ELSEVIER

Fuzzy Sets and Systems 127 (2002) 199–208

FUZZY sets and systems

www.elsevier.com/locate/fss

Empirical evaluation of a fuzzy logic-based software quality prediction model [☆]

Sun Sup So [*], Sung Deok Cha, Yong Rae Kwon

Department of Electrical Engineering & Computer Science (EECS), Korea Advanced Institute of Science and Technology, 373-1, Kusong-dong, Yusong-gu, Taejon 305-701, South Korea

Received 14 October 1998; received in revised form 17 February 2001; accepted 11 May 2001

Abstract

Software inspection, due to its repeated success on industrial applications, has now become an industry standard practice. Recently, researchers began analyzing inspection data to obtain insights on how software processes can be improved. For example, project managers need to identify potentially error-prone software components so that limited project resource may be optimally allocated. This paper proposes an automated and fuzzy logic-based approach to satisfy such a need. Fuzzy logic offers significant advantages over other approaches due to its ability to naturally represent qualitative aspect of inspection data and apply flexible inference rules. In order to empirically evaluate the effectiveness of our approach, we have analyzed published inspection data and the ones collected from two separate inspection experiments which we had conducted. χ^2 analysis is applied to statistically demonstrate validity of the proposed quality prediction model. © 2002 Elsevier Science B.V. All rights reserved.

Keywords: Software inspection; Quality prediction; Software metrics; Statistical process control; Fuzzy logic; Inspection metric

1. Introduction

Software inspection [14] is widely considered as an essential practice to develop high-quality software in cost-effective manner. Much of past studies focused on measuring the effectiveness of inspections, improving inspection processes, and empirically comparing effectiveness of inspection to other software quality assurance techniques [23,24,27].

Recently, however, researchers began applying statistical analysis on inspection data so as to improve overall software development processes and, ultimately, software quality and productivity. Christenson et al. [6] proposed a set of control charts to estimate the number of defects remaining in the inspected products. Information on the number of residual errors can guide project managers in determining which products need to be inspected again and how the current inspection process might be

[☆] S.D. Cha is affiliated to the Advanced Information Technology Research Center (AITrc) in KAIST. This work was supported, in part, by the Korea Science and Engineering Foundation (KOSEF) through the telepresence project at the AITrc.

[*] Corresponding author. Tel.: +82-42-869-5558; fax: +82-42-869-3510.

E-mail address: triples@salmosa.kaist.ac.kr (S.S. So).

improved. Similarly, Barnard and Price [1] have defined nine key metrics that project managers can use to plan, monitor, and improve inspection processes. Furthermore, they demonstrated inspection data to be useful in assessing the number of residual faults in the code after inspection.

Although statistical methods have been extensively used in the area of reliability and quality, they suffer from the unrealistic assumption of error distribution and thus often generate unsatisfactory solutions. Errors tend to be clustered rather than being evenly distributed [5,17]. Project managers, therefore, are usually more concerned about the degree of error-proneness of the modules rather than the precise estimation on the number of residual errors. If it were possible to identify potentially error-prone modules with relatively high degree of accuracy *at little or no extra cost* by analyzing existing inspection data, project managers could use such findings to maximize software development productivity. For example, they may choose to test potentially erroneous modules more extensively than others. Perhaps, formal methods could be selectively applied on the potentially error-prone modules.

Ebenau [9] showed that statistical analysis of inspection data could be used to predict potentially erroneous software components. Inspection metrics such as defect density, inspection rate, preparation rates, and the size of components were used. He created a set of control charts from which project managers can manually determine whether or not a module is potentially erroneous. Unfortunately, this approach, in addition to being inefficient, might lead to inconsistent conclusions because the boundary values used in the process is arbitrary and vary from one manager to another depending on one's project experience. Software quality has qualitative characteristics rather than quantitative in nature. Furthermore, the classes of software components are often not well separated and there are no well-defined boundaries. An automated and reproducible model to estimate software quality attributes is need.

In this paper, we describe how fuzzy logic can be used to analyze inspection data and contribute to the task of *predicting* error-prone modules. Our paper is organized as follows. In Section 2, we briefly review attempts to use inspection data to improve software development processes. In particular, we focus our review on techniques to identify potentially error-prone software components. In Section 3, we describe our fuzzy logic-based approach. We describe fuzzy membership functions used in analyzing inspection data as well as inference rules used in determining if a module should be considered error-prone. In Section 4, we evaluate effectiveness of our quality prediction model by applying it on inspection data published by Ebenau [9] as well as the ones collected from two controlled experiments the authors had previously conducted [3,22–24]. Section 5 concludes the paper and describes promising topics worthy of further research.

2. Related works

Much of past research on inspection focused on empirically validating cost-effectiveness of inspection methods [7,20,26,27]. Some variations were proposed in order to further improve inspection efficiency [2,25]. Recently, empirical studies compared cost-effectiveness of inspection method against other error detection techniques such as voting, instrumentation, testing, data-flow analysis, or code reading by stepwise refinement using the same set of programs [23,24].

Another trend in research on inspection is to apply statistical analysis on inspection data to obtain insights on how software development processes can be improved. For example, Barnard and Price [1] identified nine key metrics useful in planning, monitoring, controlling, and improving inspection processes. For example, an answer to the question "what is the quality of the inspected software?" is derived based on metrics such as average number of errors detected per KLoC (thousand lines of code), average inspection rate, and average preparation rate. Current inspection data is compared to the baseline values (e.g., historical data) accumulated from past inspection sessions. If values measuring current inspection quality are lower than the expected baseline figures, project managers may conclude that current inspection processes are ineffective and take necessary corrective actions.

Christenson et al. [6] used inspection metrics to identify characteristics of effective, questionable, or marginal inspection sessions. Classification was based primarily on preparation effort and inspection rate.

Such information was used to improve inspection process by issuing guidelines on the amount of preparation effort needed prior to formal inspection meetings and inspection rate as project goals. Furthermore, they have used inspection data to estimate density of errors remaining in the code to help project managers decide whether or not re-inspection is warranted.

It has been claimed that errors tend to cluster [5,17]. An analysis of faults detected in 27 launch interceptor program (LIP) versions supports such claim to be true. Each LIP version, developed based on the same specification, contains 15 procedures implementing various algorithms used in arriving at the firing decision [3]. There are a total of 405 modules (plus some internal routines) in the LIP programs and there are 64 known errors. Examination of error distribution revealed that about 10% of the modules contained more than 85% of known errors. Because error distribution in software does not follow normal or Poisson distribution, one cannot reliably depend on statistical analysis to estimate the number of remaining errors in the code.

Other researchers tried to predict quality of software components using metrics such as cyclomatic complexity, fan-in/fan-out, and Halstead's metrics. Specific techniques include classification trees [19], discriminant analysis [17], neural nets [16], and rule-based fuzzy logic [4,11]. Unfortunately, it is difficult to objectively rank effectiveness of such approaches because these studies used data obtained from different projects.

Ebert [12,13] evaluated the techniques listed above using data collected from the same projects using the number of misclassification errors (e.g., classifying error-prone modules as reliable modules, or vice versa) and χ^2 values. He found that fuzzy logic-based approach was the most effective and argued that there are several advantages. A prototype tool can be easily developed even when little training data are available. Furthermore, expert heuristics can be directly incorporated, and the membership functions can be tuned according to the project environment.

While it is true that each of techniques described above has merit in predicting quality attributes of software components, there are additional factors affecting software quality. These may include, but not necessarily limited to, software structure, software complexity, developer's experience, development process, software size, etc. No single factor can accurately estimate the number of defects or defect-proneness. There is no "best" metric for a single factor. Therefore, we need to take various contributing factors into consideration.

Ebenau was the first one who employed inspection metrics to identify modules that are likely to be error-prone [9]. He created a set of control charts using the inspection metrics such as defect density, inspection rate, preparation rates, or the size of components, from which project managers can manually decide the troublesome components. Major defect densities are first evaluated with respect to the upper and lower threshold values. Defect densities are expected to be located between the two threshold values. Modules are classified as being error-prone when upper threshold value is exceeded. For the modules whose defect density value lies outside the expected boundary, further analyses are conducted using additional metrics such as inspection rate, preparation rate, and product size before making the final decision. Unfortunately, this approach might lead to inconsistent conclusions because the selection of boundary of values used in the process is arbitrary. In addition, the process is applied manually and therefore inefficient.

3. Fuzzy logic-based quality prediction model

When developing a software quality prediction model, one must first identify factors that strongly influence software quality and the number of residual errors. Unfortunately, it is extremely difficult, if not impossible, to accurately identify relevant quality factors. Furthermore, the degree of influence is imprecise in nature. That is, although exact and discrete metric data are used, inference rules (or heuristics) used in drawing conclusions may be *fuzzy in nature*. Suppose, for example, that an inspection team reported an inspection rate of over 400 lines of code per hour (LoC/h) whereas typical inspection rate ranges from 150 to 200 LoC/h [27]. One can convincingly argue that such inspection rate *significantly* exceeds the reported average from industrial applications, and experts will most likely agree unanimously with the conclusion. However, such assessment is fuzzy because the term "significantly" cannot be objectively quantified. Moreover, if a team reports inspection rate of 275 LoC/h, experts are likely to differ in their opinions as to whether or not the inspection rate exceeded the industrial norm and by how much it exceeded. In other words, decision boundary is not well defined.

Linguistic or non-numeric information needs a new methodology for it to be properly analyzed. Pedrycz [18] shows that the methods for computational intelligence including fuzzy sets and neural network help exploit the notion of imprecision and approximate reasoning.

Due to its natural ability to model imprecise and fuzzy aspect of data and rules, fuzzy logic is an attractive alternative in situations where approximate reasoning is called for. A prototype system can also be developed based solely on domain knowledge without relying on extensive training data. Furthermore, performance of the system can be gradually tuned as more data become available.

A fuzzy logic-based prediction system, which we developed by following the methodology proposed by Schneidewind [21], consists of the following steps.

1. Develop a set of membership vectors for metrics, $M=\{m_1,\ldots,m_n\}$, based on n features that contain enough information to characterize an object.
2. Select a quality factor vector, F, which we are interested in measuring.
3. Develop a rule vector R mapping metric vectors M to target classes in F.
4. Collect a validation data set V to evaluate the system developed.

Step 1: We carefully analyzed various inspection data reported in literature [1,7,10,15,26] in order to select the metrics that contribute to the quality factor. Most of researchers point out that *error density* (number of major errors per KLoC), *preparation rate* (lines per hour), and *inspection rate* (lines per hour) are highly related to the quality of a module to be inspected.

For our classification system, we selected error density and inspection rate[1] as the primary quality metrics and compiled the fuzzified membership functions shown in Tables 1 and 2. The middle column, T_0, indicates a average value reported. Columns to the right, T_1 through T_3, indicate the values slightly, moderately, and significantly above the average, respectively. Similarly, columns to the left, T_{-1} through T_{-3}, represent the values slightly, moderately, and significantly below the average, respectively. The values in the tables identify the membership of the class.

[1] Since inspection rate and preparation rate are interdependent (e.g., not orthogonal to each other), use of both metrics is discouraged.

Table 1
Membership function for major error density (errors per KLoC)

Range	T_{-3}	T_{-2}	T_{-1}	T_0	T_1	T_2	T_3
$0<=x<=5$	1.0	0.3	0.0	0.0	0.0	0.0	0.0
$5<x<=10$	0.3	0.7	0.3	0.0	0.0	0.0	0.0
$10<x<=15$	0.0	1.0	0.7	0.3	0.0	0.0	0.0
$15<x<=20$	0.0	0.7	1.0	0.7	0.3	0.0	0.0
$20<x<=25$	0.0	0.3	0.7	1.0	0.7	0.3	0.0
$25<x<=30$	0.0	0.0	0.3	0.7	1.0	0.7	0.3
$30<x<=35$	0.0	0.0	0.0	0.3	0.7	1.0	0.7
$35<x$	0.0	0.0	0.0	0.0	0.3	0.7	1.0

Table 2
Membership function for inspection rate (lines per hour)

Range	T_{-3}	T_{-2}	T_{-1}	T_0	T_1	T_2	T_3
$0<=x<=50$	1.0	0.3	0.0	0.0	0.0	0.0	0.0
$50<x<=100$	0.3	1.0	0.5	0.3	0.0	0.0	0.0
$100<x<=150$	0.0	0.3	1.0	0.7	0.3	0.0	0.0
$150<x<=200$	0.0	0.0	0.5	1.0	0.7	0.0	0.0
$200<x<=250$	0.0	0.0	0.0	0.7	1.0	0.3	0.0
$250<x<=300$	0.0	0.0	0.0	0.3	0.7	0.7	0.0
$300<x<=350$	0.0	0.0	0.0	0.0	0.3	1.0	0.3
$350<x<=400$	0.0	0.0	0.0	0.0	0.0	0.7	0.7

For example, the values on the 4th row of Table 2, shaded for illustration purpose, capture the following fuzzy knowledge:

- Inspecting between 150 and 200 lines of code per hour is most likely considered, as indicated by the confidence level of 1.0, to be typical.
- Different experts might interpret the same figure as being slightly below or above the average, with the confidence level of 0.5 and 0.7, respectively. However, in such cases, the figure is more likely to be interpreted as being slightly above the average.
- It is highly unlikely that, as shown by the confidence level of 0.0, experts interpret the figure as moderately or significantly deviating from the average.

......
if inspection rate is T_0 and error density is T_3 then error-proneness is T_3.
if inspection rate is T_1 and error density is T_2 then error-proneness is T_2.
if inspection rate is T_2 and error density is T_2 then error-proneness is T_3.
if inspection rate is T_3 and error density is T_0 then error-proneness is T_{-3}.

if error density is T_0 and inspection rate is T_3 then poorly inspected is T_{-2}.
if error density is T_{-1} and inspection rate is T_2 then poorly inspected is T_1.
if error density is T_{-2} and inspection rate is T_3 then poorly inspected is T_3.
if error density is T_{-3} and inspection rate is T_0 then poorly inspected is T_{-3}.
......

Fig. 1. Inference rules for error-proneness and poorly-inspected modules.

Steps 2 and 3: We selected two factors to measure quality of software. Those factors are how poorly the code was developed and how the inspection was conducted. Next step is to create a set of heuristic rules that map the membership vectors M into the quality factors F. Heuristics are stated in conditional linguistic terms. Inference rules are shown in Fig. 1. For example, the first line indicates that the degree of error proneness is significantly high if inspection rate is average and error density is significantly above the average. Since each membership vector consists of 7 tiered classes and two factors are selected, we composed 98, or $7 * 7 * 2$, rules.

Results of approximate reasoning in which the degree of an inspection component C_i [2] being error-prone is represented as a set of fuzzy values. In our classification system, 7-*tuples* (e_{-3}, e_{-2}, e_{-1}, e_0, e_1, e_2, e_3) are used with e_0 denoting neutrality. e_3 indicates the degree that C_i is highly likely to be error- prone and e_{-3} is a degree associated with C_i's being highly unlikely to be error-prone. The result of the system has to be represented as a numeric value to indicate the degree of error proneness for C_i. The fuzzy vector is defuzzified into a numeric value by using a defuzzification scheme [28], and we have used the center of gravity method. That is, data obtained from inspection sessions will produce numeric values representing the degree of being considered error-prone.

[2] Inspection component refers to the group of codes inspected in a given session. Since each inspection session of up to 2 h is likely to review several hundred lines of code, inspection component is likely to consist of several procedures or functions.

Step 4: As a final step in our quality prediction process, we have collected training data as well as evaluation data and selected a threshold value to distinguish error-prone components from error-free components based on domain knowledge. Such a decision is bound to be subjective. For example, one expert might consider more than 5 errors per KLoC as being error-prone while another expert might select 10 errors per KLoC as the boundary value.

4. Empirical evaluation

It is relatively easy to construct quality prediction models using fuzzy logic that classify data from the past projects, because all such models can be calibrated according to the quality of fit and expert knowledge. The difficulty lies in evaluating the model to improve and stabilize it based on historical data. Validation of our quality prediction model was carried out using inspection data published by Ebenau [9] as the training data. Inspection data collected from two controlled experiments we had conducted [23,24] were used as evaluation data.

Ebenau used a statistical quality control approach in which upper and lower threshold values of the error density were used to initially select candidate modules for being error-prone or poorly inspected. In addition to the control chart for error density, other graphs for inspection rate and module size were used to finalize the selection of troublesome modules. He evaluated his approach using inspection data from the PBX200 project. PBX200 project, implementing a local tele-

Table 3
PBX200 inspection records

Module no	Component name	Inspection meeting[a]	Inspection hours	Code lines	Inspection rate (lines/h)	Defects	Errors per KLoC
1	DISP Function	I	1.25	280	224	3	10.7
2	Trans Type 1	I	2	280	140	4	14.3
3	Trans Type 1	R	1.5	280	187	2	7.1
4	Prog Keys	I	2	414	207	14	33.8
5	Status Disp	I	2.25	520	231	10	19.2
6	Trans Type 2	I	2.25	1200	533	4	3.3
7	OPS	I	1.5	240	160	1	4.2
8	LCD-Service	I	0.5	80	160	0	0.0
9	Status Bus	I	2	440	220	10	22.7
10	LCD-Term	I	0.75	85	113	2	23.5
11	OPS	I	1	81	81	1	12.3
12	Status DMA	I	2.5	330	132	8	24.2
13	LCD-User	I	1	250	250	1	4.0
14	Status Bus	R	1	440	440	3	6.8
15	Wait Display	I	1.5	350	233	0	0.0
16	Active Display	I	1	240	240	4	16.7
17	Status Disk	I	2.5	570	228	2	3.5
18	Trans Type 4	I	3	400	133	4	10.0
19	Audible Alert	I	0.5	80	160	0	0.0
20	Chan Handler	I	1.8	200	111	8	40.0
21	Dial_0 and LDN	I	0.5	36	72	2	55.6
22	Directed Recal	I	2	370	185	5	13.5
23	DMA Handler	I	0.3	40	133	2	50.0
24	Alert/CON	I	1	120	120	3	25.0
25	Do Not Distub	I	0.8	140	175	5	35.7

[a]"I" and "R" stand for initial inspection and replicated inspection, respectively.

phone switching system, had approximately 7000 lines of codes. There were 23 new inspections and 2 re-inspections conducted during detailed design and coding phase as shown in Table 3. From the PBX200 inspection data, his analysis method classified modules 4, 20, and 25 as being error-prone and module 6 as having been poorly inspected. Module 23 was subjectively excluded by considering the module size. And module 14 was subjectively excluded by considering replication of inspection.

The results of applying our proposed approach on the PBX200 inspection data are shown in Figs. 2 and 3. Fig. 2, capturing the degree of each module's error-proneness, indicate that modules 4, 20, 23, and 25 have relatively higher values than others. Next highest value, that of module 21, is 3.33, and there seems to be a big enough of a gap between the two groups. Therefore, one may conclude that four out of 25 modules should be considered as being error-prone and that further quality assurance mechanisms must be applied. It should be noted that this result is the same as the one published by Ebenau except that module 23, which was excluded due to its small module size (40 lines), is included in our results. If we were to extend heuristics so that module size is considered in the decision process, the result would become identical.

Similarly, our quality prediction model identified module 6 and 14 are the most likely candidates of poorly inspected components. Module 14 was included because our inference rules did not take into consideration on how many times the module has been previously inspected. It seems feasible to improve the accuracy of our system by taking additional metrics (e.g., module size, cyclomatic complexity, fan-in/fan-out, etc.) into consideration.

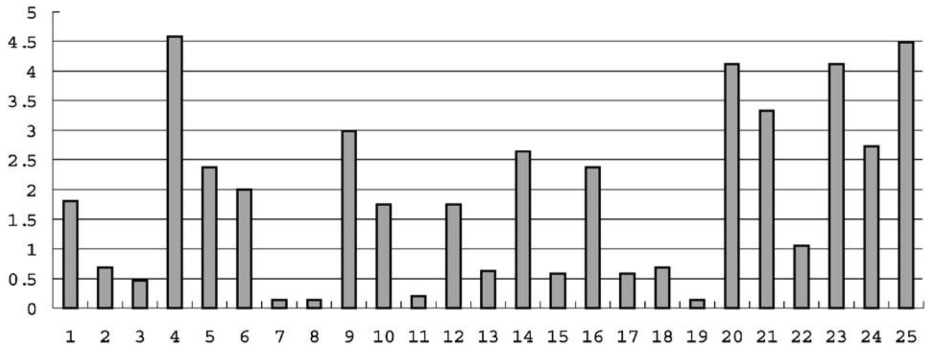

Fig. 2. Degree of error-proneness. (Fuzzy logic-based quality prediction applied on data published by Ebenau).

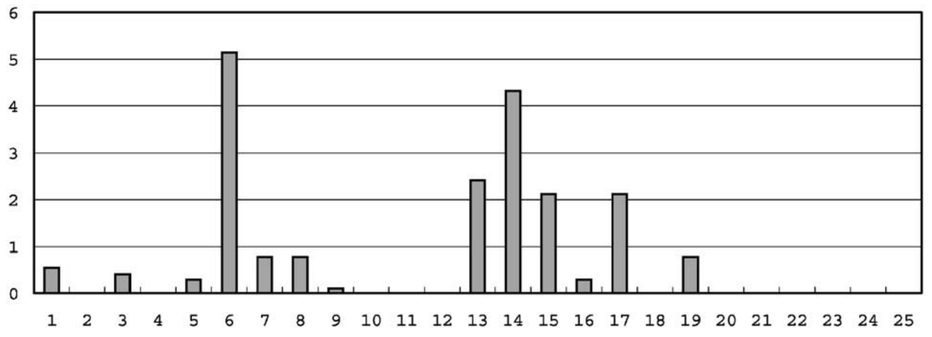

Fig. 3. Degree of poor inspection. (Fuzzy logic-based quality prediction applied on data published by Ebenau).

We conducted another case study to further validate our quality prediction model. Data collected from 38 inspection sessions, shown in Table 4, on LIP (Launch Interceptor Program) [24] and Conflict programs [23] were used. The data are sorted in the order of decreasing K-density value representing the number of known errors per KLoC. Top rows indicate the modules that are empirically known to be error prone by applying various error detection techniques such as voting, testing, inspection, etc. The purpose of this validation is to determine whether inspection rate and F-density, number of errors found per KLoC detected in inspection sessions, are useful in predicting the error proneness. That is, our quality prediction model computed the degree of error proneness based solely on inspection data and without any knowledge on the number of known errors or K-density values. The results are shown in the third column from the right in Table 4, and our model's accuracy can be measured by comparing the estimated value of error proneness to the known K-density value.

There are two cases that our prediction may fail. If an error-prone module is classified as a reliable one, it is referred to as the type I error. Similarly, our system may classify a reliable module to be error-prone. Issuing a "false alarm" is referred to as the type II error. Between the two types of errors, type I errors can pose a serious threat for quality control since it failed to issue a necessary alarm. On the other hand, type II errors are not harmful but may lead to wasted efforts. The rightmost column of the Table 4 shows two types of identification errors when density of 20 K-density is used as the boundary value. The corresponding contingency table is shown in Table 5. The χ^2 value was 7.819 and the P value was 0.005, thereby rejecting the hypothesis with a very small significance level ($\alpha \geqslant 0.005$) [8]. In other words, from the high value of χ^2 and small significance level, we infer that

Table 4
LIP and conflict data sorted by known error density

No	Component	Inspection hour	Code lines	Inspection rate (lines/h)	Errors found	F-density	Errors known	K-density	Degree of error proneness	Degree of poor inspection	Type of error ($>=20$)
1	Conflict5-6	0.67	340	510	7	20.59	27	79.4	4.5	1.3	
2	Conflict6-5	1.67	80	48	2	25.00	4	50.0	1.2	0.0	
3	Conflict2-4	1.08	354	326.77	3	8.47	17	48.0	2.0	3.0	
4	Conflict8-1	1.67	315	189	7	22.22	15	47.6	2.2	0.0	
5	Conflict5-4	1.5	225	150	3	13.33	10	44.4	1.1	0.0	
6	Conflict3-1	0.5	113	226	0	0.00	4	35.4	0.6	2.1	Type I
7	Conflict6-4	0.92	214	233.45	3	14.02	7	32.7	1.8	0.5	
8	Conflict5-2	2	257	128.5	2	7.78	7	27.2	0.3	0.2	Type I
9	Conflict2-5	0.5	193	386	1	5.18	5	25.9	2.5	4.0	
10	Conflict6-2	2.08	274	131.52	4	14.60	7	25.5	0.7	0.0	Type I
11	Conflict8-2	2.08	321	154.08	4	12.46	7	21.8	1.1	0.0	
12	Conflict5-3	2	189	94.5	2	10.58	4	21.2	0.2	0.0	Type I
13	Conflict5-5	1.92	285	148.7	5	17.54	6	21.1	1.1	0.0	
14	LIP25-1	0.5	150	300	3	20.00	3	20.0	4.0	0.8	
15	Conflict8-3	2.17	327	150.92	2	6.12	6	18.3	0.5	0.4	
16	Conflict2-1	1	291	291	2	6.87	5	17.2	1.2	1.5	Type II
17	LIP6-1	2	184	92	2	10.87	3	16.3	0.2	0.0	
18	Conflict5-1	2.08	248	119.04	2	8.06	4	16.1	0.3	0.2	
19	LIP12-1	1.5	257	171.3	1	3.89	4	15.6	0.1	0.8	
20	LIP12-3	2.92	274	93.8	4	14.60	4	14.6	0.2	0.0	
21	Conflict2-2	0.92	386	421.09	0	0.00	5	13.0	2.0	5.1	Type II
22	Conflict6-1	1.25	90	72	0	0.00	1	11.1	0.0	0.0	
23	Conflict6-3	1.92	276	144	2	7.25	3	10.9	0.3	0.2	
24	Conflict8-4	2.33	467	200.14	0	0.00	5	10.7	0.6	2.1	
25	Conflict3-2	1.5	281	187.33	0	0.00	3	10.7	0.1	0.8	
26	LIP6-3	2	387	193.5	3	7.75	4	10.3	0.5	0.4	
27	LIP3-2	0.5	259	518	0	0.00	2	7.7	2.0	5.1	Type II
28	LIP3-4	0.75	259	345.3	1	3.86	2	7.7	1.4	4.3	Type II
29	LIP3-3	1.83	393	214.8	2	5.09	3	7.6	1.0	1.4	
30	LIP3-1	2	393	196.5	1	2.54	3	7.6	0.1	0.8	
31	Conflict8-5	2.17	430	198.46	1	2.33	3	7.0	0.1	0.8	
32	Conflict2-3	2.33	316	135.43	1	3.16	2	6.3	0.1	0.6	
33	LIP25-3	3	660	220	3	4.55	3	4.5	0.6	2.1	
34	LIP12-4	0.7	233	332.9	0	0.00	1	4.3	1.4	4.3	Type II
35	LIP12-2	1.1	250	227.3	0	0.00	1	4.0	0.6	2.1	
36	LIP6-2	0.5	393	786	1	2.54	1	2.5	2.0	5.1	Type II
37	LIP25-2	2.2	510	231.8	0	0.00	0	0.0	0.6	2.1	
38	LIP6-4	1	190	190	0	0.00	0	0.0	0.1	0.8	

the threshold value of 20 is a useful discriminator. It shows that 28 of 38 modules, or 74%, have been accurately classified.

Our quality prediction model failed to completely accurately estimate quality of software modules. However, it must be emphasized that our quality prediction is based solely on existing inspection data, that the process is completely automated, and that there are no additional overheads required to initially screen a set of modules that might require special attention of quality assurance personnel.

In order to find the optimal discriminator for this case study, we repeated χ^2 tests using threshold values of 25 and 30. Table 6 illustrates how accurate

Table 5
The contingency table for 20 errors/KLoC

		Predicted		
		Error-prone	Reliable	Total
Real	Error-prone (K-density >= 20)	10	4	14
	Reliable (K-density < 20)	6	18	24
	Total	16	22	38

$\chi^2 = 7.819$, DF $= 1$, $P = 0.005$.

Table 6
χ^2 critical values according to the boundary value of error density

Boundary of error density (errors per KLOC)	χ^2	P[a]	Type I error	Type II error
20	7.819	0.005	4	6
25	5.074	0.024	4	6
30	5.315	0.021	3	5

[a]P is the smallest level of significance at which the null hypothesis of no association would be rejected.

our model was for the other scenarios. The χ^2 test was repeated with threshold values of 25 and 30, again resulting in rejection of the null hypothesis at a small significance level ($\alpha \geqslant 0.02$). We conclude the error density value 20 to be the best discriminator because we obtained the highest value of χ^2 and the smallest significance level. If tests were repeated with smaller values, most of software components would be classified as being error-prone, and discussion on accuracy would be meaningless.

5. Conclusions

In this paper, we proposed a fuzzy logic-based approach to predict error-prone modules using inspection data. Empirical evaluation of the proposed system on the published inspection data by Ebenau demonstrated effectiveness of our approach. Further empirical evaluation on LIP and Conflict inspection data as well as χ^2 analysis confirmed the model's validity. Our approach offers advantages over others in several ways. First, interpretation of much of inspection data is fuzzy in nature, and our model provides a natural mechanism to model such fuzzy data. Rules used in determining error-prone modules are fuzzy, too. Second, prototype system can be developed without having to have extensive empirical data and that the system's performance can be continuously tuned as more inspection data become available. Finally, utilization of our system requires no extra cost to the development team since our analysis is based on inspection data and analysis is automated. Our system provides software project managers with reasonably accurate and much-needed insights as to how limited resources available for software development can be best allocated.

There are several enhancements that are worthy of further research. For example, effectiveness of the proposed model needs to be further validated empirically. Such validations would provide useful insights as to how fuzzified membership functions can be enhanced. Fuzzy inference rules used in predicting error-prone modules can be further improved, too. Such enhancements would allow our model to make more accurate predictions. Our quality prediction model can be further enhanced by conducting an in-depth analysis as to why our system misclassified some of software components of LIP and Conflict programs.

References

[1] J. Barnard, A. Price, Managing code inspection information, IEEE Software 11 (1994) 59–69.

[2] D.B. Bisant, J.R. Lyle, A two-person inspection method to improve programming productivity, IEEE Trans. Software Eng. 15 (1989) 1294–1304.

[3] S.S. Brilliant, J.C. Knight, N.G. Leveson, Analysis of faults in an N-version software experiment, IEEE Trans. Software Eng. 16 (1990) 238–247.

[4] K.Y. Cai, System failure engineering and fuzzy methodology: an introductory overview, Fuzzy Sets and Systems 83 (1996) 113–133.

[5] K.Y. Cai, Software Defect and Operational Profile Modeling, Kluwer Academic Publishers, Dordrecht, 1998, pp. 18–68.

[6] D.A. Christenson, S.T. Huang, A.J. Lamperez, Statistical quality control applied to code inspections, IEEE J. Selected Areas Commun. 8 (1990) 196–200.

[7] E.P. Doolan, Experience with Fagan's inspection method, Software Practice Experience 22 (1992) 173–182.

[8] E.R. Dougherty, Probability and Statistics for the Engineering, Computing and Physical Sciences Prentice-Hall International Inc., Englewood Cliffs, NJ, 1990.

[9] R.G. Ebenau, Predictive quality control with software inspection, in software inspection: an industry best practice, IEEE Computer Society Press, Silver Spring, MD, 1996, pp. 147–156.

[10] R.G. Ebenau, S.H. Strauss, Software Inspection Process, McGraw-Hill Inc., New York, 1993, pp. 115–137.

[11] C. Ebert, Rule-based fuzzy classification for software quality control, Fuzzy Sets and Systems 63 (1993) 349–358.

[12] C. Ebert, Experiences with criticality predictions in software development, Software Eng. Notes 22 (1997) 278–296.

[13] C. Ebert, E. Baisch, Knowledge-based techniques for software quality management, in: W. Pedrycz, J.F. Peters (Eds.), Computational Intelligence in Software Engineering, World Scientific, Singapore, 1998, pp. 295–320.

[14] M.E. Fagan, Design and code inspections to reduce errors in program development, IBM Systems J. 15 (1976) 182–211.

[15] M.E. Fagan, Advances in software inspections, IEEE Trans. Software Eng. 12 (1986) 744–751.

[16] T.M. Khoshgoftaar, D.L. Lanning, A.S. Pandya, A comparative study of pattern recognition techniques for quality evaluation of telecommunications software, IEEE J. Selected Areas Commun. 12 (1994) 279–291.

[17] J.C. Munson, T.M. Khoshgoftaar, The detection of fault-prone programs, IEEE Trans. Software Eng. 18 (1992) 423–433.

[18] W. Pedrycz, J.F. Peters (Eds.), Computational Intelligence in Software Engineering, World Scientific, Singapore, 1998.

[19] A.A. Porter, R.W. Selby, Evaluating techniques for generating metric-based classification trees, J. Systems Software 12 (1990) 209–218.

[20] G.W. Russell, Experience with inspection in ultralarge-scale developments, IEEE Software 8 (1991) 25–31.

[21] N.F. Schneidewind, Methodology for validating software metrics, IEEE Trans. Software Eng. 18 (1992) 410–422.

[22] T.J. Shimeall, An experimental in software fault tolerance and fault elimination, Ph.D. Dissertation, Univ. of California Irvine, 1989.

[23] S.S. So, S.D. Cha, Y.R. Kwon, T.J. Shimeall, An empirical evaluation of six methods to detect errors in software, J. Software Testing, Verification and Reliability, in revision.

[24] S.S. So, Y. Lim, S.D. Cha, Y.R. Kwon, An empirical study on software error detection: voting, instrumentation, and fagan inspection, Proceedings of the Asia-Pacific Software Engineering Conference, IEEE Computer Society Press, Silver Spring, MD, 1995, pp. 345–351.

[25] L.G. Votta, Does Every Inspection Need a Meeting?, SIGSOFT '93-Proceedings 1st ACM SIGSOFT Symposium on Software Development Engineering, ACM Press, New York, 1993, pp. 107–114.

[26] E.F. Weller, Lessons from three years of inspection data, IEEE Software 10 (1993) 38–45.

[27] D.A. Wheeler, B. Brykczynski, R.N. Messon Jr., Software Inspection: An Industry Best Practice, IEEE Computer Society Press, Silver Spring, MD, 1996.

[28] H.J. Zimmermann, Fuzzy Set Theory and its Applications, Kluwer, Boston, 1991.