Software Qual J (2008) 16:543–562
DOI 10.1007/s11219-008-9053-8

A defect prediction method for software versioning

Yomi Kastro · Ayşe Basar Bener

Published online: 1 May 2008

Abstract New methodologies and tools have gradually made the life cycle for software development more human-independent. Much of the research in this field focuses on defect reduction, defect identification and defect prediction. Defect prediction is a relatively new research area that involves using various methods from artificial intelligence to data mining. Identifying and locating defects in software projects is a difficult task. Measuring software in a continuous and disciplined manner provides many advantages such as the accurate estimation of project costs and schedules as well as improving product and process qualities. This study aims to propose a model to predict the number of defects in the new version of a software product with respect to the previous stable version. The new version may contain changes related to a new feature or a modification in the algorithm or bug fixes. Our proposed model aims to predict the new defects introduced into the new version by analyzing the types of changes in an objective and formal manner as well as considering the lines of code (LOC) change. Defect predictors are helpful tools for both project managers and developers. Accurate predictors may help reducing test times and guide developers towards implementing higher quality codes. Our proposed model can aid software engineers in determining the stability of software before it goes on production. Furthermore, such a model may provide useful insight for understanding the effects of a feature, bug fix or change in the process of defect detection.

Keywords Software defects · Defect prediction · Neural networks

Y. Kastro (✉) · A. B. Bener
Department of Computer Engineering, Boğaziçi University, 34342 Bebek, Istanbul 34342, Turkey
e-mail: kastro@boun.edu.tr

A. B. Bener
e-mail: bener@boun.edu.tr

1 Introduction

1.1 Motivation

Although there are various formal methods and tools to improve the automation of a software delivery, the major part of software development is a human-driven process that requires intensive labor. Hence, errors are mostly introduced as a result of the human involvement in the process. There are several methods to make the software development life cycle independent of human abilities. These methods are based on well-defined processes (Fenton and Neil 1999; Ceylan et al. 2006; Boehm et al. 1995; Boehm and Basili 2000). Many research efforts are in the field of software quality (Fenton and Neil 1999; Ceylan et al. 2006; Porter and Votta 2004; Brilliant et al. 1990). These efforts include the formation of formal methods, offering new methodologies like CMM or the introduction of new tools to increase quality.

Software quality is directly correlated with the number of defects in the software. Thus, defect prediction is a significant part of the literature on software quality. Minimizing the number of defects requires a thorough testing of the software in question. On the other hand, the testing phase requires approximately 50% of the entire project schedule (Song et al. 2006; Fenton and Ohlsson 2000). This implies that testing is the most expensive, time and resource consuming phase of the software development life cycle. An effective test strategy should, therefore, consider minimizing the number of defects while using resources efficiently. Defect prediction models are helpful tools for software testing. Accurate estimates of defective modules may yield decreases in testing times, and project managers may benefit from defect predictors in terms of allocating the limited resources effectively (Tahat et al. 2001). In this research, we were motivated to reduce both testing efforts and the cost of software production in order to achieve a high quality software.

1.2 Background

Software defect prediction has been evaluated as both the prediction and the estimation of the defects in a given software module (Fenton and Neil 1999). Generally, many efforts are specifically focused on predicting the number of defects in the system, estimating the reliability of the systems as statistical functions to time-to-failure, and understanding the importance of design and testing processes on defect counts. Software metrics are attributes of software that help us understand various characteristics of the software. Software metrics are mostly used with the aim of ensuring product quality, process efficiency and risk assessment in software projects (Menzies et al. 2002). One of the most significant benefits of software metrics is that they provide information for fault prediction. Currently, there are numerous metrics that can be used by project managers to assess software risks. The early research on software metrics have focused their attention mostly on McCabe, Halstead and Lines of Code (LOC) metrics (Fenton and Neil 1999; Menzies et al. 2007).

Previous research state that trying to detect defects using only size metrics is not accurate (Padberg et al. 2004; Inoue et al. 2005). The number of defects discovered is related to the amount of testing performed. Therefore, the defect count should be interpreted within the context of the testing effort. Otherwise, the defect count itself would lead us to the wrong conclusion. For example, if we take a complex program that is not tested or used, the defect count of that program will be zero. In this case, the defect count does not tell us anything about the quality of the software since testing was not performed at all.

When basic types of metrics such as LOC, Halstead and McCabe were used in the learning process, the researchers found out that some metrics did not give similar prediction results at different stages of the software development process. In their research, Fenton and Neil explored the relationship between the number of faults and the Cyclomatic Complexity values and discovered that the values were different in pre- and post-releases. Therefore, to overcome such problems, the researchers used Bayesian Belief Network for defect modeling (Fenton and Neil 1999). Although some research stood against using static code measures (Fenton and Neil 1999; Sheppard and Ince 1994), a recent research showed that using a Naïve Bayes classifier with log-filtered static code measures yields significantly better results than rule-based methods such as the decision tree (Menzies et al. 2007). However, these models do not address the problem of predicting defects in the process of multiversion software development.

Boehm and Basili, found out through their research that finding and fixing a software problem after delivery was often 100 times more expensive than finding and fixing it during the requirement and design phase (Boehm and Basili 2001). They also claimed that about 80% of avoidable rework comes from 20% of the defects. Additionally, about 80% of the defects come from 20% of the modules, and about half the modules are defect-free. Another research focused on N-version software experiment was also conducted (Brilliant et al. 1990). Within the scope of this research, various universities used the same requirements to code a program 27 times. They tried to prove whether the defects were statistically independent. They found out that in some cases, the programmers made similar logical errors, indicating that some parts of the problem were simply more difficult than others. Finally, they concluded that minor differences in the software development environment, such as the use of different programming languages for different versions would not have a major impact on reducing the incidence of faults that caused correlated failures (Ostrand et al. 2005).

Machine-learning algorithms have been proven to be practical in poorly-understood problem domains with changing conditions (Zhang 2000). Software quality problems can be formulated similar to learning processes, and they can be classified according to the characteristics of defect. Therefore, it would be possible to apply regular machine learning algorithms to come up with a probability distribution for defect analysis (Fenton and Ohlsson 2000). Decision trees, artificial neural networks, Bayesian belief network, and classification techniques such as k-nearest neighbor are examples of the most commonly used techniques for software defect prediction problems (Zhang 2000; Mitchell 1997; Jensen 1996; Khoshgoftaar and Allen 1999).

Padberg et al. proposed a system that learns from empirical data (Padberg et al. 2004). They specifically examined the relationship between certain observable features of an inspection as well as the number of defects actually contained in the documents. They showed that some features could carry significant nonlinear information about the content of the defect. Therefore, they used a nonlinear regression technique and neural networks to solve the learning problem.

Machine learning is also used to generate models of program properties that are known to cause errors. Support vector machines and decision tree learning tools are implemented to classify and investigate the most relevant subsets of program properties. The underlying intuition is that most of the properties leading to faulty conditions can be classified within a few groups. The technique Brun and Ernst used consists of two steps: training and classification. Fault-relevant properties are utilized to generate a model, and this pre-computed function selects the properties that are most likely to cause errors and defects in the software (Brun and Ernst 2004).

In their research, Podgurski et al. used clustering over function call profiles to determine which features enabled a model to distinguish failures from non-failures (Podgurski et al. 2003). The researchers used a technique called dynamic invariant detection to pick possible invariants from a test suite and investigated violations that usually indicate an erroneous state. This method is also used to determine counter examples and to find properties that lead to correct results for all conditions (Groce and Visser 2003).

Previous research usually has dealt with the characteristics of the detectors or estimators in the defect prediction problem domains. Koru and Liu argue in their research that, besides the characteristics of the detector itself, another important property of successful models is how the input data is handled. More specifically; they claim that organizing the input data set at fine granularity level (i.e. class level abstraction against method level) will bring forth better results in defect prediction (Koru and Liu 2005).

Linear regression analysis for defect prediction treats software metrics as independent variables in order to estimate the dependent variable i.e. defect density. Munson and Khoshgoftaar investigated linear regression models and discriminant analysis to conclude that the performance of the latter is more favorable (Munson and Khoshgoftaar 1990). They used Principal Component Analysis (PCA) as a pre-processing step in order to eliminate the co-linearity in software metrics. Nagappan et al. also used linear regression analysis with PCA for the STREW metric suite (Nagappan et al. 2005). Decision tree learning is another common method that is preferred for its rule generation capabilities (Menzies et al. 2003, 2004). Such rules are easier to explain to non-technical people (Fenton and Neil 1999). There is another research that has used machine learning methods in defect identification and in estimating defect density (Ceylan et al. 2006). Ceylan et al. constructed a two-step model that predicts potentially faulty modules. They carried out their experiments with different software metric datasets, which they obtained from real-life projects. The results of the experiments show that the two-step model enhances the regression performance.

So far, many researchers have dealt with the problem of tracking change in software as a way of predicting defect density. Kung et al. examined change impact identification within the scope of object-oriented software development (Kung et al. 1994; Inoue et al. 2005). They described a formal model to capture changes and to make inferences on the changes in order to identify the affected classes. Their model includes object relation diagram, the block branch diagram and the object state diagram. They primarily focused on C++ projects. They claimed that the changed and affected classes can be tested in a cost-effective order to avoid extensive construction of test stubs.

The previous research in defect prediction has so far not taken up the problem in a multiversion software environment. Some research on tracking changes in multiversion software, on the other hand, was conducted as specific to a single programming language. We have proposed a software defect prediction model independent of language and platform where the primary focus is on the multiversion software. We mainly observe the changes between versions to propose a defect density metric in order to decide on a new stable version.

The remainder of the paper is organized as follows: The second section discusses the problem. The third section explains our proposed defect prediction framework for a multiversion software as well as the data collection and analysis process. The fourth section states the experimental design and the data sets used in the research, discusses the threats to validity and presents results for the experiments. The fifth section concludes our work and discusses the future directions.

2 The problem statement

Software can be classified into three main groups according to their defect tolerances: zero-defect (i.e. space applications), low-defect (i.e. operating systems) and high-defect (i.e. non-critical user applications) (Nagappan and Ball 2005a, b; Jorgensen 2005). In zero-defect density scenarios, the defected software cannot be used or proposed as a product. So the software development company is fully focused on finding all defects and fixing them. In the case of operating system software or non-critical user applications, the software is designated to be defected below a certain defect density that is set depending on the application. At this point, a relation is put along with the defect density and time-to-failure. Using this relation, the software development institution can target a defect density level and arrange a testing and correction phase to reach that specific defect level. In most cases, formal review methodologies for requirements and design are executed as well as the testing process of the software (Bowen and Hinchey 1995; Clarke and Wing 1996; Coppit et al. 2005). Testing effort is a major part of the overall project effort and the related cost. There are various studies on how to distribute this effort in order to have a high quality software (Nagappan and Ball 2005a, b; Jorgensen 2005).

A defect prediction solution would help the software development institution to distribute their testing resources in line with the defect density. We can argue that the modules with a higher defect density should have a higher share than regular distribution, whereas the modules with a lower defect density should have a lower share than the regular. A version-based defect density solution may also take changes into consideration. If the software institution supplies the change information quantitatively, the defect prediction solution can be predicted in relation to these changes. In this case, the regression testing effort may also be reduced.

Another important aspect of a defect prediction solution is that such a solution becomes necessary when there is a trade-off between an earlier delivery and a delivery with fewer defects. In today's software development industry, all companies and software development houses are in a severe competition, such that minimizing the development time decreases the overall project cost (Nagappan and Ball 2005a, b; Johnson et al. 2005). On the other hand, less development and testing time also increases the defect density ratio in terms of the final product. Therefore, a defect prediction solution may provide the required quantitative metric to make a decision related to the delivery of the product. The senior management of the software development company would be able to make a decision concerning the product launch if the defect density level is below a certain threshold.

2.1 Version change tracking

Software products are developing and evolving (Gregoriades and Sutcliffe 2005; Vaidyanathan and Trivedi 2005). The same product with the same name with some features added or new functions introduced is announced as a new version. A new version might differ from an older version in various ways. The most common differences are newly introduced features originating from new requirements, changes of previous modules or the bug-correction of the older version. Although most of the software houses claim that the stable version of their software does not contain bugs, many fixes can be perceived in their new versions (Clarke et al. 1996).

It is highly possible that a newer version of software will differ from the older version also in terms of LOC and the logs of Code Versioning Systems (CVS). So, the software

development institution usually employs quantitative data to measure the alterations between different versions (Jorgensen 2005). More importantly, every software product has its own developing process. Programming language, programmers' experience, algorithm complexity, or even the development methodology makes software unique. Thus, every software product should be evaluated in accordance with its own dynamics in mind. If the software development institution is able to track the bugs of the previously published version and the changes between them, these data can be analyzed through the use of a defect prediction model. The performance of a defect predictor would highly depend on the historical data. Therefore, a brand new software may not produce meaningful predictions.

2.2 Defining and tracking change

Change is the core input of any prediction algorithm or method. We would like to give a definition of change in our software versioning scope. We propose that "a change exists" when a line from the source code is modified, deleted or moved. In this scope, change can be easily analyzed by a CVS program. If the software versions are kept in a software CVS repository, then two versions can be compared easily in terms of line changes. The reason we make a clear definition of change is that it needs to be objective and it should not differ from developer to developer.

On the other hand, we can say "a change exists", if the developer of any module is able to specify the minimum modification that produces a feature or bug fix. Thus, finding the lowest level of change is human-dependent, meaning that the list of all versions' change should be prepared by the same team or person in order to overcome subjectivity. Changes should be classified into groups of features, bug fixes and other changes. At the end of change collection procedure, we should come up with either of the following for each and every module:

i. The number of lines of source code changes
ii. The number of lines of source code insertions
iii. The number of lines of source code deletions

Or

i. The number of features introduced
ii. The number of bug fixes accomplished
iii. The number of modifications done

We can claim that "a modification has been done", when a change exists other than a new feature introduction or bug fix. Improving a working algorithm, or changing a button's size are examples of modifications. Another important metric of change is the change in the size of the software. In other words, each version's size difference should also be collected.

2.3 Defect identification over versions

We have used the terms "defect" and "bug" interchangeably throughout this paper. Thus, bug tracking software is the major defect identification source for a defect prediction scheme. During the life cycle of any version of a software product, the bug reports and descriptions should be collected. The module information and the criticality of the bugs should also be specified. It is important that the entries for bug reporting are moderated,

and a team or person should approve the listing and criticality levels of the bugs to avoid subjectivity.

Most of the bug tracking software products have a built-in statistics module; therefore, it is relatively easy to get the information related to the bugs. Eventually, we should come up with the information on the number of bugs in each module for each previous version as well as the criticality level of the submitted bug. We establish a function to express the relation between changes and bugs. We assume that we have the analysis of change input and distribution data concerning bugs. If the change information is collected by classifying changes, then the following function needs to be established:

$$N_{be} = f(A_f, A_b, A_c, A_s, A_{bf}, N_f, N_b, N_c, N_s) \quad (1)$$

where:

N_{be}: is the number of total bugs expected in the next stable version.
A_f: is the array of the total number of features introduced for all past stable versions in accordance with the order of version number.
A_b: is the array of the total number of bug fixes completed for all past stable versions before publishing the version in accordance with the order of version number.
A_c: is the array of the total number of changes completed for all past stable versions in accordance with the order of version number.
A_s: is the array of the size difference of a version with the previous version in terms of kilobytes for all past stable versions in accordance with the order of version number.
A_{bf}: is the array of the total number of bugs reported for all past stable versions in accordance with the order of version number.
N_f: is the total number of features introduced for the next stable version.
N_b: is the total number of bug fixes completed for the next stable version.
N_c: is the total number of changes completed for the next stable version.
N_s: is the size difference of the next version with the previous version in terms of kilobytes.
f: is a function of these parameters that builds up to predict $N_{be.}$

If the change collection method is based on CVS logs and line changes, then the following formula should be used.

$$N_{be} = f(A_i, A_d, A_c, A_s, A_b, N_i, N_d, N_c, N_s) \quad (2)$$

where:

N_{be}: is the number of total bugs expected in the next stable version.
A_i: is the array of the total number of lines inserted for all past stable versions in accordance with the order of version number.
A_d: is the array of the total number of lines deleted for all past stable versions in accordance with the order of version number.
A_c : is the array of the total number of line changes completed for all past stable versions in accordance with the order of version number.
A_s: is the array of size difference of a version with the previous version in terms of kilobytes for all past stable versions in accordance with the order of version number.
A_b: is the array of the total number of bugs reported for all past stable versions in accordance with the order of version number.
N_i: is the total number of lines inserted for the next stable version.
N_d: is the total number of lines deleted for the next stable version.

N_c: is the total number of line changes completed for the next stable version.
N_s: is the size difference of the next version with the previous version in terms of kilobytes.
f: is a function of these parameters that builds up to predict $N_{be.}$

2.4 Building the f function

The f function has a total of nine parameters. The first five parameters, namely the A_i, A_d, A_c, A_s, A_b parameters, are used to build up the historical data and to figure out the past experiences. The last four parameters, namely N_i, N_d, N_c, $N_{s,}$ are used to supply only version-specific data in order to be able to use historical experience. This function can be divided into two parts. The first part includes five parameters that should build up the environment, and the second part includes the last four parameters that supply input to this environment for predicting the number of bugs.

An alternative approach for "function" could be the interpolation or extrapolation of arrays on polynomials. However, we worked on large datasets so that interpolation or extrapolation algorithms are not the best alternatives for the path to converge. We, therefore, have chosen another common approach that includes the use of neural networks. The first five arrays should be used to build up the neural network and train the network. Then, the last four parameters should be used to acquire the prediction data from the neural network. The f function will convert into a procedure that consists of the following steps (Fig. 1).

3 Proposed defect prediction framework

In order to build up a robust model, we need to describe various tasks to be completed. Our proposed model requires that each task is completed in a sequential manner and that each task corresponds to an independent module.

As we stated in Fig. 2, the framework consists of four layers. In the first layer, the metric data are collected. The required metric data include CVS level change, feature level data and previous versions' defect density data. Then, the collected data are reorganized and converted into input format as described in detail in Sect. 3.1. Also, data are consolidated to sum up the defect numbers and changes for each module. In the third layer, the normalized data are used to train the selected predictor. Depending on the performance of

```
Procedure P
    Build a feed-forward neural network with the five arrays
    For each item in train dataset
        Train the neural network
    End For
    Simulate the network over the last four parameters regarding the upcoming version
    Return the output of neural network
End Procedure P
```

Fig. 1 Neural network procedure

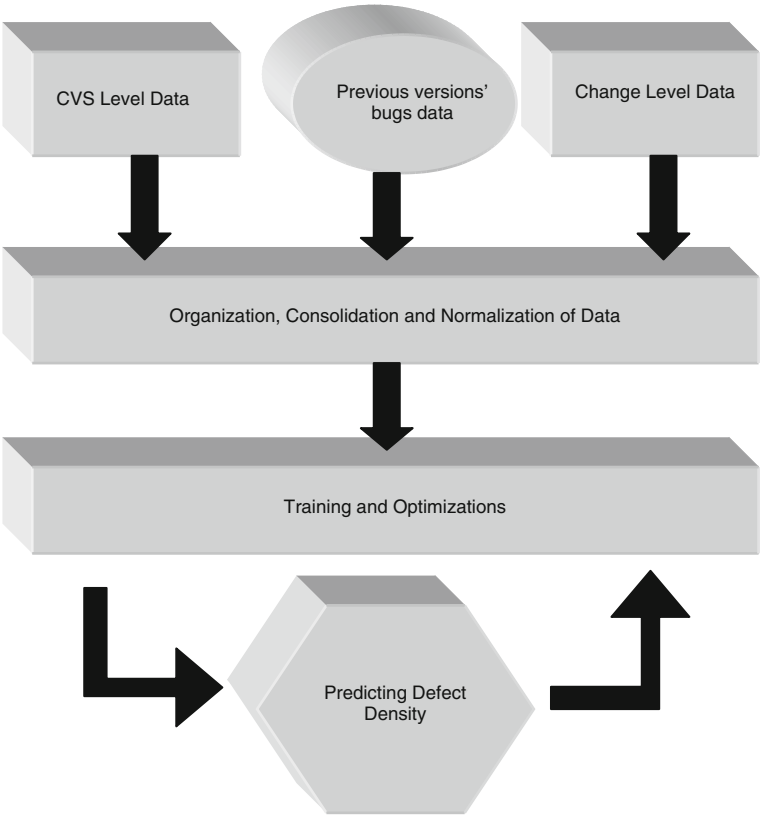


Fig. 2 Proposed defect prediction framework

the previous versions' defect predictions, optimizations are performed in this layer. These optimizations include the normalization of data within the limit range from −1 to 1, and removing non-orthogonal data from the datasets. We have used the PCA function in Matlab (Pendharkar et al. 2005). Consequently, we believe that our proposed model enables us to make accurate predictions.

3.1 Data collection and analysis process

The main focus of our research was to predict defects in multiversion software development. Therefore, defect data collection and a clear definition of defect data are two of the critical success factors for an accurate defect prediction. The bug tracking listings are the major source for analyzing the bugs. Thus, in such a research we had to find projects that have an extensive bug tracking and grouping tool containing historical data. However, some bug tracking systems may carry a risk of anyone easily entering any kind of bugs without any approval or filter. Therefore, such a public system can not be considered as a reliable bug tracking system in the scope of this research.

In selecting the bug tracking system, it was also important for us that the system included the following metrics: the related module or class of the bug, the version of the

software in which the bug is observed, the classification of the bug (user interface, function interface etc.), and the severity of the bug. A bug may cause a total failure of a software system, whereas another may only degrade the performance. In our context, the severity of the bug describes the impact of a specific bug on the software system. Since we wanted our proposed model to predict defect densities, we had to collect the bug data at module level. We defined a module as an independent software unit (i.e. user interface, database layer, communication library, etc.).

After forming the bug collection, we counted the bugs by grouping the modules and also weighted them according to their degree of severity. As a result, the final dataset contained the information of every version with the weighted bug densities. Since we focused our research on the impact of the change among versions, we also required another dataset that contained the change information. We can form a list of version changes by using two different approaches. In the first approach, the features, changes and bug fixes are collected at the level of functionality (feature level approach), whereas in the second approach only changes in CVS are collected (CVS level approach).

In the feature level approach, a version's change data should contain a number of new features introduced for each module, number of changes done for each module and number of bug fixes completed. "The feature" should be evaluated in the smallest possible piece of enhancement. The feature can be defined as a "notable property of a software system" (Harrold 2000). According to this definition, we can say that the feature introduces a better user experience, a nicer way of doing something, or it causes an increase in the definition of program requirements. In our research, we have also used the same definition. Because of the criticality of the definition of feature, we believe that the identification of features should be done by experts, and that the trained classifiers should be the secondary tools for these experts (Harrold 2000).

The other metric is the number of changes done for each module. Contrary to a feature a "change" simply changes the way a program functions. This change could be in user interface, class definitions or even in algorithms. We need to note that value-additions in algorithms should be considered as a feature. By keeping the modules that have changes or features introduced, our proposed prediction system will be able to predict not only at global but also at module level.

Another important metric in the feature level approach is the number of bug fixes completed. We assumed that every so-called stable version may contain bugs, and all the known bugs are fixed for the next stable version. This assumption is valid since we weight the bugs as per their severity, and we can ignore the ones with low impact. Consequently, the bug fixing information is the crucial data for predicting the stability of the forth-coming version.

The CVS level approach, on the other hand, does not carry an ambiguity. All we need is the source code differences based on lines inserted, deleted and changed. Most of the CVS systems are capable of supplying these metrics by comparing two versions of a source file.

The primary data used in this research come from open source applications. Open source applications are much easier to access in terms of their source code as well as the statistical information on their code. Choosing the correct project for data extraction is quite an important issue. After an extensive assessment in open source projects, we selected Azureus–Bit torrent client. Azureus is the most active project in the open-source community. 22 developers actively participate in the development process. This number is quite high concerning the developers involved in an open-source project. It has 28 stable versions for analyzing defects. Also, there are a total of 2322 bugs for analysis. Moreover, it is a user-application P2P software. Thus, it has wide range of new features. 3 stable releases before 2.0.0.0 were not containing any bug records. The 8 releases from version

2.0.0.0 to 2.0.3.2 contain bug records but not the grouping of bugs as feature, change or bugfix. As a result, Fig. 3 contains 25 versions, and Fig. 4 contains 17 versions.

We have mapped all bugs with the stable version. An important property of the bug data is the severity level. After mapping bugs, the weighted distribution was calculated by taking these severity classes into consideration. We multiplied bug counts with their severity level to come up with the defect density of the modules in the different versions. Consequently, we obtained the following distribution in Fig. 3.

This dataset gives us insight on the distribution of defect density. Each version has its own defect impact rate, which also implies the severity of the defects. Within the scope of this project, we also collected the change information for defect prediction purposes. Each version has major differences from the previous one in terms of change, bug fix and feature. The CVS logs of this project were scanned in detail, and we were able to put through a dataset and distribution of change over the versions. We collected 758 lines of change data for all versions. This data contain module-based change type with an explanation of completed task, and they are collected from the CVS and file change log of the www.sourceforge.net. The distribution of the data is shown in the Fig. 4.

The final adaptation of data merged the change information with the defect information of each version. So, two datasets were merged and grouped into versions. The output is shown in Fig. 5.

After completing an extensive data collection and pre-processing, we can use the dataset in our proposed model for defect prediction. To validate our proposed model, we need to test with metric data from different projects as well as the two different approaches we had explained earlier. The previous project was a user application for P2P network connections. We chose Linux kernels as a different project to test our proposed model. Linux kernels are more sensitive to defects. They have lower defect tolerance limits since Linux is an operating system (Sontag 1992). The dataset is chosen among sub releases of 2.4 and 2.6 versions of Linux kernels as they are considered to be the stable versions. We also chose the versions older than 6 months, because the defect-reporting pool would not be full enough with all defects for the latest versions of kernels. In the end, we came up with 33 versions of Linux kernels to observe during our experiment.

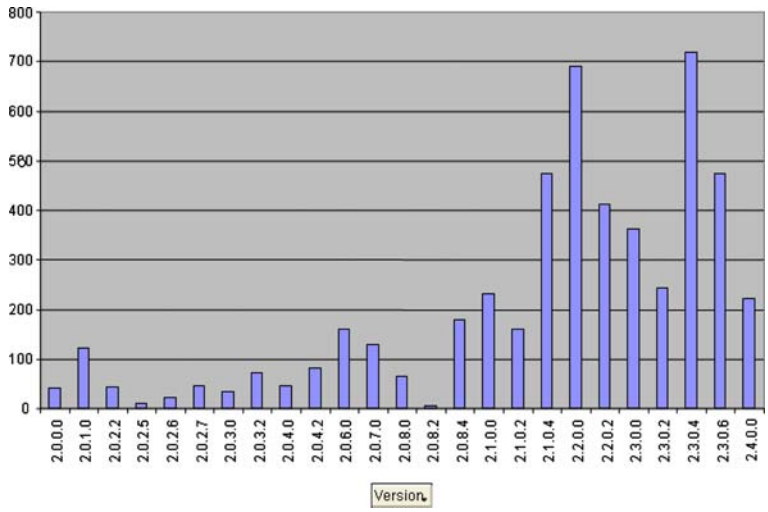


Fig. 3 Weighted defect densities versus versions

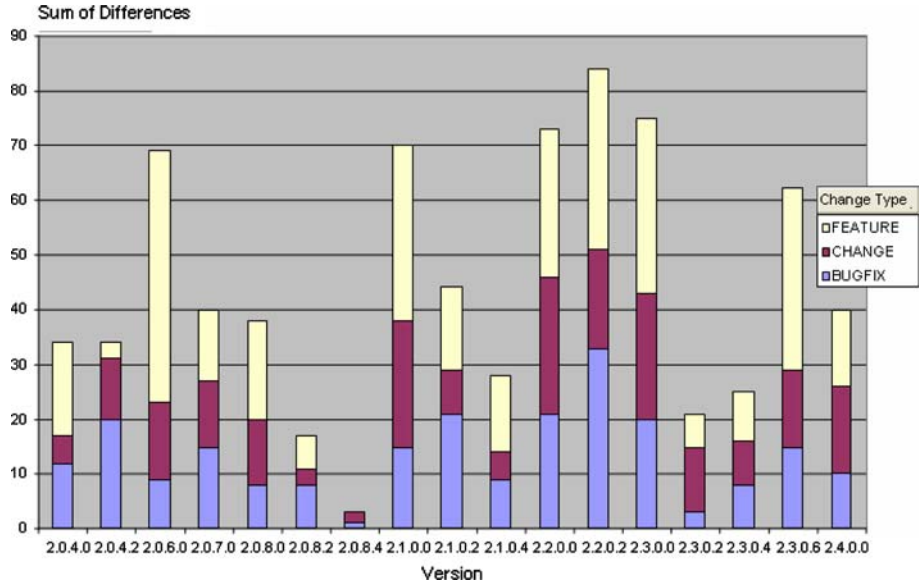
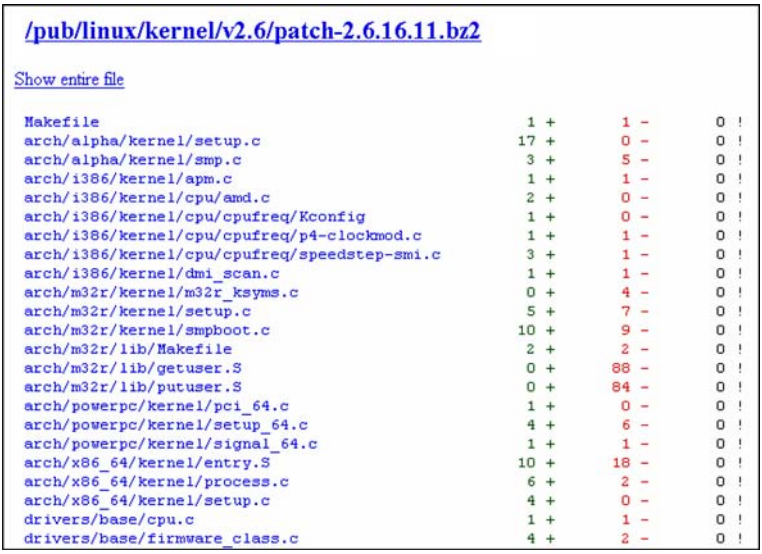


Fig. 4 Change type distributions versus versions

	INPUTS				OUTPUT
Version	BUGFIX	CHANGE	FEATURE	SIZE CHANGE	BUGS INTRODUCED
2.0.4.0	12	5	17	319	46
2.0.4.2	20	11	3	15	81
2.0.6.0	9	14	46	495	160
2.0.7.0	15	12	13	184	129
2.0.8.0	8	12	18	268	65
2.0.8.2	8	3	6	29	5
2.0.8.4	1	2	0	102	180
2.1.0.0	15	23	32	609	232
2.1.0.2	21	8	15	623	160
2.1.0.4	9	5	14	122	473
2.2.0.0	21	25	27	482	689
2.2.0.2	33	18	33	312	411
2.3.0.0	20	23	32	1215	362
2.3.0.2	3	12	6	127	244
2.3.0.4	8	8	9	136	718
2.3.0.6	15	14	33	358	472
2.4.0.0	10	16	14	537	222

Fig. 5 Final dataset containing both change and defect information

At this point, we wanted to look at CVS level changes instead of feature, change and bug fix analysis. The main purpose of using CVS level changes was to test the CVS level approach. It was also impractical to obtain feature level data for Linux kernels. The CVS file difference analysis tool helps us give the number of changed lines, deleted lines and inserted lines. Thus, we extracted the change information for all 33 versions. Another challenge for Linux kernels was extracting the defect information for each version. We have, therefore, used the query mechanisms of Linux Kernel Tracker. This site contains



/pub/linux/kernel/v2.6/patch-2.6.16.11.bz2

Show entire file

Makefile	1 +	1 -	0 !
arch/alpha/kernel/setup.c	17 +	0 -	0 !
arch/alpha/kernel/smp.c	3 +	5 -	0 !
arch/i386/kernel/apm.c	1 +	1 -	0 !
arch/i386/kernel/cpu/amd.c	2 +	0 -	0 !
arch/i386/kernel/cpu/cpufreq/Kconfig	1 +	0 -	0 !
arch/i386/kernel/cpu/cpufreq/p4-clockmod.c	1 +	1 -	0 !
arch/i386/kernel/cpu/cpufreq/speedstep-smi.c	3 +	1 -	0 !
arch/i386/kernel/dmi_scan.c	1 +	1 -	0 !
arch/m32r/kernel/m32r_ksyms.c	0 +	4 -	0 !
arch/m32r/kernel/setup.c	5 +	7 -	0 !
arch/m32r/kernel/smpboot.c	10 +	9 -	0 !
arch/m32r/lib/Makefile	2 +	2 -	0 !
arch/m32r/lib/getuser.S	0 +	88 -	0 !
arch/m32r/lib/putuser.S	0 +	84 -	0 !
arch/powerpc/kernel/pci_64.c	1 +	0 -	0 !
arch/powerpc/kernel/setup_64.c	4 +	6 -	0 !
arch/powerpc/kernel/signal_64.c	1 +	1 -	0 !
arch/x86_64/kernel/entry.S	10 +	18 -	0 !
arch/x86_64/kernel/process.c	6 +	2 -	0 !
arch/x86_64/kernel/setup.c	4 +	0 -	0 !
drivers/base/cpu.c	1 +	1 -	0 !
drivers/base/firmware_class.c	4 +	2 -	0 !

Fig. 6 The CVS log analysis of kernel files between versions

every single bug entered for any of the Linux kernel versions. It has extensive statistics and querying options. The last data that we had to collect was the size change information of Linux kernels. We collected size change data by calculating the differences from version to version (Fig. 6). The first column shows the file name that is given as an input for comparison. The second column shows the number of insertions the third column shows the number of lines deleted, whereas the last column shows the total number of changes. After extensive human work for data extraction, the change data was fully collected. The final output is shown in Fig. 7.

The experiments in this dataset were not extended to modules. Each kernel was treated as a single module. Our basic assumption was that all the bugs were fixed before the next release. We can confidently make this assumption since we multiplied the severity level with the bug counts. Therefore, the remaining unfixed bugs had a minor impact on system's stability so that we were able to neglect their existence.

4 Experimental design

We have chosen to use neural networks in our experiments because neurons have the ability to adapt to a particular situation by changing their weights and thresholds. A feed-forward neural network allows signals to travel one way only: from input to output (Sontag 1992). Throughout the experiments in this research, the multilayer perceptron (MLP) method was used in neural network experiments. MLPs are feed-forward neural networks trained with the standard back-propagation algorithm (Sontag 1992). The back-propagation networks were trained using the scaled conjugate gradient optimization in our experiments.

Bishop and Sontag claim that in MLPs with step/threshold/Heaviside activation functions, two hidden layers are sufficient for full generality (Bishop 1995; Sontag 1992). On the other hand, if there is only one input, using more than one hidden layer has no advantage. According to Sarle (1996);

	INPUTS				OUTPUT
Version	CHANGE	INSERTION	DELETION	SIZE CHANGE	BUGS INTRODUCED
2.4.18.0	1013	75299	18349	806	15
2.4.19.0	3745	549895	162806	4480	9
2.4.20.0	3462	406403	152014	3970	41
2.4.21.0	2963	366643	147765	3670	24
2.4.22.0	3895	487094	322019	4960	72
2.4.23.0	1551	171914	110551	2090	45
2.4.24.0	18	35	16	2	16
2.4.25.0	1774	223743	54469	1740	30
2.4.26.0	672	52087	38026	766	33
2.4.27.0	767	75173	35110	826	29
2.4.28.0	676	31351	15026	467	6
2.4.29.0	769	33838	10768	398	8
2.4.30.0	218	4826	2394	99	4
2.4.31.0	50	1294	459	25	8
2.4.32.0	122	2464	1152	54	6
2.6.0.0.	43	218	117	218	22
2.6.1.0	998	40596	50838	759	78
2.6.2.0	2370	177655	88369	2210	117
2.6.3.0	1906	141924	94321	1950	174
2.6.4.0	3185	192361	143740	2150	125
2.6.5.0	2261	101535	56991	1350	271
2.6.6.0	2642	173046	104312	2350	236
2.6.7.0	3772	175409	157607	2930	291
2.6.8.0	4604	250569	1153188	3520	404
2.6.9.0	4712	260926	131570	3290	370
2.6.10.0	5384	322353	289814	5	367
2.6.11.1	3	8	4	1	39
2.6.11.2	4	10	5	1	8
2.6.11.3	21	72	60	5	10
2.6.11.4	23	74	62	5	13
2.6.11.5	31	102	81	6	13
2.6.11.6	36	146	104	8	26

Fig. 7 Final dataset containing both change and defect information of Linux Kernels

> Unfortunately, using two hidden layers exacerbates the problem of local minima, and it is important to use lots of random initializations or other methods for global optimization. Local minima with two hidden layers can have extreme spikes or blades even when the number of weights is much smaller than the number of training cases. One of the few advantages of standard backprop is that it is so slow that spikes and blades will not become very sharp for practical training times.

We, therefore, used one hidden layer, considering that our total number of inputs is four (Swingler 1996). There are various proposed methods for deciding the number of units in a hidden layer (Song 2006; Tahat and Korel 2001). Although there is no "silver-bullet" to calculate the best hidden unit number, certain rules of thumb exist for general cases. Swingler claims that, "you will never require more than twice the number of hidden units as you have inputs" in an MLP with a single hidden layer (Swingler 1996). In our case, since our number of input units was four, one or two hidden units were enough for generalization. Barry and Linoff also support this claim by stating that, "One rule of thumb is that it should never be more than twice as large as the input layer" (Barry and Linoff 1997). In the light of these previous research, we have decided to use one single hidden layer with two units hidden inside, so that the hidden units will be exactly the half of the inputs and exactly twice the number of the outputs. Figure 8 represents our final network:

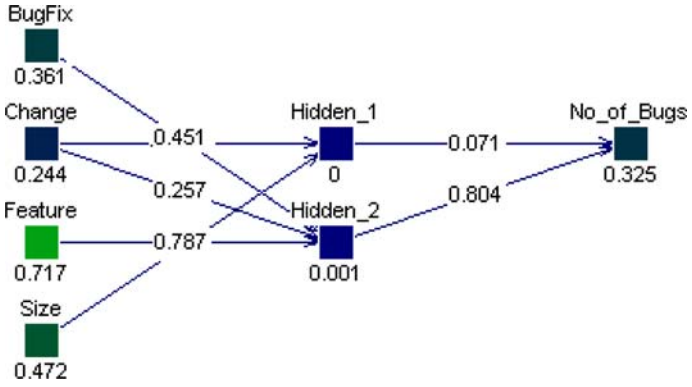


Fig. 8 Neural network model used for evaluating dataset

4.1 Threats to validity

Similar to any other empirical research, the threats to validity in our research is twofold: our results and conclusions are biased based on the data we used and the machine-learning algorithm we employed (Menzies et al. 2007).

The datasets we used come from open source projects. Project Azereus is a large open source P2P application which has a wide range of new features. Such a project would be a typical commercial application. The other dataset comes from a Linux kernel project. An operating system software is a good representative of a type of software that has a lower defect tolerance limit. We believe that choosing two different large and multiversion projects with different characteristics (product and defect tolerance) would lead us to draw relevant conclusions for the community of software engineering. Moreover, since our data are public and we share our proposed model and methodology in detail, other researchers may repeat and improve our experiments.

We have chosen neural networks for our experiments as well as the MLP method in neural networks since it gives better learning results when compared to other algorithms (Fenton and Neil 1999; Ceylan et al. 2006; Boehm et al. 1995; Boehm and Basili 2000). Since our network structure is not too complex and we have a limited number of data, we were not concerned about the running time of the algorithm. Our experiments took only 14 hours of execution and did not require a high-performance CPU. One of the major problems of neural networks is the problem of overfitting (Alpaydin 2004). To avoid overfitting, we examined error versus epoch plots for different learning rates. When we observed a plateau of error, we set our parameters. We also shuffled data in each epoch to overcome ordering and sampling bias.

4.2 Common features of experiments

We calculate the Mean Squared Error (MSE) as:

$$\mathrm{MSE} = \left(\sum_{i} (\mathrm{Real}_i - \mathrm{Predicted}_i)^2 \right) \Big/ \mathrm{N}_{\mathrm{test}} \tag{3}$$

where N_{test} = size of test set. We use MSE as a performance indicator because it gives an objective idea about the success of the prediction and it is widely used in neural network applications (Sontag 1992).

4.3 Feature level data set

The experiments are done with the feature level data set which is the Azureus project. As we explained in detail in the beginning of Sect. 4, we have chosen the learning algorithm, decided the learning rate and found the necessary epoch number in order to measure the performance results in the end. Various research experiments have been using MLP with different learning algorithms. It is mentioned that the MLP and Radial Basis learning functions have better results compared to other learning algorithms (Ceylan et al. 2006). We have decided to use MLP to train our neural network model.

The learning rate parameter is also important to find the correct convergence to the desired output. Thus, we have tested the learning rate for different inputs. We see a plateau of internal error after the 40th epoch. Only in the case of 0.5 as the learning rate, the internal error could not drop below 0.14. Considering the performances regarding the learning rate, 0.2 was selected as the best performing one on our dataset.

The dataset was used with shuffling, such that all versions were predicted at an instance of multiple runs. We have done a regression analysis of predicted and actual data (Fig. 9).

The execution was repeated 100 times, and the average values were extracted for error. The closer points to 45 degree line represent the more successful prediction. We have small deviations representing the same predicted values. MSE is calculated as 1.7374 e+004.

4.4 CVS level data set

The second group of experiments were conducted on Linux Kernel data. The dataset was used with shuffling, in a way that all versions were predicted at an instance of multiple runs. We have done a regression analysis of predicted and actual data. This analysis can be seen in Fig. 10.

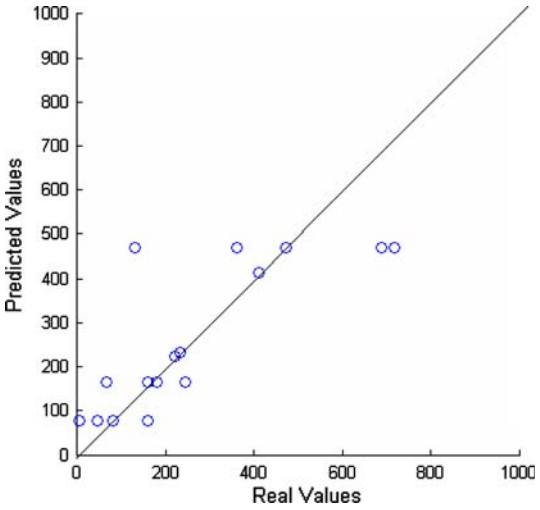


Fig. 9 Real values versus predicted values for feature level dataset

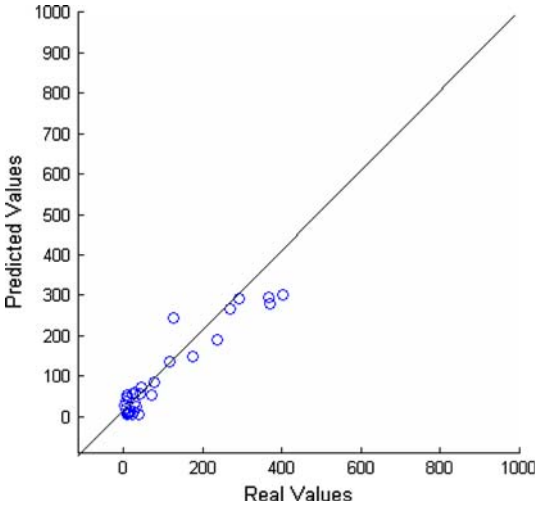


Fig. 10 Real values versus predicted values for CVS level dataset

The execution were repeated 100 times, and the average values were extracted for error. MSE was calculated as 1.5880 e+003.

4.5 Evaluation

The research in this paper focused on two diverse approaches for defect predictions. The feature level one applied the Azureus project data which had the following dimensions: the number of bug fixes completed, features introduced, changes done, and the size difference between the versions. This experiment was conducted with a large number of bugs for all versions.

The second experiment was performed with a completely different dataset. This dataset was collected from stable Linux kernel versions and from kernel bugs databases. This dataset was approximately twice as large as the previous dataset. The major difference of the CVS Level experiment was the input dimensions introduced to the model. The input dimensions were: the number of lines inserted, lines deleted, lines changed, and the size difference between te versions.

The major difference between the first and the second experiments was the approach on code change. The second experiment focused only on the changes in CVS level, whereas the first experiment took into consideration the changes in feature and the bug fix abstract level. These two experiments were executed over two different datasets with two different approaches. Since, we are working with real life data, it is practically impossible to obtain identical data structures for two different software projects of different teams.

5 Conclusions and future work

Within the scope of our study, we have proposed a defect prediction model for a multi-version software and collected the necessary data to conduct various experiments. We believe that our research has three main contributions: data, method and experiment.

The first contribution is the collection and pre-processing of extensive data on bugs and changes over versions for two different software products. The data contain important

statistical information regarding developmental life cycles of software versions. Some of the statistical analyses have been mentioned in the data collection section. The second contribution of this research has been to propose a novel method for defect prediction systems. Introducing the concept of defect prediction systems for software versioning and using difference between versions were important issues covered through this research. Our decision process to choose the correct mathematical framework and algorithms may inspire other researchers. The third contribution of this research is the experimental set-up. Having two different experiments enhanced the validity and usability of the proposed model.

There are also practical contributions of this research to software development companies. Assuming that these companies are using a well-defined metric collection framework, our proposed model can be used continuously as a version-control mechanism for upcoming software versions. Software development companies may target a defect density level for the new version. Our proposed model could help them proactively track the defect density. As a result, they can acquire a better position to prioritize testing tasks as well as allocating their limited testing resources more efficiently.

As a future work, our proposed model can be extended to include both CVS level inputs and feature level inputs. We believe that such a hybrid model can provide more accurate prediction results. If a dataset can be found in real life projects, our experiments based on two different datasets and approaches may be repeated on the same project. In addition, the identification of defect sources may also be added in the future.

Acknowledgements This work is supported in part by the Boğaziçi University research fund under grant number BAP–06HA104. Special thanks to our colleague Burak Turhan for his valuable comments on the manuscript. We would also thank Ms Cigdem Aksoy Fromm who has done the final editing of the manuscript.

References

Alpaydin, E. (2004). *Introduction to machine learning*. Cambridge, MA: MIT Press.

Barry, M. J. A., & Linoff, G. (1997). *Data mining techniques: For marketing, sales, and customer support*. New York: John Wiley.

Bishop, C. M. (1995). *Neural networks for pattern recognition*. Oxford: Oxford University Press.

Boehm, B., & Basili, V. R. (2000). Gaining intellectual control of software development. *Computer, 33*(5), 27–33.

Boehm, B., & Basili, V. (2001). Software defect reduction top 10 list. *IEEE Computer, 34*(1), 135–137.

Boehm, B., Clark, B., Horowitz, E., & Westland, C. (1995). Cost models for future software life cycle processes: COCOMO 2.0. *Annals of Software Engineering, Special Volume on Software Process and Product Measurement*(1), 57–94.

Bowen, J. P., & Hinchey, M. G. (1995). Seven more myths of formal methods. *IEEE Software, 12*(4), 34–41. doi:10.1109/52.391826.

Brilliant, S. S., Knight, J. C., & Leveson, N. G. (1990). Analysis of faults in an N-version software experiment. *IEEE Transactions on Software Engineering, 16*(2), 238–247. doi:10.1109/32.44387.

Brun, Y., & Ernst, M. (2004). *Finding latent code errors via machine learning over program executions*. Edinburgh, Scotland: ICSE 2004, 26th International Conference on Software Engineering.

Ceylan, E., Kutlubay, O., & Bener, A. (2006). Software Defect Identification Using Machine Learning Techniques. In *32nd Euromicro Conference on Software Engineering and Advanced Applications (Euromicro-SEAA 2006)*, Crotia.

Clarke, E. M., & Wing, J. M. (1996). Formal methods: state of the art and future directions. *ACM Computing Surveys, 28*(4), 626–643. doi:10.1145/242223.242257.

Coppit, D., Yang, J., Khurshid, S., Le, W., & Sullivan, K. (2005). Software assurance by bounded exhaustive testing. *IEEE Transactions on Software Engineering, 31*(4), 328–339. doi: 10.1109/TSE.2005.52.

Fenton, N., & Neil, M. (1999). A critique of software defect prediction models. *IEEE Transactions on Software Engineering, 25*(5), 675–689. doi:10.1109/32.815326.
Fenton, N., & Ohlsson, N. (2000). Quantitative analysis of faults and failures in a complex software system. *IEEE Transactions on Software Engineering, 26*(8), 797–814. doi:10.1109/32.879815.
Gregoriades, A., & Sutcliffe, A. (2005). Scenario-based assessment of nonfunctional requirements. *IEEE Transactions on Software Engineering, 31*(5), 392–409. doi:10.1109/TSE.2005.59.
Groce, P., & Visser, W. (2003). What went wrong: Explaining counterexamples. *10th International SPIN Workshop on Model Checking of Software*. Portland, Oregon, pp. 121–135.
Harrold, M. J. (2000). Testing: A roadmap. *Proceedings of the Conference on the Future of Software Engineering*, Limerick, Ireland.
Inoue, K., Yokomori, R., Yamamoto, R., Matsushita, M., & Kusumoto, S. (2005). Ranking significance of software components based on use relations. *IEEE Transactions on Software Engineering, 31*(3), 213–225. doi:10.1109/TSE.2005.38.
Jensen, F. (1996). *An introduction to Bayesian networks*. NY: Springer Verlag.
Johnson, P. M., Kou, H., Paulding, M., Zhang, Q., Kagawa, A., & Yamashita, T. (2005). Improving software development management through software project telemetry. *IEEE Software, 22*(4), 76–85. doi:10.1109/MS.2005.95.
Jorgensen, M. (2005). Practical guidelines for expert-judgment-based software effort estimation. *IEEE Software, 22*(3), 57–63. doi:10.1109/MS.2005.73.
Khoshgoftaar, T. M., & Allen, E. B. (1999). A comparative study of ordering and classification of fault-prone software modules. *Empirical Software Engineering, 4*, 159–186. doi:10.1023/A:1009876418873.
Koru, G., & Liu, H. (2005). Building defect prediction models in practice. *IEEE Software, 22*(6), 23–29. doi:10.1109/MS.2005.149.
Kung, D. C., Gao, J., Hsia, F., Wen, F., Toyoshima, Y., & Chen, C. (1994). Change impact identification in object oriented software maintenance. *Proceedings of the International Conference on Software Maintenance*, pp. 202–211, IEEE Computer Society Press.
Menzies, T., DiStefano, J., & Chapman, R. (2004). Assessing predictors of software defects. *Proccedings of Workshop on Predictive Software Models*, Chicago.
Menzies, T., DiStefano, J. S., Chapman, M., & McGill, K. (2002). Metrics that matter. *Proceedings of 27th NASA SEL Workshop on Software Engineering*.
Menzies, T., Greenwald, J., & Frank, A. (2007). Data mining static code attributes to learn defect predictors. *IEEE Transactions on Software Engineering, 33*(1), 2–13. doi:10.1109/TSE.2007.256941.
Menzies, T., D. Stefano, J. S., & Chapman, M. (2003). Learning early life cycle IV & V quality indicators. *Proceedings of Ninth International Software Metrics Symposium.*
Mitchell, T. M. (1997). *Machine Learning*. NY: McGrawHill.
Munson, J., & Khoshgoftaar, T. M. (1990). Regression modelling of software quality: Empirical investigation. *Journal of Electronic Materials, 19*(6), 106–114.
Nagappan, N., & Ball, T. (2005a). *Use of relative code churn measures to predict system defect density*. St. Louis, MO: ICSE 2005.
Nagappan, N., & Ball, T. (2005b). *Static analysis tools as early indicators of pre-release defect density*. St. Louis, MO: ICSE 2005.
Nagappan, N., Williams, L., Osborne, J., Vouk, M., & Abrahamsson, P. (2005). *Providing test quality feedback using static source code and automatic test suite metrics*. Chicago, IL: International Symposium on Software Reliability Engineering.
Ostrand, T. J., Weyuker, E. J., & Bell, R. M. (2005). Predicting the location and number of faults in large software systems. *IEEE Transactions on Software Engineering, 31*(4), 340–355. doi:10.1109/TSE.2005.49.
Padberg, F., Ragg, T., & Schoknecht, R. (2004). Using machine learning for estimating the defect content after an inspection. *IEEE Transactions on Software Engineering, 30*(1), 17–28. doi:10.1109/TSE.2004.1265733.
Pendharkar, P. C., Subramanian, G. H., & Rodger, J. A. (2005). A probabilistic model for predicting software development effort. *IEEE Transactions on Software Engineering, 31*(7), 615–624. doi:10.1109/TSE.2005.75.
Podgurski, D., Leaon, P., Francis, P., Masri, W., Minch, M., Jiayang, S., & Wang, B. (2003). *Automated support for classifying software failure reports*. Portland, Oregon: ICSE 2003.
Porter, A., & Votta, L. (2004). Comparing detection methods for software requirements inspections: A replication using professional subjects. *Empirical Software Engineering, 3*(4), 355–379.
Sarle, W. (1996). How many hidden layer should i use. *Neural Nets FAQ*, http://www.faqs.org/faqs/ai-faq/neural-nets/part3/section-9.html.

Sheppard, M., & Ince, D. C. (1994). A critique of three metrics. *The Journal of Systems and Software, 26*(33), 197–210.
Song, O., Sheppard, M., Cartwright, M., & Mair, C. (2006). Software defect association mining and defect correction effort prediction. *IEEE Transactions on Software Engineering, 32*(2), 69–82. doi:10.1109/TSE.2006.1599417.
Sontag, E. D. (1992). Feedback stabilization using two-hidden-layer nets. *IEEE Transactions on Neural Networks, 3*, 981–990. doi:10.1109/72.165599.
Sourceforge. from www.sourceforge.net.
Swingler, K. (1996). *Applying neural networks: Apractical guide*. London: Academic Press.
Tahat, L. H., Vaysburg, B., Korel B., & Bader, A. J. (2001). A requirement-based automated black-box test generation. *Proceedings of 25th Annual International Computer Software and Applications Conference*, Chicago, IL, pp. 489–495.
Vaidyanathan, K., & Trivedi, S. (2005). A comprehensive model for software rejuvenation. *IEEE Transactions on Dependable Secure Computing, 2*(2), 124–137. doi:10.1109/TDSC.2005.15.
Zhang, D. (2000). Applying machine learning algorithms in software development. *The Proceedings of 2000 Monterey Workshop on Modeling Software System Structures*, Santa Margherita Ligure, Italy.

Author Biographies

Yomi Kastro holds B.Sc. and M.Sc. degrees in Computer Engineering from Boğaziçi University. Currently, he is the Managing Partner of Inveon, a software development company in Turkey. His research interests are software quality, software testing and verification.

Ayşe Basar Bener is an assistant professor and a full time faculty member in the Department of Computer Engineering at Boğaziçi University. Her research interests are software defect prediction, process improvement and software economics. Bener has a PhD in information systems from the London School of Economics. She is a member of the IEEE, the IEEE Computer Society, and the ACM.