An Artificial Immune Recognition System-based Approach to Software Engineering Management: with Software Metrics Selection

Xin Jin[1], Rongfang Bie[1*], and X. Z. Gao[2]
[1]*College of Information Science and Technology*
Beijing Normal University, Beijing 100875, P. R. China
[2]*Institute of Intelligent Power Electronics, Helsinki University of Technology*
Otakaari 5 A, FI-02150 Espoo, Finland
*xinjin796@126.com, *corresponding author: rfbie@bnu.edu.cn, gao@cc.hut.fi*

Abstract

Artificial Immune Systems (AIS) are emerging machine learners, which embody the principles of natural immune systems for tackling complex real-world problems. The Artificial Immune Recognition System (AIRS) is a new kind of supervised learning AIS. Improving the quality of software products is one of the principal objectives of software engineering. It is well known that software metrics are the key tools in the software quality management. In this paper, we propose an AIRS-based method for software quality classification. We also compare our scheme with other conventional classification techniques. In addition, the Gain Ratio is employed to select relevant software metrics for classifiers. Results on the MDP benchmark dataset using the Error Rate (ER) and Average Sensitivity (AS) as the performance measures demonstrate that the AIRS is a promising method for software quality classification and the Gain Ratio-based metrics selection can considerably improve the performance of classifiers.

1. Introduction

The natural immune system is a powerful and robust information processing system that demonstrates several distinguishing features, such as distributed control, parallel processing, and adaptation/learning via experiences. Artificial Immune Systems (AIS) are emerging machine learning algorithms, which embody some of the principles of the natural immune system for tackling complex engineering problems [20]. The Artificial Immune Recognition System (AIRS), is a new supervised learning AIS. It has shown significant success in dealing with demanding classification tasks [18].

Software quality management is an important aspect of software project development. The Capability Maturity Model (CMM) is the *de facto* standard for rating how effective an organization's software development process is [16]. This model defines five levels of software process maturity: initial, repeatable, defined, managed, and optimization. The initial level presents the organizations with no project management system. The managed level describes the organizations, which collect information of software quality and development process, and use that information for process improvement. Finally, the optimization level describes those organizations that continually measure and improve their development process, while simultaneously explores the process innovations.

The software quality management is also an ongoing comparison of the actual quality of a product with its expected quality. Software metrics are the key tools in the software quality management, since they are essential indicators of software quality, such as, reliability, maintenance effort, and development cost. Many researchers have analyzed the connections between software metrics and code quality [8] [9] [14] [15] [16]. The methods they use fall into the following main categories: association analysis [19], clustering analysis [17], classification and regression analysis [4] [12] [21].

In this paper, we propose an AIRS for the software quality classification. We also compare this method with other well-known classification techniques. In addition, we investigate the employment of the Gain Ratio (GR) for selecting relevant software metrics in order to improve the performance of the AIRS-based classifiers.

The remainder of this paper is organized as follows. Section 2 briefly introduces the software metrics and MDP benchmark dataset. Section 3 presents our AIRS-based software quality classification method. Section 4 describes two baseline classification algorithms for comparison. Section 5 discusses the metrics selection with the Gain Ratio. Simulation results are demonstrated in Section 6. Finally, some remarks and conclusions are drawn in Section 7.

Proceedings of the Sixth International Conference on Intelligent Systems Design and Applications (ISDA'06)
0-7695-2528-8/06 $20.00 © 2006 IEEE

IEEE COMPUTER SOCIETY

2. Software Metrics

In this paper, we investigate totally 38 software metrics. Simple counting metrics, such as the number of lines of source codes or Halstead's number of operators and operands, describe how many "things" there are in a program. However, more complex metrics, e.g., McCabe's cyclomatic complexity or Bandwidth, attempt to describe the "complexity" of a program by measuring the number of decisions in a module or the average level of nesting in the module. These metrics are used in the NASA Metrics Data Program (MDP) benchmark dataset MW1 [13]. Refer to Table 1 for a more detailed description of the metrics. There are 403 modules in this dataset.

Our goal is to develop a prediction model of software quality, in which the number of defects associated with a module is projected on the basis of the values of the 37 software metrics characterizing a software module. We cast this problem in the setting of classification, and in each module, the explanatory variables are the first 37 software metrics, and the prediction variable is the defects. Software modules with no defects are in the class of *fault-none*, while those with more than one defect are in the class of *fault-prone*.

Table 1. Description of NASA MDP MW1 project dataset with characterization of software metrics [13].

Software Metrics	Descriptions
LOC_BLANK	Number of blank lines in a module.
BRANCH_COUNT	Branch count metrics.
CALL_PAIRS	Number of calls to other functions in a module.
LOC_CODE_AND_COMMENT	Number of lines containing both code and comment in a module.
LOC_COMMENTS	Number of lines of comments in a module.
CONDITION_COUNT	Number of conditionals in a given module.
CYCLOMATIC_COMPLEXITY	Cyclomatic complexity of a module.
CYCLOMATIC_DENSITY	Ratio of the module's cyclomatic complexity to its length in NCSLOC. It factors out the size component of complexity, and normalizes the complexity and maintenance difficulty of a module.
DECISION_COUNT	Number of decision points in a given module.
DECISION_DENSITY	Calculated as: Cond. / Decision.
DESIGN_COMPLEXITY	Design complexity of a module.
DESIGN_DENSITY	Calculated as: iv(G)/v(G).
EDGE_COUNT	Number of edges found in a given module, which represents the transfer of control from one module to another. (Edges are a base metric, which is used to calculate involved complexity metric).
ESSENTIAL_COMPLEXITY	Essential complexity of a module.
ESSENTIAL_DENSITY	Calculated as: (ev(G)-1)/(v(G)1).
LOC_EXECUTABLE	Number of lines of executable code for a module (not blank or comment).
PARAMETER_COUNT	Number of parameters to a given module.
HALSTEAD_CONTENT	Halstead length content of a module.
HALSTEAD_DIFFICULTY	Halstead difficulty metric of a module.
HALSTEAD_EFFORT	Halstead effort metric of a module.
HALSTEAD_ERROR_EST	Halstead error estimate metric of a module.
HALSTEAD_LENGTH	Halstead length metric of a module.
HALSTEAD_LEVEL	Halstead level metric of a module.
HALSTEAD_PROG_TIME	Halstead programming time metric of a module.
HALSTEAD_VOLUME	Halstead volume metric of a module.
MAINTENANCE_SEVERITY	Calculated as: ev(G)/v(G).
MODIFIED_CONDITION_COUNT	
MULTIPLE_CONDITION_COUNT	Number of multiple conditions in a module.
NODE_COUNT	Number of nodes found in a given module. (Nodes are a base metric, which are used to calculate involved complexity metrics).
NORMALIZED_CYLOMATIC_COMPLEXITY	
NUM_OPERANDS	Number of operands in a module.
NUM_OPERATORS	Number of operators in a module.
NUM_UNIQUE_OPERANDS	Number of unique operands in a module.
NUM_UNIQUE_OPERATORS	Number of unique operators in a module.
NUMBER_OF_LINES	Number of lines in a module. Pure and simple count from open bracket to close bracket. It includes every line in between, regardless of character content.
PERCENT_COMMENTS	Calculated as: ((CLOC+C&SLOC)/(SLOC+CLOC+C&SLOC))*100.
LOC_TOTAL	Total number of lines for a given module.
ERROR_COUNT	Number of defects associated with a module.

IEEE COMPUTER SOCIETY

3. Artificial Immune Recognition System

The Artificial Immune Recognition System (AIRS) is a new method for data mining [10] [11] [6]. In this section, we explore the application of the AIRS for software quality classification. We first prepare a pool of recognition or memory cells (data exemplars), which are the representatives of the training software modules the model is exposed to. The lifecycle of the AIRS is illustrated in Fig. 1.

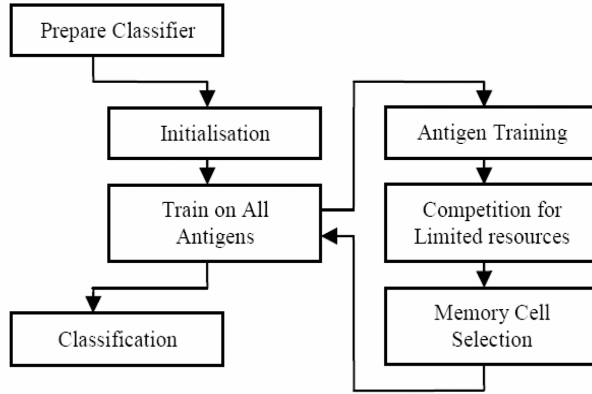

Fig. 1. Life cycle of AIRS.

3.1 Initialization

This step of the AIRS consists of preparing the data and system variables for use in the training process. The training software modules are normalized so that the range of each metrics is within [0,1]. An affinity measure is needed in the training process.

The maximum distance between two arbitrary data vectors is calculated in (1), where r is the known data range for attribute i.

$$maxdist = \sqrt{\sum_{i=1}^{n} r_i^2} \qquad (1)$$

The affinity is a similarity value, which means the smaller the affinity value the higher the affinity is (the closer the vectors are to each other):

$$affinity = maxdist/dist. \qquad (2)$$

where *dist* is the Euclidean distance.

The next step is to seed the memory cell pool. The memory cell pool is a collection of recognition elements, i.e., classifiers generated at the end of the training phase. Seeding the memory pool involves randomly selecting a number of antigens to be the memory cells.

The final step during the initialization is to choose the system variable: Affinity Threshold (AT). The AT is the mean affinity among antigens in the training dataset. It can be either calculated from a sample of the training set or the entire training set. The AT is employed in the training scheme to determine whether the candidate memory cells can replace the existing memory cells in the classifier.

3.2 Antigen Training

The recognition cells in the memory pool are stimulated by the antigen, and each cell is allocated a stimulation value (complementary affinity), as given in (3). The memory cell with the greatest stimulation is selected as the best matching memory cell for the affinity maturation process:

$$\text{stim} = 1 - \text{affinity}. \qquad (3)$$

A number of mutated clones are created from the selected memory cell and added to the Artificial Recognition Ball (ARB) pool. The ARB pool is actually a work area, where the AIRS refines the mutated clones of the best matching memory cell for a specific antigen. The number of mutated clones created from the best matching memory cell is calculated as follows:

$$\text{numClones} = \text{stim} \times \text{clonalRate} \times \text{hypermutationRate}, \qquad (4)$$

where stim is the stimulation level between the best matching memory cell and the antigen.

3.3 Competition for Limited Resources

Competition for the limited resources is used to control the size of the ARB pool and promote those ARBs with greater stimulation to the antigens being trained.

In the resource allocation procedure, the amount of resources allocated to each ARB is:

$$\text{resource} = \text{normStim} \times \text{clonalRate}. \qquad (5)$$

A user-defined parameter *total resources* specifies the maximum number of resources that can be allocated.

The ARB pool is next sorted by the allocated resources (descending), and resources are removed from the ARBs starting at the end of the list, until the total allocated resources are below the allowed level. Finally, those ARBs with zero resources are removed from the pool. The termination condition for this process of ARB refinement is met, when the mean of the normalized stimulation is above a preset threshold.

0-7695-2528-8/06 $20.00 © 2006 IEEE
IEEE COMPUTER SOCIETY

3.4 Memory Cell Selection

When the ARB refinement process is completed, the ARB with the greatest normalized stimulation scoring is selected to be the memory cell candidate.

The ARB is included in the memory cell pool, if the stimulation value of the candidate is better than that of the original best matching memory cell. We also need to remove the original best matching memory cell. This occurs only if the affinity between the candidate memory cell and the best matching cell is smaller than a replacement cut-off, which is defined as:

$$\text{cutOff} = \text{affinityThreshold} \times \text{affinityThresholdScalar}, \quad (6)$$

where the affinity threshold is chosen during the above initialization process.

3.5 Classification

When the training process is finished, the pool of memory recognition cells becomes the core of our AIRS-based classifier. The data vectors in the cells can be denormalized for the classification process. The classification is achieved using a k-nearest neighbor approach, where the k best matching with a data instance are located, and the class is determined via the majority vote.

4. Baseline Classifiers for Comparison

4.1 Naive Bayes

The Naive Bayes, a simple Bayesian classification algorithm, has been gaining popularity during recent years. In general, the software quality classification problem can be described as follows. Considering one sample only belongs to one class, for a given sample, we search for class c_i that maximizes the posterior probability $P(c_i|l;\theta^{'})$ by applying the Bayes rule:

$$P(c_i \mid l;\theta^{'}) = \frac{P(c_i \mid \theta^{'})P(l \mid c_i;\theta^{'})}{P(l \mid \theta^{'})}. \quad (7)$$

Note, $P(l|\theta^{'})$ is the same for all the classes, and l can be classified:

$$c_l = \arg\max_{c_i \in C} P(c_i \mid \theta^{'})P(l \mid c_i;\theta^{'}). \quad (8)$$

Reference [7] gives more information on estimating continuous distributions in the Bayesian classifiers.

4.2 Nearest Neighbor

Instance-based learning is a non-parametric inductive learning method storing the training instances in a memory structure, on which the predictions of new instances are based. This approach assumes that reasoning is a direct reuse of the stored experiences rather than the knowledge, such as, models or decision trees, extracted from experiences. The similarity between the new instance and an example in memory can be obtained using a distance metric.

In our experiment, we use IB1 [1], a Nearest Neighbor (NN) classifier with the Euclidian distance metric.

5. Metrics Selection

In data mining, many researchers have argued that the maximum performance cannot be achieved by utilizing all the available features, but only a "good" subset of features. This approach is named feature selection. Therefore, for the metric-based software quality classification, we need to find a subset of metrics, which can discriminate between the *fault-none* and *fault-prone* modules. In our paper, we investigate the power of Gain Ratio (GR) [5] for acquiring relevant metrics to improve the software quality classification.

Suppose there are a total of m classes denoted by $C=\{C_1, C_2,\ldots, C_m\}$ (in this study, we know m=2, and define C_1 as *fault-none* module and C_2 *fault-prone* module), and there are N training modules represented by:

$$(a(1), b(1),\ldots; t(1)),\ldots, (a(N), b(N),\ldots; t(N)) \quad (9)$$

where a(i), b(i),…are the vectors of n metrics, and $t(i)$ C is the class label. Of the N examples, N_{C_k} is in class C_k. An appropriate feature can split these examples into V partitions, each of which has $N^{(v)}$ examples. In a particular partition, the number of examples of class C_k is denotes by $N_{C_k}^{(v)}$.

The GR, which is firstly proposed by Quinlan in [5], compensates for the number of features by normalizing the information encoded in the split itself:

$$GainRatio = IG/[\sum_{k=1}^{m} -(\frac{N_{C_k}}{N})\log(\frac{N_{C_k}}{N})], \quad (10)$$

where IG is the Information Gain [5]. The higher the GR, the more relevant the metric.

0-7695-2528-8/06 $20.00 © 2006 IEEE
IEEE COMPUTER SOCIETY

6. Experiment Results

The 10-fold cross-validation on the NASA Metrics Data Program (MDP) benchmark dataset MW1, available in [13], is deployed here for evaluating the aforementioned classification and software metrics selection methods.

6.1 Performance Measure

We use the following two classification performance measures.

Error Rate (ER): Error Rate is defined by the ratio of the numbers of incorrect predictions and all the predictions (both correct and incorrect):

$$ER = N_{ip}/N_p, \quad (11)$$

where N_{ip} is the number of incorrect predictions, and N_p all predictions (i.e., the number of test samples). The ER ranges from 0 to 1, and 0 is the ideal value. Thus, the lower the ER, the better the classification performance.

Average Sensitivity (AS): We use *Sensitivity* to access how well the classifier can recognize a class (class *fault-free* is taken as an example here):

$$Sensitivity = N_{tpc} / N_c, \quad (12)$$

where N_{tpc} is the number of true predictions for the class (*fault-free* modules that are correctly classified as such), N_c is the number of modules in the class (all modules that are actually *fault-free*). The AS is the average of all the sensitivities:

$$Average\ Sensitivity = \frac{\sum_{i=1}^{n} Sensitivity_i}{n}, \quad (13)$$

where $Sensitivity_i$ is the sensitivity for class *i*, *n* is number of classes (in our case, n=2). A perfect classifier should give an AS of 1. The higher the AS, the better the classifier.

6.2 Results

TABLE I
ER OF AIRS AND THREE OTHER CLASSIFIERS ON THE MDP DATA

	AIRS	AIRS	NaiveB	NearN
ER	None	8.68	16.9	13.4
	GainRatio	7.95	11.45	11.91
AS	None	0.67	0.59	0.57
	GainRatio	0.70	0.64	0.60

NaiveB = Naive Bayes, NearN = Nearest Neighbor. None: using the original MDP full metrics, GainRatio: using the best GR selected metrics.

Table I shows the ER of the AIRS and two baseline classifiers on the MDP data. Using the original metrics, the AIRS achieves the best performance with a minimum of ER=8.68 and a maximum of AS=0.67. With the GR-based metrics selection, the performance of the AIRS is further improved with a minimum of ER=7.95 and a maximum of AS=0.70. On the other hand, the GR can improve the performances the two other classifiers: Naive Bayes and Nearest Neighbor.

Figure 2 illustrates the ER of different classifiers based on the GR selected metrics. The number of selected metrics varies form 5 to 25. The AIRS still has the best overall performance with the lowest ER. Note, the best achievement is at around top 17 selected metrics. Both Naive Bayes and Nearest Neighbor have their better performance at lower number of GR selected metrics.

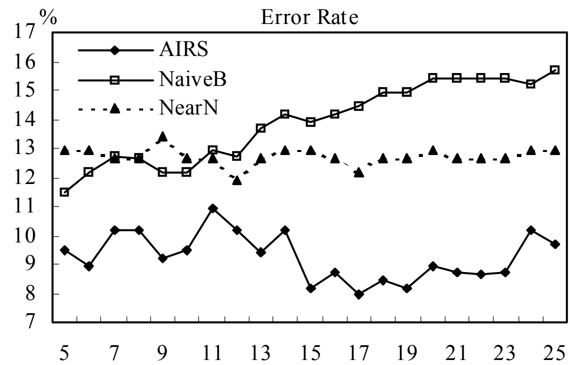

Fig. 2. ER of AIRS and two other classifiers on the MDP data. X-axis: number of selected metrics by GR.

Figure 3 gives the AS of these classifiers on the GR selected metrics. Again, the AIRS has the best performance by achieving the highest AS. The best achievement is at top 17 selected metrics.

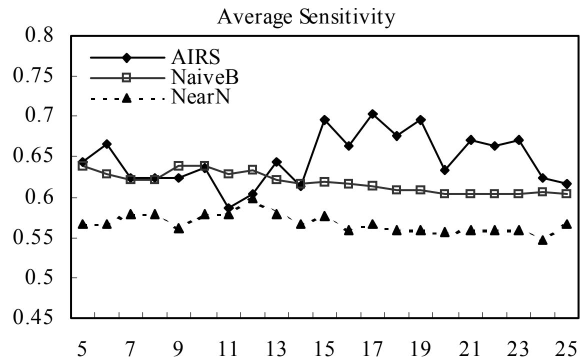

Fig. 3. AS of AIRS and two baseline classifiers on the MDP data. X-axis denotes the number of selected metrics by GR.

IEEE COMPUTER SOCIETY

7. Conclusions

In this paper, we propose an AIRS for the software quality classification. Performance comparison is also made with two baseline classification techniques: Naive Bayes and Nearest Neighbor. In addition, we use the GR for selecting relevant software metrics in order to improve the performance of classifiers. With the ER and AS as the measures, results of the MDP benchmark dataset demonstrate that our AIRS is a promising method for software quality classification, and the GR-based metrics selection can further improve the performance of existing classifiers.

Acknowledgments

Xin Jin and Rongfang Bie's work was supported by the National Science Foundation of China under the Grant No. 10001006 and No. 60273015. X. Z. Gao's research work was funded by the Academy of Finland under Grant 214144.

References

[1] I. Witten and E. Frank, *Data Mining – Practical Machine Learning Tools and Techniques with Java Implementation*. Morgan Kaufmann (2000).

[2] R. C. Holte, "Very Simple Classification Rules Perform Well on Most Commonly Used Datasets," *Machine Learning*, vol. 11, pp. 63-91 (1993).

[3] G. Buddhinath and D. Derry, "A Simple Enhancement to One Rule Classification," *Report at* http://goanna.cs.rmit.edu.au/~gjayatil/OtherLinks/Extra.php (2006).

[4] W. Pedrycz and G. Succi, "Genetic Granular Classifiers in Modeling Software Quality," *Journal of Systems and Software,* 76(3), pp. 277-285 (2005).

[5] J. R. Quinlan, *C4.5: Programs for Machine Learning*. Morgan Kaufmann, San Manteo, CA (1993).

[6] D. Goodman, L. Boggess, and A. Watkins, "An Investigation into the Source of Power for AIRS, An Artificial Immune Classification System," in *Proceedings of the 2003 International Joint Conference on Neural Networks* (2003).

[7] G. H. John and P. Langley, "Estimating Continuous Distributions in Bayesian Classifiers," in *Proceedings of the 11th Conference on Uncertainty in Artificial Intelligence*, pp. 338-345, Morgan Kaufmann, San Mateo (1995).

[8] D. Garmus and D. Herron, *Measuring the Software Process*. Prentice Hall, Upper Saddle River, NJ (1996).

[9] R. Subramanyan and M. S. Krishnan, "Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects," *IEEE Trans. Software Eng.*, vol. 29, pp. 297-310, April (2003).

[10] A. Watkins and J. Timmis, "Artificial Immune Recognition System (AIRS): Revisions and Refinements," in *Proceedings of the 1st International Conference on Artificial Immune Systems*, Canterbury, UK, pp. 173-181, September (2002).

[11] A. Watkins, J. Timmis, and L. Boggess, "Artificial Immune Recognition System (AIRS): An Immune Inspired Supervised Machine Learning Algorithm," *Genetic Programming and Evolvable Machines*, 5(1), March (2004).

[12] Xin Jin, Zhaodong Liu, Rongfang Bie, Guoxing Zhao, Jixin Ma, "Support Vector Machines for Regression and Applications to Software Quality Prediction," International Conference in Computational Science, May 28-31, Reading, UK. *Lecture Notes in Computer Science* (2006)

[13] NASA MDP, http://mdp.ivv.nasa.gov/index.html (2006).

[14] K. H. Muller and D. J. Paulish, *Software Metrics*, IEEE Press/Chapman & Hall, London, UK (1993).

[15] J. C. Munson and T. M. Khoshgoftaar, "Software Metrics for Reliability Assessment," *Handbook of Software Reliability and System Reliability*, McGraw-Hill, Hightstown, NJ (1996).

[16] S. Dick, A. Meeks, M. Last, H. Bunke, and A. Kandel, "Data Mining in Software Metrics Databases," *Fuzzy Sets and Systems*, 145(1), pp. 81-110 (2004).

[17] W. Pedrycz, G. Succi, P. Musilek, and X. Bai, "Using Self-organizing Maps to Analyze Object Oriented Software Measures," *J. of Systems and Software*, vol. 59, pp. 65-82 (2001).

[18] D. Goodman, L. Boggess, and A. Watkins, "Artificial Immune System Classification of Multiple-Class Problems," in C. H. Dagli, A. L. Buczak, J. Ghosh, M. J. Embrechts, O. Ersoy, and S. W. Kercel (eds.), *Intelligent Engineering Systems Through Artificial Neural Networks*, vol. 12, New York, NY, pp. 179-184 (2002).

[19] W. Pedrycz, G. Succi, and M. G. Chun, "Association Analysis of Software Measures," *Int. J. of Software Engineering and Knowledge Engineering*, 12(3), pp. 291-316 (2002).

[20] H. Bersini and F. Varela, "Hints for Adaptive Problem Solving Gleaned from Immune Networks," *in Proceedings of the 1st Workshop on Parallel Problem Solving from Nature*, Dortmund and Federal Republic of Germany, pp. 343-354 (1990).

[21] F. Xing, P. Guo, and M.R. Lyu, "A novel method for early software quality prediction based on support vector machine," *In Proceedings 16th International Symposium on Software Reliability Engineering, ISSRE'2005*, Chicago, Illinois, November 8-11 (2005)

IEEE COMPUTER SOCIETY