Dokuz Eylül Üniversitesi Mühendislik Fakültesi Fen ve Mühendislik Dergisi
Dokuz Eylul University Faculty of Engineering Journal of Science and Engineering

Basılı/Printed ISSN: 1302-9304. Elektronik/Online ISSN: 2547-958X

An Ordinal Classification Approach for Software Bug Prediction

Yazılım Hata Tahmini için Sıralı Sınıflandırma Yaklaşımı

Elife Öztürk Kıyak [1], **Kökten Ulaş Birant** [2], **Derya Birant** [3*]

[1] Dokuz Eylul University, The Graduate School of Natural and Applied Sciences, Izmir, TURKEY
[2,3] Dokuz Eylul University, Engineering Faculty, Department of Computer Engineering, Izmir, TURKEY
Sorumlu Yazar / Corresponding Author *: derya@cs.deu.edu.tr

Geliş Tarihi / Received: 28.11.2018 *DOI:10.21205/deufmd.2019216218*
Kabul Tarihi / Accepted: 08.01.2019 *Araştırma Makalesi/Research Article*

Atıf şekli/ How to cite: KIYAK, ÖZTÜRK, E., BİRANT, K. U., BİRANT. D. (2019). An Ordinal Classification Approach for Software Bug Prediction. DEUFMD, 21(62), 533-544.

Abstract

Software bug prediction is the process of utilizing classification and/or regression algorithms to predict the presence of possible errors (or defects) in a source code. However, current classification studies in the literature assume that the target attribute values in the datasets are binary (i.e. buggy or non-buggy) or unordered, so they lose inherent order between the class values such as zero, less and more bug levels. To overcome this drawback, this study proposes a novel approach which suggests ordinal classification methods as a solution for software bug prediction problem. This article compares ordinal and nominal versions of various classification algorithms (random forest, support vector machine, Naive Bayes and k-nearest neighbor) in terms of classification performance on real-world 38 software engineering datasets. The results indicate that ordinal classification approach achieves better classification accuracy on average than the traditional (nominal) solutions.
Keywords: *Software bug prediction, Ordinal classification, Software engineering, Software quality*

Öz

Yazılım hata tahmini, kaynak kodda bulunan olası hataların (veya kusurların) varlığını tahmin etmek için sınıflandırma ve/veya regresyon algoritmalarının kullanımı işlemidir. Fakat, literatürde bulunan sınıflandırma çalışmaları, veri setlerindeki hedef özellik değerlerini iki olasılıklı (hatalı veya hatasız) veya sırasız olarak kabul etmektedir. Bu nedenle; sıfır, az veya çok hatalı gibi sınıf değerleri arasındaki sıralama mantığını değerlendirmemektedir. Bu eksikliği gidermek amacıyla, bu çalışma, yazılım hata tahminleme problemi için sıralı sınıflandırma metotlarını kullanan yeni bir yaklaşım önermektedir. Makalede, çeşitli sınıflandırma algoritmalarının (rastgele orman, destek vektör makineleri, Naive Bayes ve k-en yakın komşu) sıralı ve itibari sürümleri, yazılım mühendisliği alanındaki 38 gerçek veriseti üzerinde sınıflandırma performansları açısından karşılaştırılmıştır. Sonuçlar, sıralı sınıflandırma yaklaşımının geleneksel (itibari) çözümlere nispeten ortalamada daha iyi bir sınıflandırma doğruluğuna ulaştığını göstermektedir.
Anahtar Kelimeler: *Yazılım hata tahmini, Sıralı sınıflandırma, Yazılım mühendisliği, Yazılım kalitesi*

1. Introduction

A *bug* (defect or fault) is an anomaly in the software that may cause it to behave incorrectly and it can be considered as the result of an error [1]. Bug prediction is a significant research topic in empirical studies of software engineering. *Software bug prediction* is the process of constructing a learning model on software metrics and defect information to predict the possible bug levels in software modules. The quality of bug prediction models is highly dependent on the selection of learning methods and algorithms. For this reason, this study investigates and compares different classification algorithms for bug prediction problem.

Classification is a major task in data analysis that is used to predict the categorical label of a specific instance based on the model. Classification tasks can be categorized into two broad types, named nominal classification and ordinal classification, based on whether their class labels are ordered or not. *Nominal classification* (binary or multi-class) assigns an instance to exactly one of the classes which are not ordered from the best to the worst such as gender (male, female) or hair color (blonde, brown, brunette), so it discards the order among instances. *Ordinal classification* is a kind of multi-class classification where there is an inherent ordering between the classes such as bad < average < good < excellent. The ordinal classification can be regarded as a special case of the nominal one, when additional information about the rank order of the classes is available. Some studies [2,3] proved that the ordinal classification approach is marginally better than the traditional multi-class classification approach.

Although there are many classification studies [4-11] performed in software bug prediction, to the best of our knowledge, there has been no prior detailed investigation for ordinal classification in this problem. To fill this gap in the literature, the study presented in this article focuses on the application of ordinal classification algorithms on real-world software bug data. Actually, the number of bugs found in a software module during test contain an ordinal response variable, since it can be categorized as high > medium > low.

The novelty and main contributions of this article are as follows: (i) it provides a brief survey of classification studies which has been revealed to predict software bugs, (ii) it is the first study that the ordinal classification methods have been implemented for software bug prediction, (iii) it compares traditional random forest (RF), support vector machine (SVM), Naive Bayes (NB) and k-nearest neighbor (KNN) algorithms with their ordinal versions, (iv) it presents experimental studies conducted on 38 different software bug datasets to demonstrate better classification performance of ordinal classification rather than conventional (nominal) classification algorithms in terms of accuracy.

The organization of the paper is as follows. Section 2 briefly summarizes the related previous studies on the subject. Section 3 explains the material and methods proposed in this paper. Section 4 contains experimental work including dataset description, experiment procedure and results on ordinal data. Finally, Section 5 gives concluding remarks and future directions.

2. Related Work

Ordinal classification has been applied in various fields, including health [2], atmospheric research [12], image classification [13], transport system [14], emergency and disaster information services [15]. A better classification performance was obtained in these studies when the order between the class labels was taken into account. There are only a few studies in software engineering related to ordinal learning. A study [16] used ordinal classification to obtain the intensity of code smells. Code smell severity is considered as an ordinal variable. Another software engineering related study [17] discovers ordinal association rules, instead of ordinal classification. However, to the best of our knowledge, our study is the first study that the ordinal classification methods have been implemented for software bug prediction.

Metric based bug prediction is to analyze a set of independent metric variables (the predictors) in the historical data and then classify the unknown ones based on these variables. Metric based bug prediction has proved to be very useful in various studies [18,19] since it deals with reducing the cost of testing as well as to improve the quality of the software end product.

Bug prediction has been widely studied for binary classification [4-7,9,11] in which the dependent variable (bugs) is divided into two groups: one with no bugs and the other with at least one. In other words, they regard bug value as binary: buggy or non-buggy. However, in the present study, we addressed the problem of multi-class classification.

While binary classification determines whether a program is defective or non-defective without taking into account the number of defects, some studies conducted to predict how many defects exist in a software module. As noted in the study [18], it is also valuable to be able to predict the number of bugs in a software project. Therefore, several studies [20,21] focused on the prediction of the number of defects with respect to independent variables by using regression technique. While [20] used statistical linear regression, [21] proposed an integrated regression model.

In the literature, various algorithms have been used by different studies when predicting bugs on a software project. For instance, neural network was used to develop a bug prediction model [4]. SVM and KNN algorithms are compared to find similarities of different files to predict defectiveness [8]. NB algorithm was preferred by some studies [9,11]. Similarly, Bayesian networks were applied to obtain the probabilities of appearances of defects [10]. However, differently from these previous studies, our study compares nominal and ordinal versions of various algorithms (RF, SVM, NB and KNN).

In the literature, there exist two types of bug prediction researches: constructing a classification model on past data of the same software project (*within-project approach*) [5] or belonging to different projects (*cross-project approach*) [9,22,23]. Each approach has its own benefits and drawbacks [5]. Another research study [11] applied the combination of within and cross-project data when historical data is limited in the project.

In order to reduce irrelevant source code metrics, some studies applied different feature selection techniques, including genetic algorithm [24], Bayesian network [10], greedy forward selection [25], and Pearson correlation method [7]. Kumar et al. [4] used several techniques one after another, including t-test, univariate logistic regression, correlation analysis and multivariate linear regression. After feature selection step, the obtained set of software metrics are considered as input to develop a bug prediction model.

An imbalanced class in the dataset is one of the main problems in software bug prediction, i.e. the number of instances that belong to the "buggy" class is far less compared to the number of instances that belong to the "non-buggy" class. Various techniques have been proposed to address this problem by specifying particular instances as missing [26], balancing data using SMOTE technique [9] or applying ensemble-based techniques [25,27]. Wijaya and Wahono [28] used random undersampling technique to deal with imbalanced data [28]. Tomar and Agarwal [29] proposed a system that assigns higher misclassification cost to the buggy instances and lower cost to the non-buggy instances. Some studies [30,31] compared different imbalance approaches such as sampling, cost-sensitive, ensemble, threshold-moving or hybrid approaches.

3. Material and Methods

3.1. Ordinal Classification

Assume that $D=\{(x_i, y_i) \mid i=1,...,n\}$ denote the dataset that has n instances, where an input vector x_i belongs to the input feature space X, the associated class label y_i belongs to the output set $Y = \{C_1, C_2,...,C_k\}$ and k is the number of classes. Ordinal classification is a form of multi-class nominal classification where there exists an inherent ordering among the class labels, i.e., $C_1 < C_2<...<C_k$. If the target attribute contains numerical values, a set of thresholds $\theta_1, \theta_2, \dots, \theta_{k-1}$ with the property $\theta_1 < \theta_2 <...<\theta_{k-1}$ divides the real number line into k disjoint segments as follows.

$$y = \begin{cases} C_1 & if\ y^* \le \theta_1, \\ C_2 & if\ \theta_1 < y^* \le \theta_2, \\ \vdots & \\ C_k & if\ \theta_{k-1} < y^*. \end{cases} \qquad (1)$$

Ordinal classification approach used in this study transforms an ordinal classification problem into a set of binary classification problems, which are separately solved by different classifiers and then class labels are predicted by combining the binary outputs [3]. As shown in Figure 1, k class problem is converted into a set of k-1 binary sub-problems. It is important to note that, the performance of the approach depends on the threshold values as well as the way of combining the outputs.

The approach aims to learn the cumulative probabilities $\Pr(y_i \leq C_i)$ instead of the distinct class probabilities $\Pr(y_i = C_i)$ as follows.

$\Pr(C_1) = 1 - \Pr(\text{class} > C_1)$

$\Pr(C_i) = \Pr(\text{class} > C_{i-1}) - \Pr(\text{class} > C_i), \quad 1 < i < k$

$\Pr(C_k) = \Pr(\text{class} > C_{k-1})$

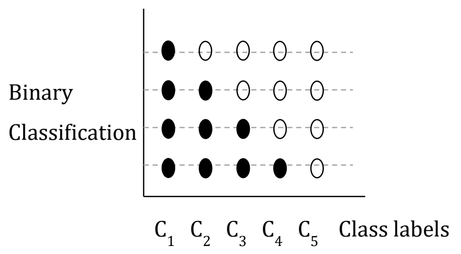

Figure 1. Conversion of *k* class ordinal data into *k*-1 binary class data.

3.2. Classification Algorithms

The classification algorithms used in this study are briefly described as follows.

Random Forest (RF) is an ensemble classification algorithm that consists of multiple decision trees and put them together to obtain more accurate prediction. Each tree is generated from bootstrap samples extracted from original dataset by replacement. It uses bootstrap aggregating and evaluates different subsets of features at each node to reduce overfitting. The predictions of all trees are gathered by a voting mechanism such as majority voting.

Support Vector Machine (SVM) is a supervised machine learning algorithm that can be used for classification or regression problems. The goal of the support vector machine is to discover an optimal hyperplane in an N-dimensional space that maximizes the margin between classes. SVM can effectively perform linear and non-linear classification. In the case of non-linear classification, SVM maps the inputs into high-dimensional spaces by using a kernel trick. SVM provides advantages in classifying small size, non-linear, complex related and high-dimensional instances.

Naive Bayes (NB) algorithm is another supervised machine learning algorithm based on Bayes Theorem which is an equation that gives the relationship of conditional probabilities to predict class labels. The aim is to obtain the probability of a class label according to the observed features. It assumes that each feature is independent of other features. This algorithm is useful when the number of attributes is high.

K-nearest neighbor (KNN) is a simple supervised classification algorithm in which an object is classified by looking at the *k* nearest objects and by selecting most frequently occurring class. A distance function (i.e. Euclidean, Minkowski, Manhattan) is used to calculate the distances of the neighbors to the given point. The KNN algorithm provides highly effective results in the presence of large training sets. The optimal *k* value is found by experiments based on the performances of the classifiers.

4. Experimental Studies

In the experimental studies, ordinal classification approach was applied on 38 software bug datasets to demonstrate its competitive superiority over the traditional nominal classification. We compared ordinal and nominal versions of four classification algorithms (random forest, support vector machine, Naive Bayes and k-nearest neighbor) separately by using Weka tool [32].

In the following subsections, dataset description, experimental work details and obtained results are explained respectively.

4.1. Dataset Description

In this study, real-world datasets from Tera-PROMISE repository [33] was used to conduct experiments. PROMISE repository [34] is one of the most preferred and the largest repositories for software engineering researches. Since datasets in the repository are publicly available, studies can be easily repeated and verified. This repository was also used by several studies [4-11,35,36]. Datasets contain source code metrics that can be used to evaluate the quality of the software or utilized to predict bugs. Table 1 shows 20 independent object-oriented metrics such as "wmc", "dit", "noc" etc. and one dependent variable "bug" which is used for prediction.

Table 1. List of software metrics

ID	Metric	Metric Full Name	Description	Data Type
1	WMC	Weighted Methods per Class	Sum of the complexities of methods defined in class	Numeric
2	DIT	Depth of Inheritance Tree	Maximum level of the inheritance hierarchy of a class	Numeric
3	NOC	Number of Children	The number of subclasses of a class	Numeric
4	CBO	Coupling Between Objects	The number of coupled classes	Numeric
5	RFC	Response for a Class	The number of all methods in a class and methods called by these methods	Numeric
6	LCOM	Lack of Cohesion in Methods	The number of set of methods that shared references to instance variables	Numeric
7	CA	Afferent Couplings	The number of classes that use a particular class	Numeric
8	CE	Efferent Couplings	The number of classes which are used by specific class	Numeric
9	NPM	Number of Public Methods	The number of all public methods stated in a class	Numeric
10	LCOM3	Lack of Cohesion in Methods	It is a variation of LCOM	Numeric
11	LOC	Lines of Code	The number of physical lines of code within the method	Numeric
12	DAM	Data Access Metric	the number of private attributes is divided by all number of attributes in the class	Numeric
13	MOA	Measure of Aggregation	The number of types of data declarations that are user-defined classes	Numeric
14	MFA	Measure of Functional Abstraction	The number of methods inherited by a class is divided by the number of methods	Numeric
15	CAM	Cohesion Among Methods of Class	The relatedness among methods based upon the parameter list of these methods	Numeric
16	IC	Inheritance Coupling	The number of parent classes to which a given class is coupled	Numeric
17	CBM	Coupling Between Methods	Total number of new methods to which all the inherited methods are coupled	Numeric
18	AMC	Average Method Complexity	Average method size which means the number of binary codes for each class	Numeric
19	MAX_CC	Maximum McCabe's Cyclomatic Complexity	Maximum number of independent paths in a method	Numeric
20	AVG_CC	Average McCabe's Cyclomatic Complexity	Average number of independent paths in a method	Numeric
21	Bug	Number of Bugs	Number of bugs detected in the class	Numeric

Table 2 shows data characteristics: the project name, its release, the number of instances, the percentage of bugs, lines of code and type. There are 28 releases of 12 open-source projects and 10 academic projects [9,11]. Bug percentage is below 10% for 5 datasets, between 10% and 20% for 20 datasets and above 20% for 13 datasets.

4.2. Experimental Work

In this study, we defined three ordinal class labels for bug proneness: bug free, less buggy, and more buggy. All instances where bug is zero were accepted as bug free classes. The instances that have only one bug were marked as less buggy, and the others that have more than or equal to two bugs were accepted as more buggy. Converting bug data from numeric to categorical enabled the implementation of ordinal classification since instances were ranked with respect to their bug values. After conversion, the popular classification algorithms were compared by keeping all the parameters as default in Weka tool.

Table 2. The basic characteristics of the datasets
(OS: open-source, AC: academic, KLOC: Kilo Lines of Code)

ID	Project	Release	# of Ins.	Bug (%)	KLOC	Type
D1		1.3	125	16		OS
D2		1.4	178	22.47		OS
D3	Ant	1.5	293	10.92		OS
D4		1.6	351	26.21		OS
D5		1.7	745	22.28	208	OS
D6	Arc	-	234	11.54	31	AC
D7	Berek	-	43	37.21		AC
D8		1.0	339	3.83	33	OS
D9	Camel	1.4	872	16.63	98	OS
D10		1.6	965	19.48	113	OS
D11	E-learning	-	64	7.81	3	AC
D12	Forest	0.7	29	17.24		OS
D13		0.8	32	6.25		OS
D14	Ivy	1.4	241	6.64	59	OS
D15		2.0	352	11.36	87	OS
D16		4.0	306	24.51	144	OS
D17	Jedit	4.1	312	25.32	153	OS
D18		4.2	367	13.08	170	OS
D19		4.3	492	2.24	202	OS
D20	Intercafe	-	125	16	11	AC
D21	Kalkulator	-	178	22.47	4	AC
D22	Log4j	1.0	135	25.19	21	OS
D23		1.2	205	92.20	38	OS
D24	Pbeans	2.0	51	19.61	15	OS
D25	Poi	2.0	314	11.78	93	OS
D26	Redaktor	-	176	15.34	59	AC
D27	Serapion	-	45	20	10	AC
D28	Skarbonka	-	45	20	15	AC
D29	Synapse	1.0	157	10.19	28	OS
D30		1.1	222	27.03	42	OS
D31	Systemdata	-	65	13.85	15	AC
D32	Termoproject	-	42	30.95	8	AC
D33	Tomcat	1.0	585	13.16	300	OS
D34	Xalan	2.4	723	15.21	225	OS
D35		2.7	909	98.79	428	OS
D36		1.2	440	16.14	159	OS
D37	Xerces	1.3	453	15.23	167	OS
D38		1.4	588	74.32	141	OS

These default parameters can be summarized as follows:

- For RF, the number of trees is set to 100, the number of features is calculated by int($\log_2$(#predictors)+1), so it is arranged as 5 for the experiments in this study.
- For SVM, the complexity constant *c* is set to 1, the kernel is selected as PolyKernel and epsilon value is assigned as 1.0E-12.
- NB uses a probabilistic model to infer the most likely class without a specific input parameter.
- For KNN, distance function is configured as Euclidean Distance and *k* value (the number of neighbors) is specified as 1.

For each dataset, 10-fold cross validation and leave-one-out cross validation techniques were applied to compare the ordinal and nominal version of algorithms. *N-fold cross validation* (nFCV) is widely used validation method that separates dataset into equal *n* subsets. One of the subsets is used as test set and other parts (*n*-1) are used as training set and this procedure is repeated *n* times so that all parts are used as test and training set. *Leave-one-out cross validation* (LOOCV) uses a single instance from the dataset as the test data, and the remaining part of the dataset as the training data. This process is repeated for all instances. We used LOOCV technique since it would be the best option for small datasets, because we would need to maximize the availability of the training data.

4.3. Experimental Results

In this study, the developed classification models were evaluated according to accuracy values. *Accuracy* is a performance measure that shows the percentage of correctly predicted observations and calculated by following formula:

$$Accuracy = \frac{(TP + TN)}{(TP + TN + FP + FN)} \quad (2)$$

where *TP* (true positives) and *TN* (true negatives) represent the number of instances correctly predicted as actual classes ("positive" and "negative"). In other words, *TP* means that the value of actual class is "positive" and predicted class is "positive", while *TN* shows that the actual class is "negative" and predicted class is "negative". However, *FP* (false positives) and *FN* (false negatives) denote the number of instances incorrectly classified. To put it another way, *FN* occurs when actual class is "positive" and predicted class is "negative". If actual class is "negative" and predicted value is "positive" then it is identified as *FP*.

In the experimental studies, we eliminated some of the datasets that have less than 70% accuracy for most of the classification algorithms. Because this means that the dataset is improper for classification task due to a reason, for example (i) insufficient training samples, (ii) imbalanced distribution of the data between classes, (iii) the presence of outlier (noisy) values, or (iv) adjacency of class's intervals.

While, Table 3 shows nFCV results, Table 4 gives LOOCV results for both nominal and ordinal versions of classification algorithms (random forest, support vector machine, Naive Bayes and k-nearest neighbor) on the datasets.

When the average results in Table 3 are examined, it is clearly seen that ordinal classification outperforms or equals to nominal classification for all algorithms. Ordinal NB (75.54%) is significantly more accurate than conventional NB (73.07%) on average. The small improvement over nominal case shows that ordering information can become more useful when the RF algorithm is used. Ordinal SVM produced a slight increment in classification accuracy. However, ordinal KNN showed no change or very slight increase when comparing with nominal KNN. Ordinal RF achieved the best performance with 83.92% accuracy on average.

Table 4 shows the leave-one-out cross validation results. When LOOCV technique is used (instead of nFCV), the difference in classification accuracy between ordinal and nominal NB algorithm remains high, 75.19% versus 72.82%. Among the algorithms applied in this study, ordinal RF has superiority in terms of classification accuracy. Improvements also exist for SVM and KNN algorithms.

Figure 2 shows the number of data sets in which the ordinal algorithms perform equal to or better than nominal versions when 10-FCV technique is used. On 30 datasets, ordinal NB is equal to or more accurate than conventional NB. Ordinal RF wins against plain RF or tied on 31 datasets of 38 ones. Similarly, Figure 3 shows the number of dataset when LOOCV technique is used. In this case, ordinal RF algorithm is better than or equal to its nominal version on 28 datasets. Compared to nominal SVM, the equal or win ratio for ordinal SVM is 24/38 for 10-FCV and 27/38 for LOOCV. KNN shows almost the same results for both 10-FCV and LOOCV (except the dataset Xerces 1.2).

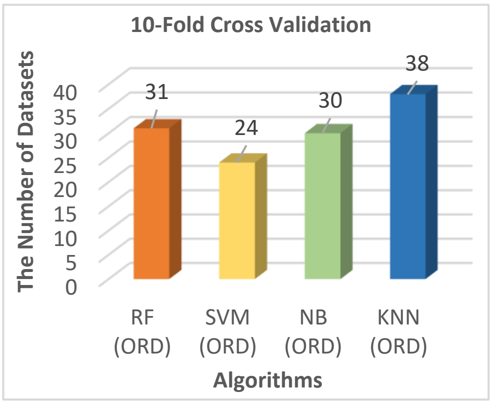

Figure 2. Comparison of algorithms in terms of the number of datasets which have better performance when 10-FCV is used

Table 3. Comparison of classification accuracies (10-fold cross validation)

Dataset Name	RF (ORD) (%)	RF (%)	SVM (ORD) (%)	SVM (%)	NB (ORD) (%)	NB (%)	KNN (ORD) (%)	KNN (%)
Ant 1.3	82.40	80.80	82.40	83.20	69.60	60.80	79.20	79.20
Ant 1.4	76.40	75.84	77.53	77.53	41.01	41.01	64.61	64.61
Ant 1.5	90.10	89.76	89.08	88.74	73.04	66.55	85.32	85.32
Ant 1.6	78.06	75.78	73.50	75.50	74.36	73.79	73.50	73.50
Ant 1.7	79.60	80.27	79.87	80.40	76.78	69.26	75.17	75.17
Arc	87.18	86.75	88.03	88.46	79.06	77.35	79.06	79.06
Berek	79.07	76.74	86.05	76.74	79.07	79.07	76.74	76.74
Camel 1.0	95.58	95.28	96.17	96.17	91.74	91.74	92.92	92.92
Camel 1.4	83.95	83.95	83.37	83.37	78.21	76.03	77.06	77.06
Camel 1.6	79.79	78.96	80.41	80.52	77.41	76.17	73.16	73.16
E-learning	90.63	90.63	90.63	90.63	82.81	84.38	85.94	85.94
Forest 0.7	82.76	82.76	79.31	79.31	72.41	82.76	79.31	79.31
Forest 0.8	93.75	93.75	93.75	93.75	93.75	93.75	87.50	87.50
Intercafe	85.19	85.19	85.19	85.19	77.78	81.48	70.37	70.37
Ivy 1.4	93.36	93.36	93.36	93.36	87.97	86.30	88.38	88.38
Ivy 2.0	88.64	89.20	88.64	88.35	80.40	79.26	83.81	83.81
Jedit 4.0	78.10	78.76	74.51	77.12	74.84	45.42	75.82	75.82
Jedit 4.1	74.68	77.56	75.96	76.28	75.64	59.61	74.04	74.04
Jedit 4.2	87.74	87.19	87.47	87.74	83.65	74.39	81.47	81.47
Jedit 4.3	97.56	97.56	97.76	97.56	93.90	93.90	96.54	96.54
Kalkulator	81.48	81.48	77.78	74.07	66.67	70.37	85.19	85.19
Log4j 1.0	76.30	75.56	77.04	76.30	78.52	73.33	68.15	68.15
Log4j 1.2	82.93	81.46	83.41	83.41	54.15	50.73	75.61	75.61
Pbeans 2	78.43	78.43	78.43	78.43	78.43	68.63	68.63	68.63
Poi 2.0	88.22	87.58	87.90	88.22	82.80	82.80	82.80	82.80
Redaktor	89.20	89.20	90.34	90.34	77.27	76.14	86.93	86.93
Serapion	80.00	80.00	82.22	84.44	71.11	75.56	73.33	73.33
Skarbonka	71.11	73.33	77.78	77.78	68.89	73.33	75.56	75.56
Synapse 1.0	87.90	87.26	89.17	89.81	75.16	75.16	83.44	83.44
Synapse 1.1	75.68	76.13	73.87	73.42	70.72	63.96	70.27	70.27
Systemdata	84.62	83.08	89.23	89.23	83.08	80.00	86.15	86.15
Termoproject	76.19	73.81	71.43	73.81	71.43	69.05	61.90	61.90
Tomcat	91.14	90.91	91.03	91.03	84.50	83.33	87.53	87.53
Xalan 2.4	83.68	84.51	84.51	84.79	78.28	78.28	79.67	79.67
Xalan 2.7	84.93	84.27	78.77	78.77	62.38	62.82	80.20	80.20
Xerces 1.2	85.00	84.55	83.86	83.86	77.50	72.95	80.68	80.68
Xerces 1.3	87.64	87.42	84.55	84.55	80.79	79.69	84.77	84.77
Xerces 1.4	79.93	79.93	59.86	60.88	45.40	47.45	73.64	73.64
AVG	**83.92**	83.66	**83.27**	83.24	**75.54**	73.07	**79.06**	79.06

Tablo 4. Comparison of classification accuracies (Leave-one-out cross validation)

Dataset Name	RF(ORD) (%)	RF (%)	SVM(ORD) (%)	SVM (%)	NB (ORD) (%)	NB (%)	KNN (ORD) (%)	KNN (%)
Ant 1.3	81.60	81.60	82.40	83.2	70.40	64.00	79.20	79.20
Ant 1.4	75.28	76.40	77.53	77.53	38.20	39.33	65.73	65.73
Ant 1.5	90.44	90.10	89.08	89.08	71.33	65.87	85.32	85.32
Ant 1.6	77.78	76.92	71.51	75.50	74.93	73.22	72.65	72.65
Ant 1.7	80.00	79.60	79.60	80.67	76.64	67.79	75.57	75.57
Arc	87.18	86.75	88.03	88.46	81.20	80.77	78.63	78.63
Berek	79.07	74.42	86.05	76.74	81.40	81.40	76.74	76.74
Camel 1.0	95.58	95.58	96.17	96.17	92.04	92.04	93.51	93.51
Camel 1.4	83.60	83.83	83.37	83.37	77.87	75.69	78.21	78.21
Camel 1.6	79.90	79.79	80.31	80.52	77.72	77.41	73.37	73.37
E-learning	89.06	89.06	90.63	90.63	84.38	81.25	87.50	87.50
Forest 0.7	79.31	86.21	79.31	79.31	68.97	82.76	79.31	79.31
Forest 0.8	93.75	93.75	93.75	93.75	93.75	93.75	87.50	87.50
Intercafe	85.19	85.19	85.19	85.19	74.07	81.48	77.78	77.78
Ivy 1.4	93.36	93.36	93.36	93.36	88.38	86.31	88.38	88.38
Ivy 2.0	87.50	88.64	88.07	88.35	80.97	79.55	85.23	85.23
Jedit 4.0	77.45	77.45	75.16	76.47	74.84	41.18	75.16	75.16
Jedit 4.1	75.64	77.24	74.68	75.96	75.64	59.62	72.12	72.12
Jedit 4.2	87.19	87.74	87.19	87.74	83.38	75.20	82.02	82.02
Jedit 4.3	97.56	97.56	97.76	97.76	93.90	94.11	96.54	96.54
Kalkulator	81.48	77.78	70.37	70.37	62.96	70.37	85.19	85.19
Log4j 1.0	77.04	76.30	78.52	75.56	79.26	73.33	66.67	66.67
Log4j 1.2	82.44	83.41	83.41	83.41	51.22	53.66	77.56	77.56
Pbeans 2	76.47	78.43	78.43	78.43	78.43	74.51	68.63	68.63
Poi 2.0	88.22	88.22	88.21	87.90	83.44	83.44	83.76	83.76
Redaktor	89.77	89.77	90.34	90.34	75.57	76.14	85.80	85.80
Serapion	82.22	77.78	84.44	84.44	71.11	73.33	73.33	73.33
Skarbonka	71.11	71.11	77.78	77.78	71.11	68.89	71.11	71.11
Synapse 1.0	87.26	87.26	89.81	89.81	74.52	75.80	82.17	82.17
Synapse 1.1	78.83	76.13	73.87	72.97	69.37	63.96	73.87	73.87
Systemdata	84.62	83.08	89.23	89.23	81.54	73.85	83.08	83.08
Termoproject	73.81	73.81	73.81	71.43	71.43	61.90	61.90	61.90
Tomcat	91.14	91.03	91.03	91.03	84.27	83.33	87.53	87.53
Xalan 2.4	84.23	83.82	84.79	84.79	78.01	78.84	80.22	80.22
Xalan 2.7	83.39	84.82	79.21	79.21	61.61	62.05	80.09	80.09
Xerces 1.2	84.55	83.64	83.86	83.86	77.50	73.86	80.23	79.77
Xerces 1.3	87.20	86.98	84.33	84.55	80.57	79.91	84.33	84.33
Xerces 1.4	79.59	80.27	60.37	60.71	45.24	47.28	75.34	75.34
AVG	**83.68**	83.55	**83.18**	83.04	**75.19**	72.82	**79.24**	79.23

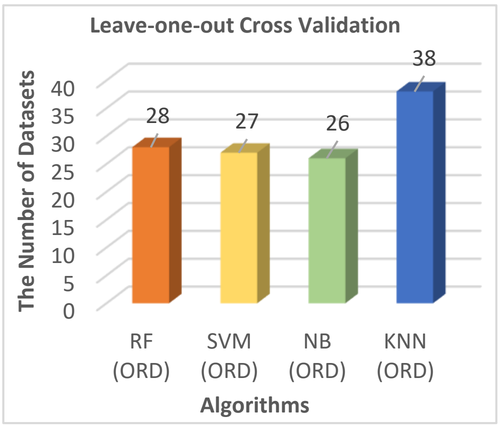

Figure 3. Comparison of algorithms in terms of the number of datasets which have better performance when LOOCV is used

The line graphs given in Figure 4 and Figure 5 show the ranks of ordinal classification algorithms for each dataset for 10-FCV and LOOCV techniques respectively. In the ranking method, each algorithm is rated according to its accuracy score on the corresponding dataset. This process is performed by assigning rank 1 to the most accurate algorithm, rank 2 to the second best and so on. In the case of tie, the average ranking is assigned to each algorithm. According to the comparative results, random forest has better performance according to others, because it generally has the lowest rank values. SVM has also good performances since its rank values are generally 1 or 2. This situation is valid for both 10-FCV and LOOCV techniques.

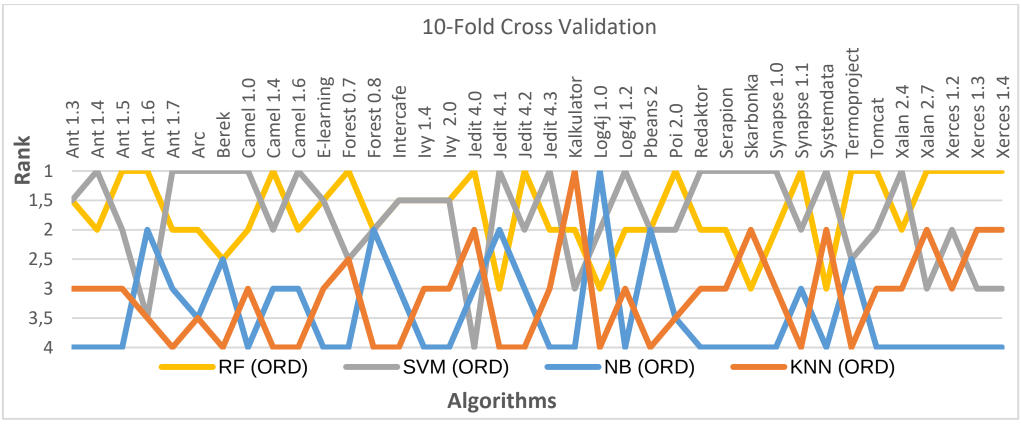

Figure 4. Rank of ordinal classification algorithms for each dataset when 10-FCV technique is used

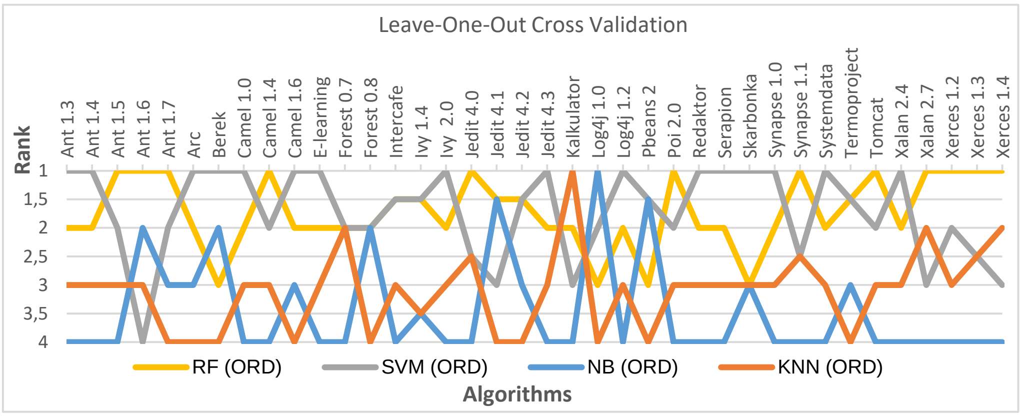

Figure 5. Rank of ordinal classification algorithms for each dataset when LOOCV technique is used

5. Conclusion and Future Work

Software bug prediction is the process of developing predictive models to improve software quality and testing efficiency. This paper presents an approach that enables standard classification algorithms to make use of ordering information for software bug prediction. The approach converts the problem into a set of binary classification problems that exploit the ordering information. The approach was implemented on 38 software bug datasets to demonstrate its competitive superiority over the traditional classification. First, the bug values of the instances were updated according to their bug tendency: bug free (=0), less buggy (=1), and more buggy (>=2). Then, the classification performances of random forest, support vector machine, Naive Bayes and k-nearest neighbor algorithms were compared with their ordinal versions. Based on the experimental studies, it is possible to say that ordinal classification methods provide better performance on software bug prediction than nominal ones.

As future work; according to the results of this study, a stand-alone application may be developed for providing bug prediction to the basic users, when historical data are fed. This tool may run several ordinal classification algorithms on given dataset without any experience on "data mining" and may give some vision about development. In addition, this tool may provide more samples to the data repository and so this may strength the inferences of this work. Another future work may be conducted by increasing the number of features in the datasets. Because the current datasets don't include any parameter about developer background, so the effect of personal characteristics doesn't taken into account. However, a more extended dataset (with developer background information) may help for better prediction.

References

[1] Burnstein, I. 2003. Practical Software Testing: A Process-Oriented Approach. 2003rd edition. Springer-Verlag New York, 710p.

[2] Georgoulas, G., Karvelis P., Gavrilis D., Stylios C. D., Nikolakopoulos G. 2017. An Ordinal Classification Approach for CTG Categorization. 39th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC), 11-15 July, 2642-2646. DOI: 10.1109/EMBC.2017.8037400

[3] Frank, E., Hall, M. 2001. A Simple Approach to Ordinal Classification. 12th European Conference on Machine Learning, Freiburg, Germany, September 5-7, 2001, Lecture Notes in Computer Science, Volume. 2167, 145-156.

[4] Kumar, L., Misra S., Rath Ku S. 2017. An Empirical Analysis of the Effectiveness of Software Metrics and Fault Prediction Model for Identifying Faulty Classes, Computer Standards & Interfaces, Volume. 53, p. 1-32. DOI: 10.1016/j.csi.2017.02.003

[5] Nucci, D. D., Palomba, F., Oliveto, R., Lucia, D. A. 2017. Dynamic Selection of Classifiers in Bug Prediction: An Adaptive Method, IEEE Transactions on Emerging Topics in Computational Intelligence, Volume. 1, Issue 3, p. 202-212. DOI: 10.1109/TETCI.2017.2699224

[6] Gupta, D. L., Saxena, K., 2017. Software Bug Prediction using Object-Oriented Metrics, Sādhanā, Volume. 42, Issue. 5, p. 655-669. DOI: 10.1007/s12046-017-0629-5

[7] Gupta, D. L., Saxena K. 2016. AUC based Software Defect Prediction for Object-Oriented Systems, International Journal of Current Engineering and Technology, Volume. 6, Issue. 5.

[8] Okutan, A., Yildiz O. T. 2016. A Novel Kernel to Predict Software Defectiveness, Journal of Systems and Software, Volume. 119, p. 109-121. DOI: 10.1016/j.jss.2016.06.006

[9] Ryu, D., Baik, J. 2016. Effective Multi-Objective Naïve Bayes Learning for Cross-Project Defect Prediction, Applied Soft Computing, Volume. 49, p. 1062-1077. DOI: 10.1016/j.asoc.2016.04.009

[10] Okutan, A., Yıldız O. T. 2014. Software Defect Prediction using Bayesian Networks, Empirical Software Engineering, Volume. 19, Issue. 1, p. 154-181. DOI: 10.1007/s10664-012-9218-8

[11] Turhan, B., Mısırlı A. T., Bener, A. 2013. Empirical Evaluation of the Effects of Mixed Project Data on Learning Defect Predictors, Information and Software Technology, Volume. 55, Issue. 6, p. 1101-1118. DOI: 10.1016/j.infsof.2012.10.003

[12] Guijo-Rubio, D., Gutiérreza, P.A. Casanova-Mateo C., Sanz-Justob, J., Salcedo-Sanzd, S., Hervás-Martíneza, C. 2018. Prediction of Low-visibility Events due to Fog using Ordinal Classification, Atmospheric Research, Volume. 214, p. 64-73. DOI: 10.1016/j.atmosres.2018.07.017

[13] Beckham, C., Pal, C. 2017. Unimodal Probability Distributions for Deep Ordinal Classification. 34th International Conference on Machine Learning, Sydney, Australia.

[14] Okyere, S., Yang, J. Aminatou, M., Tuo, G., Zhan, B. 2018. Multimodal Transport System Effect on Logistics Responsive Performance: Application of Ordinal Logistic Regression, European Transport, Issue. 68, Paper. 4.

[15] Kim, S., Kim, H., K., Namkoong, Y. 2016. Ordinal Classification of Imbalanced Data with Application in Emergency and Disaster Information Services, IEEE Intelligent Systems, Volume. 31, Issue. 5, p. 50-56. DOI: 10.1109/MIS.2016.27

[16] Fontana, F. A., Zanoni, M. 2017. Code Smell Severity Classification using Machine-Learning Techniques,

Knowledge-Based Systems, Volume. 128, p. 43-58. DOI: 10.1016/j.knosys.2017.04.014

[17] Czibula, G., Marian, Z., Czibula, I. G. 2014. Software Defect Prediction using Relational Association Rule Mining, Information Sciences, Volume. 264, p. 260-278. DOI: 10.1016/j.ins.2013.12.031

[18] Madeyski, L., Jureczko, M. 2015. Which Process Metrics Can Significantly Improve Defect Prediction Models? An empirical study, Software Quality Journal, Volume. 23, Issue. 3, p. 393-422. DOI: 10.1007/s11219-014-9241-7

[19] Prasad, M. C., Florence, L., Arya, A. 2015. A Study on Software Metrics based Software Defect Prediction using Data Mining and Machine Learning Techniques, International Journal of Database Theory and Application, Volume. 8, Issue. 3, p. 179-190. DOI: 10.14257/ijdta.2015.8.3.15

[20] Valles-Barajas, F. 2015. A Comparative Analysis between Two Techniques for the Prediction of Software Defects: Fuzzy and Statistical Linear Regression, Innovations in Systems and Software Engineering, Volume. 11, Issue. 4, p.277-287. DOI: 10.1007/s11334-015-0256-4

[21] Felix, E. A., Lee, S. P. 2017. Integrated Approach to Software Defect Prediction, IEEE Access, Volume. 5, p. 21524-21547. DOI: 10.1109/ACCESS.2017.2759180

[22] Zhang, F., Keivanloo, I., Zou Y. 2017. Data Transformation in Cross-project Defect Prediction, Empirical Software Engineering, Volume. 22, Issue. 6, p. 3186-3218. DOI: 10.1007/s10664-017-9516-2

[23] Herbold, S., Trautsch, A., Grabowski, J. 2017. A Comparative Study to Benchmark Cross-Project Defect Prediction Approaches, IEEE Transactions on Software Engineering, Volume. 44, Issue. 9, p. 811-833. DOI: 10.1109/TSE.2017.2724538

[24] Wahono, R. S., Herman, N. S. 2014. Genetic Feature Selection for Software Defect Prediction, Advanced Science Letters, Volume. 20, Issue.1, p. 239-244. DOI: 10.1166/asl.2014.5283

[25] Laradji, I. H., Alshayeb, M., Ghouti, L. 2015. Software Defect Prediction using Ensemble Learning on Selected Features, Information and Software Technology, Volume. 58, p. 388-402. DOI: 10.1016/j.infsof.2014.07.005

[26] Rana, Z. A., Mian, M. A., Shamail, S. 2015. Improving Recall of Software Defect Prediction Models using Association Mining, Knowledge-Based Systems Volume. 90, p. 1-13. DOI: 10.1016/j.knosys.2015.10.009

[27] Huda, S., Liu, K., Abdelrazek, M., Ibrahim, A., Alyahya, S., Al-Dossari, H., Ahmad, S. 2018. An Ensemble Oversampling Model for Class Imbalance Problem in Software Defect Prediction, IEEE Access, Volume. 6, p. 24184-24195. DOI: 10.1109/ACCESS.2018.2817572

[28] Wijaya, A., Wahono, R. S. 2017. Tackling Imbalanced Class in Software Defect Prediction using Two-Step Cluster-based Random Undersampling and Stacking Technique. Jurnal Teknologi, Volume. 79, Issue. 7-2, p. 45-50.

[29] Tomar, D., Agarwal, S. 2016. Prediction of Defective Software Modules using Class Imbalance Learning, Applied Computational Intelligence and Soft Computing, Volume. 2016. DOI: 10.1155/2016/7658207

[30] Rodriguez, D., Herraiz, I., Harrison, R., Dolado, J., Riquelme, J. C. 2014. Preliminary Comparison of Techniques for Dealing with Imbalance in Software Defect Prediction. Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering, 13-14 May, London, England, United Kingdom. DOI: 10.1145/2601248.2601294

[31] Wang, S., Yao, X. 2013. Using Class Imbalance Learning for Software Defect Prediction, IEEE Transactions on Reliability, Volume. 62, Issue. 2, p. 434-443. DOI: 10.1109/TR.2013.2259203

[32] Weka - Data Mining Software in Java, https://www.cs.waikato.ac.nz/ml/weka/. (Accessed: 21.11. 2018).

[33] Tera-Promise Data, https://github.com/klainfo/DefectData/tree/master/inst/extdata/terapromise/ck. (Accessed: 20.11.2018).

[34] PROMISE Software Engineering Repository http://promise.site.uottawa.ca/SERepository/ (Accessed: 20.11.2018).

[35] Li, J., He, P., Zhu, J., Lyu, M. R. 2017. Software Defect Prediction via Convolutional Neural Network. IEEE International Conference on Software Quality, Reliability and Security (QRS), 25-29 July, Praque, Czech Republic. DOI: 10.1109/QRS.2017.42

[36] Bowes, D., Hall, T., Petrić, J. 2018. Software Defect Prediction: Do Different Classifiers Find the Same Defects?, Software Quality Journal, Volume. 26, Issue. 2, p. 525-552. DOI: 10.1007/s11219-016-9353-3