

Multiple kernel ensemble learning for software defect prediction

Tiejian Wang¹ · Zhiwu Zhang² · Xiaoyuan Jing¹ ·
Liqiang Zhang¹

Received: 16 June 2014 / Accepted: 24 March 2015 / Published online: 7 April 2015
© Springer Science+Business Media New York 2015

Abstract Software defect prediction aims to predict the defect proneness of new software modules with the historical defect data so as to improve the quality of a software system. Software historical defect data has a complicated structure and a marked characteristic of class-imbalance; how to fully analyze and utilize the existing historical defect data and build more precise and effective classifiers has attracted considerable researchers' interest from both academia and industry. Multiple kernel learning and ensemble learning are effective techniques in the field of machine learning. Multiple kernel learning can map the historical defect data to a higher-dimensional feature space and make them express better, and ensemble learning can use a series of weak classifiers to reduce the bias generated by the majority class and obtain better predictive performance. In this paper, we propose to use the multiple kernel learning to predict software defect. By using the characteristics of the metrics mined from the open source software, we get a multiple kernel classifier through ensemble learning method, which has the advantages of both multiple kernel learning and ensemble learning. We thus propose a multiple kernel ensemble learning (MKEL) approach for software defect classification and prediction. Considering the cost of risk in software defect prediction, we design a new sample weight vector updating strategy to reduce the cost of risk caused by misclassifying defective modules as non-defective ones. We employ the widely used NASA MDP datasets as test data to evaluate the performance of all compared methods; experimental results show that MKEL outperforms several representative state-of-the-art defect prediction methods.

✉ Tiejian Wang
wangtiejian1116@sina.com

¹ State Key Laboratory of Software Engineering, School of Computer, Wuhan University, Wuhan, China

² School of Computer, Nanjing University of Posts and Telecommunications, Nanjing, China

Keywords Software defect prediction · Multiple kernel learning · Ensemble learning · Multiple kernel ensemble learning (MKEL)

1 Introduction

Software defect prediction is an important research topic in software engineering (Lyu 2007; Seliya et al. 2010; Nam et al. 2013). By means of metric based classification, software modules can be classified typically into two categories: defective and non-defective. Metric based defect prediction method can predict the defect proneness of new software modules with software historical defect data, which is an efficient means to relieve the burden on software code inspection or testing (Catal and Diri 2009; Hall et al. 2011). The historical defect data is obtained by using McCabe (1976), Halstead (1977) and other static software metrics, how to fully analyze and utilize the existing historical defect data, and build more precise and effective classifiers has attracted considerable researchers' interest from both academia and industry.

Software historical defect data has a complicated structure (Gray et al. 2011; Shepherd et al. 2013), which will leads to negative influence on decision of classifiers unless we focus on them when we design the software defect prediction algorithm (Ma et al. 2012; Luo et al. 2012; Gao et al. 2011; Ren et al. 2014). Kernel based learning method (Schoelkopf et al. 1998; Scholkopf et al. 1999; Muller et al. 2001), however, can map these data into a higher or even infinite dimensional kernel feature space with the original data distribution information. In this way more information is excavated and the classification performance is also improved. Compared with single kernel learning, multiple kernel learning (Ong et al. 2005; Lewis et al. 2006; Rakotomamonjy et al. 2007; Zien and Ong 2007; GÅonon and Alpaydin 2008; Damoulas and Girolami 2008; Gehler and Nowozin 2009; Kembhavi et al. 2009) can combine the mapping ability of each single basic kernel function, which solves the kernel functions selection problem effectively. It can also assemble different kernel functions with different characteristics, achieve better mapping capability by taking advantage of each basic kernel function, make the data express better in the new feature space, and improve the prediction accuracy significantly. In general, multiple kernel learning (MKL) can handle some complicated situations such as heterogeneous information or irregular multi-dimensional data, large scale problems, non-flat distribution of samples, etc.

Software historical defect data also has a marked characteristic of class-imbalance (Khoshgoftaar et al. 2010; Zhou and Liu 2006; Menzies et al. 2007; He and Garcia 2009; Gao et al. 2012), which means that defective modules are far less than non-defective ones in most software systems. The class-imbalance problem leads to negative influence on decision of classifiers (Ren et al. 2014). Many methods including resampling techniques, ensemble learning and cost-sensitive learning have been proposed to solve the class-imbalance problem (Sun et al. 2012; Dieterich 2000; Valentini and Masulli 2002; Rokach 2010). Compared with other methods, ensemble learning can obtain better predictive performance by integrating the learning results in a series of weak classifiers. As one of the most popular ensemble learning methods, Adaptive Boosting (known as "Adaboost") updates the weight of the samples dynamically according to the error rate of last learning, thus reduces the bias generated by the majority class and solves the class-imbalance problem (Freund and Schapire 1997).

Research shows that ensemble learning has better prediction capability for software historical defect data with the characteristic of “class-imbalance” in software defect prediction (Rokach 2010).

1.1 Motivation

Combining multiple kernel learning with ensemble learning is of crucial necessity to process these software historical defect data. It can not only map the historical defect data to a higher-dimensional feature space to excavate more effective information, but also solve the problem of class-imbalance. Moreover, multiple kernel learning problem can be solved through ensemble learning method, which is called multiple kernel boosting learning (Xia and Hoi 2013; Bi et al. 2004; Bennett et al. 2002). Compared with the ordinary multiple kernel learning methods, multiple kernel boosting learning can avoid the complex parameter optimization problem and adjust the accuracy of the classifier according to different application requirements.

Additionally, misclassifying a defective software module as non-defective one is much more dangerous than misclassifying a non-defective module as defective-prone one in software defect prediction (Zheng 2010). That is because classifying a software module as defective-prone implies that more testers should be invested in the verification activities, in contrast, misclassifying a defective module as non-defective carries the risk of system failure. However, this problem has yet been addressed in the process of updating the sample weight in general multiple kernel boosting learning method. Instead, the sample weight is adjusted directly according to the classification results. So when multiple kernel boosting learning method is used for software defect prediction, this problem should be focused on.

1.2 Contribution

In this paper, we propose a multiple kernel ensemble learning (MKEL) approach for software defect prediction. And the contributions are summarized as following:

1. It is the first attempt to introduce the multiple kernel learning technique into the field of software defect prediction to the best of our knowledge, although it has been effectively applied to other domains. Through boosting method we get the most optimal multiple kernel classifier, in consequence, our MKEL approach has the advantages of both multiple kernel learning and ensemble learning for software defect prediction.

2. Considering the cost of risk in software defect prediction, we design a new sample weight vector updating strategy. In the training process of MKEL, we increase the weights of the defective samples and decrease the weights of the non-defective samples according to the classification results, so as to focus more on those defective modules, reduce the cost of risk of misclassifying defective modules as non-defective ones and gain better prediction.

In this paper, we conduct experiments on twelve NASA datasets, which are public and widely used for software defect prediction. The experimental results demonstrate that the proposed approach outperforms several representative methods. The remainder part of this paper is organized as follows. Section 2 introduces the related work.

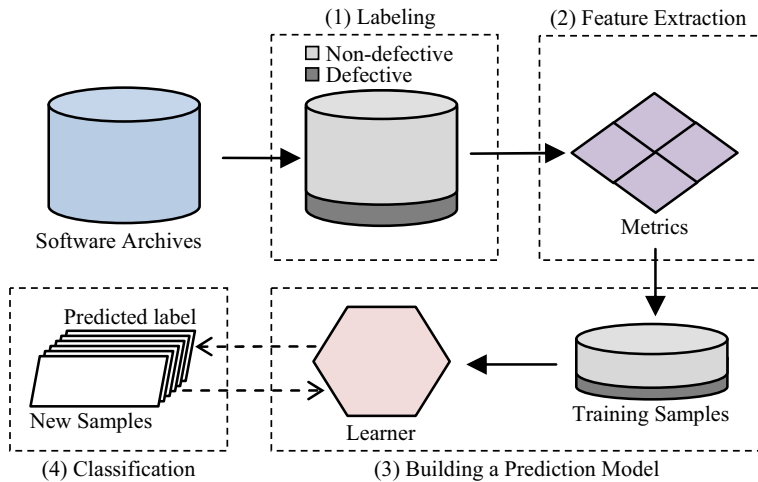


Fig. 1 Software Defect Prediction Process

Section 3 describes the proposed approach. Section 4 introduces the experimental setup. Section 5 shows the experimental results and analysis. Section 6 presents threats to validity of the models and the conclusion is drawn in Section 7.

2 Related work

Software defect prediction technology can be generally categorized into two types: static and dynamic defect prediction technology. The illustration of typical static defect prediction process is shown in Fig. 1. The first step is to collect software modules in software archives and then label them. A software module can be labeled as defective or non-defective according to whether it contains defects or not. Then, the values of defect prediction metrics, such as McCabe, Halstead metrics and et al., are used as module attributes. The module attributes and labels are used to train a learner for building a prediction model. With the prediction model, the new query modules can be whether they are defective or non-defective.

Many traditional classification methods in machine learning have been adopted for static software defect prediction, such as SVM (Xing et al. 2005; Elish and Elish 2008; Gray et al. 2009; Yan et al. 2010), Bayes (Turhan and Bener 2007, 2009; Amasaki et al. 2003; Wang and Li 2010), decision tree (Khoshgoftaar and Seliya 2002a, b; Wang et al. 2012; Breiman 2001; Gayatri et al. 2010), neural network (Thwin and Quah 2005; Paikari et al. 2012), dictionary learning (Jing et al. 2014), and etc. In order to solve the class-imbalance problem, many methods are used in software defect prediction, such as resampling (Menzies et al. 2007), ensemble learning (Sun et al. 2012; He and Garcia 2009; Wang and Yao 2013; Seiffert et al. 2009; Aljamaan and Elish 2009), and cost-sensitive (Zheng 2010; Bezerra et al. 2011; Seliya and Khoshgoftaar 2011; Sun et al. 2007). Sun et al. (2012) presented a coding-based ensemble learning method, which converts imbalanced binary-class data into balanced multiclass data and builds a defect predictor on the multiclass data with a specific coding scheme. Based on the successful

class-imbalance learning method AdaBoost.NC (He and Garcia (2009)), Wang et al. (2013) presented a dynamic version of AdaBoost.NC, which can adjust its parameter automatically during training process. Zheng (2010) presented cost-sensitive boosting algorithm to improve neural network classifiers for defect prediction, which incorporates the misclassification costs into the weight-update rule of boosting, such that the classification performance on those samples with higher misclassification costs can be improved.

In order to improve the accuracy of the prediction model, some researchers consider introducing kernel method to software defect prediction. The historical defect data were first mapped into a higher-dimensional feature space and then the software modules were classified and predicted in that kernel space. Ma et al. (2012) used a single kernel function to map the defect prediction data into a higher-dimensional feature space, and then used Kernel principal component analysis (KPCA) method for software defect prediction. Luo et al. (2012) mapped the defect prediction data into a higher-dimensional feature space in the same way, but Kernel partial least squares (KPLS) method was used for software defect prediction. Ren et al. (2014) used asymmetric Kernel partial least squares classifier (AKPLSC) and asymmetric Kernel principal component analysis classifier (AKPCAC) to solve the class imbalance problem by applying kernel function to the classifiers. All of them constructed an asymmetric classifier in kernel space, so as to improve software defect prediction by compensating the bias of the regression model caused by class-imbalance.

In general, multiple kernel learning method has obvious advantage compared with single kernel learning method, so in this paper we use multiple kernel learning method to predict the defect proneness of software modules and propose multiple kernel ensemble learning (MKEL) algorithm. Different from previous algorithms, our MKEL algorithm integrates the advantages of multiple kernel learning and ensemble learning, and applies multiple kernel learning to software defect prediction in boosting framework for the first time. Moreover, in the parts of weight vector updating, according to the characteristic of historical defect data, we take the cost of risk problem in software defect prediction account to improve the defect prediction performance.

3 Our approach

3.1 Problem formulation

For a given set of training software modules $D = \{(x_i, y_i), i = 1, 2, \dots, N\}$ and a collection of M kernel functions $K = \{k_j : X \times X \rightarrow \mathbb{R}, j = 1, 2, \dots, M\}$, where x is a vector of module attributes, $y \in \{-1, +1\}$ is the module label, k is the basic single kernel function. MKEL aims to learn a multi-kernel-based classifier $f(x)$, which is an ensemble of kernel classifiers using the collection of M kernels trained from the given training software historical defect data, and then predict the defect proneness of new software modules with the multi-kernel-based classifier $f(x)$. Typically, we express such a multi-kernel-based classifier as:

$$f(x) = \sum_{t=1}^T \alpha_t f_t(x), \quad (1)$$

where T is the total number of boosting trials, f_t is a kernel-based hypothesis learned from the t th ($1 \leq t \leq T$) boosting trial, and α_t is its associated weight in the final classifier. The main challenge of MKEL is to develop an effective boosting scheme to learn the optimal kernel-based hypothesis f_t and its combination weight α_t at each boosting trial. When T boosting trials are completed, we get T kernel-based hypothesis and their combination weights; the final MKEL classifier is an ensemble of them. Once the MKEL classifier is got, it can be used to make defect prediction for new query modules.

3.2 Multiple kernel learning

Given a supervised machine learning problem $(x_i, y_i) \in X \times Y$, we can map the input samples into a new feature space $F = \{\Phi(x)|x \in X\}$ through nonlinear mapping $\Phi : X \rightarrow F \quad x \rightarrow \Phi(x)$, then use the new representation to consider original learning problem. The kernel method is to replace the dot-product $\langle \Phi(x), \Phi(x') \rangle$ in kernel feature space by a kernel function $k(x, x')$ in the original feature space. The commonly used kernels include Polynomial kernels, Gaussian radial basis function (RBF) kernels, and etc. The Polynomial kernels has the form $k(x, x') = (\langle x, x' \rangle + \theta)^v$, and RBF kernels can be expressed as $k(x, x') = \exp(-\gamma \|x - x'\|_2^2)$. Combining different characteristics of different kernel functions contribute to get the advantages of multiple kernels and achieve better mapping capability, which is known as multiple kernel learning.

The synthetic kernel in multiple kernel learning is a convex combination of different kernel functions. It can be expressed as the weighted sum of basic kernels as $K(x, x') = \sum_{m=1}^M \beta_m k_m(x, x')$, where β_m is the weight parameter of basic kernels. Within the framework of multiple kernel, the representation of samples in the feature space is translated to how to select and confirm basic kernels and their weight parameter. The goal of a regular multiple kernel learning is to identify the optimal combination of M basic kernels, denoted by $\theta = (\theta_1, \dots, \theta_M)$ by the following maximum margin learning principle, which can be cast into the following optimization problem:

$$\min_{\theta \in \Delta} \min_{f \in H_{K(\theta)}} \frac{1}{2} \|f\|_{H_{K(\theta)}}^2 + C \sum_{i=1}^N l(f(x_i), y_i), \tag{2}$$

where $\Delta = \{\theta \in \mathbb{R}_+^M | \theta^T e_M = 1\}$, $K(\theta)(\cdot, \cdot) = \sum_{j=1}^M \theta_j k_j(\cdot, \cdot)$, $l(f(x_i), y_i) = \max(0, 1 - y_i f(x_i))$, e_M represents a vector of M dimensions with all its elements being 1. Formulation 2 can also be turned into the following min-max optimization task:

$$\min_{\theta \in \Delta} \max_{\alpha \in \Xi} \left\{ \alpha^T e_N - \frac{1}{2} (\alpha \circ y)^T \left(\sum_{j=1}^N \theta_j K^j \right) (\alpha \circ y) \right\}, \tag{3}$$

where $K^j \in \mathbb{R}^{N \times N}$ with $K_{p,q}^j = k_j(x_p, x_q)$, $\Xi = \{\alpha | \alpha \in [0, C]^N\}$, and \circ defines the element-wise product between two vectors. We can see that formulating this problem as an optimization task will lead to large amount of complex calculation, and to avoid this drawback, boosting method is used to calculate the parameters of the synthetic kernel in the multiple kernel learning.

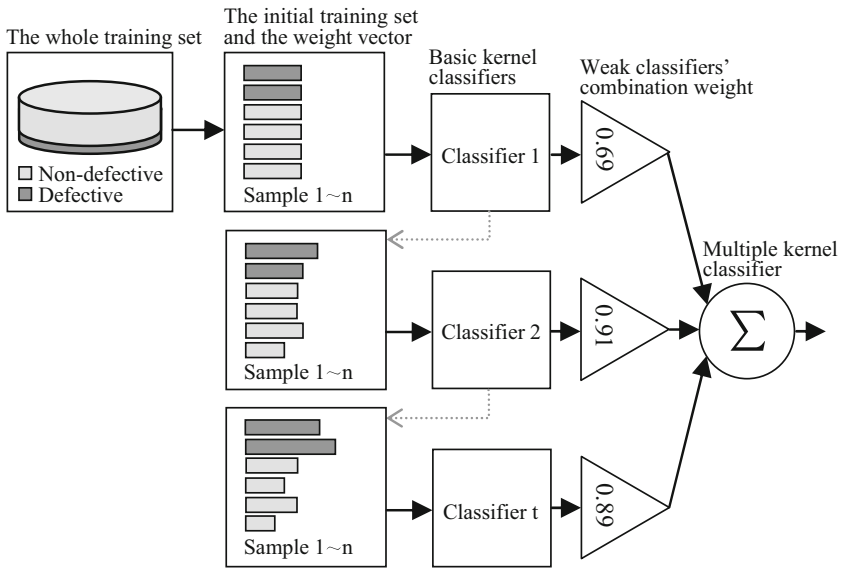


Fig. 2 Schematic representation of Adaboost

3.3 Multiple kernel ensemble learning

To learn a classifier with multiple kernels, we follow the typical procedure of a popular and successful boosting algorithm, i.e., Adaboost. Specifically we repeatedly learn some basic kernel classifiers through a series of boosting trials after the training set is initialized, and then integrated them according to their combination weights, thus we get the final MKEL classifier. The whole process can be expressed in Fig. 2.

Before boosting trials, the training set needs to be initialized firstly. We can perform a random sampling strategy directly on the entire training set, and then take these selected samples as the MKEL initial training set. After the training set initialization is completed, a distribution of weights D_t needs to be engaged to indicate the importance of the training samples for learning. Initially, these weights are all equal. At each boosting trial, we will adjust the weight vector D_t according to certain strategy, for the sake of focusing on those samples that need to be concerned at next boosting trial.

Once the initial training set and the weight vector D_t are obtained, boosting trials can be started. The key issue of the t th boosting trial is how to learn the kernel-based classifier $f_t(x)$ from these training data. In single kernel case, $f_t(x)$ can be learned by applying any regular kernel method, e.g., SVM used in our study, but in the case of multiple kernel, situation becomes more complex. In order to get $f_t(x)$, we firstly learn one single kernel classifier $f_t^j(x)$ with each single kernel k_t^j using a regular kernel method. Based on the set of M base classifiers, we can further measure the misclassification performance of each classifier $f_t^j(x)$ with kernel k_t^j over distribution D_t of the whole collection of training data:

$$\varepsilon_t = \varepsilon\left(f_t^j(x)\right) = \sum_{i=1}^N D_t(i)\left(f_t^j(x_i) \neq y_i\right). \tag{4}$$

As a result, the best classifier with the smallest misclassification rate can be figured out and taken as the weak classifier $f_t(x)$ for the t th boosting trial:

$$f_t(x) = \arg \min_{f_t^j, j \in \{1, \dots, M\}} \varepsilon(f_t^j(x)). \tag{5}$$

To get the final classification from all of these weak classifiers, each of the classifiers is assigned a weight α in AdaBoost. For the t th boosting trial, $f_t(x)$'s combination weight α_t can be determined with the error rate based on the following formula:

$$\alpha_t = \frac{1}{2} \ln \left(\frac{1 - \varepsilon_t}{\varepsilon_t} \right). \tag{6}$$

After got $f_t(x)$ and its combination weight α_t , the t th boosting trial is finished. Before next trial started, we must update the weight vector D_{t+1} in the $(t + 1)$ th boosting trial according to the result of the t th boosting trial. General boosting methods adjust the weight vector directly only according to the classification results, so that the samples that are correctly classified in the t th boosting trial will decrease in weight, and the misclassified samples will increase in weight. The goal of this effort is to focus on the misclassified samples in next boosting trial (Sun et al. 2007).

In software defect prediction we should pay more attention to defective samples, the reason is that misclassifying a defective software module as non-defective is much more dangerous than misclassifying a non-defective module as defective-prone. In order to put more emphasis on defective samples, class labels need to be taken into account when adjust the weight vector D_{t+1} . For defective samples, if they are incorrectly predicted in the t th boosting trial, their weight will be increased; if they are correctly predicted, their weight will keep unchanged; For non-defective samples, if they are incorrectly predicted in the t th boosting trial, their weight will keep unchanged, if they are correctly predicted, their weight will be decreased. The sample weight vector D_{t+1} is calculated by:

if the module is defective ($y_i = 1$):

$$D_{t+1}(i) = \frac{D_t(i)}{\text{Sum}(D)} \times \begin{cases} 1 & \text{if } f(x_i) = y_i \\ e^{\alpha_t} & \text{if } f(x_i) \neq y_i \end{cases}, \tag{7}$$

if the module is non-defective ($y_i = -1$):

$$D_{t+1}(i) = \frac{D_t(i)}{\text{Sum}(D)} \times \begin{cases} e^{-\alpha_t} & \text{if } f(x_i) = y_i \\ 1 & \text{if } f(x_i) \neq y_i \end{cases}. \tag{8}$$

After D_{t+1} is calculated, AdaBoost starts on the $(t + 1)$ th iteration. When all of the training and weight-adjusting iterations are completed, T basic kernel classifiers and their combination weights are got, the final MKEL classifier is an ensemble of them:

$$f(x) = \text{sign} \left(\sum_{t=1}^T \alpha_t f_t(x) \right), \tag{9}$$

and the details of the proposed MKEL algorithm are shown in Algorithm 1.

Algorithm 1. The MKEL algorithm

1. INPUT:
 - training data: $(x_1, y_1), \dots, (x_N, y_N)$
 - kernel functions: $k_j(\cdot, \cdot): x \times x \rightarrow \mathbb{R}, j = 1, \dots, M$
 - initial distribution $D_1(i) = 1/N, i = 1, \dots, N$
2. **for** $t = 1, \dots, T$ **do**
3. sample a set of n samples using distribution D_t
4. **for** $j = 1, \dots, M$ **do**
5. train weak classifier with kernel k_j :
 - $f_t^j: x \rightarrow \{-1, +1\}$
6. compute the training error over D_t :
 - $$\varepsilon_t = \sum_{i=1}^N D_t(f_t^j(x_i) \neq y_i)$$
7. **end for**
8. select the best classifier with the minimal error rate
 - $$f_t(x) = \arg \min_{f_t^j, j \in \{1, \dots, M\}} \varepsilon(f_t^j(x)) = \arg \min_{f_t^j, j \in \{1, \dots, M\}} \sum_{i=1}^N D_t(i)(f_t^j(x_i) \neq y_i)$$
9. choose $\alpha_t = \frac{1}{2} \ln\left(\frac{1 - \varepsilon_t}{\varepsilon_t}\right)$
10. update $D_{t+1}(i)$:
 - if** $y_i = 1$:
 - $$D_{t+1}(i) = \frac{D_t(i)}{\text{Sum}(D)} \times \begin{cases} 1 & \text{if } f_t(x_i) = y_i \\ e^{\alpha_t} & \text{if } f_t(x_i) \neq y_i \end{cases}$$
 - else if** $y_i = -1$:
 - $$D_{t+1}(i) = \frac{D_t(i)}{\text{Sum}(D)} \times \begin{cases} e^{-\alpha_t} & \text{if } f_t(x_i) = y_i \\ 1 & \text{if } f_t(x_i) \neq y_i \end{cases}$$
 - end if**
11. **end for**
12. OUTPUT: $f(x) = \text{sign}\left(\sum_{t=1}^T \alpha_t f_t(x)\right)$

4 Experimental setup

In this section, we describe the experimental setup in detail, including the benchmark datasets, evaluation measures, and experiment design.

Table 1 20 Metrics selected from NASA datasets

Metrics	Description
loc	McCabe's line count of code for each module
LOCcode	Halstead's line count of code for each module
LOCcomment	Halstead's count of lines of comments for each module
LOBLank	Halstead's count of blank lines for each module
LOCcodeAndComment	Halstead's count of code and comments for each module
v(g)	Cyclomatic complexity for each module
ev(g)	Essential complexity for each module
iv(g)	Design complexity for each module
Total_Op	Total number of operators for each module
Total_Opnd	Total number of operands for each module
Uniq_Op	Number of unique operators for each module
Uniq_Opnd	Number of unique operands for each module
n	Total operators and operands for each module
v	Volume for each module
l	Program length for each module
d	Difficulty for each module
i	Intelligent content for each module
b	Error estimate for each module
e	Programming effort for each module
t	Programming time for each module

4.1 Benchmark datasets

In this paper, we experiment with 12 datasets from NASA Metrics Data Program (MDP) to verify the applied effects of the proposed algorithm. NASA benchmark datasets are publicly available and have been widely used for software defect prediction. Each dataset represents a NASA software system or sub-system, which contains the corresponding defect-marking data and various static code metrics (Gray et al. 2011). Static code metrics are measured by static software metric methods including Halstead and McCabe measures, these metrics contain lines of code (LOC), operand and operator counts, readability, complexity and etc., we list the 20 common basic metrics and their descriptions in Table 1.

Table 2 gives the brief properties of 12 NASA datasets, including the total number of attributes, the number of defective and non-defective modules, and the ratio of them. It is obviously that every dataset has the characteristic of class-imbalance. It should be noted that the original data contains various duplicate entries, and to make our results more credible, these datasets have been cleaned to remove the duplicate parts (Gray et al. 2011).

Table 2 NASA benchmark datasets after data cleaning

Datasets	Number of attributes	Number of defective modules	Number of non-defective modules	Ratio between defective and non-defective modules
CM1	38	42	285	0.15:1
JM1	22	1672	6083	0.28:1
KC1	22	314	878	0.36:1
KC3	40	36	158	0.23:1
MC1	39	46	1930	0.02:1
MC2	40	44	81	0.54:1
MW1	38	27	228	0.12:1
PC1	38	61	650	0.09:1
PC2	37	16	729	0.02:1
PC3	38	134	939	0.14:1
PC4	38	177	1110	0.16:1
PC5	39	471	1220	0.39:1

Table 3 Defect prediction metric

	Predict as defective	Predict as non-defective
Defective modules	<i>A</i>	<i>B</i>
Non-defective modules	<i>C</i>	<i>D</i>

4.2 Evaluation measures

In software defect prediction, probability of detection (Pd), probability of false alarm (Pf), precision and accuracy are four important measures to evaluate the performance of prediction model. They can be defined by using *A*, *B*, *C*, and *D* in Table 3. Here, *A*, *B*, *C*, and *D* are the number of defective modules that are predicted as defective, the number of defective modules that are predicted as defective-free, the number of defective-free modules that are predicted as defective, and the number of defective-free modules that are predicted as defective-free, respectively.

Pd denotes the ratio is the number of defective modules correctly classified as defective to the number of defective modules, which is defined as $A/(A + B)$. Pf denotes the ratio is the number of defective-free modules wrongly classified as defective to the number of defective-free modules, which is defined as $C/(C + D)$. Precision denotes the ratio is the number of defective modules correctly classified as defect to the number of modules that are classified as defective, which is defined as $A/(A + C)$. Accuracy denotes the ratio is the number of modules that are correctly classified to the number of total modules, which is defined as $(A + D)/(A + B + C + D)$.

From the definitions of Pd and precision, it can be concluded that a higher Pd means the prediction model intends to find out defective modules as much as possible, and a

higher precision means the prediction model intends to predict defective modules as correct as possible. For software defect prediction, we hope that the prediction model not only can find out more defective modules, but also can make fewer mistakes, which means that a good prediction model desires to achieve high value of Pd and precision. However, there exists trade-off between these two measures, high Pd is on the expense of low precision, and vice versa. Therefore, a comprehensive measure to combine Pd with precision is necessary when we evaluate the performance of defect prediction model. This is the F-measure, the harmonic mean of Pd and precision, which can be defined as:

$$F - \text{measure} = 2 * Pd * \text{precision} / (Pd + \text{precision}).$$

The value of F-measure ranges from 0 to 1, the higher the value, the better the prediction model performance. Compared with other evaluate measures, F-measure can evaluate the prediction model comprehensively and effectively, meanwhile, F-measure is reasonably stable and not susceptible to the influence of parameter adjustment.

4.3 Experimental design

To verify the effect of MKEL approach, we conduct some experiments. The experimental setup is as follows: for each of the NASA MDP datasets, we randomly selected 50 % of the defective and non-defective samples as training set, and the remaining 50 % were used for test. Moreover, for the purpose of getting a more general result, we repeat each algorithm 20 times on every dataset and report average performances.

We create a set of 30 base kernels, i.e., RBF kernels with 21 different widths ($2^{-10}, 2^{-9}, \dots, 2^{10}$) on all features, Polynomial kernels of degree 1 to 9 on all features. We map every NASA MDP dataset to a higher-dimensional kernel space respectively by using these 30 base kernels. For SVM, the popular LIBSVM toolbox is adopted as the SVM solver.

The boosting training set are initialized as follows: for each NASA MDP training set, we randomly select 40 % of the training samples as suggested in [Xia and Hoi \(2013\)](#), then take them as the initial boosting training set. By default, we set the total number of boosting trials T to 100, so the final MKEL classifier will be an ensemble of 100 kernel-based weak classifiers.

5 Experimental results

In order to evaluate the performance of our MKEL approach, we compare it with two state-of-the-art class-imbalance learning methods for software defect prediction including coding based ensemble learning (CEL) ([Sun et al. 2012](#)) and dynamic version of AdaBoost.NC (DVA) ([Wang and Yao 2013](#)). And the comparison also embraces several representative software defect prediction methods, including weighted Naïve Bayes (NB) ([Wang and Li 2010](#)), Compressed C4.5 decision tree (CC4.5) ([Wang et al. 2012](#)), Cost-sensitive Boosting Neural Network (CBNN) ([Zheng 2010](#)) and Asymmetric Kernel Principal Component Classification (AKPCC) ([Ma et al. 2012](#)).

For running these previous algorithms mentioned above, we adopt their default settings and chose the suggested parameters by the literatures (Random forest is used as the basic classifier of CEL since it has better performance in the literature. DVA employs 10-fold cross-validation, and 9/10 partitions are used at each time of building models, among them, 8/9 data serve as a training set and 1/9 data serve as a validation set. CBNN employs a fivefold cross-validation and cost ratio varies from 1 to 20. RBF kernel is used for AKPCC). For all the experiments, we repeat each algorithm 20 times on every NASA MDP dataset. In this section, we present the experimental results of our MKEL approach and other compared methods.

Table 4 shows the Pd and Pf values of our approach and other compared methods on 12 NASA datasets. For each dataset, Pd and Pf values of all methods are the mean values calculated from the results of 20 runs. The average Pd and Pf values across all datasets of MKEL are 0.68 and 0.26. Due to Table 4, we can observe that the Pd values of MKEL are higher than the corresponding values of all other methods. The results indicate that the proposed MKEL approach takes the misclassification costs into consideration, which makes the prediction tend to classify the non-defective modules as the defective ones in order to obtain higher Pd values. The results of Pf values suggest that in spite of not acquiring the best Pf values on most datasets, MKEL can achieve comparatively better results in contrast with other methods.

Table 5 shows the mean and standard deviation of F-measure values of our approach and the compared methods across 20 random running on 12 NASA datasets. The average F-measure value across all datasets of MKEL is 0.48, and the standard deviation of F-measure values across all datasets of MKEL is 0.03. It can be seen that our approach achieves higher F-measure values than those of other compared methods on all datasets, which indicates that the proposed approach achieves preferable prediction effects.

To statistically analyze the F-measure results used in Table 5, we conduct a statistical test, i.e., McNemar's test (Yambor et al. 2000). This test can provide statistical significance between MKEL and other methods. Here, the McNemar's test uses a significance level of 0.05, that is, if the p value is below 0.05, the performance difference between two compared methods is considered to be statistically significant. Table 6 shows the p values between MKEL and other compared methods on 12 NASA datasets, where only one value is slightly above 0.05. According to Table 6, the proposed approach indeed makes a statistically significant difference in comparison with other methods for software defect prediction.

In above experiments, we create a set of 30 base kernels as multiple kernel learning, select 40% of the training samples randomly as the initial boosting training set, and set the total number of boosting trials T to 100. When update the sample weight vector, we use a MKEL boost sample weight vector updating strategy by gradually increasing the weights of the defective samples and decreasing the weights of the non-defective samples. We use SVM as the basic kernel classifier. The average F-measure value across all datasets of MKEL is 0.48 under these settings. In order to validate the influence of these factors, we repeat the experiment by using different number of base kernels, weight updating strategies, sampling ratios, boosting trails, and basic kernel classifiers. The average F-measure values are shown in Table 7, where M is the number of base kernels ($M = 1$ means single kernel), and μ is the initial sampling

Table 4 Experimental results: Pd and Pf comparisons on NASA's 12 datasets

Datasets	NB	CC4.5	CBBN	CEL	DVA	AKPCC	MEKL
CM1							
Pd	0.54	0.23	0.59	0.36	0.71	0.29	0.75
Pf	0.21	0.11	0.26	0.14	0.29	0.16	0.33
JM1							
Pd	0.14	0.37	0.54	0.32	0.51	0.19	0.56
Pf	0.32	0.17	0.29	0.14	0.33	0.06	0.31
KC1							
Pd	0.31	0.40	0.69	0.37	0.48	0.32	0.78
Pf	0.06	0.12	0.30	0.13	0.34	0.14	0.39
KC3							
Pd	0.46	0.41	0.51	0.29	0.65	0.29	0.65
Pf	0.21	0.16	0.25	0.12	0.43	0.13	0.33
MC1							
Pd	0.36	0.27	0.48	0.39	0.27	0.55	0.55
Pf	0.11	0.12	0.21	0.18	0.14	0.35	0.13
MC2							
Pd	0.35	0.64	0.79	0.56	0.71	0.44	0.79
Pf	0.09	0.49	0.54	0.38	0.45	0.23	0.28
MW1							
Pd	0.49	0.29	0.61	0.25	0.52	0.33	0.64
Pf	0.19	0.09	0.25	0.11	0.13	0.14	0.21
PC1							
Pd	0.36	0.38	0.54	0.46	0.65	0.38	0.69
Pf	0.11	0.09	0.17	0.13	0.45	0.09	0.31
PC2							
Pd	0.27	0.33	0.61	0.35	0.31	0.50	0.61
Pf	0.14	0.15	0.40	0.30	0.17	0.19	0.16
PC3							
Pd	0.28	0.34	0.65	0.41	0.41	0.34	0.56
Pf	0.09	0.08	0.25	0.13	0.16	0.13	0.29
PC4							
Pd	0.39	0.49	0.66	0.48	0.76	0.52	0.80
Pf	0.13	0.07	0.18	0.06	0.04	0.11	0.27
PC5							
Pd	0.32	0.50	0.79	0.37	0.69	0.50	0.81
Pf	0.14	0.02	0.08	0.13	0.16	0.24	0.08

ratio. General boost weight updating strategy adjusts the weight vector directly only according to the classification results (*if* $f(x_i) = y_i$, $D_{t+1}(i) = D_t(i)e^{-\alpha_t}/\text{Sum}(D)$, *if* $f(x_i) \neq y_i$, $D_{t+1}(i) = D_t(i)e^{\alpha_t}/\text{Sum}(D)$). The results in Table 7 indicate that

Table 5 F-measure values on 12 NASA datasets from different prediction methods

Datasets	Algorithms						
	NB	CC4.5	CBBN	CEL	DVA	AKPCC	MEKL
CM1	0.32 (0.05)	0.25 (0.09)	0.32 (0.05)	0.27 (0.10)	0.35 (0.08)	0.21 (0.06)	0.40 (0.04)
JM1	0.33 (0.01)	0.34 (0.01)	0.38 (0.04)	0.33 (0.02)	0.38 (0.06)	0.31 (0.03)	0.45 (0.02)
KC1	0.38 (0.02)	0.39 (0.03)	0.41 (0.04)	0.36 (0.04)	0.39 (0.01)	0.35 (0.04)	0.50 (0.04)
KC3	0.38 (0.06)	0.38 (0.09)	0.38 (0.06)	0.33 (0.09)	0.35 (0.10)	0.29 (0.10)	0.44 (0.05)
MC1	0.12 (0.04)	0.20 (0.10)	0.15 (0.05)	0.16 (0.03)	0.13 (0.06)	0.25 (0.07)	0.41 (0.02)
MC2	0.45 (0.09)	0.48 (0.09)	0.56 (0.05)	0.49 (0.07)	0.48 (0.08)	0.49 (0.10)	0.63 (0.03)
MW1	0.31 (0.05)	0.27 (0.11)	0.33 (0.06)	0.27 (0.05)	0.35 (0.09)	0.33 (0.11)	0.49 (0.03)
PC1	0.28 (0.05)	0.32 (0.06)	0.32 (0.04)	0.32 (0.07)	0.39 (0.03)	0.35 (0.05)	0.50 (0.01)
PC2	0.11 (0.04)	0.17 (0.10)	0.08 (0.11)	0.11 (0.03)	0.15 (0.09)	0.23 (0.06)	0.30 (0.01)
PC3	0.29 (0.05)	0.30 (0.06)	0.38 (0.03)	0.36 (0.04)	0.38 (0.06)	0.39 (0.03)	0.46 (0.05)
PC4	0.36 (0.03)	0.49 (0.06)	0.46 (0.12)	0.48 (0.04)	0.50 (0.03)	0.43 (0.10)	0.55 (0.03)
PC5	0.33 (0.06)	0.48 (0.03)	0.36 (0.04)	0.36 (0.04)	0.44 (0.05)	0.35 (0.04)	0.68 (0.02)
Average	0.31 (0.05)	0.34 (0.07)	0.34 (0.06)	0.32 (0.05)	0.36 (0.06)	0.33 (0.07)	0.48 (0.03)

multiple kernels achieve better performance than single kernel. Multiple kernels and the weight vector updating strategy improve MKEL performance mostly, by contrast, initial sampling ratio, and basic classifiers the number of boosting trials influence performance relatively inconspicuously. We conduct a statistical test to analyze the F-measure results used in Table 7, the p values are shown in Table 8.

6 Threat to validity

The study has limitations that are common with most of the empirical studies in the literature. The proposed MKEL approach uses a randomly selecting strategy to initialize the training set, which means that the initial training set is built by randomly

Table 6 P values between MKEL and other compared methods on 12 NASA datasets

Datasets	MKEL					
	NB	CC4.5	CBNN	CEL	DVA	AKPCC
CM1	3.23×10^{-5}	3.51×10^{-7}	1.61×10^{-4}	1.80×10^{-5}	1.09×10^{-8}	1.16×10^{-8}
JM1	1.29×10^{-11}	2.30×10^{-14}	0.0064	3.54×10^{-13}	1.37×10^{-12}	1.13×10^{-7}
KC1	6.38×10^{-13}	1.53×10^{-8}	9.53×10^{-7}	1.70×10^{-12}	5.34×10^{-8}	5.67×10^{-8}
KC3	3.22×10^{-6}	0.0069	1.28×10^{-4}	3.61×10^{-6}	7.38×10^{-9}	1.16×10^{-8}
MC1	2.16×10^{-9}	3.21×10^{-11}	1.63×10^{-7}	0.44×10^{-5}	2.11×10^{-12}	4.19×10^{-6}
MC2	1.15×10^{-8}	2.81×10^{-5}	1.31×10^{-5}	7.50×10^{-6}	1.29×10^{-14}	3.04×10^{-8}
MW1	1.11×10^{-5}	2.36×10^{-4}	2.60×10^{-3}	1.65×10^{-5}	3.50×10^{-13}	0.0517
PC1	1.63×10^{-8}	3.41×10^{-6}	2.53×10^{-7}	2.60×10^{-5}	3.11×10^{-22}	1.11×10^{-23}
PC2	0.41×10^{-11}	1.67×10^{-8}	0.0013	1.92×10^{-10}	6.41×10^{-9}	1.99×10^{-8}
PC3	1.01×10^{-8}	6.71×10^{-9}	4.31×10^{-5}	4.03×10^{-5}	1.38×10^{-9}	3.36×10^{-15}
PC4	2.76×10^{-16}	7.21×10^{-6}	2.75×10^{-10}	4.29×10^{-8}	3.86×10^{-17}	3.78×10^{-7}
PC5	1.80×10^{-14}	7.01×10^{-9}	3.13×10^{-11}	1.38×10^{-10}	4.56×10^{-23}	3.23×10^{-12}

Table 7 Average F-measure value of 12 NASA datasets by using different parameters

	CM1	JM1	KC1	KC3	MC1	MC2	MW1	PC1	PC2	PC3	PC4	PC5	Average
Base kernels													
<i>M</i> = 1	0.40	0.37	0.35	0.39	0.36	0.55	0.44	0.40	0.33	0.39	0.47	0.54	0.42
<i>M</i> = 20	0.38	0.48	0.39	0.39	0.37	0.60	0.49	0.45	0.28	0.41	0.51	0.60	0.45
<i>M</i> = 50	0.41	0.39	0.55	0.38	0.43	0.66	0.51	0.50	0.33	0.55	0.56	0.66	0.49
Weight updating													
General boost	0.38	0.39	0.38	0.36	0.50	0.58	0.47	0.51	0.33	0.38	0.51	0.55	0.45
MKEL boost	0.40	0.45	0.50	0.44	0.41	0.63	0.49	0.50	0.30	0.46	0.55	0.68	0.48
Sampling ratio													
μ = 20 %	0.38	0.44	0.47	0.46	0.40	0.57	0.55	0.39	0.32	0.53	0.51	0.59	0.47
μ = 60 %	0.35	0.49	0.50	0.46	0.38	0.66	0.50	0.43	0.22	0.57	0.49	0.66	0.48
Boosting trials													
<i>T</i> = 50	0.39	0.41	0.46	0.47	0.39	0.67	0.56	0.51	0.40	0.19	0.48	0.69	0.47
<i>T</i> = 150	0.40	0.44	0.50	0.43	0.50	0.55	0.49	0.50	0.32	0.44	0.51	0.63	0.47
Basic classifier													
C4.5	0.39	0.46	0.52	0.43	0.46	0.61	0.48	0.49	0.33	0.40	0.50	0.61	0.47
Naive Bayes	0.36	0.40	0.37	0.45	0.45	0.58	0.53	0.52	0.35	0.33	0.47	0.65	0.46
Random forest	0.40	0.37	0.49	0.46	0.44	0.57	0.50	0.51	0.36	0.46	0.53	0.60	0.47

Table 8 P values between MKEL and other different settings on 12 NASA datasets

Datasets	MKEL ($M = 30, \mu = 40\%, T = 100, MKEL$ weight updating, SVM basic classifier)														
	Base kernels			General weight updating			Sampling ratio			Boosting trials			Basic classifier		
	$M = 1$	$M = 20$	$M = 50$	$\mu = 20\%$	$\mu = 60\%$	$T = 50$	$T = 150$	C4.5	Naïve Bayes	Random Forest					
CMI	1.31×10^{-10}	3.38×10^{-9}	1.29×10^{-15}	2.39×10^{-9}	5.23×10^{-11}	8.08×10^{-19}	4.30×10^{-10}	7.16×10^{-19}	3.27×10^{-9}	8.21×10^{-11}					
JM1	2.23×10^{-7}	4.35×10^{-7}	2.38×10^{-5}	1.76×10^{-9}	3.07×10^{-11}	0.39×10^{-7}	2.68×10^{-7}	3.84×10^{-14}	6.59×10^{-13}	4.98×10^{-11}					
KC1	5.21×10^{-8}	3.39×10^{-11}	0.0001	2.38×10^{-3}	2.52×10^{-6}	3.15×10^{-16}	5.35×10^{-12}	0.0004	1.38×10^{-5}	2.26×10^{-4}					
KC3	4.23×10^{-8}	2.59×10^{-7}	1.53×10^{-11}	8.31×10^{-7}	4.26×10^{-7}	4.56×10^{-11}	3.86×10^{-9}	5.51×10^{-20}	0.71×10^{-4}	2.37×10^{-11}					
MC1	3.22×10^{-7}	3.55×10^{-11}	4.30×10^{-6}	6.25×10^{-8}	3.58×10^{-11}	2.31×10^{-7}	1.86×10^{-16}	3.87×10^{-14}	2.36×10^{-16}	0.76×10^{-9}					
MC2	2.33×10^{-9}	1.21×10^{-10}	2.53×10^{-9}	0.0011	2.69×10^{-7}	9.01×10^{-13}	3.95×10^{-17}	0.59×10^{-6}	1.47×10^{-9}	5.43×10^{-16}					
MW1	1.69×10^{-11}	1.66×10^{-7}	1.38×10^{-9}	3.51×10^{-8}	5.38×10^{-15}	2.37×10^{-6}	7.03×10^{-9}	2.43×10^{-15}	6.86×10^{-9}	1.39×10^{-15}					
PC1	2.13×10^{-8}	7.29×10^{-8}	1.97×10^{-6}	2.63×10^{-11}	1.28×10^{-8}	0.0032	1.02×10^{-11}	0.91×10^{-16}	6.51×10^{-15}	5.23×10^{-5}					
PC2	0.0019	3.68×10^{-9}	8.51×10^{-4}	1.37×10^{-9}	2.17×10^{-15}	0.38×10^{-21}	0.98×10^{-10}	1.29×10^{-16}	1.93×10^{-9}	4.68×10^{-9}					
PC3	1.26×10^{-10}	2.95×10^{-11}	7.41×10^{-8}	0.91×10^{-6}	5.88×10^{-21}	5.87×10^{-11}	1.33×10^{-15}	3.71×10^{-19}	2.33×10^{-13}	0.0013					
PC4	2.26×10^{-13}	8.73×10^{-6}	3.21×10^{-6}	2.20×10^{-9}	8.49×10^{-16}	4.81×10^{-13}	0.0103	5.69×10^{-8}	5.28×10^{-16}	5.52×10^{-13}					
PC5	6.38×10^{-7}	5.87×10^{-5}	6.53×10^{-8}	1.38×10^{-16}	2.93×10^{-11}	4.37×10^{-8}	1.79×10^{-8}	2.65×10^{-11}	1.51×10^{-12}	3.23×10^{-6}					

selecting samples from the original data. It is particularly conspicuous for the problem of class-imbalance in the MC1, PC1 and PC2 dataset even after data cleaning. For these datasets, it will be more likely that none of defective modules are selected in the initial training set. When the boosting training set is initialized, step must be taken to ensure that both of defective and non-defective modules are included in the initial training set.

The analysis and conclusion presented in this paper are based upon the metrics and defect data obtained from NASA projects. In spite of the generalization of our empirical results, the same analysis for another software system may provide different results, especially from a different application domain. However, the proposed MKEL approach can be extended to any software system for high quality and less testing effort. A software quality practitioner can utilize the process of developing a useful defect predictor in the presence of the problem of class-imbalance or when there are a large number of software metrics to work with.

7 Conclusion

Although multiple kernel learning has been shown effective in other domains, to the best of our knowledge, we are the first attempt towards improving software prediction performance by introducing multiple kernel learning technique. Aiming at the characteristics of defect data, we specifically design a multiple kernel ensemble learning (MKEL) classifier to predict defective modules. By using multiple kernel trick, it can fully exploit the information of historical data to improve the predict power. During the training course, we devise a new initialization strategy to make a balance between defective and non-defective software modules, and we also use a new weight update strategy which makes the defective software modules always to be focused on, so MKEL provides an effective solution for software defect prediction.

As compared with several state-of-the-art representative software defect prediction methods, the experiments on 12 NASA datasets show that the proposed MKEL approach performs better under the same experimental environment, it significantly improves the average F-measure values on all datasets. In addition, the initialization and the weight update strategy we used during the training stage solve the class-imbalance problem and decrease the cost of misclassification risk effectively, and they also make the average F-measure values improved on all datasets compared with the general strategies. All of these confirm that our MKEL approach can fully exploit the characteristics of historical data and improve the predict performance, so it is an effective solution for software defect prediction task.

References

- Aljamaan, H.I., Elish, M.O.: An empirical study of bagging and boosting ensembles for identifying faulty classes in object-oriented software. In: Proceedings of the IEEE Symposium on Computational Intelligence and Data Mining, Nashville, TN, USA, pp. 187–194 (2009)
- Amasaki, S., Takagi, Y., Mizuno, O., Kikuno, T.: A Bayesian belief network for assessing the likelihood of fault content. In: International Symposium on Software Reliability Engineering, pp. 215–226 (2003)

- Bennett, K.P., Momma, M., Embrechts, M.J.: MARK: a boosting algorithm for heterogeneous kernel models. In: Proceedings of 8th ACM-SIGKDD International Conference on Knowledge Discovery and Data Mining, Edmonton, Canada: ACM, pp. 24–31 (2002)
- Bezerra, E., Miguel, Oliveiray, A.L.I., Adeodatoz, P.J.L.: Predicting software defects: a cost-sensitive approach. *International Conference Systems, Man, and Cybernetics*, pp. 2515–2522 (2011)
- Bi, J., Zhang, T., Bennett, K.P.: Column-generation boosting methods for mixture of kernels. In: Proceedings of the 10th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Seattle, USA: ACM, pp. 521–526 (2004)
- Breiman, L.: Random forests. *Mach. Learn.* **45**(1), 5–32 (2001)
- Catal, C., Diri, B.: A systematic review of software fault prediction studies. *Expert Syst. Appl.* **36**, 7346–7354 (2009)
- Damoulas, T., Girolami, M.A.: Probabilistic multi-class multi-kernel learning: on protein fold recognition and remote homology detection. *Bioinformatics* **24**(10), 1264–1270 (2008)
- Dietterich, T.G.: Ensemble methods in machine learning. *Mult. Classier Syst.* **1857**, 1–15 (2000)
- Elish, K., Elish, M.: Predicting defect-prone software modules using support vector machines. *J. Syst. Softw.* **81**(5), 649–660 (2008)
- Freund, Y., Schapire, R.E.: A decision-theoretic generalization of on-line learning and an application to boosting. *J. Comput. Syst. Sci.* **55**(1), 119–139 (1997)
- Gao, K., Khoshgoftaar, T.M.: Software defect prediction for high-dimensional and class-imbalanced data. *SEKE*, pp. 89–94 (2011)
- Gao, K., Khoshgoftaar, T.M., Napolitano, A.: A hybrid approach to coping with high dimensionality and class imbalance for software defect prediction. *Mach. Learn. Appl.* **2**, 281–288 (2012)
- GÅonen, M., Alpaydin, E.: Localized multiple kernel learning. In: Proceedings of the 25th International Conference on Machine Learning, Helsinki, Finland: ACM, pp. 352–359 (2008)
- Gayatri, N., Nickolas, S., Reddy, A.V.: Feature selection using decision tree induction in class level metrics dataset for software defect predictions. In: The World Congress on Engineering and Computer Science, pp. 124–129 (2010)
- Gehler, P.V., Nowozin, S.: On feature combination for multiclass object classification. *IEEE Int. Conf. Comput. Vis.* **2**, 221–228 (2009)
- Gray, D., Bowes, D., Davey, N., Sun, Y., Christianson, B.: The Misuse of the NASA metrics data program data sets for automated software defect prediction. in EASE 2011. Durham (2011)
- Gray, D., Bowes, D., Davey, N., Sun, Y., Christianson, B.: Using the support vector machine as a classification method for software defect prediction with static code metrics. *Eng. Appl. Neural Netw.* **43**, 223–234 (2009)
- Hall, T., Beecham, S., Bowes, D., Gray, D., Counsell, S.: A systematic literature review on fault prediction performance in software engineering. *Softw. Eng.* **38**(6), 1276–1304 (2011)
- Halstead, M.H.: *Elements of Software Science (Operating and Programming Systems Series)*. Elsevier North-Holland, New York (1977)
- He, H., Garcia, E.A.: Learning from imbalanced data. *IEEE Trans. Knowl. Data Eng.* **21**(9), 1263–1284 (2009)
- Jing, X.Y., Ying, S., Zhang, Z.W., Wu, S.S., Liu, J.: Dictionary learning based software defect prediction. In: Proceedings of the 36th International Conference on Software Engineering, Hyderabad, India: ACM, pp. 414–423 (2014)
- Kembhavi, A., Siddiquie, B., Miezianko, R.: Incremental multiple Kernel learning for object recognition. *Int. Conf. Comput. Vis.* **2**, 638–645 (2009)
- Khoshgoftaar, M.T., Gao, K., Seliya, N.: Attribute selection and imbalanced data: problems in software defect prediction. In: *International Conference on Tools with Artificial Intelligence*, pp. 137–144 (2010)
- Khoshgoftaar, T.M., Seliya, N.: Software quality classification modeling using the SPRINT decision tree algorithm. In: Proceedings of the 14th IEEE International Conference on Tools with Artificial Intelligence, Washington, DC, USA, pp. 365–374 (2002)
- Khoshgoftaar, T.M., Seliya, N.: Tree-based software quality estimation models for fault prediction. *IEEE Symposium on Software Metrics*, pp. 203–214 (2002)
- Lewis, D.P., Jebara, T., Noble, W. S.: Nonstationary kernel combination. In: Proceedings of the 23rd International Conference on Machine Learning, Pittsburgh, USA: ACM, pp. 553–560 (2006)
- Luo, G.C., Ma, Y., Qin, K.: Asymmetric learning based on Kernel partial least squares for software defect prediction. *IEICE Trans.* **95-D**(7), 2006–2008 (2012)

- Lyu, M.R.: Software reliability engineering: a roadmap. In: Proceedings of the 2007 Future of Software Engineering (FOSE'07). Washington, DC, USA: IEEE Computer Society, pp. 153–170 (2007)
- Ma, Y., Luo, G.C., Chen, H.: Kernel based asymmetric learning for software defect prediction. *IEICE Trans. 95-D*(1), 215–226 (2012)
- McCabe, T.J.: A complexity measure. *IEEE Trans. Softw. Eng.* **4**, 308–320 (1976)
- Menzies, T., Greenwald, J., Frank, A.: Datamining static code attributes to learn defect predictors. *IEEE Trans. Softw. Eng.* **33**(1), 2–13 (2007)
- Menzies, T., Greenwald, J., Frank, A.: Data mining static code attributes to learn defect predictors. *IEEE Trans. Softw. Eng.* **33**(1), 2–13 (2007)
- Muller, K.R., Mika, S., Ratsch, G., Tsuda, K., Scholkopf, B.: An introduction to kernel based learning algorithms. *IEEE Trans. Neural Netw.* **12**(2), 181–201 (2001)
- Nam, J., Pany, S.J., Kim, S.: Transfer defect learning. In: International Conference on Software Engineering, pp. 382–391 (2013)
- Ong, C.S., Smola, A.J., Williamson, R.C.: Learning the kernel with hyperkernels. *J. Mach. Learn. Res.* **6**(7), 1043–1071 (2005)
- Paikari, E., Richter, M.M., Ruhe, G.: Defect prediction using case-based reasoning: an attribute weighting technique based upon sensitivity analysis in neural networks. *Int. J. Softw. Eng. Knowl. Eng.* **22**(5), 747–768 (2012)
- Rakotomamonjy, A., Bach, F., Canu, S.: More efficiency in multiple kernel learning. *Int. Conf. Mach. Learn.* **20**(24), 775–782 (2007)
- Ren, J., Qin, K., Ma, Y., Luo, G.: On software defect prediction using machine learning. *J. Appl. Math.* **2014**(785435), 8 (2014)
- Rokach, L.: Ensemble-based classifiers. *Artif. Intell. Rev.* **33**, 1–39 (2010)
- Schoelkopf, B., Smola, A., Muller, K.R.: Nonlinear component analysis as a kernel eigenvalue problem. *Neural Comput.* **10**(5), 1299–1319 (1998)
- Scholkopf, B., Mika, S., Burges, C.J.C., Knirsch, P., Muller, K.R., Ratsch, G.: Input space versus feature space in kernel-based methods. *IEEE Trans. Neural Netw.* **10**(5), 1000–1017 (1999)
- Seiffert, C., Khoshgoftaar, T.M., Van Hulse, J.: Improving software-quality predictions with data sampling and boosting. *IEEE Trans. Syst. Man Cybern. Part A Syst. Hum.* **39**(6), 1283–1294 (2009)
- Seliya, N., Khoshgoftaar, T.M., Hulse, J.V.: Predicting faults in high assurance software. In: IEEE International High Assurance Systems Engineering Symposium, pp. 26–34 (2010)
- Seliya, N., Khoshgoftaar, T.M.: The use of decision trees for cost-sensitive classification an empirical study in software quality prediction. *Wiley Interdiscip. Rev. Data Min. Knowl. Discov.* **1**(5), 448–459 (2011)
- Shepperd, M., Song, Q.B., Sun, Z.B., Mair, C.: Data quality: some comments on the NASA software defect data sets. *IEEE Trans. Softw. Eng.* **39**(9), 1208–1215 (2013)
- Sun, Y., Kamel, Mohamed S., Wong, Andrew K.C., Wang, Y.: Cost-sensitive boosting for classification of imbalanced data. *Pattern Recognit.* **40**(12), 3358–3378 (2007)
- Sun, Z.B., Song, Q.B., Zhu, X.Y.: Using coding based ensemble learning to improve software defect prediction. *IEEE Trans. Syst. Man Cybern. Part C* **42**(6), 1806–1817 (2012)
- Thwin, M.M.T., Quah, T.S.: Application of neural networks for software quality prediction using object-oriented metrics. *J. Syst. Softw.* **76**(2), 147–156 (2005)
- Turhan, B., Bener, A.: Software Defect Prediction: Heuristics for Weighted Naïve Bayes. In: International Conference on Software and Data Technologies, pp. 244–249 (2007)
- Turhan, B., Bener, A.: Analysis of naïve bayes' assumptions on software fault data: an empirical study. *Data Knowl. Eng.* **68**(2), 278–290 (2009)
- Valentini, G., Masulli, F.: Ensembles of learning machines. *Neural Netw.* 3–20 (2002)
- Wang, T., Li, W.H.: Naïve Bayes software defect prediction model. International Conference on Computational Intelligence and Software Engineering, pp. 1–4 (2010)
- Wang, J., Shen, B.J., Chen, Y.T.: Compressed C4.5 models for software defect prediction. International Conference on Quality Software, pp. 13–16 (2012)
- Wang, S., Yao, X.: Using class imbalance learning for software defect prediction. *IEEE Trans. Reliab.* **62**(2), 434–443 (2013)
- Xia, Hao, Hoi, Steven C.H.: MKBoost: a framework of multiple kernel boosting. *IEEE Trans. Knowl. Data Eng.* **25**(7), 1574–1586 (2013)
- Xing, F., Guo, P., Lyu, M.R.: A novel method for early software quality prediction based on support vector machine. In: Proceedings of the 16th IEEE International Symposium on Software Reliability Engineering, Chicago, Illinois, USA, pp. 213–222 (2005)

- Yambor, W.S., Draper, B.A., Beveridge, J.R.: Analyzing PCA-based face recognition algorithms: eigenvector selection and distance measures. In: Proceeding of the 2nd Workshop on Empirical Evaluation in Computer Vision, Dublin, Ireland, pp.1–15 (2000)
- Yan, Z., Chen, X.Y., Guo, P.: Software defect prediction using fuzzy support vector regression. *Adv. Neural Netw.* **6064**, 17–24 (2010)
- Zheng, J.: Cost-sensitive boosting neural networks for software defect prediction. *Expert Syst. Appl.* **37**(6), 4537–4543 (2010)
- Zhou, Z.H., Liu, X.Y.: Training cost-sensitive neural networks with methods addressing the class imbalance problem. *IEEE Trans. Knowl. Data Eng.* **18**(1), 63–77 (2006)
- Zien, A., Ong, C.S.: Multiclass multiple kernel learning. In: Proceedings of the 24th International Conference on Machine Learning. New York, USA: ACM, pp. 1191–1198 (2007)