STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LX**, Number 2, 2015

SOFTWARE DEFECT DETECTION USING SELF-ORGANIZING MAPS

ZSUZSANNA MARIAN, ISTVÁN GERGELY CZIBULA, GABRIELA CZIBULA AND SERGIU SOTOC

Abstract. This paper addresses the problem of software *defect detection*, an important problem which helps to improve the software systems' maintainability and evolution. In order to detect defective entities within a software system, a self-organizing feature map is proposed. The trained map will be able to identify, using unsupervised learning, if a software module is defective or not. We experimentally evaluate our approach on three open-source case studies, also providing a comparison with similar existing approaches. The obtained results emphasize the effectiveness of using self-organizing maps for software defect detection and confirm the potential of our proposal.

1. Introduction

In order to increase the efficiency of quality assurance, *defect detection* tries to identify those modules of a software where errors are present. In many cases there is no time to thoroughly test each module of the software system, and in these cases defect detection methods can help by suggesting which modules should be focused on during testing.

We are proposing in this paper an unsupervised machine learning method based on self-organizing maps for detecting defective entities within software systems. The self-organizing map architecture was previously applied in the literature for defect detection, but using a kind of hybrid approach, where different threshold values for some software metrics were also used [1]. To our knowledge, there is no approach in the search-based software engineering literature similar to ours. The unsupervised model introduced in this paper proved to outperform most of the similar existing approaches, considering the datasets used for evaluation.

Received by the editors: June 3, 2015.

2010 *Mathematics Subject Classification.* 68T05, 62H30.

1998 *CR Categories and Descriptors.* I.2.6[**Computing Methodologies**]: Artificial Intelligence – *Learning*; I.5.3 [**Computing Methodologies**]: Pattern Recognition – *Clustering*.

The rest of the paper is structured as follows. Section 2 presents the fundamentals of self-organizing maps as well as existing similar approaches for software defect detection. Our proposal for identifying software defects using self-organizing feature maps is introduced in Section 3. Section 4 provides an experimental evaluation of our approach, while an analysis of the obtained results and comparison with existing similar work is given in Section 5. Section 6 contains some conclusions of our paper and indicates directions for further improvement.

2. Background

In this section we aim at presenting the main characteristics of self organizing maps as well as similar approaches for software defect detection.

2.1. **Self-organizing maps.**

A *self-organizing map* (SOM) [15] is a type of artificial neural network that is trained using unsupervised learning to produce a low-dimensional (usually two-dimensional) representation of the training samples, called a *map* [5]. Self-organizing maps use a neighborhood function to preserve the topological relationships in the input space and are related to the category of *competitive learning* networks. Self-organizing maps are considered in the neural networks literature as the most innovative form of unsupervised learning.

The SOM provides a topology preserving mapping from the multi-dimensional input space to the map neurons (units). Each neuron from the input layer of a SOM is connected to each neuron from the output layer and each connection has an associated weight. The topology preservation property means that a SOM groups similar input instances on neurons that are close on the SOM [9]. The map is usually trained using the Kohonen algorithm [15].

The trained self-organizing map is able to provide clusters of similar data items [12]. This particular characteristic of SOMs makes them appropriate for data mining tasks that involve classification and clustering of data items [12]. The SOM can be used as an effective tool for clustering as well as a tool for visualizing high-dimensional data.

2.2. **Literature review.**

A review on unsupervised learning-based approaches existing in the defect prediction literature will be provided in the following.

Abaei et al. proposed in [1] a fault prediction method by utilizing self-organizing maps and thresholds. Two experiments are conducted in this paper: the first one considers the removal of the modules' labels and re-computing them afterwards by taking threshold values into account for some selected attributes (the ones for which threshold values are known). In the second experiment, a SOM is used for both clustering and evaluating the input data.

Threshold values are used as well for labeling the units from the trained SOM. For this second experiment they report a good Overall Error, and in most cases their proposed solution improves classification of unlabeled program modules in terms of FPR (False Positive Rate) and FNR (False Negative Rate). One drawback to their approach is that they do not obtain good results when the dataset is very small.

Another approach is presented in [2] that predicts software fault using a Quad Tree-based K-Means algorithm. The difference to the original K-Means algorithm is that the cluster centers are found by using Quad Trees. They evaluate their approach on various datasets, and compute the FPR, FNR and Overall Error for them. These values indicate that their approach is slightly better than other cluster center initialization techniques and they achieve slightly better results from fewer number of iterations.

The method presented in [18] uses the K-Means clustering algorithm as well, but it uses Hyper Quad Trees for determining the cluster centers. They present that Hyper Quad Trees are more efficient than simple Quad Trees because they produce more accurate centroids. After the K-Means algorithm is run, a threshold value is used to determine which cluster represents the defective and which represents the non-defective entities. The results for some public datasets confirm obtaining better outcomes in terms of FPR and Overall Error when comparing this approach to simple Quad Tree approach.

Tosun et al. used in [17] network measures to identify defective modules in software systems. Their approach uses the Naive Bayes classifier, together with a Call Graph Based Ranking (CGBR) framework. The experimental evaluation was performed on both small and large datasets and for three cases: complexity metrics only, network metrics only and a combination between them. The results show a great performance of applying network metrics for large datasets, but they do not provide significant improvement for small projects.

A clustering-based approach is presented in [3], where the Xmeans algorithm is used, an algorithm which is similar to K-means, but it can automatically determine the optimal number of clusters. The authors use the implementation of this algorithm from WEKA [7], and when the clusters are created, software metric threshold values are applied to the mean vector of each cluster in order to decide whether it represents the defective or the non-defective entities. They claim that this method proved better results than a simple threshold-based approach, fuzzy c-means and k-means.

Another unsupervised software fault prediction model is given by Park and Hong in [14] where clustering algorithms that determine the number of clusters automatically are used. They have a pre-processing step, where attribute selection is performed, using the CfsSubsetEval method from WEKA.

Results achieved with the Xmeans and EM models from WEKA (which can automatically determined the optimal number of clusters) were compared to other results produced with Xmeans (in [3]) and Quad Tree based K-Means algorithm. They conclude that both Xmeans and EM have good results if attribute selection is not performed, results that are better than the existing ones in most of the considered cases.

3. Methodology

In this section we introduce our unsupervised neural network model for defect identification in software systems.

The main idea of this approach is to represent an entity (class, module, method, function) of a software system as a multidimensional vector, whose elements are the values of different software metrics applied to the given entity. We consider that a software system S is a set of components (called *entities*) $\mathcal{S} = \{s_1, s_2, ..., s_n\}$. We are considering a feature set of software metrics $\mathcal{SM} = \{sm_1, sm_2, ..., sm_k\}$ and thus each entity $s_i \in \mathcal{S}$ from the software system can be represented as a k-dimensional vector, having as components the values of the software metrics from $\mathcal{SM}$, $s_i = (s_{i1}, s_{i2}, \ldots, s_{ik})$ (s_{ij} represents the value of the software metric sm_j applied to the software entity s_i).

For each software entity, the label of the instance (defect or non-defect) is known. We mention that the labels will be used only in the pre-processing step and for evaluating the performance of the model.

3.1. **Data pre-processing.**

The first step before applying the SOM approach is the *data pre-processing* step. During this step, the input data is scaled to [0,1] using the *Min-Max* normalization method, and then a feature selection step will be applied. Details about the feature selection step will be given in the experimental part of the paper (Section 4). After applying the feature selection step, m software metrics (features) are selected to be further used for building the SOM.

Regarding the normalization method, we have to mention that in our approach the *minimum* and *maximum* values for the software metrics (features) from the training data are used for the *Min-Max* normalization step. We have focused in this paper only on the unsupervised scenario of grouping the existing entities from a software system into *defective* or *non-defective.* In a supervised learning scenario, in which new testing data is used, it is not useful to use for normalization the *minimum* and *maximum* values for the features from the training data. Instead, it would be a good idea to use, for each software metric, the *minimum* and *maximum* values for that software metric. Further extensions of our approach will investigate this situation.

3.2. **The SOM model.** Before designing the SOM model, the input dataset is pre-processed. For the training step of the SOM, a distance function between the input instances is required. We are considering the *distance* between the high-dimensional representation of two software entities s_i and s_j as the *Euclidean Distance* between their corresponding vectors of software metrics values. We have chosen the *Euclidean distance* because it is the most often used distance measure for SOM-based approaches and because, intuitively, considering the m-dimensional instances (preprocessed as mentioned in Section 3.1), the *Euclidean distance* will assign low distances to similar entities that are very likely to have the same output class (defect or not). Nevertheless, in the future we intend to extend our approach to use other distance measures as well.

The set of pre-processed software entities from the dataset $\mathcal{S}$ are grouped into clusters using a SOM. For the self-organizing map, the *torus* topology is used (Figure 1). In geometry, a torus is a surface of revolution generated by revolving a circle in the three dimensional space about an axis coplanar with the circle. It is shown in the literature that this topology provides better neighborhood than the conventional one [11].

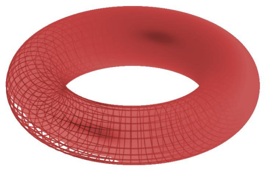

FIGURE 1. A torus

The goal of this step is to obtain two clusters corresponding to the two classes of instances: *defects* and *non-defects*. For grouping the software entities the following steps are performed.

- **Map Construction**. For a given number of *epochs* (training episodes), perform the following. Each m-dimensional training instance (software entity) is fed to the map. For each instance s_i the following steps are performed:
  (1) **Matching**. The neuron having its weight vector closest (considering the *Euclidean distance*) to instance s_i is declared the "winning" neuron. This is a competition phase, in which the output units from the map compete to match the input instance.
  (2) **Updating**. After the "winning neuron" was identified, the connection weights of the winning unit and its neighbors are updated,

such that are moved in the direction of the input instance by a factor determined by a learning rate.

- **Visualization**. After the training phase (the steps described above) was completed, in order to visualize the obtained map, the U-Matrix method [8] is used. The U-Matrix value of a particular node from the map is computed as the average Euclidean distance between the node and its closest 4 or 8 neighbors. These distances can be then be viewed as heights giving a U-Matrix landscape. The U-Matrix may be interpreted as follows [8]: high places on the U-Matrix encode data that are dissimilar while the data falling in the same valleys represent input instances that are similar. Thus, instances within the same valley can be grouped together to represent a cluster. Each cluster visualized on the map identifies a class of instances.

Once the map was built, it may also be used in a supervised learning scenario for classifying a new software entity. First, the "winning neuron" corresponding to this instance is determined (as indicated at the **Matching** step above). The cluster (class) to which the winning neuron belongs will indicate the class membership of the given entity.

3.2.1. *Testing.* For evaluating the performance of the SOM model, we are using several evaluation measures from the supervised classification literature. Since the training instances were labeled, the labels are used to compute the confusion matrix for the two possible outcomes (*non-defect* and *defect*). Considering the *defective* class as the *positive* one and the *non-defective* class as the *negative* one, the confusion matrix [16] for the defect detection task consists of: the number of *true positives* (TP), *false positives* (FP), *true negatives* (TN) and *false negatives* (FN).

Considering the values computed from the confusion matrix, the following evaluation measures will be used in this paper:

(1) False Positive Rate (FPR), computed as $\frac{FP}{FP+TN}$.
(2) False Negative Rate (FNR), computed as $\frac{FN}{FN+TP}$.
(3) Overall Error (OE), computed as $\frac{FN+FP}{FN+FP+TN+TP}$.

We have decided to use these measures, because they are used in papers presenting similar approaches, so a direct comparison of the results is possible. But the datasets used for the experiments are imbalanced, so we have decided to compute the value of a fourth performance measure as well: the *Area Under the ROC Curve (AUC)* [6]. The *ROC* curve is a two-dimensional plot of *sensitivity* vs. *(1-specificity)*, which in our case contains one single point, linked to the $(0, 0)$ and $(1, 1)$ points.

4. Experimental evaluation

In this section we provide an experimental evaluation of the SOM model (described in Section 3) on three case studies which were conducted on open source datasets. We mention that we have used our own implementation for SOM, without using any third party libraries.

4.1. **Datasets.** We have used three openly available datasets for the experimental evaluation of our model, called *Ar3*, *Ar4* and *Ar5*, which can be downloaded from [4]. All three datasets come from a Turkish white-goods manufacturer embedded software implemented in C. They all contain the value of 29 different McCabe and Halstead software metrics, computed for the functions and methods from the software systems, and one class label, denoting whether the entity is defective or not. The *Ar3* dataset contains metric values for 63 entities, out of which 8 are defective. The *Ar4* dataset contains 107 entities, out of which 20 defective, while the *Ar5* dataset has 36 entities, out of which 8 are defective.

For the SOM used in the experiments, the following parameter setting was used: the *number of training epochs* was set to 100000, the *learning coefficient* was set to 0.7, the *radius* was computed as half of the maximum distance between the neurons and the neighborhood function. We have tried out different parameter settings and we have achieved the best results with these values. However, we will perform in the future a thorough study to investigate the effect of different parameter settings.

4.2. **Data pre-processing.** In order to analyze the importance of the features, we are using the *information gain* measure. The *information gain* (IG) of a feature expresses the expected reduction in entropy determined by partitioning the instances according to the considered feature [13]. More exactly, the IG measure indicates the relevance of a feature in the defect classification task. Since the software metrics values (features values) are real numbers, in order to compute the information gain of the attributes we first discretize their values by dividing their interval of variation into ten sub-intervals.

For a better data analysis, we have computed the information gain of the features from the dataset obtained using all three datasets (*Ar3*, *Ar4* and *Ar5*) together. The information gain values for the features are shown in Figure 2.

Starting from the IG values of the software metrics, we have chosen a threshold value τ and considered only the attributes whose IG was higher than this threshold. Out of these attributes, we selected those that measure different characteristics of the software system. For the threshold τ we have selected the value 0.163, because we have achieved the best results with this value. Out of the 18 software metrics whose value was higher than τ, we have selected

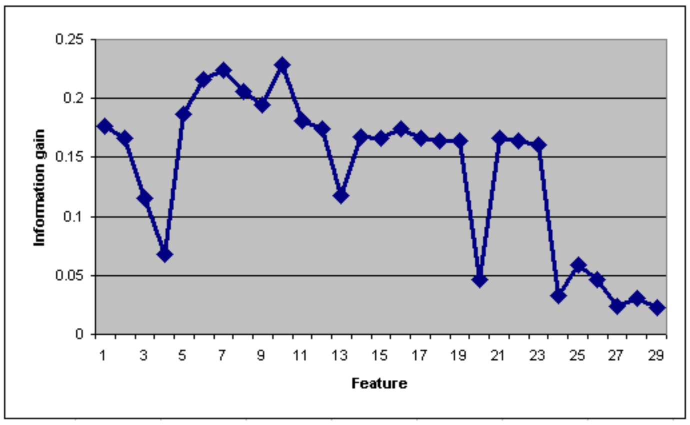

FIGURE 2. Information gain for the features.

the following 9 metrics: *halstead_vocabulary*, *total_operands*, *total_operators*, *executable_loc*, *halstead_length*, *total_loc*, *condition_count*, *branch_count*, *decision_count*. These selected attributes were used in the experimental evaluation on all three considered datasets. We mention that a different, possibly automatic, attribute selection method can also be implemented considering the IG values and will be further investigated.

4.3. **Results.** We are presenting in this section the results we have obtained by applying the SOM model on the *Ar3*, *Ar4* and *Ar5* datasets. For each dataset considered for evaluation, the experiments are conducted as follows. First, the data pre-processing step is applied and then the methodology indicated in Section 3 is used for an unsupervised construction of a torus SOM. The U-Matrix corresponding to the trained SOM will be visualized (the red labels on the U-Matrix represent the defective entities and the yellow labels represent the non-defective ones). Then, the evaluation measures presented in Section 3.2.1 will be computed for evaluating the performance of the obtained results.

4.3.1. *The Ar3 dataset.* A torus SOM, consisting of 150x8 nodes, was trained on the set of software entities from the *Ar3* dataset. Figure 3 depicts the U-Matrix visualization of the trained SOM.

Visualizing the U-Matrix for the resulting map, we have identified the two clusters, representing the defective and non-defective entities. The cluster with the defective entities contains 6 defective entities and 1 non-defective entity, thus we have 1 FP entity and 2 FN ones. All the other entities are placed

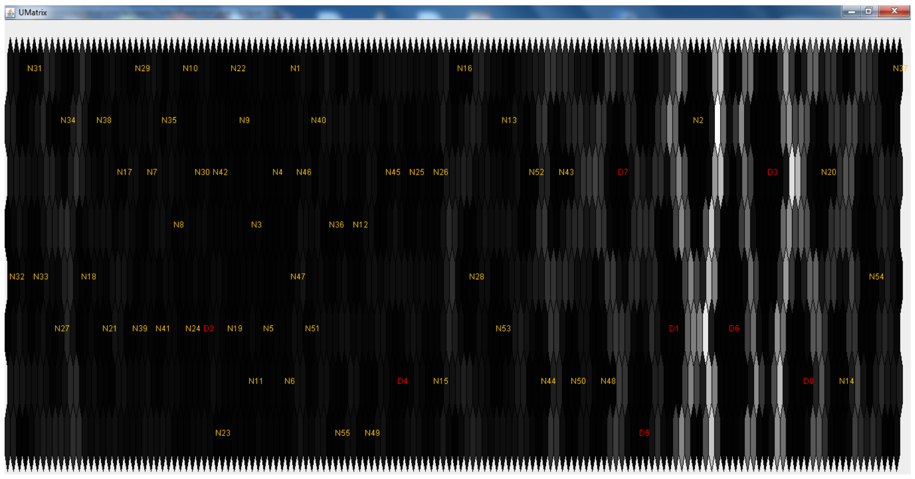

FIGURE 3. U-Matrix for the *Ar3* dataset.

in the correct cluster. The values of the performance measures from Section 3.2.1 are presented in the first three cells of the first row of Table 1.

4.3.2. *The Ar4 dataset.* A torus SOM, consisting of 150x8 nodes, was trained on the set of software entities from the *Ar4* dataset. The U-Matrix visualization of the obtained SOM is illustrated in Figure 4.

Visualizing the U-matrix for the resulting map, we have identified the two clusters which represent the defective and non-defective entities. The cluster with the defective entities contains 10 defective entities and 2 non-defective entities. Consequently we have 10 FN entities and 2 FP ones. The values of the performance measures are presented in the middle three cells of the first row of Table 1.

4.3.3. *The Ar5 dataset.* A torus SOM, consisting of 150x8 notes, was trained on the set of software entities from the *Ar5* dataset. The U-Matrix visualization of the trained SOM is presented in Figure 5.

From Figure 5 we can observe that the obtained SOM indicates a good topological mapping of the input instances, and also identifies subclasses within the defective and non-defective classes. Most of the defective entities are grouped together (there is only one FN), but there is also one non-defective entity in this cluster, so we have 1 FP as well. The values of the performance measures for this dataset are presented in the last three cells of the first row of Table 1.

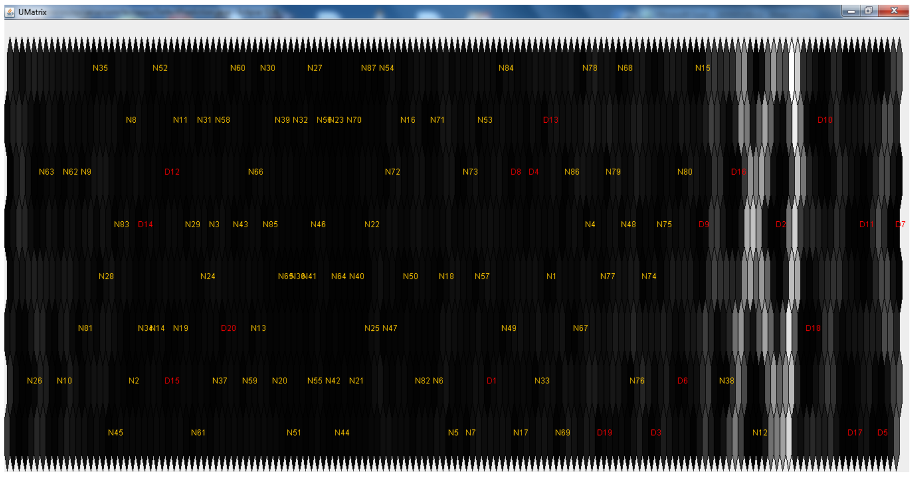
UMatrix
N35 N52 N60 N30 N27 N87 N54 N84 N78 N68 N15
N8 N11 N31 N58 N39 N32 N56 N23 N70 N16 N71 N53 D13 D10
N63 N62 N9 D12 N66 N72 N73 D8 D4 N86 N79 N80 D16
N83 D14 N29 N3 N43 N85 N46 N22 N4 N48 N75 D9 D2 D11 D7
N28 N24 N65 N36 N41 N64 N40 N50 N18 N57 N1 N77 N74
N81 N34 N14 N19 D20 N13 N25 N47 N49 N67 D18
N26 N10 N2 D15 N37 N59 N20 N55 N42 N21 N82 N6 D1 N33 N76 D6 N38
N45 N61 N51 N44 N5 N7 N17 N69 D19 D3 N12 D17 D5

FIGURE 4. U-Matrix for the *Ar4* dataset.

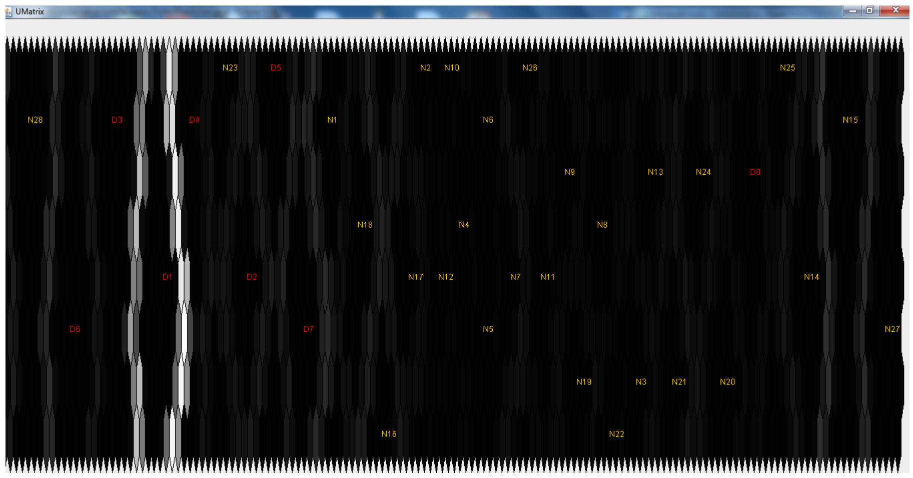

FIGURE 5. U-Matrix for the *Ar5* dataset.

5. Discussion and comparison to related work

An analysis of the approach we have introduced in Section 3 for detecting the defective entities from software systems will be provided in the following. A discussion on the obtained experimental results, as well as a comparison of them with similar approaches from the literature is conducted.

As presented in the previous section, our approach was capable of separating the defective and non-defective entities in two clusters, obtaining a good topological mapping of the input instances. Even if the separation was not perfect, for all three datasets we had both false positive and false negative entities. A major advantage of the self-organizing map is that it does not require supervision and no assumption about the distribution of the input data is made. Thus, it may find unexpected hidden structures from the data being investigated. Moreover, as seen from our experiments (Section 4), it is interesting that the SOM is able to detect, within the defective/non-defective class, subclasses of instances. This would be very useful, from a data mining perspective, since it may provide useful knowledge for the software developers.

As the accuracy of the trained SOMs depends on the choice of some parameters (number of training epochs, learning coefficient, neighborhood function), we have to measure it. One method for evaluating the quality of the resulting map is to calculate the *average quantization error* over the input samples, defined as the Euclidean norm of the difference between the input vector and the best-matching model [10]. Figures 6, 7 and 8 give a graphical representation of the average quantization error during the training steps, for each case study considered for evaluation. It can be easily seen that while the error fluctuates at the beginning of the training phase, it decreases during it, and after the training is completed, a very small average quantization error of the trained maps is obtained (1.6×10^{-6} for *Ar3*, 1.7×10^{-4} for *Ar4* and 6.7×10^{-13} for *Ar5*), which shows the accuracy of the trained maps.

Considering the subclasses identified within the clusters for the defective and non-defective entities as further work we propose to analyze these subclasses to identify the characteristics of the software entities placed in them.

Table 1 presents the values of the FPR, FNR and OE performance measures computed for the results of our approach, but it also contains values reported in the literature for some existing approaches, presented in Section 2.2. The Hyper Quad Tree-based approach presented in [18] does not report FNR values, this is marked with "NR" in the table.

From Table 1 we can see, that even if our approach does not provide the best results in each case, it has better results than most of the approaches. Out of 51 cases in total, our algorithm has a better or equal value for a performance measure in 43 cases, which represents **84.3%** of the cases.

The first line of Table 2 presents the value of the AUC measure computed for our approach. As presented in Section 3.2.1, for computing the value of the AUC measure we need to compute the *sensitivity* and *specificity* of the classification. Since *sensitivity* is equal to $1 - FNR$ and $(1 - specificity)$ is equal to *FPR*, we computed the value of the AUC measure for those approaches from the literature which report both of these values. They are presented in

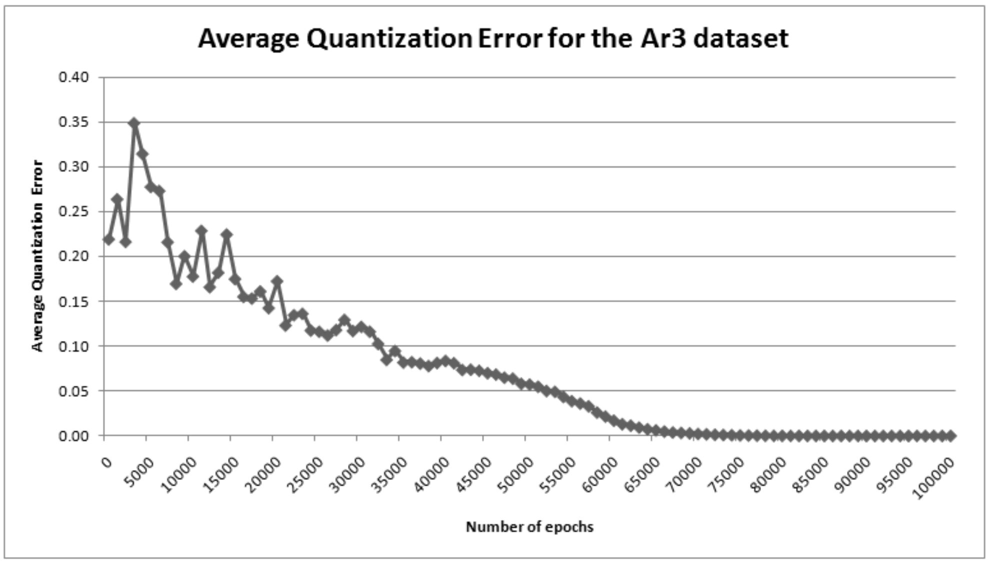

FIGURE 6. Average Quantization Error for the *Ar3* dataset.

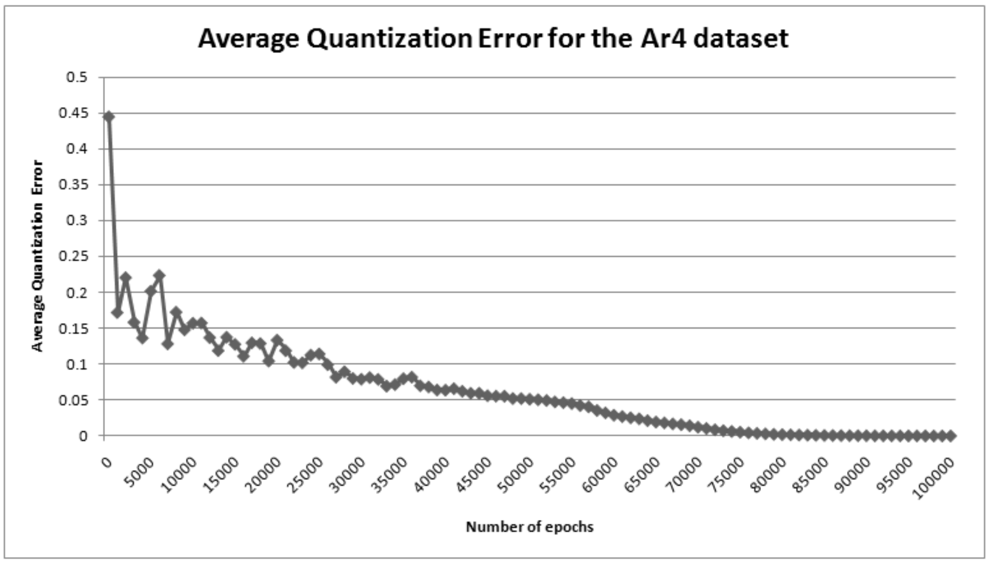

FIGURE 7. Average Quantization Error for the *Ar4* dataset.

Table 2 as well, and for each dataset the best value is marked with bold. Our approach has the highest AUC value for the *Ar5* dataset, the second highest

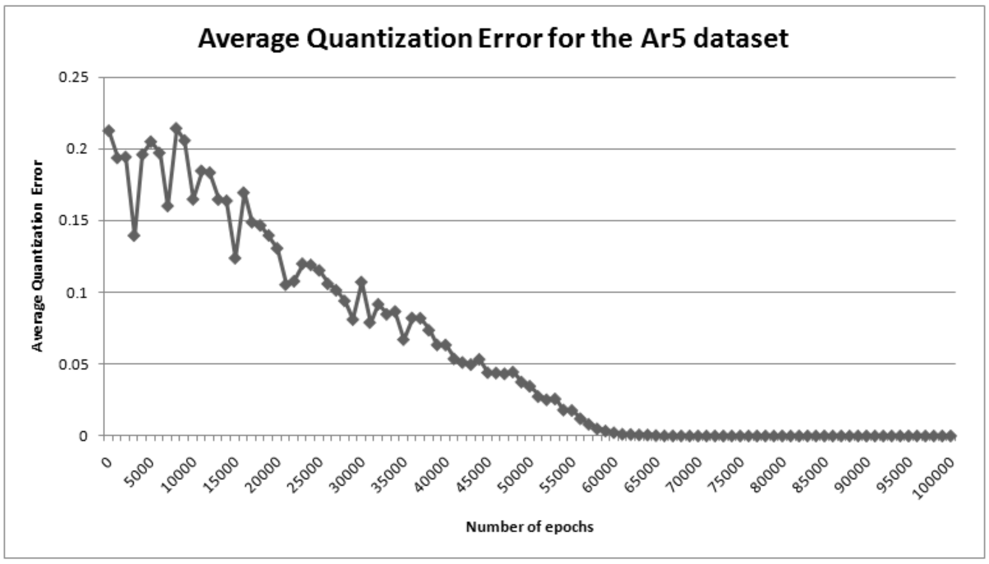

Figure 8. Average Quantization Error for the *Ar5* dataset.

	Ar3			**Ar4**			**Ar5**		
Approach	**FPR**	**FNR**	**OE**	**FPR**	**FNR**	**OE**	**FPR**	**FNR**	**OE**
Our SOM	0.0182	0.25	0.0476	0.0230	0.5	0.1121	0.0357	0.125	0.0556
SOM and threshold [1]	0	0.25	0.0556	0.1034	0	0.0938	0.0714	0.25	0.1111
K-means - QT [2]	0.3454	0.25	0.3333	0.0459	0.45	0.1214	0.1428	0.125	0.1388
K-means - Hyper QT [18]	0.0263	NR	0.0263	0.1875	NR	0.1846	0.0246	NR	0.0246
XMeans [3]	0.3455	0.25	0.3333	0.4483	0.05	0.3738	0.1429	0.125	0.1389
XMeans [14]	0.0727	0.25	0.0952	0.023	0.6	0.1308	0.149	0.125	0.1389
EM [14]	0.1091	0.25	0.127	0.023	0.6	0.1308	0.149	0.25	0.1667

Table 1. Comparison of the performance of our method to existing approaches.

value for the *Ar3* dataset and the fourth highest value for the *Ar4* dataset. Interestingly, for the *Ar3* and *Ar4* datasets the best value is achieved by the other SOM-based approach presented in the literature, suggesting that SOMs are indeed suitable for this problem.

6. Conclusions and future work

We have introduced in this paper a self-organizing feature map which may be used for an unsupervised detection of software defects. The experimental results obtained on three open-source datasets reveal a good performance of

Approach	**Ar3**	**Ar4**	**Ar5**
Our SOM	0.866	0.739	**0.92**
SOM and threshold [1]	**0.875**	**0.948**	0.839
K-means - QT [2]	0.702	0.752	0.866
XMeans [3]	0.702	0.751	0.866
XMeans [14]	0.839	0.689	0.863
EM [14]	0.820	0.689	0.801

Table 2. Comparison of AUC values.

the proposed approach, it provides better results than many of the existing approaches report in the literature.

Future work will be done in order to extend the evaluation of the proposed machine learning based model on other open source case studies and real software systems. We will also investigate the applicability of fuzzy [19] self-organizing maps for software defect detection, as well as to further consider techniques for data pre-processing and feature selection.

Acknowledgments

This work was supported by a grant of the Romanian National Authority for Scientific Research, CNCS–UEFISCDI, project number PN-II-RU-TE-2014-4-0082.

References

[1] G. Abaei, Z. Rezaei, and A. Selamat. Fault prediction by utilizing self-organizing map and threshold. In *2013 IEEE International Conference on Control System, Computing and Engineering (ICCSCE)*, pages 465–470, Nov 2013.

[2] P.S. Bishnu and V. Bhattacherjee. Software fault prediction using quad tree-based k-means clustering algorithm. *IEEE Transactions on Knowledge and Data Engineering*, 24(6):1146–1150, June 2012.

[3] C. Catal, U. Sevim, and B. Diri. Software fault prediction of unlabeled program modules. In *Proceedings of the World Congress on Engineering (WCE)*, pages 212–217, Dec 2009.

[4] Tera-promise repository. http://openscience.us/repo/.

[5] N. Elfelly, J.-Y. Dieulot, and P. Borne. A neural approach of multimodel representation of complex processes. *International Journal of Computers, Communications & Control*, III(2):149–160, 2008.

[6] T. Fawcett. An introduction to ROC analysis *Pattern Recogn. Lett.*, 27(8):861–874, 2006.

[7] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The weka data mining software: An update. *SIGKDD Explorations*, 11(1), 2009.

[8] S. Kaski and T. Kohonen. Exploratory data analysis by the self-organizing map: Structures of welfare and poverty in the world. In *Neural Networks in Financial Engineering.*

Proceedings of the Third International Conference on Neural Networks in the Capital Markets, pages 498–507. World Scientific, 1996.

[9] Andreas Khler, Matthias Ohrnberger, and Frank Scherbaum. Unsupervised feature selection and general pattern discovery using self-organizing maps for gaining insights into the nature of seismic wavefields. *Computers & Geosciences*, 35(9):1757 – 1767, 2009.

[10] Teuvo Kohonen, Ilari T. Nieminen, and Timo Honkela. On the Quantization Error in SOM vs. VQ: A Critical and Systematic Study. In *Advances in Self-Organizing Maps, 7th International Workshop, WSOM*, pages 133–144. St. Augustine, FL, USA, 2009.

[11] Peter K. Kihato, Heizo Tokutaka, Masaaki Ohkita, Kikuo Fujimura, Kazuhiko Kotani, Yoichi Kurozawa, and Yoshio Maniwa. Spherical and torus som approaches to metabolic syndrome evaluation. In Masumi Ishikawa, Kenji Doya, Hiroyuki Miyamoto, and Takeshi Yamakawa, editors, *ICONIP (2)*, volume 4985 of *Lecture Notes in Computer Science*, pages 274–284. Springer, 2007.

[12] J. Lampinen and E. Oja. Clustering properties of hierarchical self-organizing maps. *Journal of Mathematical Imaging and Vision*, 2(3):261–272, 1992.

[13] Thomas M. Mitchell. *Machine learning.* McGraw-Hill, Inc. New York, USA, 1997.

[14] Mikyeong Park and Euyseok Hong. Software fault prediction model using clustering algorithms determining the number of clusters automatically. *International Journal of Software Engineering and Its Applications*, 8(7):199–205, 2014.

[15] Panu Somervuo and Teuvo Kohonen. Self-organizing maps and learning vector quantization for feature sequences. *Neural Processing Letters*, 10:151–159, 1999.

[16] Stephen V. Stehman. Selecting and interpreting measures of thematic classification accuracy. *Remote Sensing of Environment*, 62(1):77 – 89, 1997.

[17] Ayse Tosun, Burak Turhan, and Ayse Basar Bener. Validation of network measures as indicators of defective modules in software systems. In *Proceedings of the 5th International Workshop on Predictive Models in Software Engineering, PROMISE 2009, Vancouver, BC, Canada, May 18-19, 2009*, pages 5–14, 2009.

[18] Swati Varade and Madhav Ingle. Hyper-quad-tree based k-means clustering algorithm for fault prediction. *International Journal of Computer Applications*, 76(5):6–10, August 2013.

[19] Lotfi A. Zadeh. A summary and update of "fuzzy logic". In *2010 IEEE International Conference on Granular Computing, GrC 2010, San Jose, California, USA, 14-16 August 2010*, pages 42–44, 2010.

Department of Computer Science,, Faculty of Mathematics and Computer Science,, Babeş-Bolyai University, Kogălniceanu 1, Cluj-Napoca, 400084, Romania.

E-mail address: {marianzsu, istvanc, gabis}@cs.ubbcluj.ro

E-mail address: ssic0977@scs.ubbcluj.ro