A novel kernel to predict software defectiveness

Ahmet Okutan a,*, Olcay Taner Yildiz b

a Mobipath Erenet Ltd, Maltepe, Istanbul, Turkey
b Department of Computer Engineering, Isik University, Istanbul, Turkey

A R T I C L E   I N F O

Article history:
Received 18 June 2015
Revised 3 March 2016
Accepted 3 June 2016
Available online 11 June 2016

Keywords:
Defect prediction
SVM
Kernel methods

A B S T R A C T

Although the software defect prediction problem has been researched for a long time, the results achieved are not so bright. In this paper, we propose to use novel kernels for defect prediction that are based on the plagiarized source code, software clones and textual similarity. We generate precomputed kernel matrices and compare their performance on different data sets to model the relationship between source code similarity and defectiveness. Each value in a kernel matrix shows how much parallelism exists between the corresponding files of a software system chosen. Our experiments on 10 real world datasets indicate that support vector machines (SVM) with a precomputed kernel matrix performs better than the SVM with the usual linear kernel in terms of F-measure. Similarly, when used with a precomputed kernel, the k-nearest neighbor classifier (KNN) achieves comparable performance with respect to KNN classifier. The results from this preliminary study indicate that source code similarity can be used to predict defect proneness.

© 2016 Elsevier Inc. All rights reserved.

1. Introduction

Many software engineering projects run out of budget and schedule. This is one of the biggest problems that the software development industry has met so far and many attempts have been made to increase the success rate of the software projects. One possible solution is defect prediction, that is, to predict a software defect or failure before it is observed and take necessary mitigating actions.

Not all defects have the same priority considering their effects in the maintenance phase. For example, a severe bug in an accounting software can be very critical and may have a high priority, whereas a problem in the tool tip text of some screen control may not be so important. A software module with one non-critical bug is much more preferable to a module with many critical bugs. The number of bugs together with their severity are two important factors to decide on the extent of defect proneness.

Deciding on the defectiveness of a software is very critical and important to plan the testing and maintenance phases of a software project. First, in testing period, it is possible to focus more on the defect prone modules or modules where there are more errors comparatively. Second, since more defects are fixed during the test period, the maintenance cost of the project decreases and this causes a decrease in the total cost of the project also.

There are numerous studies in the literature about defect prediction using classification and regression techniques. In the classification case, the software modules (classes or files) are marked as defective or not, that is, a binary classification problem is solved. The focus is on defect proneness rather than its extent. However in the regression case, the number of faults in each module is estimated and the emphasis is on the number of faults rather than defect proneness. There are studies where either a specific statistical method is researched or the performance of several statistical methods are compared. For example, (Pickard et al., 1999) compare the efficiency of residual analysis, multivariate regression and classification and regression trees (CART) on the software datasets which were created by simulation. Giancarlo Succi and Stefanovic (2001) compare Poisson regression model with binomial regression model to deal with software defect data that is not distributed normally. They observe that the zero-inflated negative binomial regression model shows the best ability to describe the high variability in the dependent variable. Schneidewind (2001) shows that logistic regression method is not very successful alone. But when used together with Boolean discriminant functions (BDF) it gives more accurate results.

In recent years, the amount of research done on defect prediction using machine learning algorithms has increased slightly compared to traditional methods (Catal and Diri, 2009). Many algorithms have been studied and as a consequence, some of these algorithms have been marked to be better than others on the selected data sets. We believe that, most of the time it is difficult to generalize the defect prediction results. According to Myyrtveit

* Corresponding author.

E-mail addresses: ahmet.okutan@gmail.com (A. Okutan), olcaytaner@isikun.edu.tr (O. Taner Yildiz).

http://dx.doi.org/10.1016/j.jss.2016.06.006
0164-1212/© 2016 Elsevier Inc. All rights reserved.
et al. more reliable research procedures are needed to have confidence in the comparative studies of software prediction models (Myrteit et al., 2005). Furthermore, D’Ambros et al. state that defect prediction is a field where external validity is very hard to achieve (D’Ambros et al., 2012). Menzies et al. show that an approach useful in global context is often irrelevant in local contexts in defect prediction studies (Menzies et al., 2011). Rule induction (Shepperd and Kadoda, 2001), regression (Shepperd and Kadoda, 2001; Ekanayake et al., 2009), case-based reasoning (CBR) (Khoshgoftaar et al., 1997; 2000; Shepperd and Kadoda, 2001), decision tree approaches like C4.5 (Song et al., 2006), random forest (Kim et al., 2015; Guo et al., 2004; Kaur and Malhotra, 2008), linear discriminant analysis (Munson and Khoshgoftaar, 1992), artificial neural networks (Khoshgoftaar et al., 1995; Thwin and Quah, 2002; Kaur et al., 2009; Shepperd and Kadoda, 2001), k-nearest neighbor (Boetticher, 2005), K-star (Koru and Liu, 2005), Bayesian networks (Fenton et al., 2002; Pai and Dugan, 2007; Zhang, 2000; Okutan and Yildiz, 2012) and support vector machine based classifiers (Lessmann et al., 2008; Hu et al., 2009; Jin and Liu, 2010; ShivaJee et al., 2009; Xing et al., 2005; Gondra, 2008) are machine learning algorithms that are used in the fault prediction literature.

In our previous work, we proposed a novel kernel to predict the number of defects. We showed that support vector machines (SVM) with a precomputed kernel matrix performs as good as the SVM with linear or RBF kernels, in terms of the root mean square error (RMSE). We also have shown that the proposed regression method is comparable with other regression methods like linear regression and KNN (Okutan and Yildiz, 2013). In this paper, we extend our study and focus on the software defect prediction as a classification problem and consider the similarities of code patterns among different files of a software system to predict defectiveness. Although the classification method we suggest can be used to predict defectiveness, the novelty of our work is the proposition of a new kernel rather than a new defect prediction method. To reveal the relationship between the source code similarity and defectiveness, the precomputed kernel matrix is used with K-nearest neighbor classifier in addition to SVM. Furthermore, to extract similarity between any pair of files and generate kernels, clone detection and information retrieval techniques are used.

This paper is organized as follows: In Section 2, we give a background on kernel machines. In Section 3, we present a brief review of previous work on kernel methods and software defect prediction in general. We explain our proposed approach in Section 4 and give the experiment results in Section 5 before we conclude in Section 6.

2. Kernel machines

2.1. Support vector machines

Let’s assume that we have a training set

\[ S = \{ (\mathbf{x}_i, y_i), \mathbf{x}_i \in \mathbb{R}^n, y_i \in \{-1, 1\} \}_{i=1}^m \]  

(1)

where \( y_i \)'s are either +1 (positive class) or -1 (negative class) and each \( \mathbf{x}_i \) vector (with \( t \) entries) belongs to one of these classes. Based on this assumption, a hyperplane is defined as

\[ g(\mathbf{x}) = \mathbf{w}^T \mathbf{x} - b \]  

(2)

where \( \mathbf{w} \) shows the vector normal to the optimal hyperplane and \( b/||\mathbf{w}|| \) is the offset of the hyperplane from the origin on the direction of \( \mathbf{w} \).

Support vector machines generate the optimal hyperplane (or hypersphere, depending on the kernel) that can be used for classification or regression (Cortes and Vapnik, 1995). The optimal hyperplane is found by maximizing the margin which is defined as the distance between two nearest instances from either side of the hyperplane. As it is shown in the Fig. 1, the hyperplane is \( \frac{1}{||\mathbf{w}||} \) away from each class and it has a margin of \( \frac{2}{||\mathbf{w}||} \).

In order to calculate the margin of the optimum hyperplane, we need to find the boundaries of the two classes as hyperplanes first and then take the distance among these two hyperplanes. The boundary hyperplane of the positive class can be written as \( g(\mathbf{x}) = 1 \) whereas the boundary hyperplane of the negative class is \( g(\mathbf{x}) = -1 \).

Assuming that we have a linearly separable data set, the distance to the origin is \( (b+1)/||\mathbf{w}|| \) for the first hyperplane and \( (b-1)/||\mathbf{w}|| \) for the second one. The distance between these two hyperplanes is \( 2/||\mathbf{w}|| \). The optimum hyperplane separating these two classes maximizes this margin or minimizes \( ||\mathbf{w}|| \).

For positive instances we have

\[ \mathbf{w}^T \mathbf{x}_i - b \geq 1 \]  

(3)

and for negative instances we have

\[ \mathbf{w}^T \mathbf{x}_i - b \leq -1 \]  

(4)

These two constraints can be combined as \( y_i (\mathbf{w}^T \mathbf{x}_i - b) \geq 1 \). Now our problem becomes an optimization problem of

Minimize \( ||\mathbf{w}|| \)  

s. t. \( y_i (\mathbf{w}^T \mathbf{x}_i - b) \geq 1 \)  

(5)

To control the sensitivity of SVM and tolerate possible outliers, slack variables (\( \xi_i \)) are introduced. After adding constant \( C > 0 \), which determines the relative importance of maximizing the margin and minimizing the amount of slack, the problem changes slightly and becomes an optimization problem of

Minimize \( ||\mathbf{w}|| + C \sum \xi_i \)  

s. t. \( y_i (\mathbf{w}^T \mathbf{x}_i - b) \geq 1 - \xi_i \)  

(6)

In this new representation, we observe that if \( 0 < \xi_i \leq 1 \), the data point lies between the margin and the correct side of the hyperplane. On the other hand if \( \xi_i > 1 \), then the data point is misclassified. Data points that lie on the margin are called support vectors and regarded as the most important data points in the training set.

If the data points are not linearly separable, one can use a suitable transformation function to carry the data points to a higher dimension where linear separation is possible. The transformation is based on the dot product of two vectors as \( K(\mathbf{x}_i, \mathbf{x}_j) = (\mathbf{x}_i)^T \mathbf{x}_j \). Assuming that the transformation function \( \theta \) is defined as \( \theta: \mathbf{x} \rightarrow \Phi(\mathbf{x}) \), our new dot product in the high dimensional space becomes \( K(\mathbf{x}_i, \mathbf{x}_j) = \langle \Phi(\mathbf{x}_i), \Phi(\mathbf{x}_j) \rangle \). So, the kernel function can be defined

![Fig. 1. An optimal separating hyperplane (Alpaydin, 2004).](image-url)
as a function that returns the inner product between the images of two inputs in a feature space where the image function is shown with \( \Phi \) in our representation.

Checking whether any function constitutes a valid kernel, \( K(x_i, x_j) = \langle (\Phi(x))^T, \Phi(x) \rangle \) is a difficult task. Mercer’s theorem states that every semi-positive definite symmetric function is a kernel. Checking semi-positive definiteness for a function corresponds to checking semi-positive definiteness of the matrix in the form of

\[
K = \begin{bmatrix}
K(x_1, x_1) & K(x_1, x_2) & \cdots & K(x_1, x_N) \\
K(x_2, x_1) & K(x_2, x_2) & \cdots & K(x_2, x_N) \\
\vdots & \vdots & \ddots & \vdots \\
K(x_N, x_1) & K(x_N, x_2) & \cdots & K(x_N, x_N)
\end{bmatrix}
\]

(7)

Furthermore, if we apply the positive semi definiteness test and see that a matrix \( K \) is not a valid kernel matrix, then we can produce \( K_2 = K^TK \) which is guaranteed to be a valid kernel, since for any vector \( v \)

\[
vKv = vK^Tv = \|Kv\|^2 \geq 0
\]

There are other alternative ways of converting any symmetric but non semi-positive matrix to a valid kernel matrix. Some of these transformation methods are denoise, flip, diffusion, and shift and their effect on classification using kernel machines are given by Wu et al. (2005).

### 2.2. Kernel functions

There are different types of kernels used for software defect prediction in the literature (Gray et al., 2009; Zimmermann and Nagappan, 2009). The kernel function to be used is very much dependent on the type of the problem and should be chosen very carefully (Hsu et al., 2003). For example, polynomial kernels allow us to model the feature combinations up to the order of the polynomial. On the other hand, radial basis functions allow us to generate hypersphere or circle whereas linear kernels generate hyperplane or line for classification.

String kernels are used in areas like software plagiarism detection, document classification and filtering, information retrieval, and DNA analysis (Joachims, 1998). In all of these areas, the key point is to extract and measure similarities of text passages or strings. Levenshtein distance (edit-distance) kernel, Bag of words kernel, and P-gram kernel are some of the most important string kernels that are used in the literature (Shawe-Taylor and Cristianini, 2004).

### 3. Previous work

The defect prediction models in the literature use many different types of software metrics and most of the time it is very difficult to generalize the results of the prediction models. It was shown that the results achieved in a study for one data set might not be valid for other data sets (Menzies et al., 2007; Zimmermann and Nagappan, 2009).

Hall et al. (2011) state that more complex methods like SVM perform less well compared to the simpler techniques like naive Bayes (NB) or logistic regression. We should emphasize that the performance of SVM depends on the type of the kernel used. Linear kernels are simple and usually perform well on easy data sets, but they may underfit on non-linear data sets. On the other hand, RBF kernels are more complex and can learn non-linear relationships better, but they may overfit on linear and easy data sets. Furthermore, if the data is skewed, RBF kernels might not perform well although they can generate bright results with balanced data sets.

Arisholm et al. use a Java middle-ware system from a telecommunication company to compare several data mining and machine learning techniques like SVM, neural network, logistic regression, C4.5, and Boost based on different evaluation criteria like accuracy, precision/recall, receiver operating characteristic area (ROC), and cost-effectiveness measure (CE) (Arisholm et al., 2009). They use process metrics like change or fault history besides structural source code metrics. Their results suggest that the effect of different prediction techniques on the classification accuracy is limited, however if process metrics are included together with source code metrics, the prediction performance is significantly improved, compared to the case when only structural source code metrics are used.

Gondra (2008) considers defect proneness prediction as a binary classification problem and compares the performance of Artificial Neural Network (ANN) and SVM using a data set from NASA metrics data repository. He conclude that SVM is a better defect prediction technique when compared with ANN since its accuracy is higher (The accuracy of SVM is 87.4% whereas ANN has an accuracy of 72.6%). Arisholm et al. (2007) compare the performance of SVM with eight data mining techniques like C4.5, neural networks, and logistic regression in terms of defect classification accuracy (precision, recall, and ROC) using an industrial Java system. Although they suggest that C4.5 classification tree method performs better than other techniques in general, the results are comparable in terms of ROC area and the ROC value for SVM is 83.7% which is better than the ROC value of the six out of eight techniques.

Martins (2006) reviews kernel methods like the bag-of-words kernel, the p-spectrum kernel and the weighted all-substrings kernel to measure string similarity for text authorship attribution, language detection, and cross-language document matching. He show that for the first two text categorization issues, all kernel methods yield accuracies close to 100% provided that the kernel parameters are properly fine-tuned.

While checking the similarity of source code, some code clone and plagiarism detection tools like JPlag uses greedy-string-tiling algorithm in comparing strings and generating similarity matrix. On the other hand, local alignment (Smith–Waterman) algorithm was developed to find similarities between nucleotide or protein sequences (Smith and Waterman, 1981). Local alignment finds similar contiguous subsequences of two strings and generates similarity matrix among subsequences. Ji et al. (2007) use an adaptive version of local alignment where they took into consideration the frequencies of the subsequences while generating the kernel matrix. They show that the adaptive local alignment algorithm is more robust compared to the greedy-string-tiling algorithm used in JPlag (Prechelt et al., 2000).

Roy et al. (2009) classify the clone detection techniques based on their clone detection logic and compare their performance. They define four clone types i.e., type-1, type-2, type-3, and type-4 clones and classify clone detection tools and techniques according to which clone type they are able to detect. A type-1 clone is defined as identical code fragments where only layout, comments and white spaces might be changed. A type-2 clone is viewed as identical code fragments except identifiers, literals, white spaces or comments might be changed. In type-3 clones, besides literal or identifier variations, there might be some changes, additions or deletions in the statements. Lastly, type-4 clone implies functional similarity where two programs carry out the same task but with different source codes. The first three clone types (type-1 to type-3) consider syntactic similarities whereas type-4 clone considers functional or semantic similarities between code fragments. Roy et al. (2009) show that graph based clone techniques (where functional similarity is detected) and metric based clone detection
techniques are more successful compared to other text, tree or token based techniques.

Comparing the accuracy of the software defect predictors is really difficult, since most of the time the performance of a defect prediction technique is not consistent across different data sets. Lessmann et al. (2008) compare the performance of 22 defect prediction methods according to the area under curve (AUC) across 10 data sets from the NASA metric data repository (NASA, 2010) and Promise data repository (Boeticher et al., 2007). They show that the importance of a specific classification algorithm is less important than it is assumed to be, since no significant performance difference observed among the top 17 classifiers. Although Least-Square Support Vector Machines, Random forest and Bayes net found to be the most accurate methods, their superiority is not significant statistically. Moreover, D’Ambros et al. present a benchmark to provide a comparison for bug prediction approaches.

They state that defect prediction is a field where external validity is very hard to achieve since generalizing results to different contexts/learners proved to be a partially unsuccessful endeavour. Furthermore, they point out that the only way to gain certainty towards the presence or absence of external validity in defect prediction studies is to broaden the range of case studies (D’Ambros et al., 2012).

Song et al. propose a general software defect-proneness prediction framework that includes unbiased and comprehensive comparison of competing prediction systems and conclude that one should choose different learning techniques for different data sets which means that no method can dominate in all data sets since small details changed during experiment design may generate different results (Song et al., 2011). We believe that this approach is more realistic as the nature of the data, for instance whether it is balanced or not, may affect the performance of the learning method. If the data is skewed i.e., there are not enough defective instances to learn, one technique may perform poor, although it performs well on a balanced data set. On the other hand, Hall et al. make a literature review comprising 208 fault prediction studies published from January 2000 to December 2010 and suggest that simple, and easy to use modeling techniques like naive Bayes or logistic regression perform well. On the other hand, more complex modeling techniques, such as SVM perform relatively less well (Hall et al., 2011).

Kim et al. present a novel technique to predict high risk APIs in terms of producing potential bugs, on an industrial project Tizen-wearable at Samsung Electronics. The technique is called Remi (Risk Evaluation Method for Interface testing) that is targeted for testing APIs in the development process. They find that applying the technique in the development process increases the number of detected bugs and helps to reduce the resources required for executing test cases (Kim et al., 2015). Mende et al. state that the effort reduction gained by using defect prediction models is often ignored during evaluation of the prediction models. They build and evaluate a trivial defect prediction model based on the lines of code (LoC) measure on 13 NASA MDP data sets. They find that although the model performs well when evaluated using AUC, it appears to be not better or even worse than a random selection of source files, when evaluated using their proposed performance measure that takes the size of the modules into account (Mende and Koschke, 2009).

Menzies et al. show that in defect prediction studies, an approach useful in global context is often irrelevant in local contexts. They state that lessons learned after combining small parts of different data sources (i.e., the clusters) were superior to either generalizations formed over all the data or local lessons formed from particular projects (Menzies et al., 2011). Neighboring clusters from different data sets is not possible in our case since catching similarity between files from different projects is usually not possible. That is why we have to derive local lessons and be very careful not to generalize our results across different data sets.

4. Proposed approach

We believe that the extent of similarity among source code files is somehow related with the extent of similarity in their detectiveness. Similarity must be measured in terms of both syntactic and semantic features. We use three different techniques listed below to generate three precomputed kernel matrices for each data set.

1. **Plagiarism kernel:** Plagiarism detection tools consider structural similarity together with fingerprint based approaches and can be used to measure source code similarity. We use plagiarism detection tools to extract similarities among the files of a software system and use this similarity data to generate kernel matrix.

2. **Clone kernel:** There are many clone detection techniques in the literature most of which detects lexical similarity. We extract software similarity based on the software clones to generate kernel matrix.

3. **Textual similarity kernel:** Information retrieval models, e.g., vector space model (VSM) can be used to extract textual similarities between source code files and we use VSM with term frequency-inverse document frequency (tf – idf) to generate kernel matrix.

For each data set we generate three kernel matrices using the three techniques stated above and use them with SVM and KNN classifiers to learn the relationship between the source code similarity and detectiveness. In KNN, Euclidian distance is used to decide on the class of a test instance where the class of the majority of the nearby instances are taken into account. Since most of the similarity values in the kernel matrices are zero, we use a modified KNN in our experiments, where all the instances with nonzero similarity values contribute to the class label.**

4.1. Finding source code similarity

Plagiarism detection tools measure structural similarity using fingerprint based and content comparison techniques. JPlag (Prechelt et al., 2000) and MOSS are two of the most famous and widely used plagiarism detection tools worldwide that measure structural similarity in source codes. As it is confirmed in some previous studies (Cosma, 2008), we find that JPlag is not very successful in detecting all similarities and the kernel matrix generated from the JPlag is more sparse compared to the MOSS. That is why, during our experiments, we select MOSS as plagiarism detection tool to extract similarities among files to generate the precomputed kernel matrix for the SVM and KNN classifiers.

MOSS (Measure Of Software Similarity) is a tool developed by Alex Aiken and hosted by Stanford University. Users can submit source code to MOSS through its web interface to check plagiarism. For the purpose of plagiarism detection, two parameters of MOSS i.e., m and n need to be set very carefully. m represents the maximum number of times a given passage (shared code) may appear across files. If a certain piece of code appears more than m times, then it is not regarded as plagiarism, but assumed to be a library code. For our purpose, assuming there are N files in a data set, we set m to N since all types of similarities are needed. On the other hand, n represents the maximum number of matching files to include in the results. Since we need to extract as much similarity as possible, we set n to be N2 for each data set to include all similarities.

In addition to plagiarism detection tools, we use clone detection as an alternative approach to find source code similarities. The copy paste detector (CPD) of PMD source code analyser is a
clone detection technique that uses Karp-Rabin string matching algorithm (Karp and Rabin, 1987). Karp–Rabin string matching algorithm compares the hash values of strings rather than the strings themselves. Using CPD we extract software similarity based on the software clones and generate a kernel matrix based on this similarity.

Moreover, as a third alternative, vector space model is used to extract textual similarities between source code files. Each software module is taken as a `document’ instance. After constructing the vector space models of these documents with tf-idf, we calculate each entry of the kernel by the cosine similarity (Baeza-Yates and Ribeiro-Neto, 1999).

4.2. Kernel matrix computation

We use SVM and KNN with a precomputed kernel matrix which is composed of similarity measurements of the files in a software system. We use the kernel matrix and the class information of files (defective or not) with SVM and KNN classifiers and make the software prediction process easier compared to traditional methods.

Assume that a software system is composed of one or more versions, in total of $N$ files where each $m_i$ represents a software file. Then the kernel matrix generated using one of the three techniques defined above (in Section 4) is defined as

$$K = \begin{bmatrix} K(m_1, m_1) & K(m_1, m_2) & \cdots & K(m_1, m_N) \\ K(m_2, m_1) & K(m_2, m_2) & \cdots & K(m_2, m_N) \\ \vdots & \vdots & \ddots & \vdots \\ K(m_N, m_1) & K(m_N, m_2) & \cdots & K(m_N, m_N) \end{bmatrix}$$

(9)

where $K(m_i, m_j)$ represents the similarity between files $m_i$ and $m_j$.

Although it is not a necessity that the similarity matrices we generate are semi-definite, the kernel matrices computed for the data sets in our experiments satisfy the positive semi-definiteness attribute and are therefore valid kernels. We normalize the precomputed kernel matrices to the unit norm, as this may effect their generalization ability and also result in a smaller range for $C$. The normalized matrix $K_n$ is defined as

$$K_n(x, z) = \frac{K(x, z)}{\sqrt{K(x, x) K(z, z)}}$$

(10)

for all examples $x, z$.

5. Experiments and results

We select data sets from the Promise data repository (Boetticher et al., 2007) that are large enough to perform $10 \times 10$ cross validation. So, we eliminate some of the data sets that have less than 100 entries. Moreover, in order to calculate the textual similarity, we have to use data sets whose source code is available in the open source project repositories. Based on these selection criteria, we use camel, tomatocat, poi, xalan, jedit, velocity, ant, lucene, synapse, and ivy datasets as shown in Table 2.

We also compare our proposed algorithms with standard KNN, SVM and NB, which use static code metrics (Knn-m, Svm-m, Nb-m). The details of the software metrics used in those algorithms are given in Table 1.

5.1. A sample kernel matrix

For each data set, we generate three kernel matrices that are based on the similarities of files in the source code found by using techniques in Section 4. The value at $(i, j)$ index of a kernel matrix shows how much structural parallelism exists between the $i$th and $j$th files in the source code. The parallelism is measured by the percentage of similarity between corresponding files. If no similarity is detected for any two files, then the $(i, j)$ term of the kernel matrix corresponding to these files is set to zero.

To illustrate the precomputed kernel matrix generation with a simple example, a small part of the MOSS output for lucene data set is shown in Table 3. For instance, TermQuery and SpanTermQuery are 72 % similar which is something expected since both

### Table 1

<table>
<thead>
<tr>
<th>Metric</th>
<th>Metric full name</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>wmc</td>
<td>Weighted method per class</td>
<td>Chidamber and Kemerer (1991)</td>
</tr>
<tr>
<td>dit</td>
<td>Depth of inheritance tree</td>
<td>Chidamber and Kemerer (1991)</td>
</tr>
<tr>
<td>noc</td>
<td>Number of children</td>
<td>Chidamber and Kemerer (1991)</td>
</tr>
<tr>
<td>cbo</td>
<td>Coupling between Objects</td>
<td>Chidamber and Kemerer (1991)</td>
</tr>
<tr>
<td>rfc</td>
<td>Response for class</td>
<td>Chidamber and Kemerer (1991)</td>
</tr>
<tr>
<td>ca</td>
<td>Afferent couplings</td>
<td>Schanz and Iziurieta (2010)</td>
</tr>
<tr>
<td>ce</td>
<td>Efferent coupling</td>
<td>Schanz and Iziurieta (2010)</td>
</tr>
<tr>
<td>npm</td>
<td>Number of public methods</td>
<td>Bansiya and Davis (2002)</td>
</tr>
<tr>
<td>lcom3</td>
<td>Lack of cohesion in methods</td>
<td>Henderson-Sellers (1996)</td>
</tr>
<tr>
<td>loc</td>
<td>Lines of code</td>
<td>Bansiya and Davis (2002)</td>
</tr>
<tr>
<td>dam</td>
<td>Data access metric</td>
<td>Bansiya and Davis (2002)</td>
</tr>
<tr>
<td>mfa</td>
<td>Measure of aggregation</td>
<td>Bansiya and Davis (2002)</td>
</tr>
<tr>
<td>mfa</td>
<td>Measure of functional abstraction</td>
<td>Bansiya and Davis (2002)</td>
</tr>
<tr>
<td>cam</td>
<td>Cohesion among methods of class</td>
<td>Bansiya and Davis (2002)</td>
</tr>
<tr>
<td>ic</td>
<td>Inheritance coupling</td>
<td>Tang et al. (1999)</td>
</tr>
<tr>
<td>cbm</td>
<td>Coupling between methods</td>
<td>Tang et al. (1999)</td>
</tr>
<tr>
<td>amc</td>
<td>Average method complexity</td>
<td>Tang et al. (1999)</td>
</tr>
<tr>
<td>max_cc</td>
<td>max. (McCabe’s cyclomatic complexity)</td>
<td>McCabe (1976)</td>
</tr>
<tr>
<td>avg_cc</td>
<td>avg. (McCabe’s cyclomatic complexity)</td>
<td>McCabe (1976)</td>
</tr>
</tbody>
</table>

### Table 2

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Version</th>
<th># of Instances</th>
<th>% Defective instances</th>
</tr>
</thead>
<tbody>
<tr>
<td>ant</td>
<td>1.7</td>
<td>745</td>
<td>22.28</td>
</tr>
<tr>
<td>camel</td>
<td>1.0</td>
<td>339</td>
<td>3.83</td>
</tr>
<tr>
<td>ivy</td>
<td>2.0</td>
<td>352</td>
<td>11.36</td>
</tr>
<tr>
<td>jedit</td>
<td>4.0</td>
<td>306</td>
<td>24.51</td>
</tr>
<tr>
<td>lucene</td>
<td>2.4</td>
<td>250</td>
<td>59.71</td>
</tr>
<tr>
<td>poi</td>
<td>3.0</td>
<td>442</td>
<td>63.57</td>
</tr>
<tr>
<td>synapse</td>
<td>1.2</td>
<td>214</td>
<td>33.39</td>
</tr>
<tr>
<td>tomatocat</td>
<td>6.0</td>
<td>858</td>
<td>8.97</td>
</tr>
<tr>
<td>velocity</td>
<td>1.5</td>
<td>214</td>
<td>66.36</td>
</tr>
<tr>
<td>xalan</td>
<td>2.5, 2.6</td>
<td>1688</td>
<td>47.27</td>
</tr>
</tbody>
</table>

### Table 3

<table>
<thead>
<tr>
<th>Class1</th>
<th>Class2</th>
<th>Similarity (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>QueryTermVector</td>
<td>DirectoryIndexReader</td>
<td>38</td>
</tr>
<tr>
<td>FieldWriter</td>
<td>DirectoryIndexReader</td>
<td>24</td>
</tr>
<tr>
<td>TermQuery</td>
<td>FuzzyQuery</td>
<td>42</td>
</tr>
<tr>
<td>TermQuery</td>
<td>QueryTermVector</td>
<td>38</td>
</tr>
<tr>
<td>TermQuery</td>
<td>SpanTermQuery</td>
<td>72</td>
</tr>
<tr>
<td>MultiFieldQueryParser</td>
<td>FuzzyQuery</td>
<td>26</td>
</tr>
<tr>
<td>CustomScoreParser</td>
<td>FuzzyQuery</td>
<td>22</td>
</tr>
</tbody>
</table>
are descendants of the same base class. Furthermore, the files TermQuery and FuzzyQuery are found to be 42 percent similar, which can also be explained by looking at the similar code structures and inheritance relationships in each file.

A kernel matrix obtained from the output of the Table 3 is shown in Fig. 2. Using the file names as rows and columns, each cell in the kernel matrix is filled with the percentage of similarity of files in the corresponding row and column. We observe that the diagonal values in the kernel matrix are high since a file is compared with itself. Furthermore, when no similarity is detected between any two files, the corresponding entry in the kernel matrix (for these files) is set to zero.

5.2. Experiment steps

The major steps we follow in each experiment are:

1. We choose a data set from Promise data repository. The data set must have enough entries to be able to apply cross validation and have public source code to extract similarity based kernel matrices from the source code.

2. We choose one or more versions of this data set and download corresponding sources from open source data repositories.

3. We edit the defect data and change all bug features that are greater than 1 with 1. Since we perform classification rather than regression, the bug feature must be zero for non defective files and 1 for defective ones.

4. We generate a class feature file from the changed defect data file that includes two columns i.e., the file name and the bug feature.

5. The software pipeline developed during this research takes the source code of the selected data set as input and generates three kernel matrices based on the chosen plagiarism detection, clone detection and textual similarity detection methods. Assuming there are n files in the software system chosen, the software processes the similarity output and generates an n by n kernel matrix. Each cell of this kernel matrix shows how much similarity exists between the files given in the row and column of the matrix. Then the kernel matrix is normalized and saved to a file. An example kernel matrix generated from a sample output in Table 3 is shown in Fig. 2.

6. We input the class feature file (generated in step 4) and the precomputed kernel matrices to SVM and KNN classifiers to learn the relationship between defect proneness and similarity. All classifications are done on Weka experimenter with 10 × 10 cross validation. Although we generate the kernel matrices before cross validation, in each experiment the sub kernel matrices used for the training and test are disjoint since the files included in the training and test sets are different.

5.3. Comparison of SVM kernels

In this section, we will compare the performances of Svm-p (plagiarism kernel), Svm-c (clone kernel), and Svm-t (textual similarity kernel), and Svm-r (linear kernel obtained with static code metrics listed in Table 1) classifiers. Furthermore, Svm-p, Svm-c, and Svm-t use the class feature file and the precomputed kernel matrix explained in steps 4 and 5 in Section 5.2. On the other hand, for each data set, 10 × 10 folds cross validation is applied and SVM parameters are tuned. We do not include the RBF kernel in our experiments, since RBF kernel has an overfit problem in the data sets which are linear and easy in general.

For our experiments, the data sets are not skewed, so we decide to use error metric to compare the performance of classifiers. Table 4 shows average error rates and ranks for Svm-p, Svm-c, and Svm-r, and Svm-r for all data sets. First, the results show that the performance of Svm-r is better than the performance of both other kernels and ordinary Svm-r in all experiments.
To check for the statistical significance of the results, we apply two tailed t-test (with a p-value of 0.05) to the results of Svm-p, Svm-c, Svm-t, and Svm-m. We observe that, for all data sets the difference between Svm-t and Svm-m is statistically significant. When compared with other kernels, we see that in 7 of 10 data sets the difference between Svm-t and Svm-p is significant. Similarly in 9 of 10 data sets the difference between Svm-t and Svm-c is significant.

As summarised by Roy et al. (2009), code clones can both increase and decrease the ‘quality’ (lack of defects) in a system. Our results also show similar patterns: Svm-c is the second best algorithm in camel, velocity datasets, whereas it is the worst algorithm in ant, jedit, synapse, tomcat datasets.

We think that error rates may not provide sufficient information to understand the behaviour of the classifier on data sets with different defectiveness rates. For example, given the high defect rates on some data sets, and low defect rates on others, one could expect the precision-recall to vary. That is why, to cross check our findings we use F-measure to compare the performances of Svm-p, Svm-c, Svm-t, and Svm-m. The box plots of F-measure values of Svm-p, Svm-c, Svm-t, and Svm-m are shown in Fig. 3. When we compare Svm-p, Svm-c, Svm-t, and Svm-m in terms of F-measure, we observe similar results with the error rates where the F-measure of Svm-t is better in all data sets. Furthermore, to check if the value of F-measure is influenced by a high/low recall or high/low precision, we present the precision-recall values in Table 5 and observe that although there are low precision or recall values for Svm-m and other kernels, the precision-recall values for Svm-t are compatible with F-measure results.

### 5.4. Comparison of KNN kernels

In this section, we compare the performances of Knn-p (plagiarism kernel), Knn-c (clone kernel), Knn-t (textual similarity kernel), and Knn-m. Table 6 shows average error rates and average ranks for Knn-p, Knn-c, Knn-t, and Knn-m.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Svm-p</th>
<th>Svm-c</th>
<th>Svm-t</th>
<th>Svm-m</th>
</tr>
</thead>
<tbody>
<tr>
<td>ant</td>
<td>21.67</td>
<td>62.69</td>
<td>21.17</td>
<td>62.28</td>
</tr>
<tr>
<td>camel</td>
<td>10.81</td>
<td>7.05</td>
<td>6.58</td>
<td>36.18</td>
</tr>
<tr>
<td>ivy</td>
<td>14.74</td>
<td>49.95</td>
<td>11.08</td>
<td>55.31</td>
</tr>
<tr>
<td>jedit</td>
<td>22.55</td>
<td>63.74</td>
<td>21.16</td>
<td>54.62</td>
</tr>
<tr>
<td>lucene</td>
<td>43.07</td>
<td>37.41</td>
<td>24.71</td>
<td>33.15</td>
</tr>
<tr>
<td>poi</td>
<td>31.35</td>
<td>32.56</td>
<td>20.64</td>
<td>57.20</td>
</tr>
<tr>
<td>synapse</td>
<td>26.08</td>
<td>60.82</td>
<td>21.54</td>
<td>59.44</td>
</tr>
<tr>
<td>tomcat</td>
<td>15.46</td>
<td>74.80</td>
<td>9.33</td>
<td>51.20</td>
</tr>
<tr>
<td>velocity</td>
<td>35.34</td>
<td>27.20</td>
<td>15.28</td>
<td>28.81</td>
</tr>
<tr>
<td>xalan</td>
<td>35.41</td>
<td>50.13</td>
<td>22.50</td>
<td>56.12</td>
</tr>
<tr>
<td>Average ranks</td>
<td>2.6</td>
<td>3.1</td>
<td>1.0</td>
<td>3.3</td>
</tr>
</tbody>
</table>

Table 5: Precision-recall values for Svm-p, Svm-c, Svm-t, and Svm-m.

### 5.4. Comparison of KNN kernels

In this section, we compare the performances of Knn-p (plagiarism kernel), Knn-c (clone kernel), Knn-t (textual similarity kernel), and Knn-m. Table 6 shows average error rates and average ranks for Knn-p, Knn-c, Knn-t, and Knn-m.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Knn-p</th>
<th>Knn-c</th>
<th>Knn-t</th>
<th>Knn-m</th>
</tr>
</thead>
<tbody>
<tr>
<td>ant</td>
<td>18.62</td>
<td>19.75</td>
<td>22.73</td>
<td>21.34</td>
</tr>
<tr>
<td>camel</td>
<td>6.79</td>
<td>7.05</td>
<td>6.58</td>
<td>10.23</td>
</tr>
<tr>
<td>ivy</td>
<td>13.10</td>
<td>16.33</td>
<td>11.36</td>
<td>14.94</td>
</tr>
<tr>
<td>jedit</td>
<td>21.80</td>
<td>26.76</td>
<td>25.66</td>
<td>25.47</td>
</tr>
<tr>
<td>lucene</td>
<td>36.27</td>
<td>43.05</td>
<td>38.59</td>
<td>29.93</td>
</tr>
<tr>
<td>poi</td>
<td>23.39</td>
<td>24.63</td>
<td>35.92</td>
<td>22.62</td>
</tr>
<tr>
<td>synapse</td>
<td>23.02</td>
<td>26.72</td>
<td>33.70</td>
<td>25.55</td>
</tr>
<tr>
<td>tomcat</td>
<td>11.25</td>
<td>10.94</td>
<td>9.74</td>
<td>13.40</td>
</tr>
<tr>
<td>velocity</td>
<td>43.48</td>
<td>49.47</td>
<td>33.27</td>
<td>24.27</td>
</tr>
<tr>
<td>xalan</td>
<td>31.20</td>
<td>26.51</td>
<td>49.06</td>
<td>30.11</td>
</tr>
<tr>
<td>Average ranks</td>
<td>2.0</td>
<td>3.0</td>
<td>2.7</td>
<td>2.3</td>
</tr>
</tbody>
</table>

Table 6: Average error rates and average ranks for Knn-p, Knn-c, Knn-t, and Knn-m.
5.5. Overall comparison

Although we state that the novelty of our work is the proposition of new kernels rather than a new defect prediction method, we compared the performance of our kernels with ordinary Svm-m and Knn-m defect prediction methods in the previous sections. In this section, we additionally compare our kernels with Nb-m classifier. Table 8 shows average error rates and average ranks for Svm-t, Knn-p, Knn-c, and Nb-m for all data sets. We observe that in 9 data sets, Svm-t performs better than Nb-m and in 7 data sets the difference is statistically significant. Similarly, in 7 data sets Knn-p is better than Nb-m and in 3 of these data sets the difference is significant. To cross check our findings, the box plots of F-measure values of Svm-t, Knn-p, Knn-c, and Nb-m are also shown in Fig. 5. We observe that in general the performance of the selected kernels are comparable and usually better than Nb-m classifier.

6. Threats to validity

Perry et al. list three types of validity threats to take into consideration in research studies. To alleviate these threats, we take the following measures (Perry et al., 2000):
An internal validity threat is observed if a cause effect relationship could not be established between the independent variables and the results. We address this issue by cross checking our results on different data sets and use $10 \times 10$ fold cross validation. Construct validity threats might arise when there are errors in the measurements. To mitigate this threat, we minimize the manual interventions in kernel matrix generation and learning processes.

External validity threats might arise if the results observed for one data set are not valid for other data sets. To mitigate the external validity threat, we test our proposed method on 10 open source data sets. However, although our results are good and promising and our work is unique in the sense that source code similarity based kernel matrices are used for defect proneness prediction, further research with more data sets and other similarity extraction tools is needed to justify our findings.

7. Conclusion

In this paper, we extend our precomputed kernel matrix model (Okutan and Yildiz, 2013) to be used with SVM and KNN classifiers for defect proneness prediction. Moreover, besides plagiarism detection, we use two more techniques to generate kernel matrices. The precomputed kernel matrices show the extent of the similarity
among the files of the software system and are generated from the outputs of the plagiarism detection, clone detection, and textual similarity detection tools. We compare our approach with the linear kernel SVM, KNN, and NB classifiers on 10 open source projects from Promise data repository and show that SVM and KNN with a precomputed kernel matrix could achieve comparable and even better results.

As a summary, the main contributions of this paper are:

- We propose three novel kernels that could be used with SVM and KNN for defect prediction.

- Extracting the metrics of a software system for defect prediction is necessary since most defect prediction techniques are based on the software metrics. Our method is making it possible to predict defectiveness without the need to extract the software metrics, since SVM and KNN with the precomputed kernel can be used to predict defectiveness based on the source code similarity.

As a future work, we plan to refine our research to include semantic similarity while calculating the kernel matrices. Graph based clone detection techniques are able to detect functional similarity and are more successful compared to the to-
ken based techniques in extracting semantic similarities (Roy et al., 2009). We believe that when semantic similarity is considered besides structural similarity, the accuracy of defect prediction using a precomputed kernel matrix could be higher. Furthermore, we plan to extend our work, where we want to construct a composite kernel from kernels extracted through different techniques.

References


Ahmet Okutan He received his PhD degree in computer science from Isik University in 2012. He received his BS degree in computer engineering from Bogazici University in 1998. He worked as analyst, architect, developer, tester, project manager in more than 50 middle and large scale software projects. He has professional experience regarding software project management, system analysis and design, software design and development, software testing, database management systems, and artificial intelligence. His research interests include software quality prediction and software defectiveness prediction.

Olcay Taner Yildiz He is currently a full-time associate professor at the Department of Computer Science and Engineering, Isik University, Turkey. He received the BS, MS, and PhD degrees in computer science from Bogazici University, Istanbul, in 1997, 2000, and 2005, respectively. He worked extensively on machine learning, specifically model selection and decision trees. His current research interests include software engineering, natural language processing, and bioinformatics.