Building Effective Defect-Prediction Models in Practice

A. Güneş Koru and Hongfang Liu, *University of Maryland, Baltimore County*

Static measures and defect data collected at class level can be used to build machine-learning models that predict top defect classes in practice.

Defective software modules cause software failures, increase development and maintenance costs, and decrease customer satisfaction. Effective defect prediction models can help developers focus quality assurance activities on defect-prone modules and thus improve software quality by using resources more efficiently. These models often use static measures obtained from source code—mainly size, coupling, cohesion, inheritance, and complexity measures—which have been associated with risk factors, such as defects and changes.

Emerging software repositories of publicly available data sets, such as the PROMISE repository,[1] let us build defect prediction models and obtain empirical results. These data sets facilitate an open discussion about research methodologies and results. To this end, we analyzed the CM1, JM1, KC1, KC2, and PC1 data sets in the PROMISE repository, which belong to five software products developed by NASA. Lessons learned from these analyses led us to investigate how module size relates to defect prediction performance. When we stratified the data sets according to module size, we found improved prediction performance in the subsets that included larger modules. Based on these results, we developed some guidelines that practitioners can follow in their defect-prediction efforts.

Preliminary analysis

The NASA data sets are available online (http://promise.site.uottawa.ca/SERepository/datasets-page.html). Each set includes introductory text that lists and explains the static measures and other variables. Each data point in a data set corresponds to a *module* and includes a set of statistic measures as well as a binary variable indicating whether or not a defect has been reported for the module. A defect report reflects any module changes required to correct a problem or related problems.

We built several models to predict the defective modules in these products, using the static measures as predictor variables and the binary defectiveness indicator as the response variable. (For general concepts about building defect-prediction models, see the "Predicting Defect-Prone Modules" sidebar.) To do so, we used the machine-learning algorithms available in the

Predicting Defect-Prone Software Modules

Various machine learning models, such as classification trees and neural networks, can support defect prediction. These relatively sophisticated models are preferable to simple linear regression and correlation models because the relationship between defects (response variable) and static measures (predictor variables) might not be a monotonous linear relationship.[1]

To build a prediction model, you must have defect and measurement data collected from actual software development efforts to use as the learning set. A trade-off usually exists between how well a model fits to its learning set and its prediction performance on additional data sets (test sets). Therefore, you should evaluate a model's performance by comparing the predicted defectiveness of the modules in a test set against their actual defectiveness.

In the absence of additional data sets, it is a common practice to use a portion of the available data as the learning set to build a model and the remainder as the test set to evaluate prediction performance. One widely used evaluation method, *10-fold cross validation*, partitions the data set into 10 equal portions. This method uses each portion once as the test set to evaluate the model built using the remaining nine portions.

Figure A shows the commonly used prediction performance measures: precision, recall, and F-measure.

A model's *precision* is defined as the ratio of the number of modules correctly predicted as defective, or true positive (*tp*), to the total number of modules predicted as defective in the set (*tp* + *fp*). A model's *recall* is defined as the ratio of the number of modules predicted correctly as defective (*tp*) to the total number of defective modules in the set (*tp* + *fn*).

To perform well, a model must achieve both high precision and high recall. However, a trade-off exists between precision and recall. For example, if a model predicts all modules as defective, its recall will be 1 but the precision will be low. Obviously, in this case, we can't say that the model performs well. On the other hand, if a model predicts only one module as defective and the prediction turns out to be correct, the model's precision will be 1. Yet again, we can't consider this model to have good performance.

In this study, we use *F-measure* to measure prediction performance. F-measure considers precision and recall equally important by taking their harmonic mean (see figure A for the formula). Like precision and recall, F-measure is always valued between 0 and 1, and a higher value indicates better prediction performance.

Reference

1. A.G. Koru and J. Tian, "An Empirical Comparison and Characterization of High Defect and High Complexity Modules," *J. Systems and Software*, vol. 67, no. 3, 2003, pp. 153–163.

Confusion matrix

Predicted

Actual	DEF	NODEF
DEF	*tp*	*fn*
NODEF	*fp*	*tn*

tp : number of true positives
fp : number of false positives
fn : number of false negatives
tn : number of true negatives
Precision = *tp*/(*tp* + *fp*)
Recall = *tp*/(*tp* + *fn*)
F-measure = 2 * Recall * Precision/(Recall + Precision)

Figure A. Measures of prediction performance on a typical confusion matrix, which shows actual and predicted defectiveness. DEF means "defective," and NODEF means "not defective."

WEKA (Waikato Environment for Knowledge Analysis) open source tool.[2]

The prediction performance was not discouraging but not very satisfactory either, as other researchers have also found. For example, Tim Menzies and his colleagues worked on JM1 and KC2 and found a low probability of detecting defective modules.[3] Taghi Khoshgoftaar and Naeem Seliya performed an extensive comparison of many machine-learning algorithms on the same two data sets.[4] They also observed low prediction performance, and they didn't see dramatic improvements by using different techniques. Lan Guo and her colleagues had similar results, although they found that a random-forests technique produced better prediction results than other machine-learning algorithms.[5]

Related results

While considering possible explanations for these results, we observed that the NASA data sets considered the smallest unit of functionality—that is, a C function or a C++ method—to be a module. We also observed that the studies reporting successful defect prediction results considered larger code chunks as modules. For example, Khoshgoftaar and his colleagues accepted a set of source files as a module.[6] They measured 13 million lines of code for 7,000 modules, with an average module size of ap-

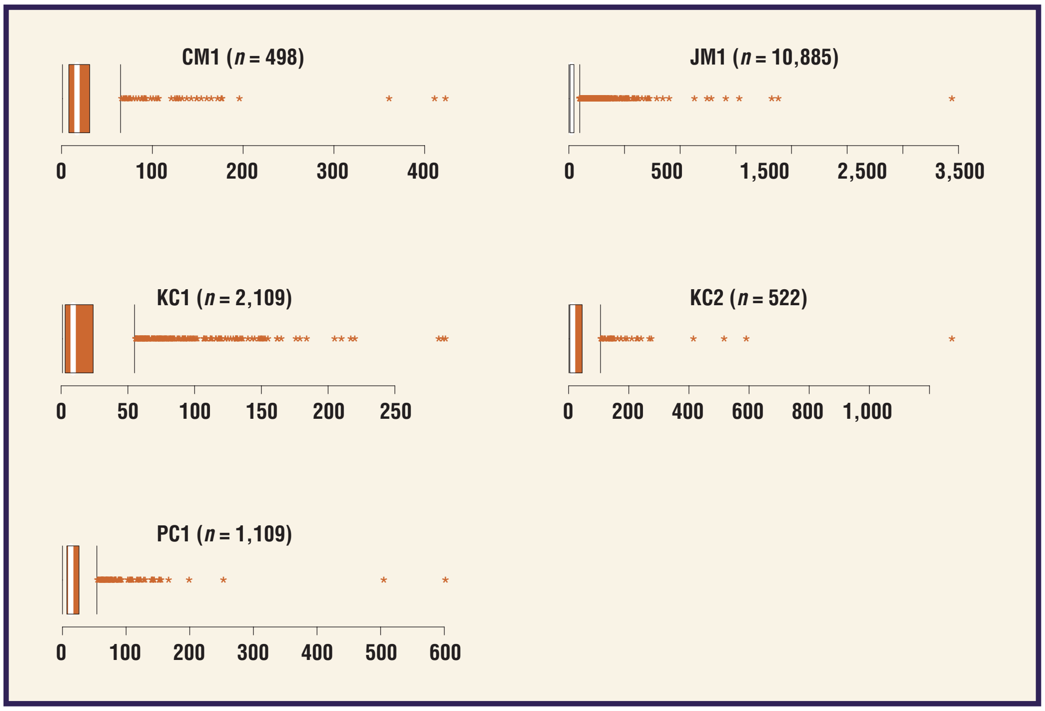


Figure 1. Box-whisker plots for module sizes in NASA products. The *x*-axis shows lines of code, and *n* is the number of data points. The white stripe indicates the median. The boxes extend to the first and third quartiles. Outliers that fall outside the whiskers appear as stars.

proximately 1,860 LOC. In a recent study, Thomas Ostrand and his colleagues accepted each file as a module having an average size of 250–300 LOC in the systems they studied.[7] Ramanath Subramanyam and Mayuram Krishnan took each class as a module with an average size of 163 LOC for C++ classes.[8]

These liberal interpretations of what constitutes a module can be expected given that the *IEEE Standards for Software Engineering Terminology* offers two definitions that allow such interpretation:[9]

- A program unit that is discrete and identifiable with respect to compiling, combining with other units, and loading.
- A logically separable part of a program.

However, these variations affect prediction results by determining the granularity level at which defect predictions are made. The predictions made using the original NASA data sets were at the finest granularity level among the studies cited earlier. Figure 1 shows the box-whisker plots for the module sizes in the NASA data sets.

Module size and defect prediction

If we use the average module size of 163 LOC reported by Subramanyam and Krishnan[8] as a threshold to characterize the NASA modules relatively as either small or large, we can characterize the great majority of our data sets' modules as small (see figure 1). Only a small percentage of modules can be characterized as large modules.

When small modules predominate in a data set, the machine-learning algorithm's effectiveness in finding defect patterns will be limited. For small modules, the static measures other than LOC will also be small because they usually depend on size. In other words, the characteristics of small modules will show little variation, which will make it difficult for a machine-learning algorithm to distinguish between small defective and nondefective modules. Because a large portion of the small modules is usually nondefective, most machine-learning algorithms will be biased toward predicting small modules as nondefective.

For large modules, the measurement values will have a better chance to show variation. However, the percentage of large modules in a data set will be insufficient to train a machine-learning model to distinguish between large defective and nondefective modules.

Aggregating the measurement and defect data when building prediction models offers a way to overcome these problems. This approach requires a system with enough data points to lead to statistically meaningful conclusions after aggregation. At the class level, instead of using only a binary (true/false) value to

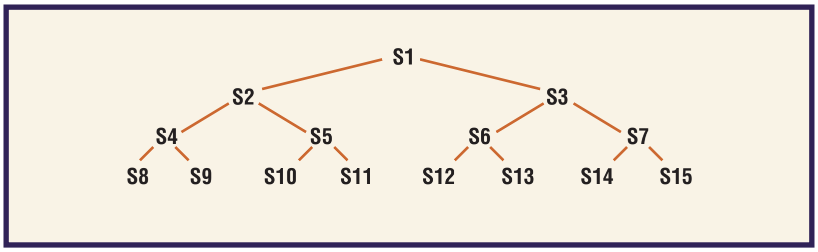


Figure 2. Naming scheme for data subsets.

indicate defectiveness, we can obtain a defect count and use a threshold value to label defect-prone classes. Then we can build prediction models to predict those defect-prone classes.

Predicting defects on data subsets

To show the relationship between module size and defect prediction performance, we stratified the data sets according to module size. For each original data set, we obtained 15 subsets by recursively partitioning in half after sorting the modules according to their size. We mixed the data points in each subset randomly to provide an unbiased learning scheme for the cross-validation runs. Figure 2 shows the naming scheme for these subsets.

First, we ranked the modules in S1, which represents the whole set, according to their size. We put the modules whose sizes were smaller than the median LOC value into S2 and modules whose sizes were larger than the median LOC value into S3. We repeated this process recursively. At any level, the leftmost subset included the smallest modules (for example, S8 in figure 2) and the rightmost subset included the largest modules (S15). We found it appropriate to stop at the third level where the subsets still included enough data points for machine learning.

We then ran several machine-learning algorithms on these subsets in an exploratory manner using 10-fold cross validation and observing the prediction performance (see the sidebar). Normally, the choice of machine-learning algorithm and model parameter adjustments affect prediction performance. Here, we report our results obtained using J48 learner, WEKA's implementation of the C4.5 algorithm.[10] J48 is a decision-tree learner, which Menzies and his colleagues used on the JM1 and KC2 data sets.[3] Decision-tree learners generate a simple tree structure where nonterminal nodes represent tests on one or more attributes and terminal nodes reflect decision outcomes. J48 has the useful feature of generating tree-based models that human experts can easily interpret.

We developed a set of Perl programs and shell scripts that used a comprehensive set of parameter-value combinations to call the WEKA class relevant to executing J48. As a result, we built several models with distinct parameter value combinations. Among those models, our programs recorded the ones with the best precision, recall, and F-measure. F-measure takes both precision and recall equally into account (see the sidebar).

Table 1 shows the results for each product. In general, at each level, the subsets that contain larger modules have higher F-measures than those that contain smaller modules. For example, in KC1, the F-measure was 0.4 for S1. After the first partitioning, the F-measure was 0.26 for S2 and 0.46 for S3. After the second partitioning, F-measure values were 0, 0.15, 0.40, and 0.49 for S4, S5, S6, and S7, respectively. After the third partitioning, again the subsets with larger modules had higher F-measures.

We observed the same behavior not only with J48 but also with the other algorithms that performed reasonably well, such as K-Star and Random Forests.[2]

Table 1

F-measure values obtained using J48 for each subset-product pair

Level	Subset	CM1	JM1	KC1	KC2	PC1
1	S1	0.28	0.33	0.40	0.57	0.26
2	S2	0	0.27	0.26	0	0.25
2	S3	0.15	0.36	0.46	0.56	0.37
3	S4	0	0.25	0	0	0.31
3	S5	0	0.25	0.15	0	0
3	S6	0.29	0.21	0.40	0.21	0.27
3	S7	0.16	0.43	0.49	0.66	0.36
4	S8	0	0.21	0	0	0
4	S9	0	0.27	0	0	0
4	S10	0	0.29	0	0	0
4	S11	0	0.25	0.21	0	0
4	S12	0.50	0.08	0.39	0.15	0.35
4	S13	0.18	0.24	0.37	0.17	0.27
4	S14	0.15	0.26	0.35	0.50	0.10
4	S15	0.22	0.52	0.61	0.76	0.42

Defect prediction using KC1 class-level data

While looking for additional data that belong to the same products but are collected at a higher abstraction level, we realized that the NASA Metrics Program Web site (http://mdp.ivv.nasa.gov) included both method- and class-level measures for KC1. We aggregated the available method-level measures to class level by calculating their minimum, maximum, sum, and average values. Complete names of the measures and more information appear in the preambles of the WEKA format arff files that we posted at the PROMISE Web site, and in the glossary posted at the NASA Metrics Program site (http://mdp.ivv.nasa.gov/mdp_glossary.html). The NASA Web site also includes a defect file that associates each defect entry with a method in KC1.

We obtained each class's defect count by adding the defect counts for its methods. We made the resulting data sets and the R function used to create them publicly available at the PROMISE Web site. (R is a free software environment for statistical computing and graphics; www.r-project.org.) KC1's class-level data included 145 data points. The median and average values for class size were 162 and 296.3 LOC, respectively.

KC1 analysis

Using the class-level data, we built two J48 models: one to predict the classes that had at least one defect (DF) and another to predict the top 5 percent of classes in the defect ranking (Top5%DF). The 145 data points yielded 60 DF classes and eight Top5%DF classes. (Since 5 percent of 145 yields slightly more than seven classes, we rounded up to eight for Top5%DF.)

We fed all static measures obtained at class level to the models to make binary predictions for each class: DF or non-DF and Top5%DF or non-Top5%DF. The specific WEKA program call was

```
weka.classifiers.trees.J48
   -C0.25 -M8
```

We set the minimum number of instances at any leaf node to eight and again used 10-fold cross validation. To obtain even more realistic estimates of prediction accuracy, we calculated the averages for precision, recall, and F-measure over 10 different runs, using different randomization seed values for cross-validation shuffling in each run. Table 2 shows the resulting precision, recall, and F-measure values.

Table 2

Values for all defective classes and top 5% defective classes

	Precision	Recall	F-Measure
DF	0.62	0.68	0.65
Top5%DF	0.50	0.63	0.56

KC1 results

When compared to the F-measure values for the S1 subset of KC1 (whole set) in table 2, the F-measure value was higher (DF = 0.65), indicating better prediction performance. However, you could still argue that predicting the defective modules isn't very difficult because 60 modules have at least one defect.

When we predicted the top 5 percent of classes in defect ranking, which was a much restricted set, the precision, recall, and F-measure values were 0.50, 0.63, and 0.56, respectively. Table 1 shows the F-measure values obtained for the S1 subset of KC1 to be around 0.4—less than the F-measure value we obtained by predicting the top defect classes. In fact, using class-level data, we correctly predicted five of the eight classes in Top5%DF.

Although we could obtain tree-based models with more levels and rules, the Top5%DF decision tree was one level defined by two rules:

```
sumNUM_UNIQUE_OPERATORS <= 285:
  Not in Top5%DF
sumNUM_UNIQUE_OPERATORS > 285:
  In Top5%DF
```

This simple tree-based model divides the whole set of KC1 classes into two parts using a cut-off value (285) of the aggregated measure (sum of the number of unique operators). The model predicts that classes with values equal to or smaller than the cut-off value will not be in Top5%DF, while classes with values greater than the cut-off will be. The first rule predicts that 135 classes will not be in Top5%DF; in fact, 132 of them were not. The second rule predicts that 10 classes will be in Top5%DF, and five were.

Defect prediction guidelines

Our results show the ineffectiveness of microlevel analysis for predicting defective software modules. We therefore advise against performing defect prediction at a fine granularity level, such as method level. Instead, practition-

Determining class-level defect counts requires consistent counting of the changes needed in each class to solve problems.

ers should collect measurement and defect data at a macro level, shifting the static measures and defect data to a higher abstraction level (similar to what we did by shifting from method level to class level for KC1). The following steps lead to effective defect prediction models in their development settings.

Obtain static measures

At this step, practitioners can use various tools to measure particular snapshots of their source code and to obtain method-level and class-level static measures. In our research, we often use the tool Understand C++/Java, available from Scientific Toolworks (www.scitools.com), which provides a broad array of static measures at both levels.

Aggregate measures

A simple way to aggregate method-level measures to class level is to calculate the sum over all the methods in a class. In some of our models, the aggregated measures turned out to be more related to defects than the original class-level measures. Therefore, you should merge the data set that includes the aggregated method-level measures with the one that includes original class-level measures. The machine-learning models will later take the merged data set as input.

Collect defect data

Determining class-level defect counts requires consistent counting of the changes needed in each class to solve a problem (for example, a failure) or combination of related problems. Each such change counts as one defect, which you should count from the time the measurements were performed. Using defect counts supports more versatile data analysis than using a binary variable to show defectiveness. For example, you can use a defect count threshold to label certain classes as high risk or, as in our analysis, to label a certain percentage of top-defect modules as risky. Alternatively, some prediction models, such as regression trees, can use defect counts directly.

Build a prediction model using WEKA

The open source WEKA machine-learning tool is available for download and installation from its project Web site (www.cs.waikato.ac.nz/~ml/weka). By collecting all class-level data in a spreadsheet, you can then feed it to WEKA as comma-separated values. WEKA's GUI makes it easy to build various machine-learning models.[2]

To estimate prediction performance for the models, we recommend using 10-fold cross validation over other methods such as holdout methods. Model parameters usually require adjustment to optimize or balance precision, recall, and F-measure values. The higher precision, the less wasted testing and inspection effort; and the higher recall, the fewer defects go undetected (see the sidebar).

Predict defect-prone classes

The created model can be used to make predictions about newly added classes or classes in the next release or in products developed within the same development environment. The models that include a set of rules with measures and their cutoff values—for example, the models produced by J48—are easy for practitioners to interpret and use in characterizing and predicting defect-prone classes.

Improve prediction models

Over time, defect patterns can change. As new measurement and defect data become available, you can include them in the data sets and rebuild the prediction model.

Normally, defect prediction models will change from one development environment to another according to specific defect patterns. For example, if programmers lack object-oriented programming skills, measures such as inheritance measures might turn out to be associated with defects. The products in the PROMISE repository were developed at different sites, and a model built at one site might not work well on another one. Within the same development environment, practitioners should build defect-prediction models as soon as defect and measurement data become available, then update them continuously.

Finally, we believe the PROMISE repository will continue to provide a common ground for discussions among software engineering researchers and practitioners. Over time, these efforts can lead to some generalizable defect-prediction models to serve as a starting point in development environments that have no historical data. Practitioners can then evolve those models to be highly effective in specific development environments.

Acknowledgments

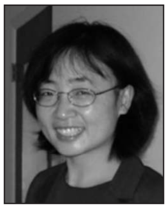
We thank the special issue editor, Bojan Cukic, and the anonymous reviewers for their useful and constructive comments. We thank Tim Menzies and Jelber S. Shirabad for their efforts in putting together the Promise Web site, data repository, and workshop. We also thank Carolyn Seaman and Chris Law for reading the earlier drafts of this article.

References

1. J.S. Shirabad and T.J. Menzies, "The Promise Repository of Software Engineering Databases," School of Information Technology and Engineering, University of Ottawa, Canada, 2005; http://promise.site.uottawa.ca/SERepository.
2. I.H. Witten and E. Frank, *Data Mining: Practical Machine Learning Tools with Java Implementations*, Morgan Kaufmann, 2000.
3. T. Menzies et al., "Mining Repositories to Assist in Project Planning and Resource Allocation," *Proc. 1st Workshop on Mining Software Repositories* (MSR 04), 2004, http://msr.uwaterloo.ca/papers/Menzies.pdf.
4. T.M. Khoshgoftaar and N. Seliya, "The Necessity of Assuring Quality in Software Measurement Data," *Proc. 10th Int'l Symp. Software Metrics* (Metrics 04), IEEE CS Press, 2004, pp. 119–130.
5. L. Guo et al., "Robust Prediction of Fault-Proneness by Random Forests," *Proc. 15th Int'l Symp. Software Reliability Eng.* (ISSRE 04), IEEE CS Press, 2004, pp. 417–428.
6. T.M. Khoshgoftaar et al., "Application of Neural Networks to Software Quality Modeling of a Very Large Telecommunications System," *IEEE Trans. on Neural Networks*, vol. 8, no. 4, 1997, pp. 902–909.
7. 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 Eng.*, vol. 31, no. 4, 2005, pp. 340–355.
8. R. Subramanyam and M. Krishnan, "Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects," *IEEE Trans. on Software Eng.*, vol. 29, no. 4, 2003, pp. 297–310.
9. *IEEE Std. 610.12-1990, IEEE Standard Glossary of Software Engineering Terminology*, IEEE Press, 1999, pp. 10–33.
10. J.R. Quinlan, *C4.5: Programs for Machine Learning*, Morgan Kaufmann, 1993.

About the Authors

A. Güneş Koru is an assistant professor in the Department of Information Systems at the University of Maryland, Baltimore County. His research interests include software quality, measurement, maintenance, and evolution. He received his PhD in computer science from Southern Methodist University. He is a member of the IEEE and ACM. Contact him at 1000 Hilltop Cir., Dept. of Information Systems, UMBC, Baltimore, MD, 21250; gkoru@umbc.edu.



Hongfang Liu is an assistant professor in the Department of Information Systems at the University of Maryland, Baltimore County. Her research interests include software engineering, machine learning, data mining, and biomedical text mining. She received her PhD in computer science at the Graduate School of City University of New York. Contact her at 1000 Hilltop Cir., Dept. of Information Systems, UMBC Baltimore, MD, 21250; hfliu@umbc.edu.