interesting papers in that category include [1, 2, 3, 4, 5, 7, 8, 11, 19, 21, 25]

There has also been extensive work with various ways of assessing the success of fault prediction models including work reported in [1, 9, 10, 13, 22].

To the best of our knowledge, however, only Mockus and Weiss [16] studied the relationship between faults and four different developer variables, mostly assessing the developer's experience. They used a logistic regression model to predict

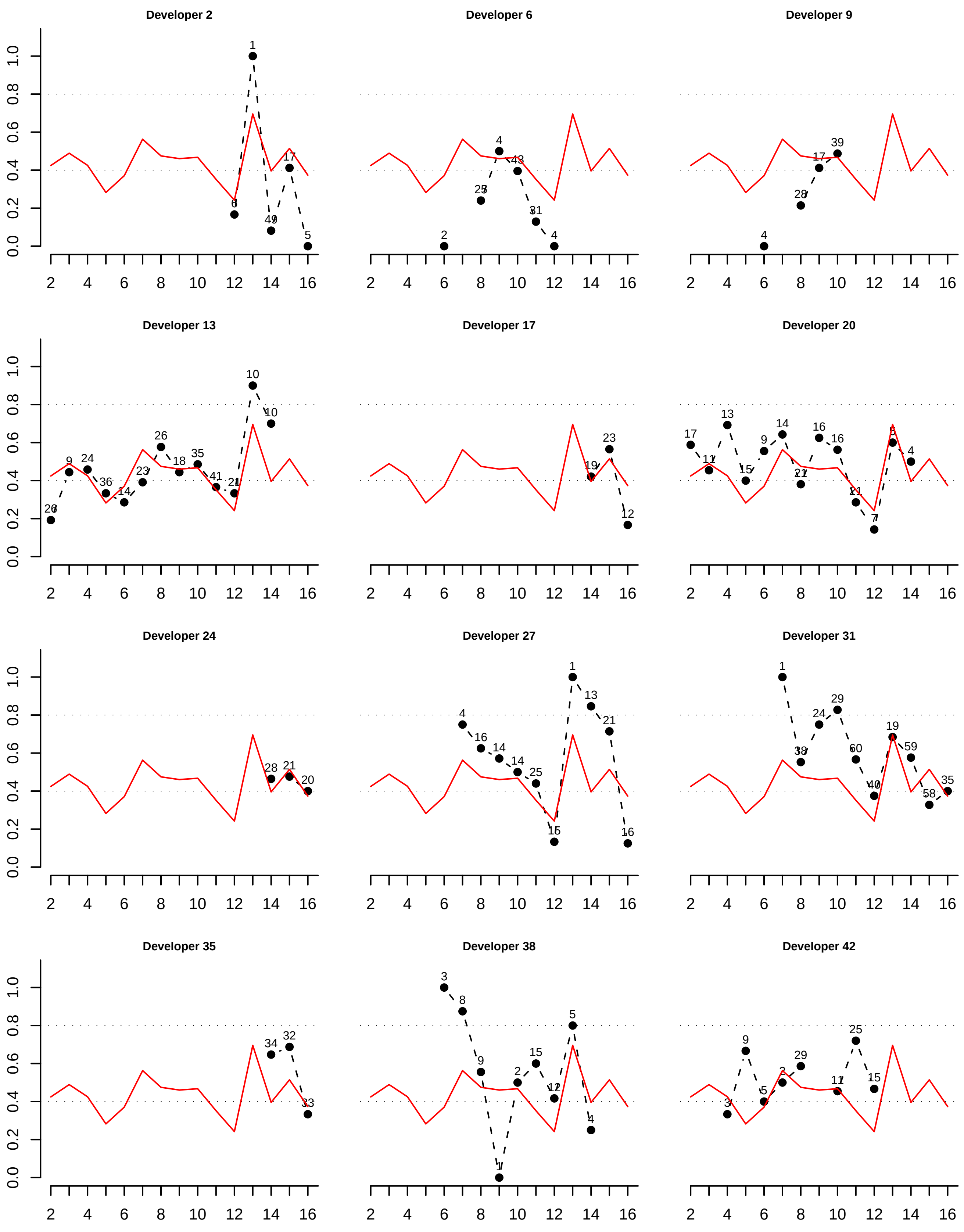

**Figure 3: Selected developers bug ratios**

whether a specific change leads to a failure using a large, mature telephone switching system. The four developer variables they used included: a measure of the developer's overall experience, their recent experience, and their experience with a specific subsystem, as well as the number of developers who made modifications to satisfy a Modification Request. They found that the only statistically significant variable was the one that measured the developer's overall experience level.

Our augmented models also included the number of developers who modified a file, but our use of individual developer information considered using a fault ratio on the files changed by each developer as a predictor variable, instead of the experience measures that Mockus and Weiss used.

It is interesting to note that while we found both in our earlier study of three systems [23] and for the new system studied here, that the cumulative number of different developers changing a file was statistically significant, Mockus and Weiss found that their developer count variable was not statistically significant. Of course, we found both here and in our earlier work that two other count variables, the number of new developers changing a file during the prior release, and the total number of developers changing a file during the prior release, were not statistically significant.

## 9. CONCLUSIONS

We have provided more confirmatory evidence that we can successfully predict which files will be most fault-prone in the next release of a software system. We have now used a standard model and automated tool to make predictions for many releases of seven different software systems at many different levels of maturity. In all cases we have seen strikingly similar results. We see that we can pinpont 20% of the files as particularly likely to be fault-prone and they will typically contain from 75 - 95% of the faults.

In our environment, this is considered very valuable help for development teams as it allows them to better focus their resources. Also critically important for acceptance is the fact that the entire prediction process is fully automated which means that no data mining or statistics expertise is needed nor any knowledge of the types of metrics that contribute to the prediction.

We have also confirmed our earlier findings that using cumulative developer information can improve prediction results, but only by a small amount. In contrast, we did not find evidence that indicates an individual developer's past performance is an effective predictor of future bug locations. While some evidence was found of persistent variation among developers in the prevalence of faults among their changed files, most of that variation appears to be explained by code mass, which is already included in our prediction models.

## 10. REFERENCES

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

[2] V.R. Basili, L.C. Briand, and W.L. Melo. A Validation of Object-oriented Design Metrics as Quality Indicators. *IEEE Trans. Software Eng.*, 22(10), pp.751-761, 1996.

[3] A. Binkley, and S. Schach. Validation of the Coupling Dependency Metric as a Predictor of Run-time Failures and Maintenance Measures. *Proc. 20th Int. Conf. Software Engineering*, pp. 452-455, 1998.

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

[5] Y. Jiang, B. Cukic, T. Menzies, and N. Bartlow. Comparing Design and Code Metrics for Software Quality Prediction. *Proc. 4th International Workshop on Predictor Models in Software Engineering (PROMISE'08)*, pp. 11-18.

[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] 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.

[8] 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.

[9] S. Lessmann, B. Baesens, C. Mues and S. Pietsch. Benchmarking Classification Models for Software Defect Prediction: A Proposed Framework and Novel Findings. *IEEE Trans. Software Eng.*, 34(4), pp.485-496, 2008.

[10] Y. Ma and B. Cukic. Adequate and Precise Evaluation of Quality Models in Software Engineering Studies. *Proc. International Workshop on Predictor Models in Software Engineering (PROMISE'07)* 2007.

[11] A. Marcus, D. Poshyvanyk, and R. Ferenc. Using the Conceptual Cohesion of Classes for Fault Prediction in Object-oriented Systems. *IEEE Trans. Software Eng.*, 34(2), pp. 287-300, 2007.

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

[13] T. Mende and R. Koschke. Revisiting the Evaluation of Defect Prediction Models. *Procl. 5 International Conference on Predictor Models in Software Engineering (PROMISE'09)*, 2009.

[14] T. Menzies, J. Greenwald and A. Frank. Data Mining Static Code Attributes to Learn Defect Predictors. *IEEE Trans. Software Eng.*, 33(1), pp. 2-13, 2007.

[15] T. Menzies, B. Turhan, A. Bener, G. Gay, B. Cukic and Y. Jiang. Implications of Ceiling Effects in Defect Predictors. *Proc. 4th Int. Workshop on Predictor Models in Software Engineering (PROMISE08)*, pp. 47-54, 2008.

[16] A. Mockus and D.M. Weiss. Predicting Risk of Software Changes. *Bell Labs Technical Journal*,

April-June 2000, pp. 169-180.

[17] N Nagappan and T. Ball. Use of Relative Code Churn Measures to Predict System Defect Density. *Proc. 27th Int. Conference on Software Engineering (ICSE05)*, pp.284-292, 2005.

[18] N. Nagappan, T. Ball, and A. Zeller. Mining Metrics to Predict Component Failures. *Proc 28th Int. Conference on Software Engineering (ICSE06)*, pp. 452-461, 2006.

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

[20] T.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.

[21] Y. Shin, R. Bell, T. Ostrand, and E. Weyuker. Does Calling Structure Information Improve the Accuracy of Fault Prediction? *Proc. Mining Software Repositories (MSR09)*, May 2009

[22] E.J. Weyuker, R.M. Bell, and T.J. Ostrand. We're Finding Most of the Bugs, but What are we Missing? *Proc International Conference on Software Testing*, Paris, April 2010.

[23] E.J. Weyuker, T.J. Ostrand, and R.M. Bell. Do Too Many Cooks Spoil the Broth? Using the Number of Developers to Enhance Defect Prediction Models. *Empirical Software Eng. Journal*, October 2008.

[24] E.J. Weyuker, T.J. Ostrand, and R.M. Bell. Comparing the Effectiveness of Several Modeling Methods for Fault Prediction. *Empirical Software Eng. Journal*, 2009.

[25] T. Zimmermann and N. Nagappan. Predicting Defects Using Network Analysis on Dependency Graphs. *Proc. 13th Int. Conference on Software Engineering (ICSE'08)*, p.531-540, 2008.