Cost-Benefit Analysis of Software Quality Models

Taghi M. Khoshgoftaar (taghi@cse.fau.edu)
Florida Atlantic University, Boca Raton, Florida USA

Edward B. Allen (edward.allen@computer.org)*
Mississippi State University, Mississippi USA

Wendell D. Jones and John P. Hudepohl
Nortel, Research Triangle Park, NC 27709 USA

Abstract. Software reliability is increasingly important in today's marketplace. When traditional software development processes fail to deliver the level of reliability demanded by customers, radical changes in software development processes may be needed. Business process reengineering (BPR) is the popular term for comprehensive redesign of business processes. This paper focuses on the business processes that produce commercial software, and illustrates the central role that models have in implementation of BPR. Software metrics and software-quality modeling technology enable reengineering of software development processes, moving from a static process model to a dynamic one that adapts to the expected quality of each module.

We present a method for cost-benefit analysis of BPR of software development processes as a function of model accuracy. The paper defines costs, benefits, profit, and return on investment from both short-term and long-term perspectives. The long-term perspective explicitly accounts for software maintenance efforts.

A case study of a very large legacy telecommunications system illustrates the method. The dependent variable of the software-quality model was whether a module will have faults discovered by customers. The independent variables were software product and process metrics. In an example, the costs and benefits of using the model are compared to using random selection of modules for reliability enhancement. Such a cost-benefit analysis clarifies the implications of following model recommendations.

Keywords: software reliability, software quality model, cost-benefit analysis, return on investment, software metrics, business process reengineering, BPR

1. Introduction

Software is the medium for implementing increasingly sophisticated features throughout a system's lifetime (Hudepohl et al., 1992). Consequently, software reliability is a strategic business weapon in today's competitive marketplace (Hudepohl, 1990). When traditional software development processes fail to deliver the level of reliability demanded

* This work was performed while Edward B. Allen was at Florida Atlantic University.

by customers, radical changes in software development processes may be needed.

Business process reengineering (BPR) is the popular term for comprehensive redesign of business processes to gain dramatic improvements in performance measures, such as cost, quality, service, and speed (Motwani et al., 1998; Orman, 1998). Hammer (1990) coined BPR for the radical but systematic change to business processes needed to exploit fully the potential offered by information technology (Biazzo, 1998). He later popularized the concept in a book with Champy (Hammer and Champy, 1993).

This paper focuses on the business processes that produce commercial software. Typical software development processes treat all modules similarly according to a static model of the development life cycle. For example, each software module goes through the same processes of design, reviews, coding, and testing. Finally, the software is released to customers. The introduction of software metrics and software-quality modeling technology has precipitated reengineering of software development processes, moving from the static process model to a dynamic one that adapts to the expected quality of each module. A dynamic development process can result in dramatic improvements in software quality delivered to customers, with attendant gains in customer satisfaction and major reductions in rework costs.

Software-metrics technology measures attributes of artifacts and processes during development. For example, common software product metrics quantify attributes of a module's source code (Fenton and Pfleeger, 1997). Software process metrics quantify events in a module's history (Khoshgoftaar et al., 1998). Both kinds of measurements are related to mistakes during development and hence, software reliability. Reliability is an important aspect of software quality. The level of reliability is often indicated by the faults ("bugs") discovered by customers. In this paper, a software quality model predicts whether each module will be considered fault-prone after release, based on measurements of its product and process history as independent variables. A software quality model is a key technology that empowers developers to reprioritize modules dynamically during development, so that enhancement efforts are applied in a cost-effective manner.

For example, suppose a software quality model indicates that certain modules are at-risk when changed. After a review, the design team may choose to redesign these modules first before continuing their implementation, so that their reliability is enhanced. Moreover, during maintenance, any changes proposed for high-risk modules may require more stringent justification and inspection. Risk assessments can also be used for assigning technical staff to test, test automation, and maintenance efforts, matching skill level and experience with complexity.

Reengineering software development from a static model to dynamic allocation of resources promises dramatic improvement in quality. Can using a software quality model really deliver these benefits? This paper presents a method for cost-benefit analysis of a software quality classification model. We illustrate the method with a case study of a very large legacy telecommunications software system. The case study used logistic regression as the modeling technique (Hosmer and Lemeshow, 1989) in conjunction with a generalized classification model which we have proposed (Khoshgoftaar and Allen, 2000). Other classification techniques could have been used, but our goal is not a comparison of techniques. Our focus here is how to assess the costs and benefits of model accuracy, so that the implications of following model recommendations will be clear. Remaining sections present the generalized classification model, a description of the system studied, case study results, the cost-benefit analysis, and conclusions.

2. A Generalized Classification Model

Given a set of software modules, our goal is to predict the class of each module. We define the classes *fault-prone* and *not fault-prone* by a threshold on the number of faults over a period of interest, such as the operational life of a release of the software. This section presents a generalized classification model that we have proposed for use with software quality models (Khoshgoftaar and Allen, 2000).

The following scenario is the context for our analysis of the costs and benefits of using a software quality classification model. Development of each release of a system is considered a "project". Our case study focused on faults discovered by customers. The scenario could be adjusted for other definitions of *fault-prone.*

1. Build and evaluate a software quality classification model using data on a historical project(s), such as a prior release(s).

2. Collect data from the current project as early in development as possible for independent variables.

3. Apply the model to predict which modules will be *fault-prone*, i.e., modules "recommended" by the model for reliability enhancement.

4. Enhance the reliability of the recommended modules using special processes, such as extra reviews, extra testing, or redesign, and fix faults that are consequently discovered in the *fault-prone* modules.

5. Release the system for operational use.

6. Fix faults that are discovered by customers.

A Type I misclassification is when the model identifies a module as *fault-prone* which is actually *not fault-prone*. A Type II misclassification is when the model identifies a module as *not fault-prone* which is actually *fault-prone*. Table I summarizes notation. In a standard development process, the expected proportion of *fault-prone* modules is π_{fp}, and similarly for *not fault-prone* modules, π_{nfp}. As explained in Table I, the expected number of modules in various subsets are shown in Table II, where in addition to totals, the rows signify actual classes of modules, and the columns correspond to classes predicted by a model.

There is a tradeoff between the Type I misclassification rate and the Type II misclassification rate. As one goes down, the other goes up. We have observed that it is often difficult to determine a practical balance for this tradeoff in terms of Type I and Type II misclassification rates. Therefore, we translate these rates into measures of the *effectiveness* and *efficiency* of a model, which are more closely related to project management concerns.

Following the model's recommendation, the proportion of modules that receive reliability enhancement that are actually *fault-prone* is $\Pr(fp|fp)\pi_{fp}$. We define *effectiveness* as the proportion of *fault-prone* modules that received reliability enhancement treatment out of all the *fault-prone* modules.

$$\textit{effectiveness} = \frac{\Pr(fp|fp)\pi_{fp}}{\pi_{fp}} = 1 - \Pr(nfp|fp) \tag{1}$$

One can maximize *effectiveness* by minimizing the Type II misclassification rate, $\Pr(nfp|fp)$.

When one applies a reliability enhancement process to a *not fault-prone* module, it will probably be a waste of time, because the reliability is already satisfactory. We define *efficiency* as the proportion of reliability enhancement effort that is not wasted. This is equivalent to the proportion of *fault-prone* modules that received reliability enhancement treatment out of all modules that received it.

$$\textit{efficiency} = \frac{\Pr(fp|fp)\pi_{fp}}{\Pr(fp|nfp)\pi_{nfp} + \Pr(fp|fp)\pi_{fp}} \tag{2}$$

One can maximize *efficiency* by minimizing the Type I misclassification rate, $\Pr(fp|nfp)$.

Our goal is to allow appropriate emphasis on *effectiveness* and *efficiency* according to the needs of the project. The following rule enables

Table I. Notation

G_{nfp}	the *not fault-prone* class (group) of modules
G_{fp}	the *fault-prone* class (group) of modules
k	index of classes
n	the number of modules in the software system under study
i	index of modules
$\mathbf{x}_i$	the vector of independent variables of the i^{th} module
$Class(\mathbf{x}_i)$	the i^{th} module's class, predicted by a model's classification rule
$f_k(\mathbf{x}_i)$	a likelihood function for the i^{th} module's membership in the class k
$\hat{f}_k(\mathbf{x}_i)$	an estimate of $f_k(\mathbf{x}_i)$
p_i	the probability that a module is *fault-prone*
$\hat{p}_i$	an estimate of p_i; in a logistic regression context, $\hat{f}_{nfp}(\mathbf{x}_i) = 1-\hat{p}_i$, $\hat{f}_{fp}(\mathbf{x}_i) = \hat{p}_i$
π_k	the prior probability of membership in G_k
n_k	the expected number of modules in G_k
$\Pr(nfp \vert nfp)$	the probability that a model correctly classifies a module as in G_{nfp}
$\Pr(fp \vert fp)$	the probability that a model correctly classifies a module as in G_{fp}
$\Pr(fp \vert nfp)$	the probability that a model misclassifies a module as in G_{fp} which is actually in G_{nfp}, i.e., the Type I misclassification rate
$\Pr(nfp \vert fp)$	the probability that a model misclassifies a module as in G_{nfp} which is actually in G_{fp}, i.e., the Type II misclassification rate
$\hat{n}_{nfp,nfp}$	the expected number of modules that a model correctly classifies as in G_{nfp}
$\hat{n}_{fp,fp}$	the expected number of modules that a model correctly classifies as in G_{fp}
$\hat{n}_{nfp,fp}$	the expected number of modules that a model misclassifies as in G_{fp} which are actually in G_{nfp}
$\hat{n}_{fp,nfp}$	the expected number of modules that a model misclassifies as in G_{nfp} which are actually in G_{fp}
C_I	the cost of a Type I misclassification
C_{II}	the cost of a Type II misclassification

a project to select the best balance between the misclassification rates, and consequently, between *effectiveness* and *efficiency*.

$$Class(\mathbf{x}_i) = \begin{cases} G_{nfp} & \text{if } \frac{\hat{f}_{nfp}(\mathbf{x}_i)}{\hat{f}_{fp}(\mathbf{x}_i)} \geq c \\ G_{fp} & \text{otherwise} \end{cases} \tag{3}$$

where c is a constant which we experimentally choose. Each classification technique provides its own way to estimate the likelihood, $\hat{f}_k(\mathbf{x}_i)$,

Table II. Notation for Module Counts

Class	Model		Total
	G_{nfp}	G_{fp}	
Actual			
G_{nfp}	$\hat{n}_{nfp,nfp}$	$\hat{n}_{nfp,fp}$	n_{nfp}
G_{fp}	$\hat{n}_{fp,nfp}$	$\hat{n}_{fp,fp}$	n_{fp}
Total			n

of belonging to each class (group), G_k. Given a candidate value of c, we estimate *effectiveness* and *efficiency*. If the balance is not satisfactory, we select another candidate value of c and estimate again, until we arrive at the best c for the project.

For example, one approach is to choose c so that *effectiveness* is at least a minimum acceptable level. In another approach, one chooses c such that Type I and Type II misclassification rates are equal (Khoshgoftaar et al., 1998; Seber, 1977). This value of c is especially appropriate when $\pi_{fp} \ll \pi_{nfp}$. In practice, we can achieve only approximate equality due to finite discrete data sets. A third approach sets a maximum on the number of modules recommended for enhancement. Many other strategies are also possible.

3. System Description

We conducted a case study of a very large legacy telecommunications system, written in a high level language, and maintained by professional programmers in a large organization. The entire system had significantly more than ten million lines of code. This embedded computer application included numerous finite state machines and interfaces to other kinds of equipment.

This case study focused on faults discovered by customers after release. A module was considered *fault-prone* if any faults were discovered by customers, and *not fault-prone* otherwise. Fault data was collected at the module level by the problem reporting system. A module consisted of a set of related source code files.

This case study considered only updated modules, i.e., those with at least one update to source code since the prior release. These modules had several million lines of code in a few thousand modules. The proportion of modules with no faults among the updated modules was $\pi_{nfp} = 0.926$, and the proportion with at least one fault was $\pi_{fp} = 0.074$.

Enhanced Measurement for Early Risk Assessment of Latent Defects (EMERALD) is a sophisticated system of decision support tools used by software designers and managers to assess risk and improve software quality and reliability at Nortel (Northern Telecom) (Hudepohl et al., 1996). EMERALD provides access to metrics and software quality models based on those software metrics.

Our goal was a model of updated modules where predictions could be made after beta testing. Software product metrics were collected from source code by the EMERALD system, and were aggregated to the module level. Development process data was largely drawn from a development metrics data base derived from configuration management and problem reporting data. EMERALD interfaces with this data base. Our tools collected over sixty metrics. Pragmatic considerations usually determine the set of candidate independent variables. We do not advocate a particular set of software metrics for software quality models, to the exclusion of others recommended in the software engineering literature. Because marginal data collection costs are modest, we prefer the data-mining approach, analyzing a broad set of metrics, rather than limiting data collection according to predetermined research questions. Preliminary data analysis selected metrics that were appropriate candidates for modeling purposes. The candidate independent variables for a module consisted of those listed in Table III.

4. Case Study Results

We impartially divided the available data on updated modules into approximately equal *fit* and *test* data sets, so that each data set had sufficient observations for statistical purposes. The *fit* data set was used to build the model, and the *test* data set was used to evaluate its accuracy.

Stepwise logistic regression on the *fit* data set selected significant variables at the 15% level. The following model was estimated based on the *fit* data set. Variables are in order of significance; *VARSPNMX* and *CUST_FIX* were not significant.

$$\begin{aligned}\log\left(\frac{\hat{p}}{1-\hat{p}}\right) &= -4.9780 + 0.3940\, \mathit{UNQ\_DES} \\ &\quad +0.0165\, \mathit{FILINCUQ} + 1.0921\, \mathit{USAGE} \\ &\quad +0.9057\, \mathit{BETA\_PR} + 0.4845\, \mathit{BETA\_FIX} \\ &\quad -0.3958\, \mathit{VLO\_UPD} + 0.00039\, \mathit{SRC\_GRO} \\ &\quad +0.0137\, \mathit{LGPATH} - 0.00037\, \mathit{UPD\_CAR} \qquad (4)\end{aligned}$$

Table III. Data Elements

Symbol	Description
Software Product Metrics	
FILINCUQ	The number of distinct include-files, such as header files.
LGPATH	The base 2 logarithm of the number of independent paths in the control flow graph.
VARSPNMX	The maximum span of variables, where the span of a variable is the number of line of code between its first and last use.
USAGE	The deployment percentage of the module.
Development Process Metrics	
BETA_PR	The number of problems found in this module during beta testing of the current release.
BETA_FIX	Total number of different problems that were fixed for the current development cycle where the problems originated from issues found by beta testing of a prior release.
CUST_FIX	Total number of different problems that were fixed for the current development cycle where the problems originated from issues found by customers in a prior release.
SRC_GRO	Net increase in lines of code due to software changes.
UNQ_DES	Number of different designers that updated this module.
UPD_CAR	The total number of software updates that designers had made in their company careers at the time each updated this module.
VLO_UPD	Number of updates by designers who each had 10 or less total software updates in entire company career.

As an example, we chose c to balance the Type I and Type II misclassification rates. Based on resubstitution of the *fit* data set into the model, $c = 15.98$ yielded the Type I and Type II misclassification rates which were approximately equal.

Using the generalized-classification model on the *test* data set, Table IV shows the effect of various values of c on the accuracy of the model. (The row for $c = 15.98$ is bold.)

Figure 1 is a graph[1] of the fraction of modules that the model predicted to be *fault-prone*, i.e., the percent recommended for reliability enhancement, for various values of c, as listed in Table IV. For $c = 15.98$, the fraction recommended was 31.36%.

Figure 2 depicts the tradeoff between Type I and Type II misclassification rates for various values of c, as listed in Table IV. For $c = 15.98$, the Type I misclassification rate was 27.71% and the Type II misclassification rate was 22.96%.

[1] All figures in this paper use a logarithmic scale for c.

Table IV. Model Accuracy

test data set

c	Recommended by Model	Error Rate Type I	Type II	Effectiveness	Efficiency
0.10	0.49%	0.06%	94.07%	5.93%	100.00%
0.25	0.88%	0.30%	91.85%	8.15%	68.74%
0.50	1.37%	0.59%	88.89%	11.11%	59.99%
1.00	2.30%	1.07%	82.22%	17.78%	57.13%
2.00	3.67%	2.07%	76.30%	23.70%	47.75%
5.00	08.28%	5.92%	62.22%	37.78%	33.76%
10.00	18.64%	15.22%	38.52%	61.48%	24.40%
15.00	29.06%	25.46%	25.93%	74.07%	18.86%
15.98	**31.36%**	**27.71%**	**22.96%**	**77.04%**	**18.17%**
20.00	38.21%	34.87%	20.00%	80.00%	15.49%
25.00	48.85%	46.00%	15.56%	84.44%	12.79%
30.00	57.51%	55.18%	13.33%	86.67%	11.15%
50.00	77.08%	75.84%	7.41%	92.59%	8.89%
100.00	95.56%	95.20%	0.00%	100.00%	7.74%

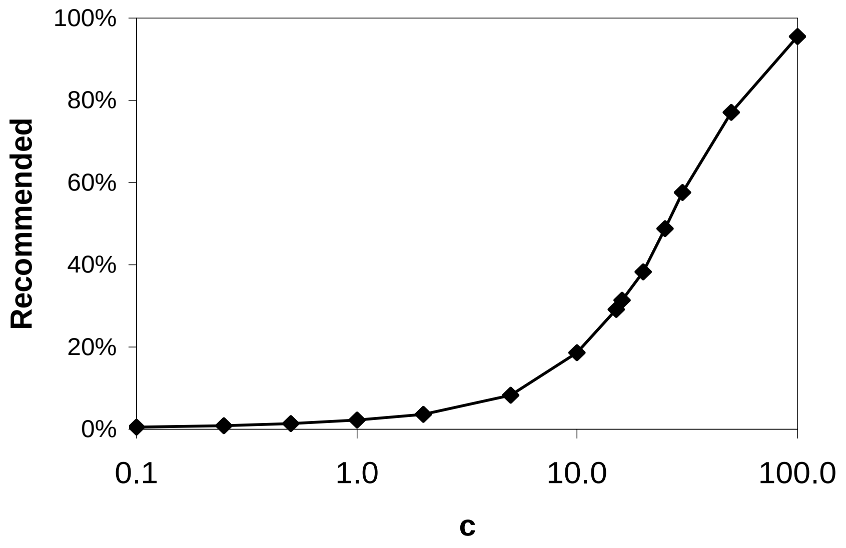

Figure 1. Modules Predicted to be Fault-Prone

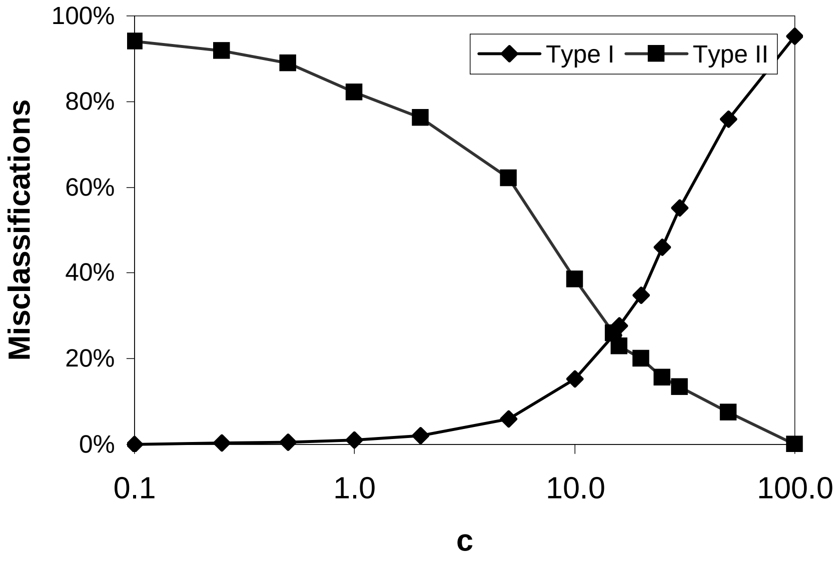

Figure 2. Misclassification Rates

Figure 3 shows the same tradeoff in terms of *effectiveness* and *efficiency*, as listed in Table IV. For $c = 15.98$, *effectiveness* was 77.04% and *efficiency* was 18.17%. In other words, if model recommendations were followed, over three quarters of the *fault-prone* modules would be enhanced before release, instead of during system operations, and more than one in six recommended modules actually were *fault-prone.* *Efficiency* of 18.17% was a marked improvement over selecting modules at random where $\pi_{fp} = 7.4\%$.

5. Cost-Benefit Analysis

Many classification techniques assume that the costs of Type I and Type II misclassifications are equal. However, in software engineering practice, the penalty for a Type II misclassification is often much more severe than for a Type I (Khoshgoftaar and Allen, 1998). The cost of a Type I misclassification, C_I, is the time and expense wasted trying to enhance the reliability of a module that is already *not fault-prone.* In other words, let C_I be the cost of reliability enhancement processes to discover faults in one module. A reliability enhancement technique, such as extra reviews (Ackerman et al., 1989; Jones, 1996; McCarthy et al., 1996) or extra testing, typically has modest direct cost per module. On the other hand, the cost of a Type II misclassification,

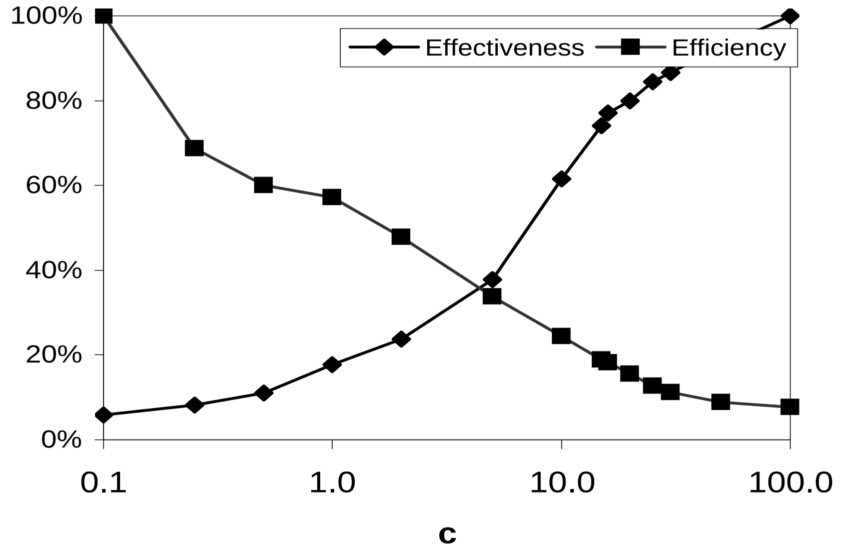

Figure 3. Effectiveness and Efficiency

C_{II}, is the lost opportunity to correct faults early. In other words, let C_{II} be the cost difference between fixing faults that were discovered by customers in one *fault-prone* module, and fixing faults discovered by reliability enhancement processes prior to release. The consequences of letting a fault go undetected until after release can be very expensive. C_{II} may include the cost of distributing and installing fixes at customer sites after release. One might even include the cost of consequential damages during operations due to software faults. This section takes the distinction between C_I and C_{II} into account as we analyze costs and benefits of using a software quality classification model.

Let us consider the cost of a Type I misclassification, C_I, to be a "cost unit". We call C_{II}/C_I the "cost ratio". In this paper, we model C_I and C_{II} as constants, and we assume the cost of a correct classification is zero. A more sophisticated cost model is a topic for future research.

We define the *Cost* of using a model as the extra effort entailed beyond the budgeted costs of normal development. In this paper, our focus is on using a model, and thus, we assume the costs of building a model, and data collection are already budgeted; we do not include them in the cost-benefit analysis. An expanded definition of *Cost* is a topic for future research. We define *Benefit* as the benefit due to using the model. From these quantities we can calculate *Profit* and return on

Table V. Example Predictions
Number of modules

	Predicted *nfp*		Predicted *fp*	
c	Total	Actually *fp*	Total	Actually *fp*
0.10	995	70	5	4
0.25	991	68	9	6
0.50	986	66	14	8
1.00	977	61	23	13
2.00	963	56	37	18
5.00	917	46	83	28
10.00	814	29	186	45
15.00	709	19	291	55
15.98	**686**	**17**	**314**	**57**
20.00	618	15	382	59
25.00	512	12	488	62
30.00	425	10	575	64
50.00	229	5	771	68
100.00	44	0	956	74

Total actually *nfp* = 926
Total actually *fp* = 74
Total modules = 1,000

investment, *ROI*.

$$Profit = Benefit - Cost \tag{5}$$

$$ROI = \frac{Profit}{Cost} \tag{6}$$

Future research will consider more sophisticated financial models, such as those incorporating inflation and the cost of money.

5.0.0.1. *Example.* Suppose we had 1,000 updated modules to analyze with the model from a project similar to our case study's or a subsequent release. We expect 74 to be actually *fault-prone*, because the prior probability was $\pi_{fp} = 0.074$. Table V applies the case-study results in Table IV to this hypothetical example, listing the number of modules predicted to be in each class and the number of actually *fault-prone* modules expected in each group, rounded to integers.

The number of modules that could receive reliability enhancement, such as extra reviews or additional testing, is often limited by practical factors, such as budget, schedule, or availability of key people. A graph like Figure 1 can be used to choose a practical value of c for the current project. For example, the fraction recommended for $c = 15.98$ was 31.4% of the modules. Suppose this was an acceptable fraction of the modules that could be enhanced. Let us consider the cost implications of misclassifications for $c = 15.98$ as an example. (Rows for $c = 15.98$ in tables below are bold.) This example illustrates the computation of a cost-benefit analysis. It is easily recomputed, if one were to choose a different value of c, and thereby different levels of *effectiveness* and *efficiency*, and/or if one had a different cost ratio.

Continuing our hypothetical example, as shown in Table V, for $c = 15.98$, the model identifies $314 = (1 - 0.2296)(74) + (0.2771)(1000 - 74)$ modules as *fault-prone.* When we apply a reliability enhancement technique to the 314 recommended modules, we discover that $57 = (1 - 0.2296)(74)$ are actually *fault-prone* and the remaining $257 = 314 - 57$ are not. Over the operational life of the software, customers find faults in $17 = 74 - 57$ additional modules which the model indicated as *not fault-prone.*

5.1. Short-Term Analysis

A short-term perspective on costs and benefits focuses on costs prior to release. Our short-term analysis considers the costs of fixing all customer-discovered faults to be already budgeted as part of the baseline maintenance process. Consequently, *Cost* and *Benefit* have short-term definitions here. *Cost* is defined as the direct costs of reliability enhancement for all modules recommended by the model.

$$Cost \;=\; C_I\,(\hat{n}_{nfp,fp} + \hat{n}_{fp,fp}) \qquad (7)$$

If one does not follow model recommendations, then $Cost = 0$. *Benefit* is defined as the cost-avoidance of maintenance-phase fixes for the *fault-prone* modules that are recommended by the model.

$$Benefit \;=\; C_{II}\,\hat{n}_{fp,fp} \qquad (8)$$

If one does not follow model recommendations, the $Benefit = 0$. Thus, according to Equation (5), *Profit* is calculated as

$$Profit \;=\; (C_{II} - C_I)\,\hat{n}_{fp,fp} - C_I\,\hat{n}_{nfp,fp} \qquad (9)$$

If the model is not used, $Profit = 0$. “Nothing ventured, nothing gained.”

The following sections discuss the example with a short-term analysis at cost ratios of 10 and 200. Figures present additional results for cost ratios of 50 and 100.

5.1.1. *Example: Cost Ratio = 10*

Let us suppose that there is a benefit of $C_{II} = 10$ units for identifying a *fault-prone* module at or prior to the end of beta testing rather than

Table VI. Short-Term Cost-Benefit Analysis
Cost Ratio, $C_{II}/C_I = 10$

c	Cost	Benefit	Profit	ROI
0.10	5	40	35	7.0000
0.25	9	60	51	5.6667
0.50	14	80	66	4.7143
1.00	23	130	107	4.6522
2.00	37	180	143	3.8649
5.00	83	280	197	2.3735
10.00	186	450	264	1.4194
15.00	291	550	259	0.8900
15.98	**314**	**570**	**256**	**0.8153**
20.00	382	590	208	0.5445
25.00	488	620	132	0.2705
30.00	575	640	65	0.1130
50.00	771	680	-91	-0.1180
100.00	956	740	-216	-0.2259

Cost of not using model = 0
Benefit of not using model = 0

waiting until a fault is discovered in that module by a customer. This cost ratio (C_{II}/C_I) of 10:1 is only an example, but we believe that 10:1 is plausible for many projects. Each project must determine its own cost ratio.

Table VI shows relationships among c, profit, and return on investment for a hypothetical cost ratio of $C_{II}/C_I = 10$. For $c = 15.98$, the cost is one unit per recommended module, 314 units, and the benefit is the number of *fault-prone* modules enhanced times the value of each enhancement, $570 = 57 \times 10$.

Figure 4 depicts the relationship between *Profit* and c for various cost ratios, including the profit listed in Table VI. For a cost ratio of 10, profit's maximum is at $c \approx 10$. For higher values of c, the Type I misclassification rate becomes very high (see Figure 2), overwhelming any benefit from identifying more *fault-prone* modules, due to the lower Type II rate. At very high values of c, *Profit* becomes a loss (not shown in the figure). At $c = 15.98$, *Profit* $= 256$, which is near the maximum.

Figure 5 shows the relationship between c and short-term return on investment, at a cost ratio of 10, as listed in Table VI. Figures 6 and 7 show the the same relationship for cost ratios of 50 and 100 respectively.

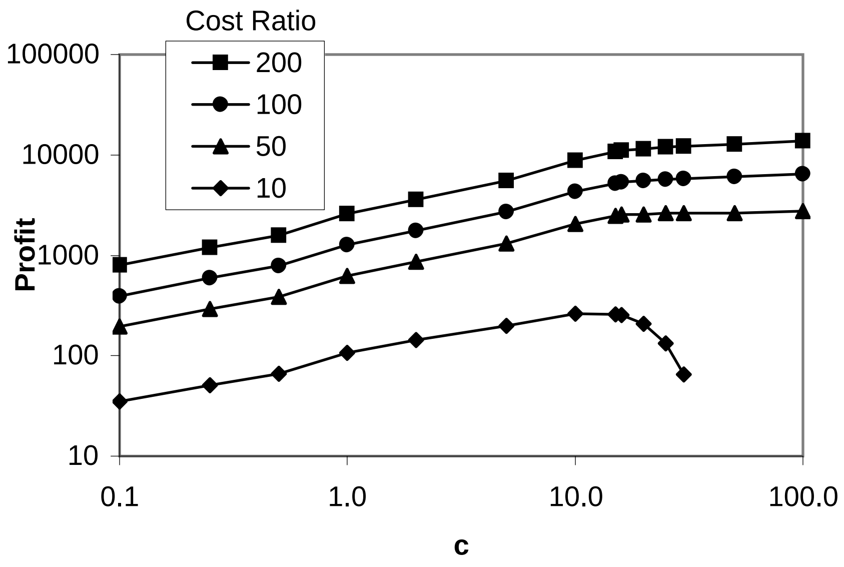

Figure 4. Profit

At $c \approx 10$, which maximizes profit for a cost ratio of 10, we see that $ROI = 1.42$, and at $c = 15.98$, $ROI = 0.82$.

5.1.2. *Example: Cost Ratio = 200*
Let us suppose that there is a benefit of $C_{II} = 200$ units for identifying a *fault-prone* module with the model rather than waiting for a customer to discover faults. Even though this cost ratio (C_{II}/C_I) of 200:1 is only an illustration here, we believe that costs associated with customer-discovered faults for the case study's telecommunications system are so high that 200:1 is plausible. The value of the cost ratio is determined by the special circumstances of a project.

Table VII shows profit and return on investment for various values of c for a cost ratio of 200. At $c = 15.98$, like the above example, the cost of reliability enhancement when implementing this model's recommendations is one unit per module, 314 units. The benefit for using the quality model is $11,400 = 57 \times 200$ units for identifying some *fault-prone* modules correctly.

Figure 4 includes profit as a function of c, as listed in Table VII. At a cost ratio of 200, profit is maximized when reliability enhancement processes are applied to all modules, i.e., when c is so large that the Type II misclassification rate is zero (see Figure 2). In other words, the value of discovering faults early, which would otherwise be discovered by customers, is so high that it pays to enhance the reliability of as

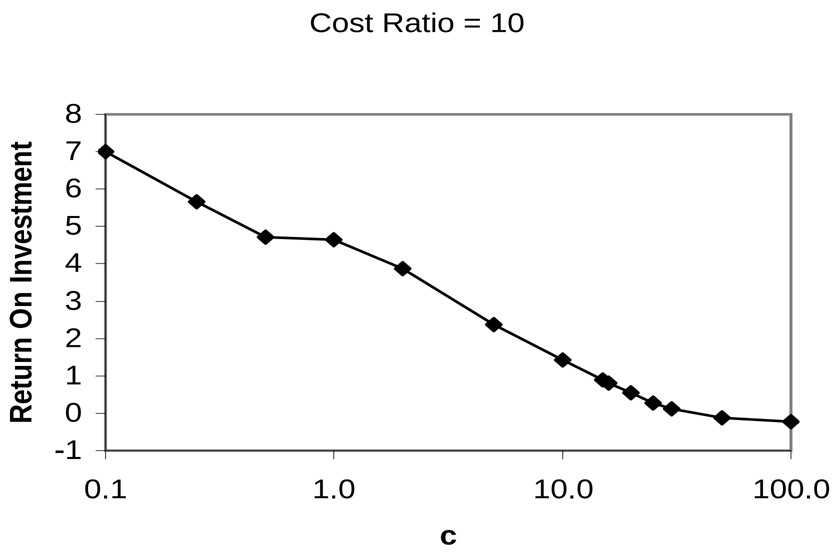

Figure 5. Short-Term Return On Investment

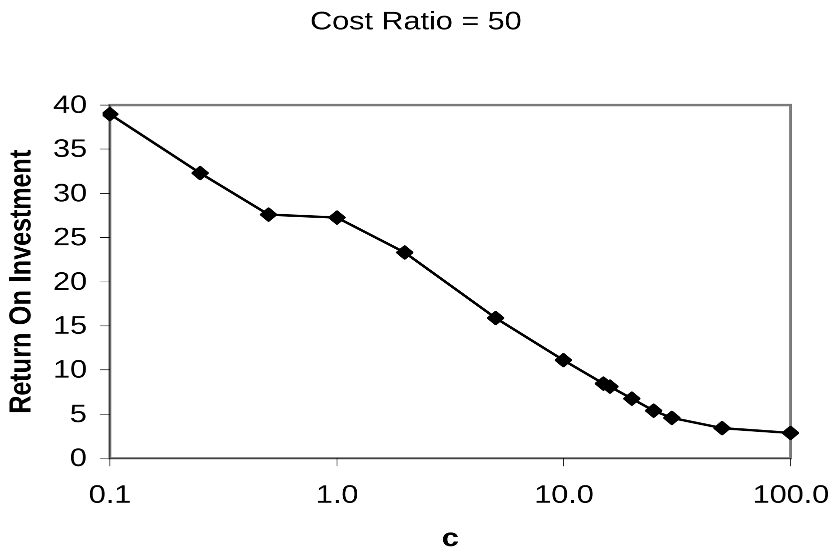

Figure 6. Short-Term Return On Investment

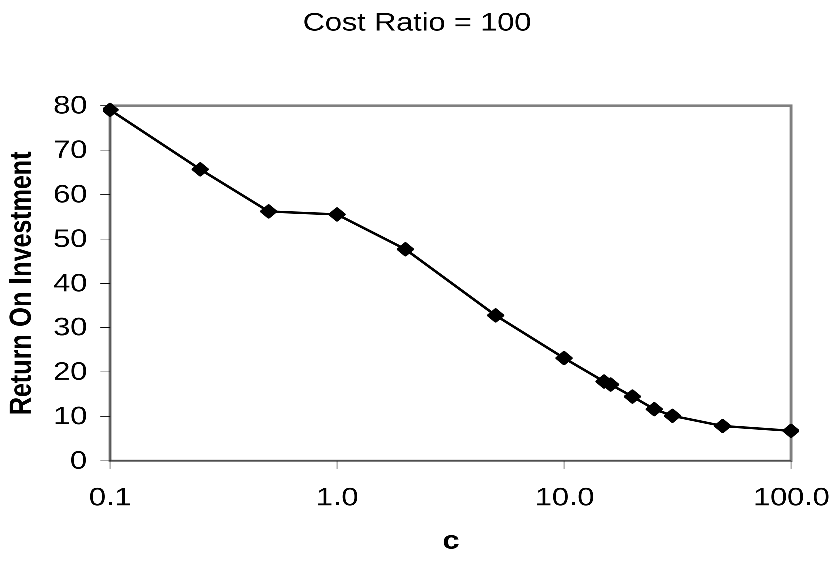

Figure 7. Short-Term Return On Investment

Table VII. Short-Term Cost-Benefit Analysis
Cost Ratio, $C_{II}/C_I = 200$

c	Cost	Benefit	Profit	ROI
0.10	5	800	795	159.0000
0.25	9	1200	1191	132.3333
0.50	14	1600	1586	113.2857
1.00	23	2600	2577	112.0435
2.00	37	3600	3563	96.2973
5.00	83	5600	5517	66.4699
10.00	186	9000	8814	47.3871
15.00	291	11000	10709	36.8007
15.98	**314**	**11400**	**11086**	**35.3057**
20.00	382	11800	11418	29.8901
25.00	488	12400	11912	24.4098
30.00	575	12800	12225	21.2609
50.00	771	13600	12829	16.6394
100.00	956	14800	13844	14.4812

Cost of not using model = 0
Benefit of not using model = 0

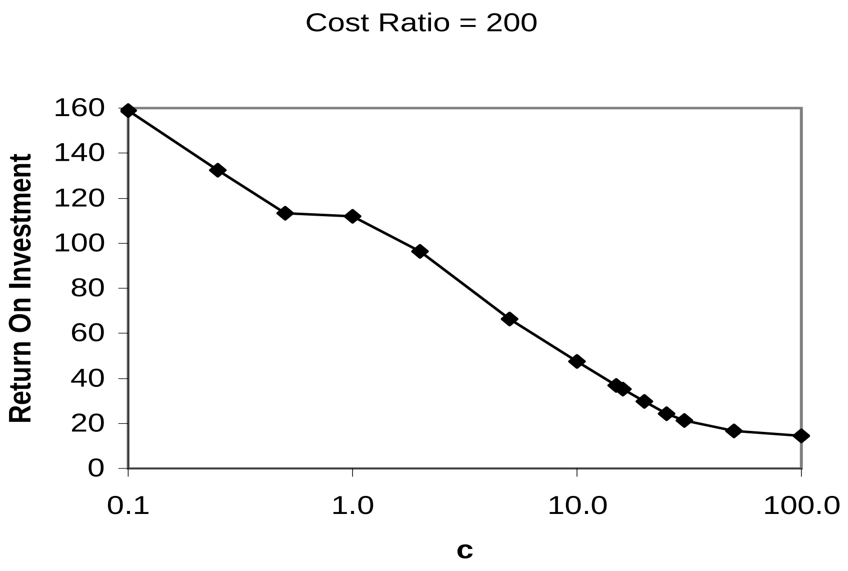

Figure 8. Short-Term Return On Investment

many modules as possible. However, it is usually not practical to apply special processes to all modules. In this example, we supposed that 31.4% of the modules is the practical maximum that the model should recommend, corresponding to $c = 15.98$. Thus, the profit of using the model is $11,086 = 11,400 - 314$ units.

Figure 8 shows the relationship between c and return on investment for a cost ratio of 200, as listed in Table VII. The shape of Figure 8 is similar to other cost ratios, but ROI is approximately scaled by the cost ratio. For small values of c, the Type II misclassification rate is so high that very few modules were recommended for enhancement. Because most of the recommendations were correct (see Figure 2), ROI approaches the cost ratio. However, an extreme classification model, such as this, that finds only a few fault-prone modules is not very useful to the project. At a value of $c = 15.98$, the return on investment is $ROI = 35.31 = 11,086/314$.

For comparison, consider enhancing 314 randomly chosen modules. The number of *fault-prone* modules in this group is $23 = 314 \times 0.074$. The benefit is $4,600 = 23 \times 200$ units. The profit is $4,286 = 4,600 - 314$ units. The return on investment is $ROI = 13.65 = 4,286/314$. Thus, using the model more than doubles the profit (11,086 *vs.* 4,286) and return on investment (35.31 *vs.* 13.65) of randomly selecting the same number of modules for reliability enhancement.

5.2. Long-Term Analysis

A long-term perspective on costs and benefits focuses on costs throughout the life cycle, including the maintenance phase. Our long-term analysis considers the costs of fixing all faults, irrespective of when they are discovered. Some are discovered before release due to using the model, and others are discovered by customers after release. Consequently, *Cost* and *Benefit* have long-term definitions here. The long-term perspective is especially appropriate for the maintenance phase.

Cost is defined as the cost of enhancements recommended by the model, plus the cost of fixing faults discovered by customers.

$$Cost = C_I \left(\hat{n}_{nfp,fp} + \hat{n}_{fp,fp}\right) + C_{II}\, \hat{n}_{fp,nfp} \tag{10}$$

If one does not follow model recommendations, then $Cost = C_{II}\, n_{fp}$. *Benefit* is defined as the value of diagnosing and fixing all faults, as if the model were not used, namely, the number of modules times the cost of fixing a customer-discovered fault.

$$Benefit = C_{II} \left(\hat{n}_{fp,nfp} + \hat{n}_{fp,fp}\right) \tag{11}$$

$$Benefit = C_{II}\, n_{fp} \tag{12}$$

Thus, *Benefit* does not depend on the misclassification rates, $\Pr(nfp|fp)$ and $\Pr(fp|nfp)$, and consequently, does not depend on c. If one does not follow model recommendations, then the benefit is still given by Equation (12). According to Equations (5), (10), and (12), *Profit* is calculated as

$$Profit = C_{II}\, n_{fp} - C_I \left(\hat{n}_{nfp,fp} + \hat{n}_{fp,fp}\right) - C_{II}\, \hat{n}_{fp,nfp} \tag{13}$$

$$Profit = (C_{II} - C_I)\, \hat{n}_{fp,fp} - C_I\, \hat{n}_{nfp,fp} \tag{14}$$

Equations (9) and (14) reveal that the *Profit* of using the model is the same for both the short-term perspective and the long-term perspective. If the model is not used, then cost equals benefit and $Profit = 0$. "Nothing ventured, nothing gained."

The following sections discuss the example with a long-term analysis at cost ratios of 10 and 200. Figures show additional results for cost ratios of 50 and 100.

5.2.1. *Example: Cost Ratio = 10*

As we did above, let us suppose that $C_{II}/C_I = 10$. Table VIII shows the resulting relationships among c, profit, and return on investment. For $c = 15.98$, according to Equation (10), the cost is 484 units, and the benefit is $740 = 74 \times 10$.

Table VIII. Long-Term Cost-Benefit Analysis
Cost Ratio, $C_{II}/C_I = 10$

c	Cost	Benefit	Profit	ROI
0.10	705	740	35	0.0496
0.25	689	740	51	0.0740
0.50	674	740	66	0.0979
1.00	633	740	107	0.1690
2.00	597	740	143	0.2395
5.00	543	740	197	0.3628
10.00	466	740	274	0.5880
15.00	481	740	259	0.5385
15.98	**484**	**740**	**256**	**0.5289**
20.00	532	740	208	0.3910
25.00	608	740	132	0.2171
30.00	675	740	65	0.0963
50.00	821	740	-81	-0.0987
100.00	956	740	-216	-0.2259

Cost of not using model = 740
Benefit of not using model = 740

For a given cost ratio, the long-term profit is the same as the short-term profit. Figure 4 shows the profit for a cost ratio of 10:1.

Figure 9 depicts the long-term return on investment as a function of c at a cost ratio of 10. Figures 10 and 11 show results for cost ratios of 50 and 100 respectively. Due to the long-term perspective, at a cost ratio of 10, return on investment for low values of c is also low. The c maximizing ROI is the same as the value maximizing profits. High values of c result in losses and ROI is negative. At $c \approx 10$ which maximizes profits, we see that $ROI = 0.59$, and for $c = 15.98$, $ROI = 0.53$.

5.2.2. *Example: Cost Ratio = 200*

As we did above, let us suppose that $C_{II}/C_I = 200$. Table IX shows relationships among c, profit, and return on investment for this example. For $c = 15.98$, according to Equation (10), the cost is 3,714 units, and the benefit is $14,800 = 74 \times 200$.

Because the long-term profit is the same as the short-term profit, Figure 4 shows the profit for a cost-ratio of 200:1.

Figure 12 shows the relationship between c and return on investment. Because of the long-term perspective, for a cost ratio of 200,

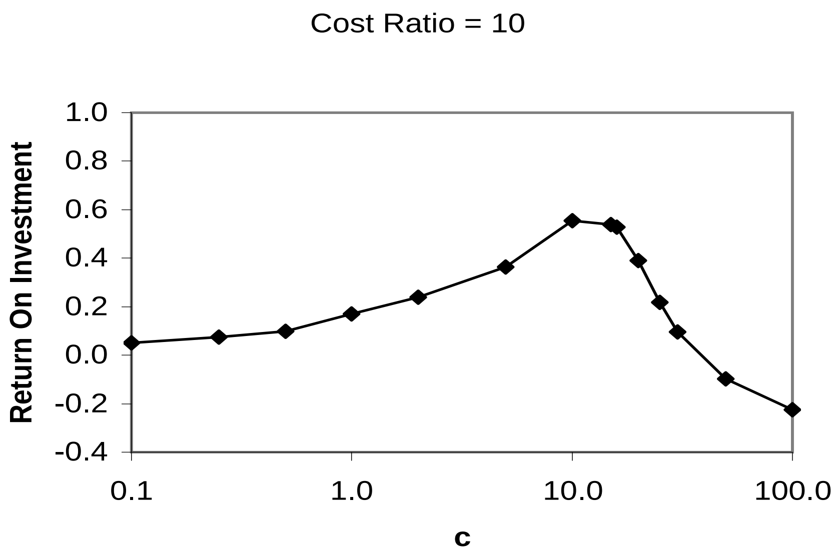

Figure 9. Long-Term Return On Investment

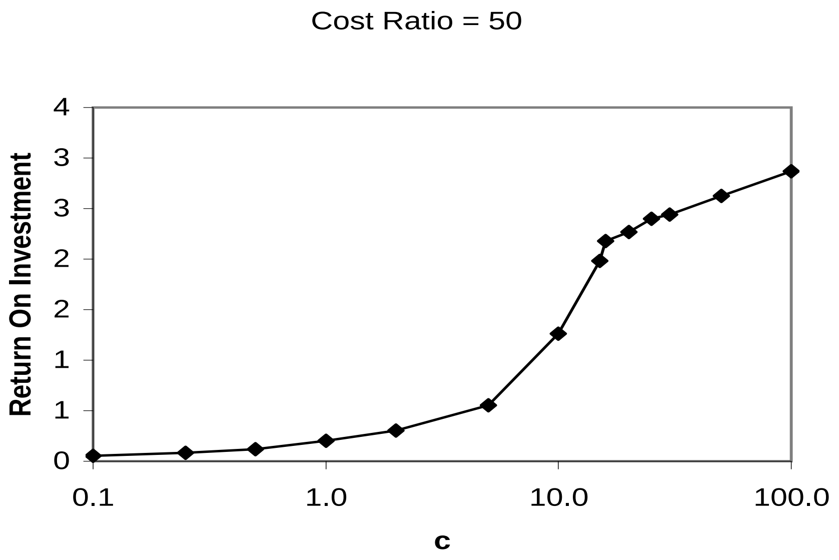

Figure 10. Long-Term Return On Investment

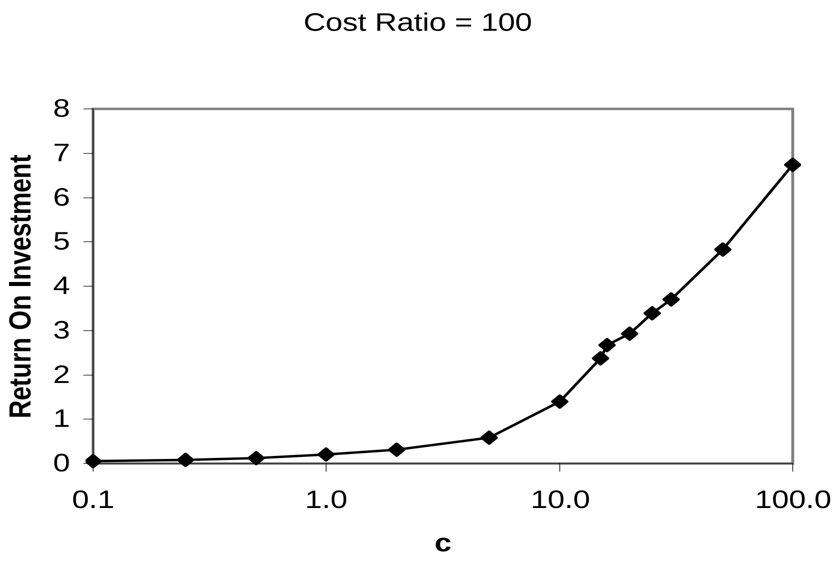

Figure 11. Long-Term Return On Investment

Table IX. Long-Term Cost-Benefit Analysis
Cost Ratio, $C_{II}/C_{I} = 200$

c	Cost	Benefit	Profit	ROI
0.10	14005	14800	795	0.0568
0.25	13609	14800	1191	0.0875
0.50	13214	14800	1586	0.1200
1.00	12223	14800	2577	0.2108
2.00	11237	14800	3563	0.3171
5.00	9283	14800	5517	0.5943
10.00	5786	14800	9014	1.5579
15.00	4091	14800	10709	2.6177
15.98	**3714**	**14800**	**11086**	**2.9849**
20.00	3382	14800	11418	3.3761
25.00	2888	14800	11912	4.1247
30.00	2575	14800	12225	4.7476
50.00	1771	14800	13029	7.3569
100.00	956	14800	13844	14.4812

Cost of not using model = 14,800
Benefit of not using model = 14,800

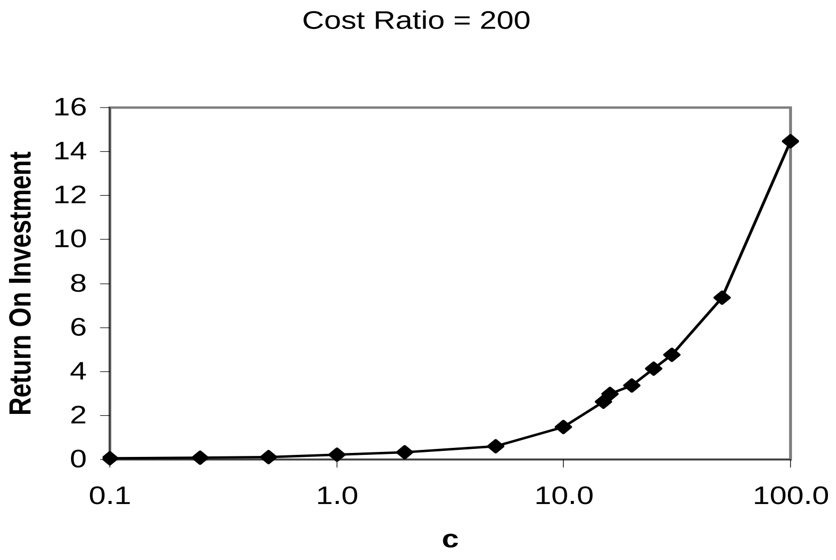

Figure 12. Long-Term Return On Investment

low values of c result in low ROI. Extremely high values of c yield the highest ROI, because at $C_{II} = 200$, it is profitable to enhance the reliability of as many modules as possible. However, special enhancement of all modules is not practical. At a value of $c = 15.98$, the return on investment is $ROI = 2.98 = 11,086/3,714$.

For comparison, consider enhancing 314 randomly chosen modules: The cost of using the model is $Cost = 10,514 = 314 + (200)(74 - 23)$ units, and $Benefit = 14,800 = 74 \times 200$ units. $Profit = 4,286 = 14,800 - 10,514$ units, and the return on investment is $ROI = 0.41 = 4,286/10,514$. Like the short-term analysis, using the model more than doubles the long-term profit (11,086 *vs.* 4,286) and yields more than seven times the return on investment (2.98 *vs.* 0.41) of randomly selecting the same number of modules.

6. Conclusions

Many businesses are finding that development of reliable software has become increasingly important in today's marketplace. A static model of the software development life cycle that treats all modules similarly, is becoming inadequate to the task. The need to achieve a new level of quality is mandating reengineering of software development processes to a dynamic model.

Business process reengineering (BPR) is the popular term for radical redesign of business processes (Hammer and Champy, 1993). This paper focuses on reengineering the business processes that produce commercial software to take advantage of software-quality modeling technology. In particular, this paper presents how to analyze the costs and benefits of the accuracy of a software quality classification model. The cost-benefit analysis method considers both a short-term and long-term perspective. The long-term perspective is especially well-suited to the maintenance phase. A cost-benefit analysis gives insight into the implications of implementing the recommendations of a software quality model in the context of a dynamic development process.

A case study illustrated the method. The case study examined a very large legacy telecommunications system with strict reliability requirements. A small fraction of the modules in the case study had faults discovered by customers (less than 8%). This small set of modules can be difficult to identify. Logistic regression in conjunction with a generalized classification model which we have proposed predicted whether each module was *fault-prone* or not (Khoshgoftaar and Allen, 2000). The example illustrated how a dynamic process using a software quality model can result in much higher profit and return on investment than a static approach that randomly selects modules for reliability enhancement. The case study shows that the approach used here scales up to large systems. We anticipate that a refinement of this model will be incorporated into Nortel's EMERALD system.

Future research will refine the measure of cost and benefit for improved realism, will validate these results when data on subsequent releases become available, and will study other classification techniques.

Acknowledgements

We thank the EMERALD team at Nortel for collecting the data. This work was supported by a grant from Nortel, through the Software Reliability Department, Research Triangle Park, NC. The findings and opinions in this paper belong solely to the authors, and are not necessarily those of the sponsor. Moreover, our results do not in any way reflect the quality of the sponsor's software products.

References

Ackerman, A. F., L. S. Buchwald, and F. H. Lewski: 1989, 'Software Inspections: An effective verification process'. *IEEE Software* **6**(3), 31–36.

Biazzo, S.: 1998, 'A Critical Examination of the Business Process Re-engineering Phenomenon'. *Internation Journal of Operations and Production Management* **18**(9/10), 1000–1016.

Fenton, N. E. and S. L. Pfleeger: 1997, *Software Metrics: A Rigorous and Practical Approach*. London: PWS Publishing, 2d edition.

Hammer, M.: 1990, 'Reengineering Work: Don't Automate, Obliterate'. *Harvard Business Review* **68**(4), 104–112.

Hammer, M. and J. Champy: 1993, *Reengineering the Corporation: A Manifesto for Business Revolution*. New York: Harper-Collins.

Hosmer, Jr., D. W. and S. Lemeshow: 1989, *Applied Logistic Regression*. New York: John Wiley & Sons.

Hudepohl, J. P.: 1990, 'Measurement of Software Service Quality for Large Telecommunications Systems'. *IEEE Journal of Selected Areas in Communications* **8**(2), 210–218.

Hudepohl, J. P., S. J. Aud, T. M. Khoshgoftaar, E. B. Allen, and J. Mayrand: 1996, 'EMERALD: Software Metrics and Models on the Desktop'. *IEEE Software* **13**(5), 56–60.

Hudepohl, J. P., W. Snipes, T. Hollack, and W. Jones: 1992, 'A Methodology to Improve Switching System Software Service Quality and Reliability'. In: *Proceedings of IEEE Global Telecommunications Conference*. pp. 1671–1678.

Jones, C.: 1996, 'Software Defect-Removal Efficiency'. *Computer* **29**(4), 94–95.

Khoshgoftaar, T. M. and E. B. Allen: 1998, 'Classification of Fault-Prone Software Modules: Prior Probabilities, Costs, and Model Evaluation'. *Empirical Software Engineering: An International Journal* **3**(3), 275–298.

Khoshgoftaar, T. M. and E. B. Allen: 2000, 'A Practical Classification Rule for Software Quality Models'. *IEEE Transactions on Reliability* **49**(2). In press.

Khoshgoftaar, T. M., E. B. Allen, R. Halstead, G. P. Trio, and R. Flass: 1998, 'Process Measures for Predicting Software Quality'. *Computer* **31**(4), 66–72.

McCarthy, P., A. Porter, H. Siy, and L. G. Votta: 1996, 'An Experiment to Assess Cost-Benefits of Inspection Meetings and their Alternatives'. In: *Proceedings of the Third International Software Metrics Symposium*. Berlin.

Motwani, J., A. Kumar, J. Jiang, and M. Youssef: 1998, 'Business Process Reengineering: A Theoretical Framework and an Integrated Model'. *Internation Journal of Operations and Production Management* **18**(9/10), 964–977.

Orman, L. V.: 1998, 'A Model Management Approach to Business Process Reengineering'. *Journal of Management Information Systems* **15**(1), 187–212.

Seber, G. A. F.: 1977, *Linear Regression Analysis*. New York: John Wiley and Sons.

Authors' Vitae

Taghi M. Khoshgoftaar
is a professor of the Department of Computer Science and Engineering, Florida Atlantic University and the Director of the Empirical Software Engineering Laboratory. His research interests are in software engineering, software complexity metrics and measurements, software reliability and quality engineering, computational intelligence, computer performance evaluation, multimedia systems, and statistical modeling. He has

published more than 150 refereed papers in these areas. He has been a principal investigator and project leader in a number of projects with industry, government, and other research-sponsoring agencies. He is a member of the Association for Computing Machinery, the American Statistical Association, the IEEE Computer Society, and IEEE Reliability Society. He served as the general chair of the 1999 International Symposium on Software Reliability Engineering (ISSRE'99), and the general chair of the 2001 International Conference on Engineering of Computer Based Systems. He has served on technical program committees of various international conferences, symposia, and workshops. He has served as North American editor of the *Software Quality Journal*, and is on the editorial board of the *Journal of Multimedia Tools and Applications.*

Edward B. Allen
received the B.S. degree in engineering from Brown University, Providence, Rhode Island USA, in 1971, the M.S. degree in systems engineering from the University of Pennsylvania, Philadelphia, Pennsylvania USA, in 1973, and the Ph.D. degree in computer science from Florida Atlantic University, Boca Raton, Florida USA, in 1995. He is currently an assistant professor in the Department of Computer Science at Mississippi State University. He began his career as a programmer with the U.S. Army. From 1974 to 1983, he performed systems engineering and software engineering on military systems, first for Planning Research Corp. and then for Sperry Corp. From 1983 to 1992, he developed corporate data processing systems for Glenbeigh, Inc., a specialty health care company. His research interests include software measurement, software process, software quality, and computer performance modeling. He has more than 60 refereed publications in these areas. He is a member of the IEEE Computer Society and the Association for Computing Machinery.

Wendell D. Jones
received his Ph.D. in Mathematical Science (Statistics emphasis) from Clemson University in 1987. He is Chief Researcher of EMERALD, a Business Unit of Nortel Networks.

John P. Hudepohl
is manager of EMERALD, a Business Unit of Nortel Networks. He has more than 20 years experience in the reliability, maintainability and quality fields for both hardware and software.

Address for Offprints: Readers may contact the authors through Taghi M. Khoshgoftaar, Empirical Software Engineering Laboratory, Dept. of Computer Science and Engineering, Florida Atlantic University, Boca Raton, FL 33431 USA. Phone: (561)297-3994, Fax: (561)297-2800, Email: taghi@cse.fau.edu, URL: www.cse.fau.edu/esel/.