Feature Selection and Clustering in Software Quality Prediction

Qi Wang
Dept. of Electronic Engineering, Shanghai Jiaotong University
No. 1954, HuaShan Rd., Shanghai, P.R.China, 200030
Email: wangqi@alcatel-lucent.com

Jie Zhu
Dept. of Electronic Engineering, Shanghai Jiaotong University
No. 1954, HuaShan Rd., Shanghai, P.R.China, 200030
Email: zhujie@sjtu.edu.cn

Bo Yu
System Verification Test Dept. of Lucent Technologies Optical Networks Co., Ltd
No. 388, TianLin Rd., Shanghai, P.R. China, 200233
Email: boyu@alcatel-lucent.com

Software quality prediction models use the software metrics and fault data collected from previous software releases or similar projects to predict the quality of software components in development. Previous research has shown that this kind of models can yield predictions with impressive accuracy. However, building accurate software quality prediction model is still challenging for following two reasons. Firstly, the outliers in software data often have a disproportionate effect on the overalls predictive ability of the model. Secondly, not all collected software metrics should be used to construct model because of the curse of dimension. To resolve these two problems, we present a new software quality prediction model based on genetic algorithm (GA) in which outlier detection and feature selection are executed simultaneously. The experimental results illustrate this model performs better than some latest raised software quality prediction models based on S-PLUS and TreeDisc. Furthermore, the clustered software components and selected features are easier for software engineers and data analysts to study and interpret.

Keywords: Software quality prediction; genetic algorithm; clustering; feature selection

1. INTRODUCTION

The typical way to estimate software quality is using software metrics and fault data collected from previous system releases or similar projects to construct a quality-prediction or quality-classification model. Then engineers use this model to predict the fault proneness of software components in development. Previous research [1] has shown that software quality models based on software metrics can yield predictions with useful accuracy. Such models can be used to predict the response variable that can either be the class of a component (e.g. fault-prone or not fault-prone) or a quality factor (e.g. number of faults) for a component. The former is usually referred to as classification models [2] while the latter is usually referred to as prediction models [3]. The focus of this paper is on the former, i.e., classification models. Quite often, predicting the number of faults is not necessary. Rather, identifying components

that are probably fault-prone may be sufficient for practical purposes [4]. Software metrics used by the model and the response variable are referred to as the independent variables and dependent variable respectively. Over the last few decades many software quality modelling techniques have been developed and used in real life software quality predictions. A few commonly used modelling techniques for software quality estimation include regression trees [2, 3], artificial neural network [5, 6], case-based reasoning [7, 8], multiple linear regression [9], and fuzzy logic [10]. Many of these techniques facilitate software quality estimation modelling using both classification and prediction models.

However, building accurate quality estimation models is challenging because the outliers in the real-world software measurements usually degrade trained models' performance. An outlier is an instance that lies an abnormal distance from other values in a data set. In a sense, this definition leaves it up to the analyst (or a consensus process) to decide what will be considered abnormal. Identifying outlying observations is an important aspect of the software quality estimation model-building process. Outlying observations should be identified because of their potential effect on the fitted model. That is, occasionally certain observations will have a disproportionate effect on the overalls predictive ability of the model. To deal with this kind of problem, a few clustering approaches, such as CLARANS [11], DBSCAN [12] and BIRCH [13] are developed with exception-handling capacities. In the cluster-based outlier detection method, small clusters are identified as outliers [14]. The outlier detection approach introduced in this paper is also a kind of cluster-based approach.

Another main challenge is how to select the appropriate software features to construct estimation model. In software measurement phase, the main purpose is to gather more informative software metrics. But in the model construction phase, it is not all the features that should be considered as input of the estimation model because of the curse of dimension. Thus the feature selection techniques should be used during model construction. The goal of feature selection is, given a data set that describes a target concept using n attributes, to find the minimum number m of relevant attributes which describe the concept as well as the original set of attributes do. Feature selection algorithms can be classified in two classes according to the type of information extracted from the training data and the type of the induction algorithm [15]. Feature selection can be accomplished independently from the performance of a specific learning algorithm. Optimal feature selection is achieved by maximizing or minimizing a criterion function. Such an approach is referred to as the filter feature selection model. Conversely, if the result of feature selection is directly related to the performance of the learning algorithm, usually in terms of its predictive accuracy, this is called wrapper model. This paper presents an effective solution to the feature selection issue, based upon the genetic algorithm paradigm, which fits the filter model.

In this paper, the above two functions: outlier detection and feature selection are integrated in the same software quality prediction model. This software quality prediction model is based on genetic algorithm. The criterions of clustering and feature selection are combined together in the fitness function. Consequently, the gene evolution is directed to generate better clustering and selected better features. After evolution complete, we get a classification model built by the selected features. And the experimental results show that this kind of GA model can significantly improve the classification accuracy. Besides, the selected features and clustered software components are easier for experts to do further analysis.

This paper is organized as follows. The related works are introduced in section 2. Some details of the genetic algorithm are presented in section 3. Section 4 describes the modelling methodology used in our study. In section 5, the experiments results and some discussion are presented. Section 6 is the conclusion of our study.

2. RELATED WORKS

2.1 Outlier detection

Recently, outlier mining started drawing more attentions. The current outlier mining approaches can be classified into five categories: statistic based [16], distance based [17, 18], density based [19], clustering based [14, 20, 21], and deviation based [23]. Our method belongs to the cluster based approaches, so we mainly introduce this kind of methods here. Su *et al.* proposed a two-phase clustering algorithm for outlier detection [14]. They first modify the traditional k-means algorithm in Phase 1, which results that the data points in the same cluster may be most likely all outliers or all non-outliers. And then they construct a minimum spanning tree in Phase 2 and remove the longest edge. The small clusters, the tree with less number of nodes, are selected and regarded as outlier. Zengyou He *et al* introduced two new definitions in [21]: cluster-based local outlier definition and the definition of outlier factor, CBLOF (Cluster-based Local Outlier Factor). Based on this definition, they proposed an outlier detection algorithm, CBLOF. The overall cost of their CBLOF is *O(2N)* by using the squeezer clustering method [22]. In their method, outlier detection is tightly coupled with the clustering process, data are grouped into clusters first, and then outliers are mined; i.e. the clustering process takes precedence over the outlier-detection process.

2.2 Genetic algorithms in feature selection

This section presents a review of the most interesting research works presented in the literature on the general feature selection issue and for the specific genetic-based implementations. A direct approach to using GA's for feature selection was introduced by Siedlecki and Sklansky [24]. In their work, a GA is used to find an optimal binary vector, where each bit is associated with a feature (Fig. 3). If the *i*th bit of this vector equals 1, then the *i*th feature is allowed to participate in classification; if the bit is a 0, then the corresponding feature does not participate. Each resulting subset of features is evaluated according to its classification accuracy on a set of testing data using a nearest neighbor classifier. This technique was later expanded to allow linear feature extraction, by Punch *et al.* [25] and independently by Kelly and Davis [26]. The single bit associated with each feature is expanded to a real-valued coefficient, allowing independent linear scaling of each feature, while maintaining the ability to remove features from consideration by assigning a weight of zero.

The GA feature extraction technique has been expanded to include a binary masking vector along with the feature weight vector on the chromosome [27]. If the mask value for a given feature is zero, the feature is not considered for classification. If the mask value is one, the feature is scaled according to the associated weight value, and is included in the classifier. The inclusion of the mask vector allows the GA to more rapidly sample feature subsets while simultaneously optimizing scale factors for features included in the classifier. An additional improvement specific to the *k* nearest neighbor classifier was the inclusion of the value of *k* on the GA chromosome. This modification allows the GA to optimize the feature weights and value for better classification accuracy.

3. GENETIC ALGORITHM

GA is a class of robust problem-solving techniques based on a population of solutions, which evolve through successive generations by means of the application of three genetic operators: selection, crossover, and mutation [23]. GA is suited to perform a search in huge search spaces, where other methods (local or gradient searches) cannot provide good results. The clustering and feature selection problem that encoded with a binary representation is one of these cases. This chapter will introduce the method of how to integrate clustering and feature selection in the same evolution.

3.1 Individual representation

Considering that there are *N* objects to be clustered, denoted as {O_1, O_2…O_N}. A length *N* object string represents a subset of {O_1, O_2…O_N}. If O_i is in the subset, the i^{th} position of the string will be 1; otherwise, the i^{th} bit will be 0. Suppose each object has *M* features to be clustered, denoted as {F_1, F_2… F_M}. A length *M* feature string represents a subset of {F_1, F_2… F_M}. If F_i is in the subset, the i^{th} position of the string will be 1; otherwise, the i^{th} bit will be 0. A chromosome is composed of these two strings as showed in Fig 1. The first *M* bits represent the selected features while the last *N* bits stand for the seeds for a clustering because we will use each object in the subset represented by this string as a seed to generate a cluster.

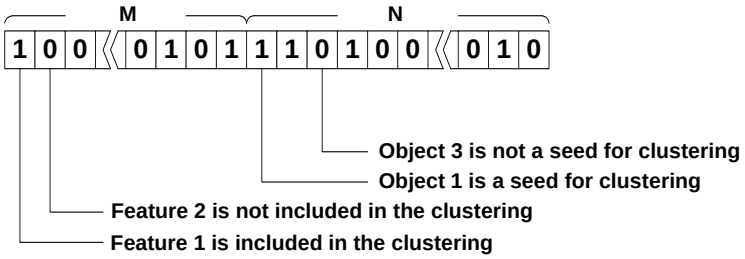

FIGURE 1: chromosome, comprising a feature string and an object string

The chromosomes are initialed in such a way that the numbers of 1's in the feature strings are normal distributed within [1, M] and the numbers of 1's in the object strings are almost uniformly distributed within [1, *N*]. Let *T*= {T_1, T_2… T_n} be a subset of the objects and *U*= {U_1, U_2… U_m} is a subset of the features. Initial clusters C_i={T_i} for *i=1,2…n* and initial centers of clusters S_i=T_i for *i=1,2…n*. The generation of the clusters proceeds as follows:

The objects in {O_1,O_2,…,O_N}-*T* are taken one by one and the distance between the taken object O_i and each S_j is calculated using the following distance function.

$$d(O_i, S_j) = \sum_{q=1}^{m} \left| O_{jq} - S_{jq} \right| \qquad (1)$$

Where O_i=(o_{i1},o_{i2},…,o_{im}) and S_j=(s_{j1},s_{j2},…,s_{jm}), *m* is the number of selected features.

Then, O_i belongs to C_j if $d(O_i,S_j) \le d(O_i,S_k)$ for all *k* such that $1 \le k \le n$ and $k \ne j$.

If O_i is classified as in cluster C_j, the center S_j of the cluster C_j will be recomputed using the following function when O_i is added to C_j.

$$S_{jq} = \frac{\sum_i o_{iq}}{\left| C_j \right|} \qquad (2)$$

Where the summation is over all *i* such that $o_i \in C_j$ and $\left| C_j \right|$ is the number of objects in C_j.

After all objects in {O_1, O_2…O_N}-*T* have been considered, we will obtain *n* clusters C_1, C_2…C_n with centers S_1, S_2…S_n. Each cluster C_j is generated by the seed T_j. We define {C_1, C_2…C_n} to be the set of clusters generated by this chromosome.

3.2 Fitness function

The fitness function must reflect the way each individual is, in comparison to the others, becoming closer to the optimal solution. The calculation of the fitness is a very important part of our algorithm. The fitness of string R is the sum of two scores, namely, *SCORE1* and *SCORE2*. Let {C_1,C_2,…,C_n} be the set of clusters generated by string R. Let C_j' be defined as follows.

$$C_j' = \{ \; O_i \mid O_i \in C_j \text{ and } d(O_i,S_j) \le d(O_i,S_k) \qquad (3)$$

for all k such that $1 \le k \le n$ and $k \neq j$}

Note that C_j' is a subset of C_j and contains those subjects of C_j that are indeed closer to the center of C_j than to other centers. Also note that C_j' may be a proper subset of C_j. In other words, there may be some objects of C_j that are closer to other centers than to the center of C_j. Now, *SCORE1* is defined as follows.

$$SCORE1 = \sum_{j=1}^{m} |C_j'| \quad (4)$$

Suppose $O_i \in C_j$, we define *Dintra*(O_i) and *Dinter*(O_i) as:

$$Dintra(O_i) = d(O_i, S_j) \quad (5)$$

$$Dinter(O_i) = \min_{1 \le k \le n, k \neq j} d(O_i, S_k) \quad (6)$$

where $d(O_i,S_j)$ and $d(O_i,S_k)$ are defined in (1). So *SCORE1* is defined below:

$$SCORE2 = \begin{cases} \sum_{i=1}^{n} (Dinter(O_i) * w + Dintra(O_i)) & ,if \quad SCORE1 = N \\ 0 & ,if \quad SCORE1 < N \end{cases} \quad (7)$$

where w is a weight. If the value of w is small, we emphasize the importance of *Dintra*(O_i). This tends to produce more clusters and each cluster tends to be compact. If the value of w is chosen to be large, we emphasize the importance of *Dinter*(O_i). This tends to produce less clusters and each cluster tends to be loose.

3.3 Crossover and mutation

After the calculation of fitness for each string in the population, the reproduction operator is implanted by using a roulette wheel with slots sized according to fitness. In the crossover phase, for each chosen pair of strings, two random numbers are generated to decide which pieces of the strings are to be interchanged. Each random number is an integer in [1, N]. For example, if two random numbers are 2 and 5, then position 2 to 5 of this pair of strings is interchanged. For each pair of chosen strings, the crossover operator is done by probability p_c. In the mutation phase, bits of the strings in the population will be chosen with probability p_m. Each chosen bit will be change from 0 to 1 or from 1 to 0.

4. METHODOLOGY

4.1 Software data

The training and testing data used in this paper are collected over two large telecommunication system, written in high-level language (C and C++), namely Ruby and Sapphire. They are maintained by professional programmers in a large organization. Four consecutive releases of each product are considered in our study. Some details of the four releases are listed in table 1. A software component consisted of a set of related source code files. Fault data are collected at the component-level by the fault reporting system. A software component is considered:

- *Fault-prone* if any faults discovered by customers or system test engineers and resulted in changes to source code in the component,
- *Not fault-prone*, otherwise.

The *fault-prone* components are labeled as 1 and the *not fault-prone* components are labeled as 0. The first three releases of each product are used to train the software quality estimation model. The last release is deployed as training set to evaluate our model.

TABLE 1: Software data used in this study

Releases	Ruby		Sapphire	
	Fault-prone	**Not fault-prone**	**Fault-prone**	**Not fault-prone**
1	167	743	98	430
2	189	802	76	467
3	135	860	72	511
4	102	911	67	523

Software metrics for these two products are extracted on both file level and routine level; the metrics collected included 18 routine level metrics and 14 file level metrics (Table 2 and Table 3).

TABLE 2: Software metrics on routine level

Symbol	Description
RtnArgXplSum	Sum of the explicit argument numbers (actual parameters) passed to other function by all the explicit function calls made in the routine
RtnCalXplNbr	Number of explicit function/method calls in the routine
RtnCastXplNbr	Number of explicit type casts in the routine
RtnComNbr	Number of comment sections in the routine scope (between the routine brackets {...}), without considering the comments within nested classes or routines
RtnComVol	Size in characters of all the comments in the routine, without considering the comments within nested classes or routines
RtnCplCtlSum	Total (sum) complexity of the control predicates (test expressions) composing the decision and loop statements within the routine
RtnCplCycNbr	Cyclomatic number of the routine
RtnCplExeSum	Total (sum) complexity of the executable statements within the routine
RtnStmDecNbr	Number of declarative statements in the routine
RtnStmDecObjNbr	Number of variable/object declaration statements in the routine
RtnStmDecPrmNbr	Number of parameters of the routine
RtnStmDecRtnNbr	Number of function/routine declaration statements within the routine
RtnStmDecTypeNbr	Number of type/class declaration statements in the routine
RtnLnsNbr	Number of lines of the routine
RtnScpNbr	Number of scopes within the scopes of the routine
RtnScpNstLvlSum	Sum of nesting level values for all scopes in the routine
RtnStmCtlNbr	Number of control-flow statements in the routine
RtnStmExeNbr	Number of executable statements in the routine

TABLE 3: Software metrics on file level

Symbol	Description
FilComGlbNbr	Number of comment sections in the global scope of a file (thus excluding the comments inside routine or class scopes within the file)
FilComGlbVol	Size in characters of all the comments in the global scope of a file (without considering the comments inside routine or class scopes within the file)
FilComTotNbr	Total number of comment sections in the file (considering the comments inside routine or class scopes within the file)
FilComTotVol	Size in characters of all the comments in the file (considering the comments inside routine or class scopes within the file)
FilDecClaNbr	Number of classes declared within the file.
FilDecGncTypNbr	Number of generic classes (template class declaration) declared within the file.
FilDecGndTypTotNbr	Total number of generated classes (template class instances) declared within the file.
FilDecStruNbr	Number of *struct* types declared within the file.
FilDecObjExtNbr	Number of extern objects declared within the global scope of the file.
FilDefObjGlbNbr	Number of global variables/objects defined within the global scope of the file.
FilDefRtnNbr	Number of functions/routines defined within the file.
FilIncNbr	Total number of files included by the current file
FilIncDirNbr	Total number of files directly included by the current file
FilLnsNbr	Number of lines of the file

4.2 Software quality prediction modeling process

We standardize the measurements to a mean of zero and a variance of one for each metric first. Many raw software metrics have incompatible units of measure. This step converts them all to a unit of one standard deviation. In training

Training Phase

Start
Input: training set
Step 1: use the training set to train GA model
Output: *j* clusters and *m* selected features
Step 2: classify the clusters into three group
    If (quantity of cluster members < 5)
        {belongs to outlier group G_o}
    else if (purity > 80%)
        {belongs to decided label group G_d}
    else {belongs to undecided label group G_u}
Output: 3 groups G_o, G_d and G_u
Step 3: train MLP classifier with G_u
Output: trained model
End

Testing Phase

Start
Input: each instance of test set
Step 1: assigned a cluster by trained GA model
Output: belongs to cluster C_i
Step 2: check C_i belongs to which group
Output: belongs to group G_o, G_d or G_u
Step 3: processing the instance depends on its group
    switch (group label)
        { case G_o: not considered; End
          case G_d: get the proneness label; End
          case G_u: goto step 4 }
Step 4: classified by trained MLP classifier
Output: proneness label
End

FIGURE 2: Detail process during training phase and testing phase

phase, the first step is training GACF model. After that, the training set is clustered to *j* clusters with *m* selected features. Then the clusters will be classified into three groups. If the quantity of cluster members is less than 5, it will belong to outlier group, labeled as G_o. All of the members of G_o will be considered as outliers. If the cluster's purity is more than 80%, it will belong to decided group, labeled as G_d. Otherwise it will belong to undecided group G_u. The purity of a cluster is defined as the percentage of the most-dominated category (*fault-prone* or *not fault-prone*) in the cluster. The clusters which belong to G_u are used to train MLP classifier to classify the similar data in the test set. In testing phase, each of the instances in the test set will be assigned a cluster C_i by trained GACF model first. If C_i belongs to G_o, the instance will be considered as outlier and its proneness will not be estimated. If C_i belongs to G_d, the instance's proneness is the most-dominated category of cluster C_i. If C_i belongs to G_u, this instance will be classified by MLP classifier which is trained by similar data in the training phase.

5. EXPERIMENTAL RESULTS

GA is running with following parameters: the crossover rate which is expressed as expected number of crossovers per individual pre generation equals to 0.8; the mutation rate which is expressed as expected number of mutations per bit per generation equals to 0.001; the population size equal to 200. The run length is limited to 200 generations. Release 4 is used to evaluate the model's estimation performance. Specifically, we report Type I misclassification (the model identifies a component as *fault-prone* which is actually *not fault-prone*), Type II misclassification (the model identifies a component as *not fault-prone* which is actually *fault-prone*) and overall misclassification rate as classification result. The value of *w* is chosen between 0.1 and 10. Once *w* is decided, GA is executed three times and the least misclassification rate is set to be the classification result of this model. Table 4 and Table 5 show the experimental results of these two products respectively.

TABLE 4: Classification result over Ruby

w	Type I misclassification rate	Type II misclassification rate	Overall misclassification rate	Clusters			Selected features
				G_o	G_d	G_u	
0.1	18.3%	61.5%	22.6%	9	56	12	12
0.3	17.2%	62.3%	21.7%	7	50	9	13
0.5	16.6%	59.5%	20.9%	6	44	10	11
1	20.1%	60.8%	24.2%	5	38	12	14
1.5	21.2%	55.4%	24.6%	4	32	12	15
2	20.3%	47.9%	23.0%	5	25	11	12
4	19.2%	43.7%	21.6%	6	20	13	14
6	**16.5%**	**48.2%**	**19.6%**	**5**	**16**	**14**	**13**
10	17.1%	45.9%	20.0%	5	9	15	13

TABLE 5: Classification result over Sapphire

w	Type I misclassification rate	Type II misclassification rate	Overall misclassification rate	Clusters			Selected features
				G_o	G_d	G_u	
0.1	17.2%	34.2%	19.1%	5	30	10	11
0.3	16.3%	31.1%	17.9%	5	26	9	13
0.5	15.5%	32.5%	17.4%	4	22	11	12
1	16.9%	33.8%	18.8%	5	17	13	15
1.5	14.3%	32.9%	16.4%	3	16	13	13

2	13.3%	28.7%	15.1%	3	14	14	14
4	12.7%	23.5%	13.9%	4	14	15	11
6	11.8%	23.4%	13.1%	3	10	17	12
10	**11.5%**	**24.8%**	**13.0%**	**3**	**9**	**17**	**12**

It can be found in Table 4 and 5 that the misclassification rate decreases steadily with the increase of *w*. The lowest misclassification rate is marked with bold font.

5.1 Clustering and feature selection

The influence of *w* on clustering result is described in Fig 3 and 4. It can be observed in the figures that with the increase of *w,* the number of decided label group G_d declined steadily. This is because when *w* is small, GA model generates more clusters and these clusters are very compact. In these cases, the purity of most clusters is higher than 80%. It results in most of them belonging to the decided label group G_d. With the increase of *w*, the number of clusters becomes smaller and looser. The clusters' purity decrease and few clusters' labels are certain. Consequently, more clusters' label are not decided and need to be classified by MLP classifier further.

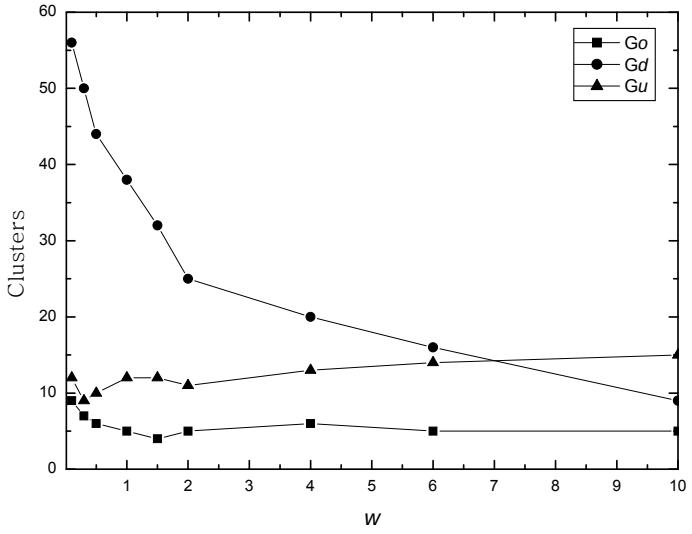

FIGURE 3: The influence of w on clustering results (Ruby)

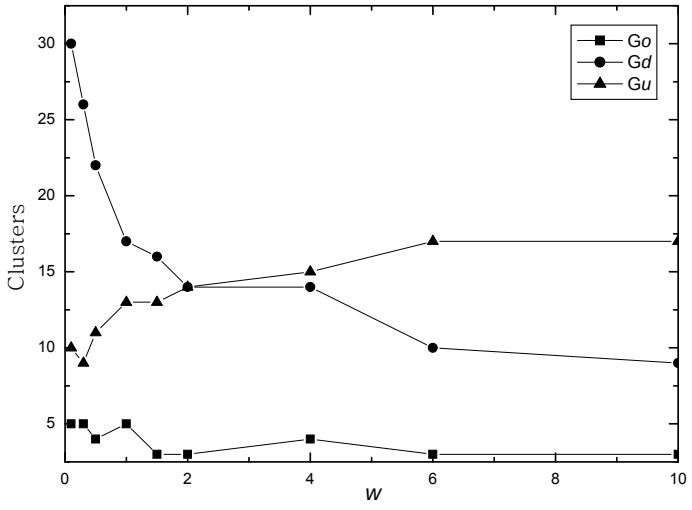

FIGURE 4: The influence of w on clustering results (Sapphire)

Another thing worthy to be noticed is that the quantity of selected features in our experiment is between 11 and 15. And there are 6 features are in all of these features which are RtnStmDecPrmNbr, RtnStmDecObjNbr, RtnArgXplSum, RtnStmCtlNbr, FilDecObjExtNbr and FilDefRtnNbr. We use these 6 common selected features to train a C4.5 classification tree. What surprised us is this classification tree can also generate good classification accuracy (more than 70%). It can be concluded that these 6 common selected features are the key features which influence final prediction result.

5.2 Comparison of classification accuracy

Figure 5 reports the Type I, Type II, and overall misclassification rate for GA classification model, compared to S-PLUS [28] and TreeDisc [29] model. We choose these two classifiers for comparison because they are latest used and known for good at software quality prediction. For each classification technique, we first obtain several candidate classification models. The one with a preferred balance between the Type I and Type II misclassification rate is chosen as the final model. Such a strategy is a practical solution for high-assurance software systems because supervised classification is difficult for such systems and often yields a classifier that predicts most of all components as not fault prone. The S-PLUS performed worse over Sapphire and only slightly better in terms of overall misclassification rate over Ruby than C4.5. However, the performance of GA is impressive over both of two products. Its overall misclassification rate is lower than S-PLUS and TreeDisc. Besides, the distance between Type I and Type II error rate is smaller than S-PLUS and TreeDisc. Which means it can work well in both of the two kinds of classification.

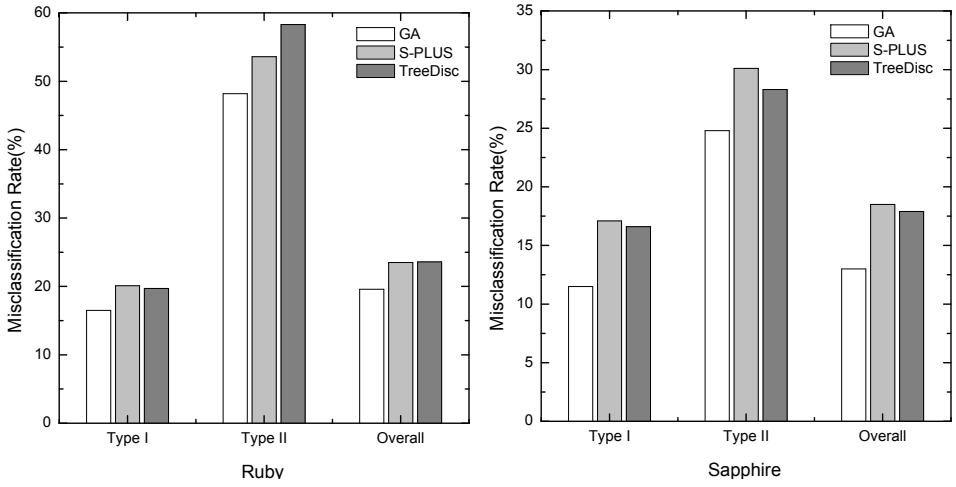

FIGURE 5: Classification result comparison of GA, S-PLUS and TreeDisc

6. CONCLUSION

This paper presents a software quality prediction model base on genetic algorithm in which outlier detection and feature selection are executed simultaneously. The clusters whose members are less than 5 are considered as outliers and cleaned from the data set. The other clusters are divided by purity further. For the clusters whose purity is high enough (more than 80%), proneness labels can be easily decided by majority label. Otherwise, clusters need extra classification executed by trained MLP. Both the criterions of clustering and feature selection are integrated in the fitness function of GA so that these two functions are accomplished in same evolution. Besides the outlier detection and feature selection, the trained software quality prediction model has excellent classification ability. The experimental results illustrate that this model performs better than some latest software quality classifiers such as S-PLUS and TreeDisc. In the future research, we plan to continue discussions with software engineers to better

evaluate and interpret this software quality prediction model. It's possible to build a more interactive system for software engineers and data analysts to find the patterns hided in the software development.

ACKNOWLEDGMENT

We thank the software engineers in Lucent Technologies for their constructive comments and suggestions, and Belle Wu for her patient review of the manuscript.

REFERENCES

[1] Khoshgoftaar, T. M., Allen, E. B., Jones, W. D., and Hudepohl, J. P. Accuracy of software quality models over multiple releases, in Annals of Software Engineering 9(1–4): 103–116.

[2] Khoshgoftaar, T. M., Allen, E. B., and Deng, J. Controlling overfitting in software quality models: experiments withregression trees and classification, in Proceedings: 7th International Software Metrics Symposium. London UK, 190–198. 2001.

[3] Gokhale, S. S., and Lyu, M. R. Regression tree modeling for the prediction of software quality, in H. Pham (ed.): Proceedings: 3rd International Conference on Reliability and Quality in Design. Anaheim, California, USA, 31–36. 1997.

[4] Taghi M. Khoshgoftaar, Edward B. Allen, Nishith Goel, Amit Nandi, and John McMullan. Detection of software modules with high debug code churn in a very large legacy system, in Proceedings of the Seventh International Symposium on Software Reliability Engineering, pages 364--371, White Plains, NY, October 1996. IEEE Computer Society.

[5] Finnie, G. R., Wittig, G. E., and Desharnais, J. M. A comparison of software effort estimation techniques: using function points with neural networks, case-based reasoning, and regression models. Journal of Systems and Software 39: 281–289. 1997.

[6] Khoshgoftaar, T. M., and Lanning, D. L. A neural network approach for early detection of program modules having high risk in the maintenance phase, Journal of Systems and Software 29(1):85–91. 1995.

[7] Ganesan, K., Khoshgoftaar, T. M., and Allen, E. B. Case-based software quality prediction, International Journal of Software Engineering and Knowledge Engineering 10(2): 139–152. World Scientic Publishing. 2000.

[8] Kolodner, J. Case-Based Reasoning. San Mateo, California, USA: Morgan Kaufmann Publishers, Inc. 1993.

[9] Berenson, M. L., Levine, D. M., and Goldstein, M. Intermediate Statistical Methods and Applications: A Computer Package Approach. Englewood Cliffs, NJ, USA: Prentice Hall. 1983.

[10] Xu, Z. Fuzzy logic techniques for software reliability engineering, Ph.D. thesis, Florida Atlantic University, Boca Raton, Florida USA. 2001.

[11] R. T. Ng and J. Han. Efficient and Effective Clustering Methods for Spatial Data Mining. In Proceedings of the 20th VLDB Conference, pages 144--155, Santiago, Chile, 1994.

[12] Ester M., Kriegel H.-P., Sander J., Xu X.: A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise, Proc. 2nd Int. Conf. on Knowledge Discovery and Data Mining, AAAI Press, 1996.

[13] Tian Zhang, Raghu Ramakrishnan, and Miron Livny. BIRCH: An Efficient Data Clustering Method for Very Large Databases. In Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, pages 103--114, Montreal, Canada, 1996.

[14] Jiang, M.F., S.S. Tseng, and C.M. Su. Two-phase clustering process for outliers detection, Pattern Recognition Letters, Vol 22, No. 6-7, pp. 691-700.

[15] Wettschereck D., Mohri T. and Aha D. W. A review and Comparative Evaluation of Feature Weighting Methods for Lazy Learning Algorithms, AI Review Journal. 1995.
[16] V.BARNETT, T.LEWIS. Outliers in Statistic Data, John Wiley's Publisher.
[17] Knorr, Edwin M. and Raymond T. Ng. A Unified Notion of Outliers Properties and Computation, 3rd International Conference on Knowledge Discovery and Data Mining Proceedings, 1997, pp. 219-222.
[18] Knorr, Edwin M. and Raymond T. Ng. Algorithms for Mining Distance-Based Outliers in Large Datasets, in Very Large Data Bases Conference Proceedings, 1998, pp. 24-27.
[19] Markus M. Breunig, Hans-Peter Kriegel, Raymond T. Ng, Jorg Sander. LOF: Identifying Densitybased Local Outliers, Proc. ACM SIGMOD 2000 Int. Conf. On Management of Data, Dalles, TX, 2000.
[20] Spiros Papadimitriou, Hiroyuki Kitagawa, Phillip B. Gibbons, Christos Faloutsos. LOCI: Fast Outlier Detection Using the Local Correlation Integral, 19th International Conference on Data Engineering, March 05 - 08, 2003, Bangalore, India.
[21] He, X. Xu, S.Deng. Discovering Cluster Based Local Outliers, Pattern Recognition Letters, Volume24, Issue 9-10, June 2003, pp.1641-1650.
[22] He, Z., X., Deng, S., 2002. Squeezer. An efficient algorithm for clustering categorical data, Journal of Computer Science and Technology.
[23] Arning, Andreas, Rakesh Agrawal, and Prabhakar Raghavan. A Linear Method for Deviation Detection in Large Databases, 2nd International Conference on Knowledge Discovery and Data Mining Proceedings, 1996, pp. 164-169.
[24] W. Siedlecki and J. Sklansky. A note on genetic algorithms for large scale feature selection, Pattern Recognit. Lett., vol. 10, pp. 335–347, 1989.
[25] W. F. Punch, E. D. Goodman, M. Pei, L. Chia-Shun, P. Hovland, and R. Enbody. Further research on feature selection and classification using genetic algorithms, in Proc. Int. Conf. Genetic Algorithms, 1993, pp. 557–564.
[26] J. D. Kelly and L. Davis. Hybridizing the genetic algorithm and the k nearest neighbors classification algorithm. in Proc. 4th Int. Conf. Genetic Algorithms Appl., 1991, pp. 377–383.
[27] M. L. Raymer, W. F. Punch, E. D. Goodman, P. C. Sanschagrin, and L. A. Kuhn. Simultaneous feature scaling and selection using a genetic algorithm, in Proc. 7th Int. Conf. Genetic Algorithms (ICGA), Th. Bäck, Ed. San Francisco, CA: Morgan Kaufmann, 1997, pp. 561–567.
[28] Khoshgoftaar, T. M., and Seliya, N. Software quality classification modeling using the SPRINT decision tree algorithm. In Proceedings: 14th International Conference on Tools with Artificial Intelligence. Washington, DC, USA: IEEE Computer Society, November, pp. 365–374. 2002.
[29] Khoshgoftaar, T. M., and Allen, E. B. Controlling overfitting in classification-tree models of software quality. Empirical Software Engineering 6(1): 59–79. 2001.