A Novel Approach Using Fuzzy Self-Organizing Maps for Detecting Software Faults

Istvan-Gergely CZIBULA, Gabriela CZIBULA,
Zsuzsanna MARIAN, Vlad-Sebastian IONESCU

Babeş-Bolyai University,
1, M. Kogălniceanu Street, Cluj-Napoca, 400084, Romania,
{istvanc, gabis, marianzsu, ivlad}@cs.ubbcluj.ro

Abstract: As software projects become more complex, there is an increased focus on their analysis and testing. Detecting software faults is a problem of major importance for improving the quality of the software development related processes and the efficiency of the software testing process. In order to detect faults in existing software systems, we introduce in this paper a novel approach, based on *fuzzy self-organizing feature maps*. A fuzzy map will be trained, using unsupervised learning, to provide a two-dimensional representation of the *faulty* and *non-faulty* entities from a software system and it will be able to identify if a software module is or not a defective one. Five open-source case studies are used for the experimental evaluation of our approach. The obtained results are better than most of the results already reported in the literature for the considered datasets and emphasize that a *fuzzy* self-organizing map is more efficient than a crisp one for the case studies used for evaluation.

Keywords: Software defect detection, Machine learning, Self-organizing map, Fuzzy clustering.

1. Introduction

Software defect detection represents the activity of identifying software modules which contain errors and it contributes to increasing the effectiveness of the quality assurance process. Fault detection methods would be helpful for suggesting to the developers which software modules should be focused on during testing, particularly when, from lack of time, the modules cannot be systematically tested.

Code review is frequently used in agile development processes for maintaining the quality of the software. During code review, an experienced programmer reviews the source code in order to identify vulnerabilities, security problems and other problems overlooked by the initial implementer. Since code review is a time consuming and costly activity, *software defect detection* can be used to guide the code review process by identifying parts of the source code where the code review is most likely to identify problems.

Software defect detection is intensively investigated in the literature and an active area in the software engineering field, as shown by a systematic review published in 2011, which collected 208 fault prediction studies published between 2000 and 2010 [12]. Detecting software faults is a complex and difficult task, mainly for large scale software projects. In the literature there are a lot of machine learning-based approaches for predicting faulty software entities. From a supervised learning perspective, defect prediction is a hard problem, particularly because of the imbalanced nature of the training data (the number of *non-defective* training instances is much higher than the number of *defective* ones). Moreover, it is not a trivial problem to identify a set of software metrics that would be relevant for discriminating between *faulty* and *non-faulty* modules.

Even if there are a lot of methods already developed for detecting software defects, researchers are still focusing on improving the performance of existing classifiers. We are introducing in this paper an unsupervised machine learning method based on *fuzzy self-organizing maps* for detecting faults within software systems. To the best of our knowledge, our approach is novel in the search-based software engineering literature and proved to outperform most of the existing similar approaches, considering the case studies we have used for evaluation.

The rest of the paper is structured as follows. Section 2 presents the importance of the problem approached in this paper and gives a motivation for our work. Several existing approaches similar to ours are given in Section 3. Our proposal is introduced in Section 4. Section 5 provides the experimental results which were obtained on several open-source case studies and Section 6 analyses the experimental results and compares them to

existing similar work from the literature. The conclusions of the paper and directions for future research are outlined in Section 7.

2. Problem Relevance. Motivation

Since software systems are continuously growing in size and complexity, predicting the reliability of software has a fundamental role in the software development process [25]. Clark and Zubrow consider in [7] that there are three main reasons for which the analysis and prediction of software defects is essential. The first one is to help the project manager to measure the progress of a software project and to plan activities for defect detection. The second reason is to contribute to the process management, by evaluating the quality of the software product and measuring the process performance [7]. Finally, information about software faults, their location within the software and the distribution of defects may contribute to improving the efficiency of the testing process and the quality of the next version of the software.

Many of the machine learning-based defect predictors existing in the literature have been built using historical data collected by mining software repositories [13]. Unfortunately, there are studies carried out in the defect prediction literature (like [3]) which have revealed that defect data extracted from change logs and bug reports may contain noise [13]. Other machine learning-based software defect predictors use openly available datasets, like the NASA datasets, where only the software metric values computed for the modules of the software system are available, but not the source code. Unfortunately, there can be noise in these datasets as well, as shown by [11]. Therefore, there is a need to build classifiers which can cope with the lack of information, imprecision and noise. *Fuzzy* techniques [22] [9] are known in the *soft computing* literature to be able to better deal with noisy data than the crisp methods and may lead to the development of more robust systems.

In consequence, we consider a *fuzzy self-organizing* map approach towards software fault detection to be a pertinent choice for both coping with uncertainty and for overcoming the drawbacks of supervised learning.

3. Related Work on Software Defect Detection

In the following, we will briefly review several machine learning-based approaches from the defect detection literature which are somehow related to our approach (are based on *unsupervised learning* or are using the same *case studies* as in this paper).

An approach that uses a combination of self-organizing maps and threshold values is presented in [1]. After the SOM is trained, threshold values are used to label the trained nodes: if any of the values from the weight vector is greater than the corresponding threshold, the node will represent the defective entities. Classification is done by finding the best matching unit for the given instance and using the label of the node.

We have introduced an approach for detecting defective entities using self-organizing maps in [16]. After an attribute selection based on the Information Gain [18] of the attributes, a map was trained to visualize the defective entities. While we had encouraging results, we have realized that in many cases defective and non-defective entities are quite similar, they are close to each other on the map. These observations led us to the use of fuzzy maps, which can handle such situations.

There are several approaches in the literature that use different clustering algorithms to group defective and non-defective entities. One such approach is presented in [4], where K-Means algorithm is used and the centers of the clusters are found using Quad Trees. Since determining the optimal number of clusters is not a simple task, some approaches use clustering algorithms where the number of clusters is automatically determined. Such an approach is presented in [6] where the X-means algorithm from Weka is used for clustering. After the clusters are created software metric threshold values are used to determine which clusters represent the defective and which represent the non-defective entities. The X-means algorithm (together with a clustering algorithm that is capable of automatically determining the optimal number of clusters, EM) is used in [20] as well, together with different attribute selection techniques implemented in Weka.

Yu and Mishra in [24] investigate the problem of building cross-project detection models,

which are models built from data taken from one software system, but used and tested on a different software system. They use binary logistic regression on the *Ar* datasets, and build two models: self-assessment, when the model is tested on the dataset from which it was built, and forward-assessment, when some datasets are used for building a model and a different one is used for testing it. They conclude that self-assessment leads to better performance measures, but forward-assessment gives a more realistic measure of the real performance of the binary logistic regression model.

The problem of cross-project defect detection is approached in [19] as well. The authors consider situations when the software metrics from the datasets on which a model was built are not the same as the metrics computed for the system to be tested. They introduce an approach which tries to match the software metrics from the different sets to each other, based on correlation, distribution, and other characteristics. To compare this approach to other existing ones, they use 28 datasets (including the *Ar* datasets) and Logistic Regression from Weka.

Multiple Linear Regression and Genetic Programming are used in [2] to evaluate the influence and performance of different resampling methods for the problem of defect detection. The *Ar* datasets are used as case studies to compare five different resampling methods: hold-out, repeated random sub-sampling, 10-fold cross validation, leave-one-out cross-validation and non-parametric bootstrapping. The results of the study show that, considering the AUC performance measure, there is no significant difference between the resampling methods, but the authors claim that this can be caused by the imbalanced datasets or the high number of attributes.

A comparison of statistical and machine learning methods for defect prediction is presented in [15]. They compare logistic regression with six machine learning approaches. The models were evaluated on two *Ar* datasets, and the best performance was obtained using Decision Trees.

4. Methodology

In this section we introduce our *fuzzy self-organizing map* model for detecting faults in existing software systems.

The software entities (classes, modules, methods, functions) from a software system are represented as high-dimensional vectors (an element from this vector is the value of a software metric applied to the considered entity). As shown in [16], the software system *Soft* is viewed as a set of instances (called *entities*) $Soft=\{e_1, e_2, \ldots, e_n\}$. A set of software metrics will be used as the feature set characterizing the entities from the software system, $M=\{m_1, m_2, \ldots, m_l\}$. Therefore, an entity $e_i \in Soft$ from the software system can be represented as a *l*-dimensional vector, $e_i=\{e_{i1}, e_{i2}, \ldots, e_{il}\}$ (e_{ij} denotes the value of the software metric m_j applied to the entity e_i).

For each entity from the software system, its label is known (D=defect or N=non-defect). The labels of the instances will not be used for building the *fuzzy SOM*, since the learning process will be completely unsupervised. The labels will be used only for pre-processing the input data and for evaluating the performance of the resulting classification model.

Before applying the fuzzy SOM approach, the data is *pre-processed*. First, the data is normalized using the *Min-Max* normalization method, and then a feature selection step is used in order to identify a subset of features (software metrics) that are highly relevant for the fault detection task (details will be given in the experimental part of the paper). As a result of the feature selection step, *p* features (software metrics) will be selected and will be further used for building the *fuzzy SOM*.

4.1 The *fuzzy SOM* model. Our proposal

The dataset pre-processed as indicated above, will be used for the unsupervised training of the map. As for the classical SOM approach, a distance function between the input instances is needed. We are using as *distance* between two software entities e_i and e_j *Euclidean Distance* between their corresponding vectors.

We are proposing, in the following, a *fuzzy self-organizing map* algorithm (FSOM) for building the *fuzzy* map. Our algorithm does not reproduce any existing algorithm from the

literature, but it combines the existing viewpoints related to *fuzzy SOM* approaches. The underlying idea in FSOM is the classical SOM algorithm, combined with the concept of *fuzziness* employed in *fuzzy clustering* [14].

The FSOM algorithm enhances the classical Kohonen algorithm for building a SOM with the idea (employed in *fuzzy clustering*) of using a fuzzy *membership matrix*. In *fuzzy clustering*, instead of using a *crisp* assignment of an object to a cluster, an object can belong to multiple clusters. The degree to which an input object belongs to each cluster is indicated by the set of *membership levels* expressed by the columns of the *membership matrix*. In building the *fuzzy SOM*, we will use the *fuzzy membership* idea related to the computation of the "winning neuron". Instead of using a *crisp* best-matching unit (BMU), as used in the classical SOM algorithm, the membership matrix will be used to specify the degree to which an input instance belongs to an output neuron (cluster). This means that an input instance is not mapped to a single neuron (its BMU), but to all the neurons (clusters) from the map (but with a certain *membership degree*).

Intuitively, an input instance will have the larger *membership degree* to the neuron representing its BMU. The idea of updating the winning neuron and its neighbours is kept from the classical SOM, but if the input instance has a larger membership degree (level) to a neighbouring neuron, this neuron will be "moved" closer to the input instance than the other neurons (i.e., the updating rule considers the computed membership levels). Through these updating rules, the FSOM algorithm maintains the main characteristic of the classical SOM of "moving" the winning neuron and its neighbourhood towards the input instance, but it may express a better updating scheme than the crisp approach.

Let us consider, in the following, that the input layer of the map consists of p neurons (the dimensionality of the input data after feature selection) and the computational layer of the map consists of c neurons disposed on a two dimensional grid, in which an output neuron i is represented as a p-dimensional vector of weights, $w_i=\{w_{i1}, w_{i2}, \ldots, w_{ip}\}$ (w_{ij} represents the weight of the connection between the j-th neuron from the input layer and the i-th neuron from the computational layer).

Let us denote by u the *membership matrix*, where: $u_{ik}\in[0,1]$, $\forall 1\le i\le c$, $1\le k\le n$. These values are used to describe a set of fuzzy c-partitions for the n entities, and u_{ik} represents the degree to which entity e_k belongs to the output neuron (cluster) i.

The main steps of the FSOM algorithm are described in the following.

Step 1. Weights initialization. The weights are initialized with small random values from [0,1].

Step 2. Membership degrees computation. The values from the membership matrix are computed as in Formula (1) (as for the *fuzzy c-means clustering algorithm* [14]). m is a real number, greater than 1 and represents the *fuzzifier*. The role of the *fuzzifier* is to control the overlapping between the clusters [14].

$$u_{ik}=\frac{1}{\sum_{j=1}^{c}\left(\frac{\|x_k-w_i\|}{\|x_k-w_j\|}\right)^{\frac{2}{m-1}}} \quad (1)$$

Step 3. Sampling. Select a random input entity e_t and send it to the map.

Step 3.1. Matching. Find the "winning" neuron j^*, as the output neuron which maximizes the *membership degree* of the input entity e_t to the neuron, i.e. $j^*=\underset{1\le j\le c}{argmax}\, u_{jt}$.

Step 3.2. Updating. After identifying the "winning neuron", update the connection weights of the winning unit and its neighbouring neurons, such that the neurons are "moved" closer to the input instance. When updating the weights for a particular neuron, we will consider the *membership degree* of the considered entity to that neuron. More precisely, for each output neuron j, $(\forall 1\le j\le c)$, its weights w_{ji}, $(\forall 1\le i\le p)$ will be updated with a value Δw_{ji} computed as in Formula (2).

$$\Delta w_{ji}=\eta\cdot T^*_{jj}\cdot(e_{ti}-w_{ji})\cdot u^m_{jt} \quad (2)$$

where η is the learning rate and T_{jj^*} denotes the neighbourhood function usually used in the classical Kohonen's algorithm [21] and whose radius decreases over time.

Step 4. Iteration. Repeat steps 2-3 for a given number of iterations.

If we look at Step 2 of the FSOM algorithm, we observe that an input entity will have the largest *membership degree* to the neuron

(cluster) representing its BMU. Intuitively, the degrees to which the entity belongs to the other neurons from the map (others than its BMU) have to decrease as the distance from the entity and the neurons increases. Another characteristic of the *fuzzy* algorithm (compared to the crisp variant) is the fact that the weights of particular neurons from the neighbourhood of the "winning neuron" (see Step 3) are updated differently depending on the degree to which the current entity belongs to the neuron. This updating method may lead to final weights which would give a better representation of the input space.

After the map was trained using the FSOM algorithm described above, in order to visualize the obtained map, the U-Matrix method is used. The U-Matrix value of a particular node (neuron) from the map is calculated as the average distance between the node and its 4 neighbours. If one interprets these distances as heights, the U-Matrix may be interpreted as follows: high places on the U-Matrix represent entities that are dissimilar with those from low places, while the data falling around the same height represent entities that are similar and can grouped together to represent a cluster.

Since the fault prediction problem is a binary classification one, our goal is to identify on the trained map two clusters corresponding to the two classes of entities: *defects* and *non-defects*.

Even if the *fuzzy SOM* was built using unsupervised learning, after its creation it may also be used in a supervised learning scenario for classifying a new software entity. First, the "winning neuron" corresponding to this entity is determined (as indicated in Step 3.1). Then, the class (*defect* or *non-defect*) to which the winning neuron belongs will indicate the result of classifying the new software entity.

For evaluating the performance of the FSOM model trained as shown above, we compute the confusion matrix for the two possible classes, considering the *defective* class as the *positive* one and the *non-defective* class as the *negative* one. For computing the values from the confusion matrix, we use the known labels (classes) of the training entities.

Since defect prediction data is highly imbalanced the number of *defects* is much smaller than the number of *non-defects*) the main challenge in software fault prediction is to increase the *true positive rate* (i.e., maximize the number of *defective* entities that are classified as *faults*), or, equivalently, to decrease the *false negative rate* (i.e., minimize the number of *defective* entities that are wrongly classified as *non-faults*). For the problem of defect detection, having *false negatives* is a more serious problem than having *false positives*. The first situation denotes an undetected fault in the system, which can cause serious problems later, while in case of the second situation some time is lost to thoroughly test a fault-free entity that was classified as faulty. In the case of imbalanced data, the evaluation measure that is relevant for representing the performance of the classifiers is the *Area Under the ROC Curve (AUC)* measure [10] (larger AUC values indicate better classifiers).

5. Computational Experiments

In this section we provide an experimental evaluation of the FSOM model (described in Section 4) on five open source datasets which were previously used in the software defect detection literature. We mention that we have used our own implementation for FSOM, without using any third party libraries.

5.1 Datasets

The datasets used in our experiments are publicly available for download at [17] and are called *Ar1*, *Ar3*, *Ar4*, *Ar5* and *Ar6*. All five datasets were obtained from a Turkish white-goods manufacturer embedded software implemented in C [16]. The software entities from these datasets are functions and methods from the considered software and are represented as 29-dimensional vectors containing the value of different McCabe and Halstead software metrics. For each instance within the datasets, we also know the class label, denoting whether the entity is *defective* or *non-defective*.

We depict in Table 1 the descriptions of the *Ar1-Ar6* datasets used in our case studies. For

Table 1. Description of the datasets used for the experimental evaluation.

Dataset	Defects	Non-defects	Difficulty
Ar1	9 (7.4%)	112 (92.6%)	0.666
Ar3	8 (12.7%)	55 (87.3%)	0.625
Ar4	20 (18.69%)	87 (81.31%)	0.7
Ar5	8 (22.22%)	28 (77.78%)	0.375
Ar6	15 (14.85%)	86 (85.15%)	0.666

each dataset, the number of *defects* and *non-defects* is illustrated, as well as the *difficulty* of the dataset. The measure of *difficulty* for a dataset was introduced by Boetticher in [5] and is computed as the percentage of entities for which the nearest neighbour (ignoring the label of the entity when computing the distances) has a different label. Since our datasets are imbalanced, when computing the difficulty of the datasets, we considered only the percentage of defective entities for which the nearest neighbour is non-defective.

From Table 1 one can observe that all datasets are strongly imbalanced, with all number of *defects* much smaller than the number of *non-defects*. Moreover, it can be seen that the task of accurately classifying the defective entities is very difficult. *Ar1*, *Ar4* and *Ar6* seem to be the most difficult datasets from the defect classification point of view. The complexity of the software fault prediction task for the *Ar1* and *Ar6* datasets is highlighted in Figure 1, which depicts a two dimensional view of the data obtained using t-SNE [23]. T-distributed Stochastic Neighbour Embedding (t-SNE) is a method for visualizing high-dimension data in a way that better reflects the initial structure of the data compared to other techniques, such as PCA. From a visualization point of view, the method has been shown to produce better results than its competitors on a significant number of datasets.

5.2 Results

For the *fuzzy self-organizing map*, we used the *torus* topology in our experiments, since it is shown in the literature that this topology provides better neighborhoods than the conventional one. The parameters used for building the map are the following: 200000 *training iterations* and the *learning coefficient* was set to 0.7. For controlling the overlapping degree in the fuzzy approach, the *fuzzifier* was set to 2 (shown in the literature as a good value for controlling the fuziness degree).

For the feature selection step, we have used the analysis that was performed in [16] on the *Ar3*, *Ar4* and *Ar5* datasets. For determining the importance of the software metrics for the defect detection task, the *information gain* (IG) measure was used. From the software metrics whose IG values were higher than a given threshold, a subset of metrics that measure different characteristics of the software system were finally selected. Therefore, 9 software metrics were selected in [16] to be representative for the defect detection process: *halstead_vocabulary*, *total_operands*, *total_operators*, *executable_loc*, *halstead_length*, *total_loc*, *condition_count*, *branch_count*, *decision_count* [16]. The previously mentioned software metrics will also be used in our FSOM approach.

In the following, we present the results we have obtained by applying the FSOM model on the *Ar1*, *Ar3*, *Ar4*, *Ar5* and *Ar6* datasets. After the data is preprocessed, the FSOM algorithm introduced in Section 4 is applied and the U-Matrix corresponding to the trained FSOM will be used to identify the class of *defects* and *non-defects*. Then, for each instance from the training dataset, we compare the class provided by our FSOM with the entity's true class label (known from the training data). Finally, the AUC measure is computed.

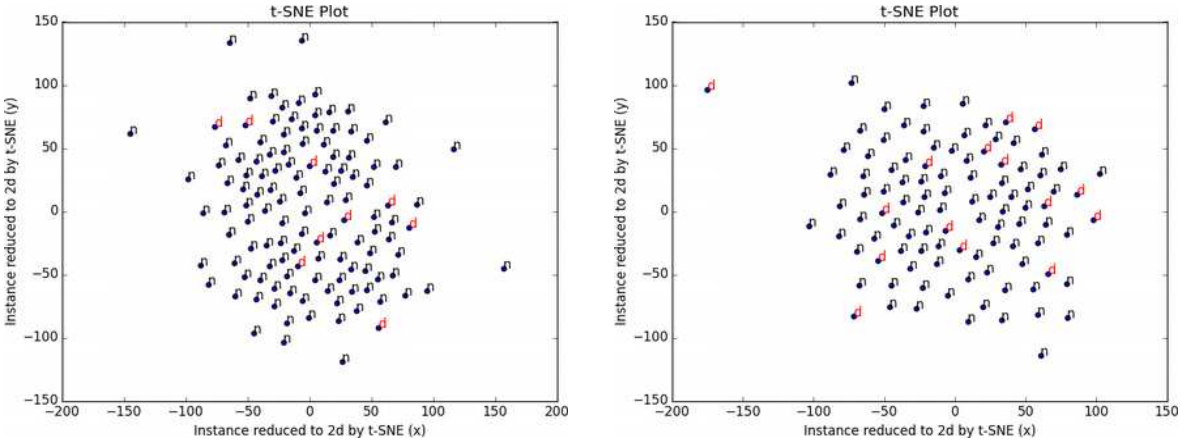

Figure 1. t-SNE plots for the *Ar1* and *Ar6* datasets.

Figure 2 depicts the U-Matrix visualizations of the best FSOMs obtained on the five datasets used in our experiments. On each neuron from the maps we represent software entities which were mapped on that neuron, i.e., instances for which the neuron was their BMU. The red circles represent the defective entities and the green circles represent the non-defective entities. Each neuron is also marked with the number of defects **D** and non-defects **N** which are represented on it.

Visualizing the U-Matrices from Figure 2, one can identify two distinct areas: one containing lightly coloured neurons, whereas the second area consists of darker neurons. The two areas represented on the maps correspond to the clusters of *defective* and *non-defective* software entities. Since the percentage of software faults from the software systems is significantly smaller than the percentage of non-faulty entities (see Table 1), the area from the map containing a larger number of elements is considered to be the *non-defective* cluster. The remaining area from the map corresponds to the *defective* cluster.

Table 2 illustrates, for each dataset, the configuration used for the FSOMs (number of rows and columns of the maps) as well as the values from the *confusion matrix* (*false positives*, *false negatives*, *true positives* and *true negatives*).

Table 2. Results obtained using FSOM on all experimented datasets.

Dataset	rows x cols	FP	FN	TP	TN
Ar1	3x2	26	1	8	86
Ar3	2x3	1	2	6	54
Ar4	2x3	18	4	16	69
Ar5	3x2	4	0	8	24
Ar6	3x3	18	4	11	68

6. Discussion and Comparison to Related Work

As presented in Section 5 and graphically illustrated in Figure 2, our FSOM approach was able to provide a good topological mapping of the entities from the software system and successfully identified two clusters corresponding to the *faulty* and *non-faulty* entities. Even if the separation was not perfect, which is extremely difficult for the software defect detection task, for all five datasets we obtained good enough *true positive rates* (at least 73% detection rate for the defects). For the *Ar5* dataset, our FSOM succeeded in obtaining a perfect defect detection rate, misclassifying only 4 non-defective entities.

The AUC measure is often considered to be the best performance measure to compare classifiers [10]. However, it is usually suitable for methods which, instead of directly returning

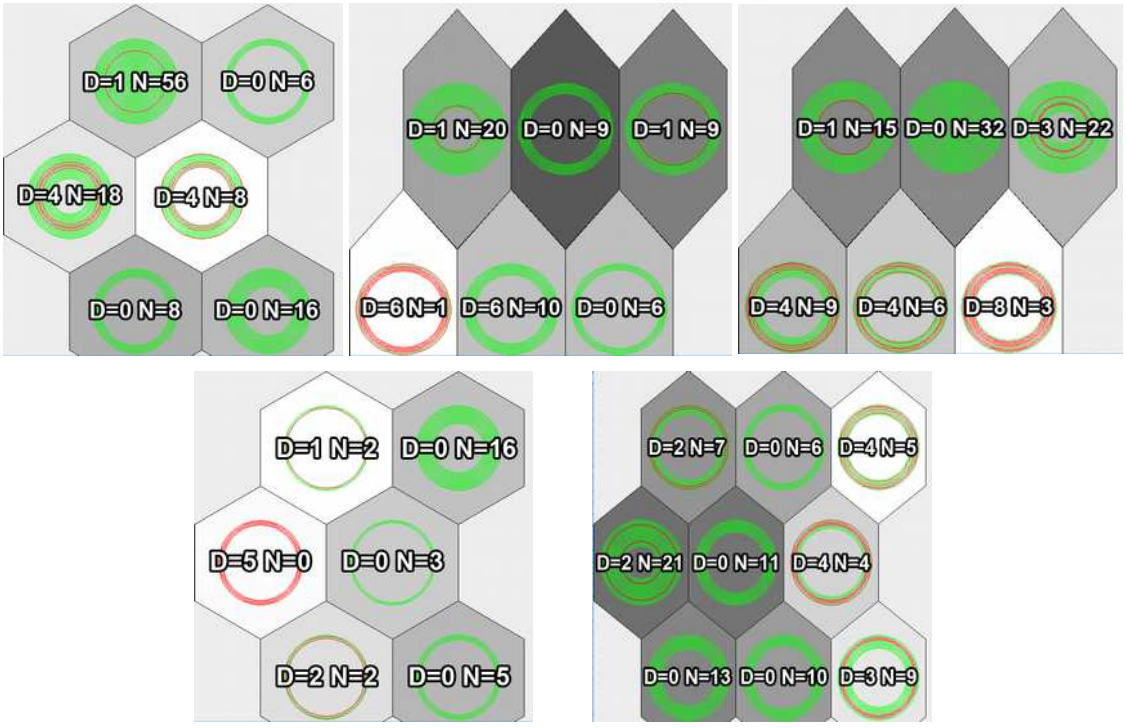

Figure 2. U-Matrix for the datasets used in our experiments.

the classification of an instance, return a score which is transformed into classification using a threshold. In such cases, different thresholds lead to different (*sensitivity*, *1-specificity*) points on the ROC curve, and AUC measures the area under this curve. For methods where no threshold is used (for example, in our approach) the ROC curve contains one single point, which is linked to the points (0, 0) and (1, 1), thus providing a curve and making the computation of the AUC measure possible.

Table 3 presents the values of the AUC measure computed for the results we have obtained using our approach, but it also contains values reported in the literature for some existing similar approaches ([1], [6], [20], [4], [24]), presented in Section 3. If an approach does not report results on a particular dataset, we marked it with "n/a" (*not available*). In case of approaches that do not report the value of the AUC measure, but report other measures (for example *false positive rate*, *false negative rate*) if it was possible, we computed the values from the confusion matrix of these measures and used them to compute the value for the AUC measure, as in case of our approach. The best results obtained for the AUC are marked with bold in the table.

We would like to mention that the results from [2] for the Multiple Linear Regression and Genetic Programming approaches are the best values reported by the authors and they were usually achieved for different resampling settings. In the case of the cross-project defect prediction approach, [24], we have reported only the results of the experiments when the same dataset was used both for building the model and testing it.

From Table 3 we observe that our FSOM approach has better results than most of the approaches existing in the literature and considered for comparison. Out of 54 comparisons, our algorithm has a better or equal value for the AUC performance measure in 48 cases, which represents **89%** of the cases.

It has to be noted that the *fuzzy SOM* method introduced in this paper proved to have a better or equal performance, for all datasets, than the *crisp* approach previously introduced in [16].

Table 3. Comparison of our AUC values with the related work.

Approach	Ar1	Ar3	Ar4	Ar5	Ar6
Our FSOM	0.829	0.87	0.80	**0.93**	0.762
SOM [16]	0.695	0.87	0.74	0.92	0.726
SOM with Threshold [1]	n/a	**0.88**	**0.95**	0.84	n/a
K-means with Quad-Trees [4]	n/a	0.70	0.75	0.87	n/a
Clustering Xmeans [20]	n/a	0.84	0.69	0.86	n/a
Clustering EM [20]	n/a	0.82	0.69	0.80	n/a
Clustering Xmeans [6]	n/a	0.70	0.75	0.87	n/a
Genetic Programming [2]	0.530	0.67	0.65	0.67	0.630
Multiple Linear Regression [2]	0.550	0.61	0.62	0.55	0.590
Binary Logistic Regression [24]	0.551	0.87	0.73	0.39	0.722
Logistic Regression [19]	0.734	0.82	0.82	0.91	0.640
Logistic Regression [15]	0.494	n/a	n/a	n/a	0.538
Artificial Neural Networks [15]	0.711	n/a	n/a	n/a	0.774
Support Vector Machines [15]	0.717	n/a	n/a	n/a	0.721
Decision Trees [15]	**0.865**	n/a	n/a	n/a	**0.948**
Cascade Correlation Networks [15]	0.786	n/a	n/a	n/a	0.758
GMDH Network [15]	0.744	n/a	n/a	n/a	0.702
Gene Expression Programming [15]	0.547	n/a	n/a	n/a	0.688

For the *Ar3* and *Ar6* datasets, the FSOM performed similarly to the classical SOM. For the other three datasets the FSOM outperformed the SOM. For the *Ar1* dataset, the FSOM obtained a significantly better AUC value than the classical SOM. These results highlight the effectiveness of using a *fuzzy* approach rather than a crisp one.

Analyzing the results from Table 3 we observe that our FSOM approach has the highest AUC value for the *Ar5* dataset, the second highest value for the *Ar1* and *Ar3* datasets and the third highest value for the *Ar6* dataset. Interestingly, the results that we have obtained are perfectly correlated with the difficulties of the considered datasets (given in Table 1). More precisely, the best result was obtained for the "easiest" dataset, *Ar5*, while the worst results were provided for the datasets which are more "difficult", *Ar6* and *Ar4*. Even for the hardest datasets, the AUC values obtained by the FSOM are better than most of the AUC values from the literature.

Figure 3 depicts the AUC value obtained by our FSOM and the average AUC value reported in related work from the literature for each dataset (see Table 3). The first dashed bar from this figure corresponds to our FSOM. One can observe that the AUC value provided by our approach is better, for each dataset, than the average AUC value from the related work.

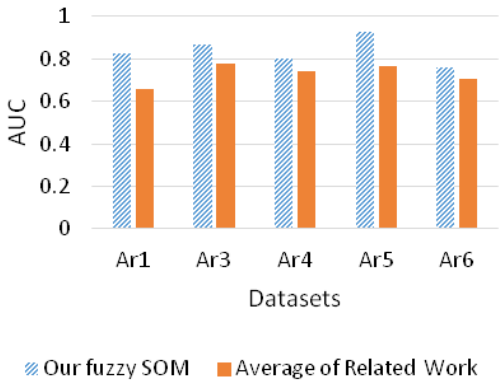

Figure 3. Comparison to related work.

7. Conclusions and Future Work

A fuzzy self-organizing feature map has been introduced in this paper for detecting, in an unsupervised manner, those software entities which are likely to be defective. The experiments we have performed on five open-source datasets used in the software defect detection literature highlight a very good performance of the proposed approach, providing results better than most of the similar existing approaches. Moreover, the *fuzzy* approach [8] introduced in this paper proved to outperform, for the considered case studies, the crisp SOM approach.

Other open-source case studies and real software systems will be further used in order to extend the experimental evaluation of the fuzzy self-organizing map model proposed in this paper. We also aim to investigate the applicability of other *fuzzy* models for software defect detection, as well as to identify software metrics appropriate for software fault detection.

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. ABAEI, G., Z. REZAEI, A. SELAMAT, **Fault Prediction by Utilizing Self-organizing Map and Threshold**, in ICCSCE, 2013, pp. 465-470.
2. AFZAL, W., R. TORKAR, R. FELDT, **Resampling Methods in Software Quality Classification**, International Journal of Software Engineering and Knowledge Engineering, vol. 22, no. 2, 2012 pp. 203-223.
3. ARANDA, J., G. VENOLIA, **The Secret Life of Bugs: Going Past the Errors and Omissions in Software Repositories**, in Proceedings of the 31st International Conference on Software Engineering, USA, 2009, pp. 298-308.
4. BISHNU, P., V. BHATTACHERJEE, **Software Fault Prediction Using Quad Tree-Based K-Means Clustering Algorithm**, IEEE Transactions on Knowledge and Data Engineering, vol. 24, no. 6, June 2012, pp. 1146-1150.
5. BOETTICHER, G. D., **Advances in Machine Learning Applications in Software Engineering**, IGI Global, 2007.
6. CATAL, C., U. SEVIM, B. DIRI, **Software Fault Prediction of Unlabeled Program Modules**, in WCE, 2009, pp. 212-217.

7. CLARK, B., D. ZUBROW, **How Good is the Software: A Review of Defect Prediction Techniques**, in Software Engineering Symposium, Carnegie Mellon University, 2001, pp. 1-35.

8. DINU, S., **Multi-objective Assembly Line Balancing Using Fuzzy Inertia-adaptive Particle Swarm Algorithm**, Studies in Informatics and Control, vol. 24, no. 3, 2015, pp. 283-292.

9. DRAGOMIR, O., F. DRAGOMIR, V. STEFAN, E. MINCA, **Adaptive Neuro – Fuzzy Inference Systems – An Alternative Forecasting Tool for Prosumers**, Studies in Informatics and Control, vol. 24, no. 3, 2015, pp. 351-360.

10. FAWCETT, T., **An Introduction to ROC Analysis**, Pattern Recognition Letters, vol. 27, no. 8, 2006, pp. 861-874.

11. GRAY, D., D. BOWES, N. DAVEY, Y. SUN, B. CHRISTIANSON, **The Misuse of the NASA Metrics Data Program Data Sets for Automated Software Defect Prediction**, Proceedings of the Evaluation and Assesment in Software Engineering, 2011.

12. HALL, T., S. BEECHMAN, D. BOWES, D. GRAY, S. COUNSELL, **A Systematic Literature Review on Fault Prediction Performance in Software Engineering**, IEEE Transactions on Software Eng., vol. 38(6), 2011, pp. 1276-1304.

13. KIM, S., H. ZHANG, R. WO, L. GONG, **Dealing with Noise in Defect Prediction**, in Proceedings of the 33rd International Conference on Software Engineering, New York, NY, USA, 2011, pp. 481-490.

14. 14.KLAWONN, F. and HÖPPNER, F., **What Is Fuzzy about Fuzzy Clustering? Understanding and Improving the Concept of the Fuzzifier**, LNCS 2810, Springer, 2003, pp. 254-264.

15. MALHOTRA, R., **Comparative Analysis of Statistical and Machine Learning Methods for Predicting Faulty Modules**, Applied Soft Computing, vol. 21, 2014, pp. 286-297.

16. MARIAN, Z., G. CZIBULA, I.-G. CZIBULA, S. SOTOC, **Software Defect Detection using Self-Organizing Maps**, Studia Univ. Babes-Bolyai, Informatica, vol. LX, no. 2, 2015 pp. 55-69.

17. MENZIES, T., R. KRISHNA, D. PRYOR, **The Promise Repository of Empirical Software Engineering Data**, http://openscience.us/repo/, North Carolina State Univ., Dep. of Computer Science.

18. MITCHELL, T. M., **Machine learning**, McGraw-Hill, New York, USA, 1997.

19. NAM, J. S. KIM, **Heterogeneous Defect Prediction**, in Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, 2015, pp. 508-519.

20. PARK, M. E. HONG, **Software Fault Prediction Model using Clustering Algorithms Determining the Number of Clusters Automatically**, Int. Journal of Software Engineering and Its Applications, vol. 8, no. 7, 2014, pp. 199-205.

21. SOMERVUO, P., T. KOHONEN, **Self-organizing Maps and Learning Vector Quantization for Feature Sequences**, Neural Processing Letters, vol. 10, 1999, pp. 151-159.

22. TEODORESCU, H.-N. L., **Coordinate Fuzzy Transforms and Fuzzy Tent Maps – Properties and Applications**, Studies in Informatics and Control, vol. 24, no. 3, 2015, pp. 243-250.

23. VAN DER MAATEN, L., G. HINTON, **Visualizing Data using t-SNE**, Journal of Machine Learning Research, vol. 9, 2008, pp. 2579-2605.

24. YU, L., A. MISHRA, **Experience in Predicting Fault-Prone Software Modules Using Complexity Metrics**, Quality Technology & Quantitative Manag., vol. 9, no. 4, 2012, pp. 421-433.

25. ZHENG, J., **Predicting Software Reliability with Neural Network Ensembles**, Expert Systems with Applications , vol. 36, no. 2, Part 1, 2009, pp. 2116-2122.