Using Statistical Models to Predict Software Regressions

Alexander Tarvo
Microsoft Corporation, Redmond, WA 98052
alexta@microsoft.com

Abstract

Incorrect changes made to the stable parts of a software system can cause failures – software regressions. Early detection of faulty code changes can be beneficial for the quality of a software system when these errors can be fixed before the system is released. In this paper, a statistical model for predicting software regressions is proposed. The model predicts risk of regression for a code change by using software metrics: type and size of the change, number of affected components, dependency metrics, developer's experience and code metrics of the affected components. Prediction results could be used to prioritize testing of changes: the higher is the risk of regression for the change, the more thorough testing it should receive.

1. Introduction

Despites all efforts of engineering community, bugs in software systems are still inevitable today. Probably, one of the most unpleasant classes of bugs is *software regressions* – undesired changes in the behavior of already stable parts and features of the software system. Software regressions can lead to significant problems for the software manufacturer, if are not detected and fixed early. If software regression is not detected during the testing, customers will be affected by undesirable side effects of the change. Such situation will result not only in a financial loss for a manufacturer (in any case, the issue needs to be fixed and update issued), but its reputation will also suffer.

The key method of avoiding negative consequences of software regressions is testing of all code changes. However, cost and time restrictions often prevent engineers from doing exhaustive testing of all changes, so some method of test prioritization is necessary. Probably, one of the most straightforward and widely used of such methods is assessment of the risk, associated with each code change, by an expert or group of experts. The higher is the risk of regression, associated with the change (which can be also called regression proneness of the change), the more thorough testing it should receive.

However¸ manual risk estimation is costly and subjective: it relies on skills and experience of experts. To address this problem, we developed a statistical model to predict risk of regressions. The model utilizes knowledge of the software system, represented in a form of software metrics, and provides an objective quantification of regression risk for each code change. Predicted risk is used by test engineers to plan testing activities for changes: high-risk changes should pass through an extensive testing to discover possible regressions while changes with low probability of regression can pass just sanity testing.

This paper presents an industrial case study, where we built the system to predict software regressions by using historical data on changes in Windows XP operating system. Analysis of system's accuracy shows it could be successfully used to predict software regressions.

2. Data collection

In our work we concentrate on post-release changes, or *fixes*, which are made after software system is released to the market. These changes include bug fixes, new features, reliability or performance improvements. Information on fixes is stored in the bug-tracking database in the form of *bug records*. If the bug record describes a fix for a software regression, a link to the bug which resulted in regression is provided in the record. This allows for identifying fixes which caused regressions – regressed fixes. Only regressions, caused by changes in a source code of the software system, are considered.

Since a fix typically results in a code change, the corresponding bug record can be related to the set of changes in the program source code. These changes are grouped in one or more *check-ins* – atomic changes of a source code, recorded in a version control system. Each check-in contains a list of changed source files, differences between old and new versions of these files, date of the change, name of a developer and a brief description of the change. By identifying check-ins,

1071-9458/08 $25.00 © 2008 IEEE
DOI 10.1109/ISSRE.2008.21

IEEE computer society

related to the fix, it is possible to detect components of the software system, affected by it.

Three major types of metrics are used to describe code changes in this study: these are (1) change metrics, (2) code metrics and (3) dependency metrics.

2.1. Change metrics

Change metrics describe properties of a software change itself: size of the change, number of changed components, experience of the developers and other properties. Some major classes of change metrics are described below.

Number of changed components is one of the most important properties of the fix: complex fixes that cause changes in a large number of components are expected to be more regression prone than small fixes. By analyzing source and binary code of the software system, following metrics were defined:

- ***AddedFunctionsCount:*** number of functions, added because of the fix;
- ***DeletedFunctionsCount:*** number of functions, deleted because of the fix;
- ***ChangedFunctionsCount:*** number of functions, changed because of the fix;
- ***AddedSourceFilesCount:*** number of source files, added because of the fix;
- ***DeletedSourceFilesCount:*** number of source files, deleted because of the fix;
- ***ChangedSourceFilesCount:*** number of source files, changed because of the fix;
- ***BinariesAffected:*** number of binary modules affected by the fix.

Another group of metrics is related to the experience of developer, who performed the change – we call these metrics *experience metrics*. It can be possible that change done by an experienced person can be of a higher quality than that of an inexperienced programmer [9]. One possible method to estimate developer's experience is looking at all fixes done by this person in past: the more fixes the developer worked on in past, the higher is his/her experience. Developer is considered experienced if he/she did more check-ins than 75% of other developers during the same period. Correspondingly, inexperienced developer did less than 25% of fixes in comparison to other developers.

To measure overall programming experience of the developer we defined global experience metrics, based on the number of changes he/she made into the whole software system during the past 12 months:

- ***CheckinsLastYear:*** overall number of check-ins by this developer;
- ***ExperiencedDeveloper:*** 1, if developer did 15 or more check-ins; 0 otherwise;
- ***InxperiencedDeveloper***: 1, if developer did 2 or less check-ins; 0 otherwise;

However, not only overall experience of developer matters, but his/her knowledge of the particular area of source code should be considered as well. To measure developer's knowledge of the affected area, the number of changes he/she had made in this part of the system during previous 12 months was counted:

- ***CheckinsLastYearInComponent***: number of check-ins developer did in the source code of the component, that is affected by a fix;
- ***ExperiencedDeveloperInComponent***: 1, if developer did 7 or more check-ins in the source code of the affected component, 0 otherwise;
- ***InxperiencedDeveloperInComponent***: 1, if developer did 1 or less check-in in the source code of the affected component, 0 otherwise.

The last group of fix metrics is fix characteristics, which describe its nature and the overall fix process:

- ***FixForRegression***: 1, if this is a fix for a known regression, 0 otherwise;
- ***IsNewFeature***: 1, if this fix is a new feature, 0 if it is a bug fix;
- ***CheckinCount***: number of check-ins required to make the change;
- ***DeveloperCount***: number of developers, working on the change;
- ***BugLinesDelta***: summary change in size (LOC) of affected functions.

2.2. Code metrics

It has been shown that fault proneness of the component can be successfully predicted by using its code metrics: size, complexity and historical code churn are positively correlated with the number of failures spotted in the component [3]. Since regressions are actually consequences of failures, we assume that complex components with high historical code churn might have a higher number of regressions as well. Also, making a fix in a large and complex component is a complicated task for a developer and can increase chances of making a mistake, so code metrics were included in the set of predictor variables. Three major classes of code metrics were used in this study:

- *Complexity metrics* describe internal complexity of the component. Examples of complexity metrics are component size or number of global parameters in it;
- *Object-oriented metrics* describe complexity of components developed with using of object-oriented methodology. Examples of object-oriented metrics are number of classes in the module, size of the class hierarchy, number of methods in the class;
- *Code churn* describe the history of changes in the component. Examples of churn metrics are number of

changed code lines or functions in the component, as well as number and properties of bugs fixed in it.

Complexity and object-oriented metrics were collected for each binary module and function using MaX framework [10]. MaX is an automated tool that can collect code metrics at binary module and function level. Churn metrics were collected for each binary module, source file and function using custom-developed tool called Binary Change Tracer (BCT).

2.3. Dependency metrics

Components in a software system do not exist in isolation: they interact in a number of ways. For example, an application can load a dynamically linked library and call functions or access data structures located in it. Obviously, if the library is not present in the system, application will not work properly, so it can be said that application *depends* on the library.

Dependencies between all components of software system form a dependency graph – a directed graph $G=\{C, D\}$, where components form a set of vertices $C=\{c_1,\dots,c_N\}$ and dependencies form set of edges $D=\{d_1,\dots,d_M\}$. Number of dependencies M can be as high as $M = N \cdot N$. If component c_i depends on another components c_j, there exists an edge in the graph $d_{ij}=(c_i, c_j) \in D$. For component c_i this dependency is called an *outgoing* dependency, for component c_j it is an *incoming* dependency.

It has been shown [1] that data and call dependencies can be useful predictors of component fault proneness. In this study dependency metrics are used as predictors of the fix regression proneness.

Dependency metrics for a code change were defined by using information about changed components and a structure of the call graph. Suppose that fix affects a subset of components $C_c \in C$. Then dependencies, linking these components to any other components in software system, are constituting a set of affected dependencies $D_c \in D$. Affected dependency $d = (c_i, c_j)$ is a part of set D_c if $c_i \in C_c$ or $c_j \in C_c$.

We distinguish two basic types of affected dependencies (see Figure 1):

- ***External dependency*** $d_{ext} = (c_i, c_k)$ is a dependency between a changed component $c_i \in C_c$ and non-changed component $c_k \notin C_c$
- ***Internal dependency*** $d_{int} = (c_i, c_j)$ is a dependency between two changed components $c_i, c_j \in C_c$

In this study dependency data was collected at function and binary levels. If data is collected at function level, a set of changed functions $F_c=\{f_1,\dots,f_K\}$ is defined. Correspondingly, at the binary level, a set of changed binaries $B_c=\{b_1,\dots,b_L\}$ is defined. It allows us to define four dependency metrics for each fix:

- ***BinaryExternalDependenciesAffected:*** total number of external dependencies for binary modules B_c, affected by the fix;
- ***BinaryInternalDependenciesAffected:*** total number of internal dependencies between binary modules B_c, affected by the fix.

Similarly, ***FunctionExternalDependenciesAffected*** and ***FunctionInternalDependenciesAffected*** metrics were defined for the functions F_c, affected by the fix.

Number of dependencies for each affected function was also considered, resulting in following metrics:

- ***IncomingDependenciesCount*** – number of incoming dependencies of the function;
- ***OutgoingDependenciesCount*** – number of outgoing dependencies of the function;

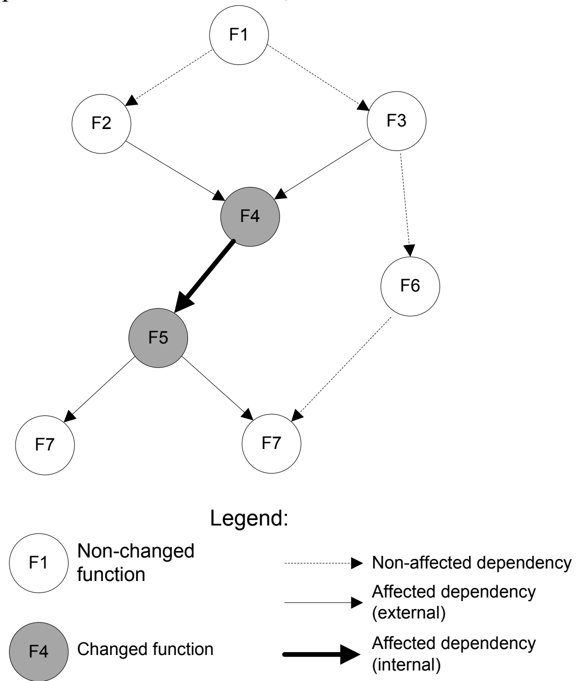

Figure 1. Dependency metrics

MaX framework was used to extract dependency information for software system. In addition to the code metrics, MaX can also collect call and data dependency information for each function in the software system.

3. Measuring accuracy

Model, developed during this study, can be viewed as a binary classifier: it is classifying fixes as regression prone and non-regression prone. There are four possible outcomes of classifications:

- ***True Positive (TP)***: fix is regression prone and was classified as regression-prone;

- ***False Positive (FP)***: fix is not regression prone, but was classified as regression-prone;
- ***True Negative (TN)***: fix is not regression prone and was classified as not regression prone;
- ***False Negative (FN)***: fix is regression prone, but was classified as not regression prone.

Based on these outcomes, a number of metrics to measure classification performance have been developed [11, 12]:

$$Precision = \frac{TP}{TP + FP}$$

$$Recall = True\ Positive\ Rate = \frac{TP}{TP + FN}$$

$$False\ Positive\ Rate = \frac{FP}{FP + TN}$$

However, many statistical methods, such as logistic regression, do not directly specify if a fix is regression prone or not. Instead, they produce a number, representing a probability of regression for a fix. To convert this number into an actual class label it is necessary to define a threshold. Fix is considered as regression prone if output of the classifier is higher than the threshold and as non-regression prone, if output is below the threshold. To measure change of classifier performance depending on the threshold, we used a technique called Receiver Operating Characteristic (ROC) graphs.

ROC graph is a two-dimensional graph, where true positive rate (recall) is plotted on the Y axis and false positive rate is plotted on the X axis [11]. ROC curve for an ideal classifier is a straight line from (0,1) to (1,1). Line from (0,0) to (1,1) implies a worst possible classifier that is no better than a random guessing. Area under ROC curve (AUC) can serve as a single number to measure classifier's performance. It can vary from 0.5 for the worst classifier to 1.0 for the best possible one.

4. Model building

Subject of this study is Microsoft Windows XP - an operating system from Microsoft Corporation. It is a large software system composed of several millions lines of code which constitute thousands of binary modules. At the moment of writing this paper Windows XP has undergone two major maintenance releases (service packs) and numerous smaller fixes. To train the model we scanned the whole code of the system and selected data on a large number of bug records (fixes) made after Windows XP SP2 was released (August 2004). Only unique bug records (ones that point to different sets of check-ins) were included into the dataset. To make sure that all regressions in selected fixes were revealed, no data was collected during the last 18 months of the study. Nevertheless, dataset appeared to be sparse: only a small fraction of these fixes caused software regressions.

A number of metrics were collected for each fix: these metrics include change metrics, dependency metrics and code metrics. Code metrics were collected for each Windows component, affected by the fix. These include complexity metrics, object-oriented metrics and pre-release code churn. Pre-release code churn for each component was collected during Windows XP SP2 development (from September 2003 to August 2004). Three different levels of granularity were used to collect code metrics: code churn and bug data were collected at the level of binary modules, source files and functions; while complexity and object-oriented metrics were collected at function and binary module levels only.

A classic table-based approach was used to build the model. For each fix, a vector $\bar{x}$ of independent variables (predictors) and a dependent variable y (response) were defined. Independent variables are fix metrics, which describe a change. Dependent variable is an occurrence of regression for a certain fix.

Stepwise logistic regression [5] was used to build a statistical model. As many other statistical methods, like decision trees or neural networks, logistic regression require dimension of the vector $\bar{x}$ be a constant across all data points. However, a single fix can lead to changes in multiple components: it is not uncommon for a large complex fix to affect a number of source files or even binary modules. Thus if code metrics for all affected variables are included into the vector $\bar{x}$, its size will vary. To make dimensions of the $\bar{x}$ vector constant, we aggregated values of code metrics for all components, affected by the single fix [12].

Suppose the fix affects components $(c_1,c_2,\ldots,c_n)$, and code metrics $(m_1(c_j),\ldots,m_k(c_j))$ are defined for any component $c_j \in (c_1,c_2,\ldots,c_n)$. In this case, an aggregated value $\widehat{m_1}$ of metric m_1 across components $(c_1,c_2,\ldots,c_n)$ is

$$\widehat{m_1}\ (c_1,c_2,\ldots,c_n) = f(m_1(c_1),m_1(c_2),\ldots,m_1(c_n)),$$

where $f(m_1(c_1),m_1(c_2),\ldots,m_1(c_n))$ is an aggregating function.

In this study $max(m_1(c_1),m_1(c_2),\ldots,m_1(c_n))$ and $median(m_1(c_1),m_1(c_2),\ldots,m_1(c_n))$ functions were used to aggregate values of metrics $m_1(c_1),m_1(c_2),\ldots,m_1(c_n)$.

To measure a predictive power of the model, data splitting technique [12] was used. 50 random splits were done; during each split, 70% of fixes were selected to form a training set and the remaining 30% formed a test set. Averaged ROC graph were created as it was discussed in [11].

To evaluate relative importance of different groups of metrics, four models were built separately for dependency metrics, experience metrics, fix metrics and code metrics. For each model, 50 random splits were done, and, during each random split, ROC curve was built. Based on these 50 values of AUC, mean value of area under the curve

μ(AUC), as well as its standard deviation σ(AUC) were calculated. Results are reported in the Table 1.

Table 1. Model performance for the different groups of metrics

Metric group	μ(AUC)	σ(AUC)
Fix metrics (no experience)	0.73	0.046
Code metrics	0.70	0.040
Dependency metrics	0.69	0.049
Experience metrics	0.54	0.044

To see if these models provide statistically different values of AUC, author performed a series of unpaired t-tests to compare resulting AUCs. According to these tests, fix metrics appeared to be better predictors of the regression proneness for a fix (p-value $p<0.001$) than code or dependency metrics. No statistically significant difference ($p=0.56$) was found between the accuracy of the models, based on dependency and code metrics.

Surprisingly, experience metrics didn't prove to be informative ones ($p<0.001$). One of possible explanations is that the most complex and risky fixes are done by the most experienced programmers, and simple fixes are left to novice engineers. Anyway, this phenomenon might be left as an opportunity for a future investigation.

To build the final model, all metrics were used. Stepwise logistic regression reported five most significant predictors (see Table 2):

- ***SourceFilesChanged***: number of source files changed due to the fix;
- ***NewFeature***: 1, if this fix is a new feature, 0 otherwise;
- ***PreReleaseFunctionsDeleted***: number of functions deleted from the binary module during the pre-release timeframe;
- ***MaxFunctionLocalCoupling***: maximum value of FunctionLocalCoupling metric across all functions in the binary module. FunctionLocalCoupling is the number of calls to other classes, whose instances are created as local variables in the function;
- ***MaxSubClasses***: Maximum number of sub classes across all high-level classes in the binary module;

FunctionsDeleted, ***MaxFunctionLocalCoupling*** and ***MaxSubClasses*** are code metrics, defined for binary modules. That's why median values of these metrics for all binaries, affected by the fix, were taken as predictors.

Table 2. Most significant predictors

Metric name	Type	p-value
SourceFilesChanged	Fix	0.001
NewFeature	Fix	0.03
Median(FunctionsDeleted)	Code	0.02
Median(MaxFunctionLocalCoupling)	Code	0.002
Median(MaxSubClasses)	Code	<0.001

Resulting ROC graph for the final model is shown in Figure 2. Its mean area under ROC curve is **μ(AUC) = 0.77** and standard deviation **σ(AUC)=0.040**, which is significantly better than random draw (AUC=0.5). What is more important, the model can detect the most risky fixes with sufficient accuracy, which allows to intensify testing activities for them and so enable regressions to be discovered in these fixes early.

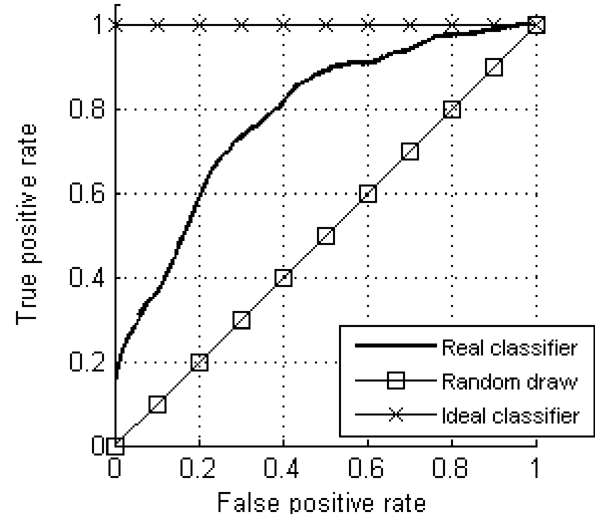

Figure 2. ROC and precision-recall graphs for the model

In addition to logistic regression, author evaluated an accuracy of regression prediction models, based on other machine learning algorithms, available in MATLAB (see Table 3). Unpaired t-tests shown that logistic regression model has somewhat higher accuracy than multilayer perceptron with Principal Component Analysis ($p=0.029$), while regression tree has the worst accuracy ($p<0.001$).

Table 3. Comparative performance of different machine learning algorithms

Machine learning algorithm	μ(AUC)	σ(AUC)
Logistic regression	0.77	0.040
Multilayer perceptron + PCA	0.75	0.058
CART tree	0.67	0.069

5. Related work

Using statistical models to predict software risk is a widely used technique. Almost all studies are concentrated on predicting fault proneness [1, 2, 3, 6, 7] of components in the software system. All of these works are using various types of code metrics to predict fault proneness of the components: one kind of such metrics is code churn, which is defined as an amount of changes in

the software system [3]. Another class of metrics is complexity metrics, like code complexity, size or number of functions in the component [7]. For object-oriented programs special types of object-oriented metrics can be defined [4].

Different statistical methods were used to build models, including decision trees, neural networks and logistic regression [1, 7, 8]. Many works recommend using PCA [2, 10] to reduce dimensionality of input data and eliminate multicollinearity between various metrics.

Prediction of software regressions appears to be a less explored area. By the time of writing this paper, only the work of A. Mockus and D. Weiss on predicting risk of software changes [9] was known to the author. This work shows a possibility to successfully predict risk of software change. Fix metrics, such as fix size, experience of a developer, number of affected subsystems were used to predict risk of software changes.

Presented work extends state of the art by considering additional metrics for regression prediction like pre-release code churn, dependency, code complexity and object-oriented metrics. Relative importance of different groups of metrics is also evaluated and best set of metrics is selected for building the model.

6. Experience and lessons learned

In a presented paper we have shown that risk of regression can be successfully predicted for a code change and pointed to metrics which are good predictors of regression risk. As a result of this study, a practical system for regression prediction has been developed and deployed in the Windows Serviceability team.

Once a development of the new fix is done, the system automatically analyzes changes, caused by the fix: it extracts fix metrics and calculates risk of regression for that fix. In addition to that, the system also conducts change impact analysis for the fix. Resulting report is presented to the test engineer: the report contains both change impact information as well as a predicted risk of regression. These two pieces of knowledge complement each other: change impact information tells the engineer which Windows components might be impacted because of dependencies on changed components, and which tests should be run to verify changed code. At the same time, regression risk gives a hint how much testing should be done: it is recommended that risky fixes with high probability of regression should receive more testing, than low-risk fixes.

The model has been deployed in March 2008 and quickly gained popularity among test engineers: based on usage logs we estimate that at least 70% of all test engineers in the Windows Serviceability team are using regression risk reports in their work.

To improve model's accuracy, we plan to collect more data on different versions of Windows, and introduce more metrics, such as presence of the code review for the fix. Also, in addition to logistic regression, we are going to experiment with different machine learning methods, such as Naïve Bayes and Support Vector Machines. To see if metrics, selected by the model, can be used as risk predictors for different types of software projects, we plan to evaluate our risk prediction model on other Microsoft products, such as SQL Server and Office.

7. References

1. T. Zimmermann, N. Nagappan, "Predicting Subsystem Failures Using Dependency Graph Complexities", *Proceedings of the 18th IEEE International Symposium on Software Reliability*, 2007, pp. 227-236.
2. N. Nagappan, T. Ball, A. Zeller, "Mining metrics to Predict Component Failures", *Proceedings of the 28th international conference on Software engineering*, 2006, pp. 452-461.
3. N. Nagappan, T. Ball, "Use of Relative Code Churn Measures to Predict System Defect Density", *Proceedings of the 27th international conference on Software engineering*, 2005, pp. 284-292.
4. E. Arisholm, L. C. Briand , "Predicting Fault-prone Components in a Java Legacy System", *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, 2006, pp. 8-17.
5. Larose D. T., "*Data Mining Methods and Models*", Wiley-Interscience, Hoboken, NJ , 2006.
6. T.M. Khoshgoftaar et al.,"Early quality prediction: a case study in telecommunications", *IEEE Software* Vol. 13 No 1, 1996, pp. 65-71.
7. T. Menzies, J. Greenwald, A. Frank, "Data Mining Static Code Attributes to Learn Defect Predictors," *IEEE Transactions on Software Engineering*, vol. 33, no. 1, pp. 2-13, Jan., 2007
8. E. Arisholm, L.C. Briand, M. Fuglerud, "Data Mining Techniques for Building Fault-proneness Models in Telecom Java Software", *18th IEEE International Symposium on Software Reliability*, 2007, pp. 215-224.
9. A. Mockus, D. Weiss, "Predicting risk of software changes", *Bell Labs Tech Journal*, Vol. 5, no. 2, 2000, pp. 169-180.
10. A. Srivastava, Thiagarajan. J., Schertz, C., "Efficient Integration Testing using Dependency Analysis," *Microsoft Research Technical Report*, MSR-TR-2005-94, 2005.
11. T. Fawcett, "An Introduction to ROC analysis", *Pattern Recognition Letters*, 26, 2006, pp. 861–874
12. Hand D.J., Mannila H., Smyth P., "*Principles of Data Mining*", The MIT Press, 2001.