Journal of Zhejiang University SCIENCE A
ISSN 1673-565X (Print); ISSN 1862-1775 (Online)
www.zju.edu.cn/jzus; www.springerlink.com
E-mail: jzus@zju.edu.cn

Predicting the fault-proneness of class hierarchy in object-oriented software using a layered kernel

Peng HUANG, Jie ZHU[‡]
(*Department of Electronic Engineering, Shanghai Jiao Tong University, Shanghai 200240, China*)
E-mail: superhp@sjtu.edu.cn; zhujie@sjtu.edu.cn

Received Nov. 22, 2007; revision accepted June 26, 2008

Abstract: A novel kernel learning method for object-oriented (OO) software fault prediction is proposed in this paper. With this method, each set of classes that has inheritance relation named class hierarchy, is treated as an elemental software model. A layered kernel is introduced to handle the tree data structure corresponding to the class hierarchy models. This method was validated using both an artificial dataset and a case of industrial software from the optical communication field. Preliminary experiments showed that our approach is very effective in learning structured data and outperforms the traditional support vector learning methods in accurately and correctly predicting the fault-prone class hierarchy model in real-life OO software.

Key words: Object-oriented software, Fault-proneness, Support vector machine, Structured kernel
doi:10.1631/jzus.A0720073 **Document code:** A **CLC number:** TN914; TP311

INTRODUCTION

In the last decade, object-oriented (OO) method has become one of the most common paradigms in software design and development. However, evaluating the quality of OO software before release is always a challenge. Usually, OO metrics are used for the prediction of software module quality. Many researchers (Basili *et al.*, 1996; Khoshgoftaar *et al.*, 1997; Tang *et al.*, 1999; Briand *et al.*, 2000; Cartwright and Shepperd, 2000; Briand and Wust, 2002) worked on the development of prediction models for the classification of fault-prone modules. Chidamber and Kemerer (1994) introduced a suite of metrics for OO environments. Later, Tang *et al.*(1999) investigated the correlation between these CK metrics and three types of OO faults. Briand *et al.*(2000) used a multivariate statistical method to investigate the defect proneness of 64 metrics, divided into 4 groups. Following Briand's metrics, Kanmani *et al.*(2007) evaluated a predictor based on a neural network for some academic software.

However, the traditional procedure (Berard, 1998; Cartwright and Shepperd, 2000; Briand and Wust, 2002; Kanmani *et al.*, 2007) always treats class as a basic module, thus partly ignoring relational information hidden in the inheritance related classes. Furthermore, a single class is incomplete and practically meaningless because the definition of its inherited method is often located in another class. In this paper, a new learning method of defect prediction for OO software is introduced. We use the set of all inheritance related classes, the class hierarchy, as the basic module in defect prediction.

Recently, kernel methods (Shawe-Taylor and Cristianini, 2004) such as support vector machines (SVM) (Cortes and Vapnik, 1995; Vapnik, 1998; Scholkopf *et al.*, 2000) have prompted an explosion of interest in the machine learning and data mining communities. Some authors (Xing *et al.*, 2005; Jin *et al.*, 2006) reported on software quality prediction based on SVM. The attractiveness of kernel methods comes from the fact that they use only the inner products of the vector representations when they access the examples. The function computing the inner products is called the 'kernel'. As long as the

[‡] Corresponding author

kernel function is available, kernel methods can be applied in high dimensional feature spaces without suffering from the high cost of computing the mapped data. When we handle more complex objects such as sequences, trees, and graphs, the proper vector representation is not obvious. For a tree-structured class hierarchy, even though there are various metrics for single classes, the step of feature extraction is still difficult and time consuming.

Thus, many kernel functions (Zhang and Shasha, 1989; Haussler, 1999; Collins and Duffy, 2002; Gartner *et al.*, 2002; 2004; Bille, 2005) for structured data were devised for specific situations. Haussler (1999) proposed convolution kernels, a general framework for handling discrete data structure using kernel methods. Gartner *et al.*(2004) introduced several kernels for term, list, set, and multi-set. The first effective tree kernel was proposed by Collins and Duffy (2002) in an application of natural language processing. It was named ‘sparse tree kernel’, and was actually a special case of the convolution kernel (Haussler, 1999). The sparse tree kernel is defined by recursive comparison of every node of two trees. The node attribute used by the sparse tree kernel is the ingress degree, i.e., the number of the sub-nodes, which are merely structure-related. There are several improved versions deduced from the sparse tree kernel. Kashima and Koyanagi (2002) and Huang and Zhu (2007) introduced a labeled ordered tree kernel (Zhang and Shasha, 1989) and its extensions to allow elastic structure matching and label mutations.

The main aim of this study was to design a kernel and develop a method for learning and predicting the fault-proneness of the class hierarchy. To handle the class hierarchy in the OO software, a layered kernel was first designed for its special tree-like data structure. Before this kernel-based learning method was applied to real-life OO software, an artificial dataset was designed to prove its effectiveness. A case study was then executed to analyze some industrial software written in C++. The results showed that the SVM combined with the structured kernel achieved relatively high accuracy compared with the conventional attribute-value learning method in predicting the fault-proneness of code. Section 2 discusses the methodology of the model construction and prediction process. Various methods of predicting fault-proneness were applied to the artificial and real-life datasets. Results produced by the different methods are compared and discussed in Section 3 and our conclusions are summarized in Section 4.

METHODS

To achieve our aims, a dedicated methodology was developed and applied (Fig.1) for training and validating the fault prediction model for OO software. The same methodology was used for both a real-life dataset and an artificial dataset, except for some slight adjustments in a few steps.

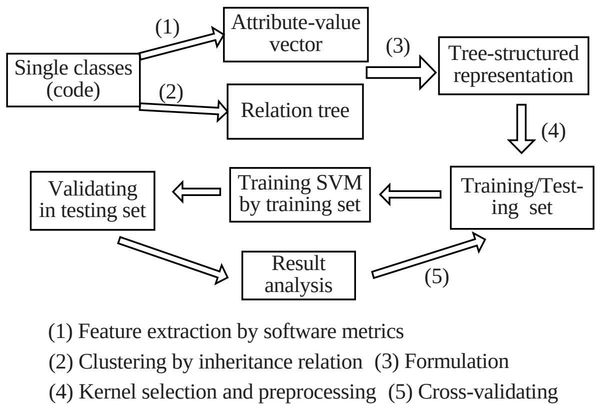
Single classes (code)
(1)
Attribute-value vector
(2)
Relation tree
(3)
Tree-structured representation
(4)
Training/Testing set
Training SVM by training set
Validating in testing set
Result analysis
(5)
(1) Feature extraction by software metrics
(2) Clustering by inheritance relation (3) Formulation
(4) Kernel selection and preprocessing (5) Cross-validating

Fig.1 Methodology for developing and validating the OO software fault-proneness predictor

The procedure of the methodology is as follows:

Step 1: Construct a proper vector for each class by measurement of a sample of *n* single classes, using *m* metrics.

Step 2: Formularize the tree-structured representation for each class hierarchy based on proper vectors for a single class. Suppose the class hierarchy dataset has *l* instances after the clustering.

Step 3: Prepare the training and testing dataset for cross-validation. The whole dataset is divided into several parts of equal size, and each time one part is selected for testing and the others for training.

Step 4: Train the SVM with the training dataset. The performance of the layered kernel was compared with that of four other kernels. Thus, a total of five models were constructed.

Step 5: Predict the fault-proneness of each class hierarchy (fault or non-fault) in the testing dataset.

Step 6: Compare the prediction result with the actual label in each instance. Calculate the evaluation parameters—misclassification rate, Type I and Type II error rates, and compare the prediction accuracy of the five models.

Step 7: Return to Step 3 until the cross-validation covers the whole dataset.

Software metrics for a single class

To construct the proper vector for a single class, the OO metrics already proposed and validated in the literature (Chidamber and Kemerer, 1994; Berard, 1998; Briand *et al.*, 2000) were considered. Because the software metrics were not our major focus, several commonly used OO metrics were selected from each of four groups (Briand *et al.*, 2000; Briand and Wust, 2002) to measure each class. In fact, fewer than 10 well-selected metrics are quite powerful in application (Berard, 1998). According to some authors (Briand *et al.*, 2000; Kanmani *et al.*, 2007), many existing OO metrics are based on similar ideas and hypotheses and therefore are almost always interrelated and somewhat redundant. Thus, a large number (e.g., 20 to 30, or more) of different metrics have to be used. Our analysis enabled us to identify the most useful metrics.

In this study, 21 software metrics (Table 1) were considered, covering the core set reported in other papers (Chidamber and Kemerer, 1994; Briand *et al.*, 2000; Kanmani *et al.*, 2007). The effectiveness inspection in the next section proved that they were sufficient for our real-life dataset. The Krakatau toolkit, developed by Power Soft (http://www.powersoftware.com/kp/), was used to measure the C++ code. Using Krakatau, the raw measurement for each class is obtained. All the measurements were standardized to a mean of zero and a variance of one for each metric to give a standardized metric-value vector for each class.

Representation of the class hierarchy

For class hierarchies, one needs a knowledge representation formalism that can accurately and conveniently depict these structured objects. The tree structure is a simple and natural choice. The tree-structured representation for class hierarchy, termed a 'hierarchy tree' (HT for short) in this study, is based on the vectors for each class. Each HT has a unique root node and its structure is extracted from the inheritance relation (Fig.2).

Table 1 Selected metrics

Abbreviation	Full name
CBO	Coupling between object classes
CSAO	Class size (attributes & operations)
CSA	Class size (attributes)
CSI	Class specialization index
CSO	Class size (operations)
DIT	Depth of inheritance tree
LOC	Lines of code
LOCM	Lack of cohesion methods
NAAC	Number of attributes added in the class
NAIC	Number of attributes inherited in the class
NAOC	Number of operations added in the class
NOIC	Number of operations inherited in the class
NPavgC	Average number of method parameters in the class
NSUB	Number of sub-classes
OSavg	Average operation size
PA	Private attribute usage
PPPC	Percentage of public/protected members
RFC	Response for the class
SLOC	Source lines of the code
TLOC	Total lines of the code
WMC	Weighted methods in the class

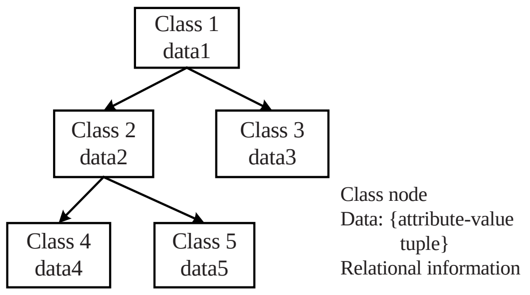

Fig.2 Hierarchy tree structure

Inheritance is a common OO programming technique intended to aid code sharing and re-use. The inheritance mechanism allows a class (derived class) to inherit behaviors and features from another class, called the 'base class'. There are various inheritance mechanisms for OO programming languages, such as single inheritance, multiple inheritance, interface inheritance and association inheritance. Among these, only single and multiple inheritances that determine HT structure need separate discussion below. The other inheritance mechanisms can be handled by the kernel.

If only single inheritance exists in the OO programming language (such as Java or VB.Net), the construction of an HT is straightforward and explicit. But when multiple inheritance mechanism (such as in C++ or Perl) is allowed, two base classes may have the same derived class. Thus, one class may appear in several different HTs after the classes are clustered by the inheritance relation (Fig.3).

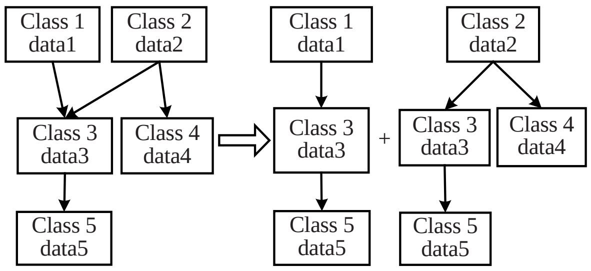

Fig.3 Hierarchy tree structure for multiple inheritance relation

Layered kernels

Gartner *et al*. introduced a set kernel (Haussler, 1999; Gartner *et al.*, 2004) derived from definition of the convolution kernel:

$$K_{\mathrm{SET}}(X,X'):=\sum_{x\in X,x'\in X'}K_{\chi}(x,x'),\tag{1}$$

where K_χ is a kernel defined on χ, and $X, X'\subseteq\chi$.

The flexible tree kernel (Zhang and Shasha, 1989) is a recent development of the sparse tree kernel (Collins and Duffy, 2002; Kashima and Koyanagi, 2002). This kernel is based on counting the number of all possible tree mapping (Zhang and Shasha, 1989; Kuboyama and Shin, 2006) under some constraint. The tree representation for OO class hierarchy is unordered and the left-to-right relation among the nodes in the same layer is not respected. Thus, the recursive computation according to node-to-node relation can be substituted by layer-to-layer to avoid relatively high computational cost. Here we introduce a layered tree kernel deduced from the flexible tree kernel (Kuboyama and Shin, 2006) using set kernel. The recursive definition is given as follows. The function K_{L} is trivially symmetric and positive semi-definite, so it is a kernel.

Definition 1 Layered kernel

$$K_{\mathrm{L}}(F,\phi)=K_{\mathrm{L}}(\phi,F)=0,\tag{2}$$

$$\begin{aligned}K_{\mathrm{L}}(F_1,F_2)=&K_{\mathrm{SET}}(l_{\mathrm{t}}(F_1),l_{\mathrm{t}}(F_2))[1+K_{\mathrm{L}}(l_{\mathrm{r}}(F_1),l_{\mathrm{r}}(F_2))]\\&+K_{\mathrm{L}}(l_{\mathrm{r}}(F_1),F_2)+K_{\mathrm{L}}(F_1,l_{\mathrm{r}}(F_2))-K_{\mathrm{L}}(l_{\mathrm{r}}(F_1),l_{\mathrm{r}}(F_2)),\end{aligned}\tag{3}$$

where $l_{\mathrm{t}}(F)$ and $l_{\mathrm{r}}(F)$ return the first layer and remaining structure of the forest F, respectively.

Support vector machine

In this paper, the problem is limited to classification. The support vector classifier (C-SVC) is formulated as the following quadratic optimization problem (Cortes and Vapnik, 1995; Pontil and Verri, 1998; Vapnik, 1998):

$$\begin{aligned}&\min\left(\frac{1}{2}\langle w,w\rangle+C\sum_i\xi_i\right)\\&\text{s.t.}\quad y_i\left(\langle w,x_i\rangle+b\right)\ge 1-\xi_i,\ \xi_i\ge 0.\end{aligned}\tag{4}$$

Here, the chosen classifier is a soft-margin C-SVC (Cortes and Vapnik, 1995; Vapnik, 1998; Chang and Lin, 2001) that allows relaxed constraints on the boundary. Usually, the primal problem is transformed into the dual space for the convenient solution:

$$\begin{aligned}&\min\left(\frac{1}{2}\sum_i\sum_j\alpha_i\alpha_j y_i y_j\langle x_i,x_j\rangle-\sum_i\alpha_i\right)\\&\text{s.t.}\quad 0\le\alpha_i\le C,\ \sum_i y_i\alpha_i=0.\end{aligned}\tag{5}$$

The decision function is

$$f(x)=\mathrm{sgn}\left(\sum_i y_i\alpha_i\langle x_i,x\rangle+b\right).\tag{6}$$

The application and implementation of most of the training and testing below is simplified by using the SSVM (simple SVM) toolkit (Vishwanathan *et al.*, 2003), which is written in MATLAB.

Evaluation of the results

Validation of the prediction method varies depending on which evaluation approaches are used (Khoshgoftaar *et al.*, 1997). In this study, the instances are classified into two categories—the fault-prone (positive) and the non-fault-prone (negative). Here n_1 denotes the number of positive instances that are wrongly classified as negative (Type I errors); n_2 denotes the number of false-positive instances that are negative but classified as positive (Type II errors); n_3 and n_4 denote the total number of positive and negative instances, respectively. We will compute the proportion of Type I (n_1/n_3), Type II (n_2/n_4) and total misclassification errors ($(n_1+n_2)/(n_3+n_4)$) in the whole dataset. The correct prediction percentage is then calculated to compare the various methods.

APPLICATION AND EXPERIMENTS

In this section, the artificial dataset and real-life software are described first. The real-life codes were processed and divided into three different datasets, giving a total of four datasets for the experiment. The experimental results and comparisons listed in the tables are discussed.

Artificial dataset

Before our scenario was applied to real-life or industrial software, an artificial dataset was designed to verify its effectiveness. A dataset of 100 randomly generated bi-trees (Fig.4) was constructed. The maximum depth of each bi-tree was limited to 3. For each node in the tree, an integer ranging from 1 to 30 was randomly assigned for its data. The learning task to investigate was a 2-category classification problem according to the indicator function below:

$$\mathrm{sum}(T) = \sum_{Node} Node.data \times \alpha_{Node},\ \ Node \in T, \quad (7)$$

$$label(T) = \begin{cases} +1, & \mathrm{sum}(T) \geq 120, \\ -1, & \mathrm{sum}(T) < 120. \end{cases} \quad (8)$$

Note that α is the structure-related parameter for a node.

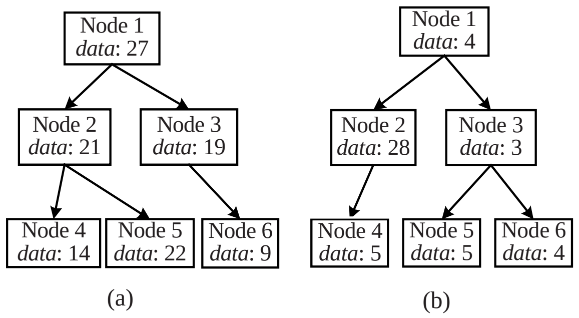

Fig.4 Bi-trees in the artificial dataset. (a) A positive bi-tree instance; (b) A negative bi-tree instance

The bi-tree dataset has 52 positive and 48 negative instances. Here α is the depth of the node in the tree and the similarity between the nodes is defined as

$$\sigma = 1 - (|data1 - data2|)/30.$$

Real-life telecommunication software

The real-life software case used in this study was taken from the optical communication domain. This software mainly implements commuting and transporting data from 2k to 10G bits (Fig.5) in synchronous digital hierarchy (SDH) networks. It was developed by the software team (SW) in the optical networks department of Alcatel-Lucent, Shanghai. Another independent system verification team (SVT) from the same department undertakes the testing. After the codes pass the functional test performed by the SW, they are verified by the SVT in the communication environment. Hundreds of test cases, including both manual and automation tests, are executed to ensure product quality. If the test results do not meet requirements, a modification request (MR) is proposed. According to the following rules, each MR is assigned a severity level:

Severity 5: Complete or partial service outage, significant decrease in capacity, loss of critical function.

Severity 4: Loss or degradation of some system functionality, but no loss of service.

Severity 3: Loss or degradation of non-essential functionality, or failure to conform to specifications, but overall operation is unimpaired.

Severity 2: Minor problem or enhancement request.

Severity 1: Scheduled work.

In our work, the classification label of the class hierarchy instance is based on the number of MRs and

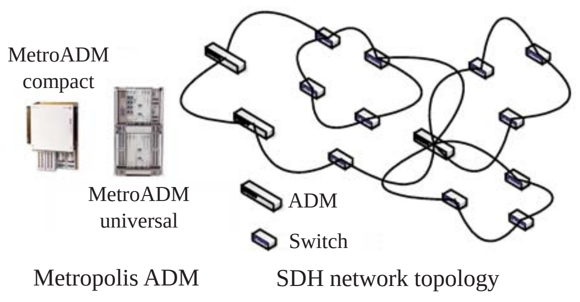

Fig.5 Alcatel-Lucent product in a modern communication system

their severity. Generally, if an instance's weighted sum of the related MRs is more than 2.5, it is viewed as a fault category.

The software release comprises about 17000 lines and a total of 425 classes. Among these classes, 126 are mainly implementing interaction and command execution, 184 are handling the data access and memory allocation in the embedded system and 63 are realizing switching protocol such as generic framing procedure (GFP) and rapid spanning tree protocol (RSTP). After clustering by inheritance relation, the whole dataset has 175 instances including 100 single classes. The mean inheritance height of all instances and instances with more than one class is 1.57 and 2.33, respectively. Because multiple inheritance mechanism tends to result in code collision and misunderstanding, it is seldom used in this release.

Since the software release delivered to the SVT has already been carefully debugged, the distribution of positive and negative instances in the whole dataset tends to be unbalanced (Osuna *et al.*, 1997; Seliya and Khoshgoftaar, 2007). According to the MRs related to all the 175 grouped instances, only 23 instances are labeled as faulty (negative) compared with 152 labeled as non-faulty (positive). To deal with the unbalanced data, some researchers (Osuna *et al.*, 1997; Chang and Lin, 2001) proposed using different penalty parameters in the SVM training. Here, different penalty parameters *C* (refer to Eq.(5)) are assigned to the positive and negative instances. Only the best prediction results are listed in the tables. Besides the whole dataset, two other sub-sets were constructed to inspect the performance of our method under different instance distributions.

Results and discussion

For the artificial dataset, 10-fold cross-validation was used to compute the proportion of Type I error, Type II error, and the total misclassification error. The experimental results (Table 2) showed that the proper kernels had similar classification accuracy, and that the set kernel had slightly better performance. The layered kernel performed best, being about 20% more accurate than the set kernel. The layered kernel also gave the best results in terms of Type I and Type II error rates.

For the real-life dataset, metrics should first be selected to measure attributes of each class. A total of 21 metrics (Table 1) were considered, of which 20 were used to construct vector for each class after eliminating the measures with the maximum number of zeros, as recommended by Briand *et al.*(2000). Because OO metrics were not our focus, we evaluated only effectiveness. Ten-cross-validation among all 420 instances (each class) was roughly taken (RBF kernel) first, and the result (90.62% accuracy) supports the usefulness of the selected metrics. Table 3 shows the prediction results of the five trials over the whole dataset. When five-cross-validation is used, the proper kernels still had a similar performance, and the accuracy of the set kernel was low. The layered kernel performed best. The problem is that all the kernels had high Type II error rates.

Table 2 Comparison of results using the bi-tree dataset

Method	Accuracy (%)	Type I error (%)	Type II error (%)	Total error (%)
Layered kernel	**91.00**	**13.17**	**4.44**	**9.00**
Set kernel	72.00	21.67	25.63	28.00
Linear	67.00	61.17	14.17	33.00
Gaussian	68.00	61.11	12.22	32.00
RBF	64.00	51.50	22.32	36.00

Highlighted number refers to the best result among the different methods

Table 3 Comparison of results for the whole dataset (unbalanced raw dataset)

Method	Accuracy (%)	Type I error (%)	Type II error (%)	Total error (%)
Layered kernel	**87.71**	**1.93**	87.14	**12.29**
Set kernel	61.71	52.38	**61.71**	38.29
Linear	81.14	8.02	90.42	19.86
Gaussian	80.43	4.75	92.14	19.57
RBF	78.86	10.52	90.48	21.14

Highlighted number refers to the best result among the different methods

Tables 4 and 5 show the results from dataset 1 (100 instances in total) and dataset 2 (50 instances in total), which both include all 23 fault instances and a random selection of others. The same five trials were executed on the two sets. Table 4 shows that for dataset 1 the layered kernel still had the best performance but the Type II error rate was still high. For dataset 2 (Table 5), the layered kernel had a much lower Type II error rate and still led the other kernels in prediction accuracy, which proves the effectiveness of the layered kernel and our method. The reason for the better result is that dataset 2 was more balanced than dataset 1. The high Type II error rate in the raw dataset and dataset 1 mainly comes from the unbalanced distribution of the instances, but not the kernel. The low Type I error rates and high Type II error rates mean that the fault class hierarchy cannot always be detected, but the benign class hierarchy is hardly ever misclassified. Therefore the classification process can be usefully applied before testing.

Table 4 Comparison of results from dataset 1 (100 instances)

Method	Accuracy (%)	Type I error (%)	Type II error (%)	Total error (%)
Layered kernel	**84.00**	4.36	73.33	**16.00**
Set kernel	66.50	52.38	79.23	33.50
Linear	76.00	**3.93**	90.00	24.00
Gaussian	76.00	9.11	73.62	23.00
RBF	70.00	16.96	**71.67**	30.00

Highlighted number refers to the best result among the different methods

Table 5 Comparison of results using dataset 2 (balanced data, 50 instances)

Method	Accuracy (%)	Type I error (%)	Type II error (%)	Total error (%)
Layered kernel	**89.10**	**9.20**	**13.10**	**10.90**
Set kernel	46.00	**0.00**	100.00	54.00
Linear	84.00	10.00	23.30	16.00
Gaussian	52.00	90.00	**0.00**	48.00
RBF	48.00	100.00	0.00	52.00

Highlighted number, except the zero type I/II error rate, which is a minimum but not reasonable result, refers to the best result among the different methods

CONCLUSION AND FUTURE WORK

Bringing kernel methods and relational data structure together is a promising direction for practical software quality prediction. This requires both designing an effective data representation for the software and defining a positive definite kernel on the data structure. In this paper, we discussed the applicability of a kernel method to class hierarchy quality prediction in OO software. We modeled each class hierarchy by tree-structured representation and introduced a layered kernel deduced from the flexible tree kernel (Kuboyama and Shin, 2006) and set kernel (Haussler, 1999; Gartner *et al.*, 2002). First, we applied the layered kernel to an artificial dataset to show the general validity of our method. Next, we performed some experiments on a case of real-life software from the optical communication domain. Although the experiments in this paper were limited, we could demonstrate at least to some extent that our method has some predictive power for the class hierarchy in OO software, and that our layered kernel can efficiently capture the structural information among the classes. The experimental results are encouraging and show the potential applicability of the structured kernel for analyzing real-life OO software. Our layered kernel and tree data structure performed better than the traditional attribute-value support vector learning. In the future, we plan to extend our work to more complex authentic OO software quality prediction, and experiment with other structured kernels potentially better suited for the class hierarchy.

References

Basili, V., Briand, L., Melo, W., 1996. A validation of object-oriented design metrics as quality indicators. *IEEE Trans. on Software Eng.*, **22**(10):751-761. [doi:10.1109/32.544352]

Berard, E.V., 1998. Metrics for Object-oriented Software Engineering. Available at http://www.ipipan.gda.pl/~marek/objects/TOA/moose.html

Bille, P., 2005. A survey on tree edit distance and related problems. *Theor. Comput. Sci.*, **337**(1-3):217-239. [doi:10.1016/j.tcs.2004.12.030]

Briand, L., Wust, J., 2002. Empirical Studies of Quality Models in Object-oriented Systems. Proc. Advances in Computers. Elsevier, London, p.98-167.

Briand, L., Wust, J., Daly, J., Victor, P.D., 2000. Exploring the relationships between design measures and software quality in object-oriented systems. *J. Syst. Software*, **51**(3):245-273. [doi:10.1016/S0164-1212(99)00102-8]

Cartwright, M., Shepperd, M., 2000. An empirical investigation of an object-oriented software system. *IEEE Trans. on Software Eng.*, **26**(8):786-796. [doi:10.1109/32.879814]

Chang, C.C., Lin, C.J., 2001. LIBSVM: A Library for Support Vector Machines. Available at http://www.csie.ntu.edu.tw/~cjlin/libsvm

Chidamber, S., Kemerer, C., 1994. A metrics suite for object-oriented design. *IEEE Trans. on Software Eng.*, **20**(6):476-493. [doi:10.1109/32.295895]

Collins, M., Duffy, N., 2002. Convolution Kernels for Natural Language. Proc. Advances in Neural Information Processing Systems. MIT Press, Cambridge, MA, USA, p.625-632.

Cortes, C., Vapnik, V., 1995. Support vector networks. *Machine Learning*, **20**(3):273-297. [doi:10.1007/BF00994018]

Gartner, T., Flach, P.A., Kowalczyk, A., Smola, A.J., 2002. Multi-instance Kernels. Proc. 19th Int. Conf. on Machine Learning. Morgan Kaufmann Publishers, San Francisco, CA, USA, p.179-186.

Gartner, T., Lloyd, J., Flach, P.A., 2004. Kernels and distance for structured data. *Machine Learning*, **57**(3):205-232. [doi:10.1023/B:MACH.0000039777.23772.30]

Haussler, D., 1999. Convolution Kernels on Discrete Structures. Technical Report USSC-CRL 99-10. Department of Computer Science, University of California at Santa Cruz, Santa Cruz, CA, USA.

Huang, P., Zhu, J., 2007. Decomposition method for tree kernels. *LNCS*, **4492**:593-601.

Jin, X., Liu, Z.D., Bie, R.F., Zhao, G.X., Ma, J.X., 2006. Support vector machines for regression and applications to software quality prediction. *LNCS*, **3994**:781-788. [doi:10.1007/11758549_105]

Kanmani, S., Uthariaraj, V.R., Sankaranarayanan, V., 2007. Object-oriented software fault prediction using neural networks. *Inf. Software Technol.*, **49**(5):483-492. [doi:10.1016/j.infsof.2006.07.005]

Kashima, H., Koyanagi, T., 2002. Kernels for Semi-structured Data. Proc. 9th Int. Conf. on Machine Learning. Morgan Kaufmann Publishers, San Francisco, CA, USA, p.291-298.

Khoshgoftaar, T.M., Allen, E.B., Hudepohl, J.P., Aud, S.J., 1997. Application of neural networks to software quality modeling of a very large telecommunications systems. *IEEE Trans. on Neural Networks*, **8**(4):902-909. [doi:10.1109/72.595888]

Kuboyama, T., Shin, H.K., 2006. Flexible Tree Kernels Based on Counting the Number of Tree Mapping. Proc. Machine Learning on Graphs. Springer, Berlin, Germany, p.69-77.

Osuna, E., Freund, R., Girosi, F., 1997. Support Vector Machines: Training and Applications. Technical Report AI Memo 1602. Massachusetts Institute of Technology, Cambridge, MA, USA.

Pontil, M., Verri, A., 1998. Properties of support vector machines. *Neural Comput.*, **10**(4):955-974. [doi:10.1162/089976698300017575]

Scholkopf, B., Smola, A., Williamson, R., Bartlett, P.L., 2000. New support vector machine. *Neural Comput.*, **12**(5):1207-1245. [doi:10.1162/089976600300015565]

Seliya, N., Khoshgoftaar, T.M., 2007. Software quality estimation with limited fault data: a semi-supervised learning perspective. *Software Qual. J.*, **15**(3):327-344. [doi:10.1007/s11219-007-9013-8]

Shawe-Taylor, J., Cristianini, N., 2004. Kernel Methods for Pattern Analysis. Cambridge University Press, New York. [doi:10.2277/0521813972]

Tang, M.H., Kao, M.H., Chen, M.H., 1999. An Empirical Study on Object-oriented Metrics. Proc. 6th Int. Conf. on Software Metrics Symp.. IEEE Computer Society, Washington, DC, USA, p.242-249. [doi:10.1109/METRIC.1999.809745]

Vapnik, V., 1998. Statistical Learning Theory. Wiley, New York.

Vishwanathan, S.V.N., Smola, A.J., Murty, M.N., 2003. SimpleSVM. Proc. 20th Int. Conf. on Machine Learning. IEEE Computer Society, Washington, DC, USA, p.760-767.

Xing, F., Guo, P., Lyu, M.R., 2005. A Novel Method for Early Software Quality Prediction Based on Support Vector Machine. Proc. 16th IEEE Int. Symp. on Software Reliability Engineering. IEEE Computer Society, Washington, DC, USA, p.213-222. [doi:10.1109/ISSRE.2005.6]

Zhang, K., Shasha, D., 1989. Simple fast algorithm for the editing distance between trees and related problems. *SIAM J. Comput.*, **18**(6):1245-1262. [doi:10.1137/0218082]