STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LXVI**, Number 1, 2021
DOI: 10.24193/subbi.2021.1.04

DETECTING THE MOST IMPORTANT CLASSES FROM SOFTWARE SYSTEMS WITH SELF ORGANIZING MAPS

ELENA-MANUELA MANOLE

Abstract. Self Organizing Maps (SOM) are unsupervised neural networks suited for visualisation purposes and clustering analysis. This study uses SOM to solve a software engineering problem: detecting the most important (key) classes from software projects. Key classes are meant to link the most valuable concepts of a software system and in general these are found in the solution documentation. UML models created in the design phase become deprecated in time and tend to be a source of confusion for large legacy software. Therefore, developers try to reconstruct class diagrams from the source code using reverse engineering. However, the resulting diagram is often very cluttered and difficult to understand. There is an interest for automatic tools for building concise class diagrams, but the machine learning possibilities are not fully explored at the moment. This paper proposes two possible algorithms to transform SOM in a classification algorithm to solve this task, which involves separating the important classes - that should be on the diagrams - from the others, less important ones. Moreover, SOM is a reliable visualization tool which able to provide an insight about the structure of the analysed projects.

Introduction

Nowadays, many software engineering problems are tackled by means of computational intelligence. The idea is to reformulate some difficult, repetitive, expensive, or even boring software engineering activities as search problems that can benefit from the advantages of Artificial Intelligence. Various

Received by the editors: 24 May 2021.

2010 *Mathematics Subject Classification.* 68T05, 68T20.

1998 *CR Categories and Descriptors.* I.2.6 [**Artificial Intelligence**]: Learning – *Connectionism and neural nets*; D.2.2 [**Software Engineering**]: Design Tools and Techniques – *Computer-aided software engineering (CASE).*

Key words and phrases. self organizing maps, artificial neural networks, machine learning, classification algorithms, search based software engineering, software visualisation, key class detection, condensing class diagrams .

problems can be approached: defect detection [2] [21], cost estimation [13], requirements analysis [8], software maintenance [9], test oracles [28], maintaining legacy systems [29].

The focus of this paper is a less popular, yet challenging software engineering problem: detecting the key classes from a software system for automatic diagram construction. From a machine learning perspective, we have a data set containing OOP classes and the goal is to classify them in groups of relevant and non-relevant ones. The relevant (key) classes are meant to be represented on the UML diagrams while the classes with lowest scores are ignored.

Self Organizing Maps (SOM) have been previously used for software analysis and visualization but was never applied to the class detection problem before. The paper is structured as follows: the first chapter presents the basics of Self Organizing Maps. In the second section we review some related articles of the domain. Then, the key detection problem and its applications are highlighted. The fourth chapter deals with our methodology for resolving this problem, where we describe the steps of our approach, beginning with data representation and visualization and continuing with SOM algorithm and its transformation in a hybrid method for classification purposes. The experimental setting and the performance results are presented in the fifth chapter, followed by the concluding remarks section.

1. Self Organizing Maps

The SOM algorithm was developed in the 1980's by Professor Teuvo Kohonen [11]. SOM are presented as special unsupervised neural networks. Their main purpose is to create a low dimensional representation of the input space of the training data. This representation forms a map of neurons where they compete and organize themselves in such a way that the topological properties of the input space is preserved. This means that if two instances from the input space are close to each other, they will be near each other in the map, too. Only one neuron is activated at any time (the winning neuron), because of the competition process which induce inhibitory connections.

The SOM algorithm is inspired from the neural cortex of the brain. Different sensors (motor, visual, auditory) are mapped in certain areas of the cortex. They form a map (topographic map), where each piece of input information is stored in its neighbourhood and where neurons responsible with closely related pieces of information is kept close together so that they can communicate fast via short synapses. The location of a neuron in a SOM corresponds to a particular instance from the input space [11].

The architecture of SOM is fairly simple. The Kohonen map has a feed-forward structure with an input layer and a computational layer with neurons arranged in a matrix. Each neuron is connected to all nodes from the input layer. The number of input units is equal to the number of dimensions from the input space. In the training phase, the map is built using the input data. Then, a new input instance can be automatically classified using the mapping phase, by matching it to its nearest neuron.

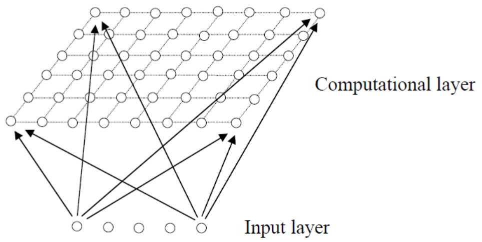


FIGURE 1. SOM architecture

The main components of any self organizing map are:

- **Initialization phase**: The connection weights of each neuron j from the computational layer, $w_j = (w_{j1}, w_{j2}, ..., w_{jD})$ are initialized at random.
- **Competition**: For each input instance, the distance between the input vector: $x = (x_1, x_2, ..., x_D)$ and the weight vector: $w_j = (w_{j1}, w_{j2}, ..., w_{jD})$ is computed as: $d_j^2(x) = \sum_{i=1}^{D}(x_i - w_{ji})^2$, if we use the Euclidean distance. Other distances can be used as well. The neuron with the smallest distance from the input instance becomes the winning neuron or the Best Matching Unit (BMU).
- **Cooperation**: Now, the winning neuron influences the other neighboring neurons, by signaling them. The neurons which are closer to the BMU are excited more than the neurons which are far away. For this, we define a topological neighborhood function, which decays with distance: $T_{j,I(x)} = exp(-S^2_{j,I(x)}/2\sigma^2)$, where S_{ij} is the distance between neurons i and j and $I(x)$ is the index of the winning neuron. σ is a time dependence, for example the exponential decay: $\sigma(t) = \sigma_0 exp(-t/\tau_\sigma)$
- **Adaption**: In this phase, the weights of the BMU and its neighbors are updated. The excited neurons are moved closer to the data point, by adjusting the associated connection weights. The BMU is modified by a greater amount than the neighboring neurons. The update rule is: $\Delta w_{ji} = \eta(t) \times T_{j,I(x)}(t) \times (x_i - w_{ji})$, where η is the learning rate, defined as: $\eta(t) = \eta_0 exp(-t/\tau_\eta)$.

The steps from above are repeated for a predefined number of iterations or until there are not significant changes in the map organization.

As stated before, SOM is usually used for visualisation purposes. An idea is to simply draw the matrix of neurons according to their positions from the map. Each neuron is equivalent with a reduced representation of an input instance (or of a group of instances). Another common technique is to represent a matrix of distances (called U-Matrix) between the weight vectors of adjacent neurons. The matrix is coloured with values of different intensities. Lighter colour between two neurons denotes a small distance, while a darker colour denotes a large distance. Figure 2 is an example of a U-matrix. The black dots denote the neurons.

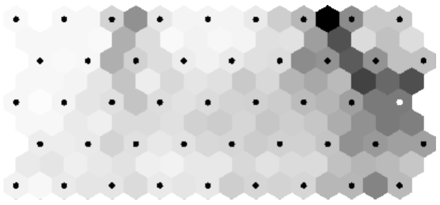


Figure 2. SOM U-Matrix

The U-matrix is useful especially for cluster analysis. The dark areas in the map are interpreted as boundaries between the clusters of the underlying data.

The neurons from the map are activated with various frequencies. Some neurons are activated multiple times, others are activated just once, others are never activated. To see the overall distribution of activated neurons, activation frequency map can be used.

2. Related work

In this section we give a brief overview of SOM models used for software engineering problems.

Czibula et al. [17] [3] discuss about software restructuring and refactoring via clustering with SOM. In their experiments, SOM was able to distinguish clusters and proved to be effective for visualization purposes. Even though the U-matrix can be used to identify the boundaries between clusters, it is argued that its interpretation is a matter of subjectivity. To overcome the need of a reliable clustering tool, their key-idea was to apply a hierarchical clustering algorithm on the trained units.

SOM is also effective for defect prediction, as proposed in [16]. The goal is to detect two clusters, one containing non-defective entities and another one containing defects. An analysis is performed by observing the U-matrix and the boundaries of high distances. The results showed a good performance of SOM in terms of the Area Under The Receiver Operator Characteristic Curve (AUC) and quantization error.

Another very interesting approach for default prediction is studying a hybrid SOM in a semi-supervised manner described by Abaei [1]. The technique is suited when we face a limited amount of defect data. The algorithm has several phases. Firstly, SOM is trained and an initial clustering is performed. Then, the neurons from the map that are not activated are removed from the system. The weights of the remaining (activated) neurons are assigned to labels, based on some thresholds given by certain software measurements. These weights are fed to a neural network for further training. It has been observed that the neurons of SOM tend to resemble the input data as they move towards it. Using some thresholds to stop the neurons from approaching too close to the input patterns is a justified heuristic.

In terms of visualization and analysis of software engineering data, the research conducted by MacDonell [14] presents a range of situations in which one would benefit from SOM. One application involves clustering software artefacts in groups with low, medium and high defect counts. The clusters can be interogated for statistics purposes. Another useful application is building component maps, depicting the distribution of one software metric per map. Groups with similar values for the analyzed metric are detected (similar number of attributes per class, the depth in the inheritance tree, the number of child classes, the number of lines per code). The visualization with SOM reveals useful information about the distribution of artefacts. Pedrycz et al. [23] analyses the Linguist open source Java project in terms of software metrics. Besides, in [22] they identified relationships between classes, for example, one cluster will contain classes having a specific keyword, or specific type (helper classes, error handling classes, interfaces, etc.). Some patterns between the software metrics were identified as well.

3. Key class detection and condensing class diagrams

The problem that we study can be found in the literature with various names: detecting class importance, condensing reversed engineering class diagrams or key class detection. We dedicate this separate chapter to review the existing approaches of the domain.

Detecting key classes using an automatic approach can be an interesting research topic and has several applications: condensing reversed-engineered class diagrams [19] [27], helping program comprehension, analyzing design evolution [7], prediction of future changes [31], recommending potentially relevant files that the developer should view (easy navigation) [25] and others.

Although several existing Computer Aided Software Engineering tools can be configured to remove several properties in a class diagrams, they are unable to automatically identify classes that are less important [19]. Some experimental research showed that developers experience more difficulties in finding the information they need in reverse engineered diagrams and also find the level of detail in "forward" designed ones more appropriate [5].

What makes a class important? There is a research debate about what are the attributes which determine the relevance of a class. It is claimed that an important class represents a key concept that is usually found in the documentation and that has a higher degree of control within the application. This control can be measured, for example, by identifying the tightly coupled classes. The software can be also seen as a network, where classes are vertices and the dependencies between them are edges. A class which is used by many other classes is a good candidate to be an important one, representing a fundamental information or business model. Also, a class which is using other important classes may be an important one as well.

The topic can also be treated subjectively because different groups of developers could define sets of different sizes with key classes. The key class should respect the level of detail that the developer wants to deal with.

Another issue we may face when implementing an automatic key detection system is the imbalanced data. In a real world system, only a few classes are used to document the architectural design. For instance, the developers of Tomcat 5.5 thought that only six classes would be enough to represent the important concepts of the system. Nevertheless, there are many other classes that can be included, if a developer wants to see a more detailed view.

There are a few automatic approaches in the literature for this problem. An interesting article by Șora [18] describes the importance of a class by the amount and types of interactions it has with other classes. The approach is based on an graph ranking algorithm based on Page Rank in order to model the dependencies of the system. Vale and Maia [4] use Trace Extractor which is a tool that saves files with invocation trees for each triggered concurrent thread in the studied program. These trees are used to compute the importance of the involved classes. The algorithm has a tree compression phase to remove identical parts, then a phase which classifies the subtrees as relevant and non relevant. This was done with a Naive Bayes classifier. The final step is to identify the key classes from the subtrees by considering some of the roots as the important classes (or the subtree can further split and assessed).

Osman [19] and Thung [27] solved the problem of condensing reversed engineering class diagrams by identifying only the important classes. They employed supervised methods, among which random forest was the best performing one. Later, they proposed an optimistic classification strategy for dealing with data points with unknown label. To our knowledge, they did not study SOM, so in this research we guide our experiments towards this unexplored direction. In Section 4 we use their research as reference for the approaches documented in the present article.

We found one unsupervised approach with K-means [30] and ensemble learning using the same data sets and we mention its performance as well.

4. Methodology

This chapter presents our approach of SOM for classification, which is quite unusual for this type of machine learning model. The idea is to separate the key classes from the less important ones via a modified SOM. We propose two types of algorithms: a majority voting technique on the trained map units and hybrid approach combining SOM with a classic neural networks.

4.1. **Input data.**

For this research we employ the same data sets and preprocessing strategy from Osman [19] and Thung [27]. They prepared and published nine data sets, each representing a different open-source software project. Obviously, the instances are in fact OOP classes represented as numeric feature vectors. The label is denoted by the last column of the data set, called "In Design", which tells whether the current instance appears or not on the design UML diagram of the project. The nine projects are described in Table 1.

The features that characterize the instances are in fact the values for different software metrics associated with the underlying class. Generally speaking, software measures refer to quantifiable and scalar descriptions of properties of software artefacts [22].

Osman and Thung focused their work on finding metrics serving the goal to distinguish the classes that represent important concepts for the UML diagram.

In one of their earliest studies [19], they chose a set of 11 metrics, called "design metrics", which are well known measures that can be found in any software engineering article: the number of attributes, the number of operations, getters and setters, or various types of dependencies with other classes and coupling measures. The authors conducted a survey [20] about class diagram comprehension which revealed that the metrics related to size and coupling are preferred among developers. The second argument was that coupling is an important structural element in object oriented systems.

Project	Description	Total Classes	Classes In Design Diagram
ArgoUML	UML diagramming application	903	44
JGAP	A framework for performing genetic algorithms and genetic programming	171	18
JPMC	A collection of automated intelligent agents in financial sector	121	24
JavaClient	A framework for developing robotics applications	214	57
Mars	An application for creating simulation of possible human settlement on Mars	840	29
Maze	An application for solving maze puzzles	59	28
Neuroph	A framework for developing neural network architectures	161	24
Wro4J	An application for optimizing web resources	87	11
xUML	A software for producing executable and testable systems from specified data models and associated state machines	84	37

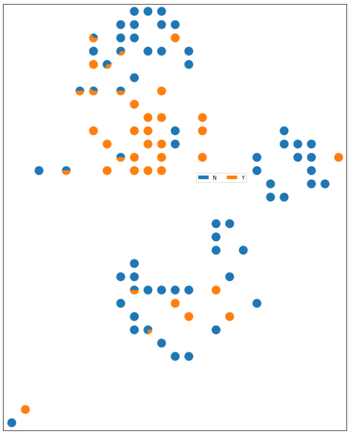
TABLE 1. The datasets [27]

Later, they addressed a new question: *How can we represent the relationships between classes from a new perspective in order to assess class importance?* This led to the concept of network metrics, which greatly improved the accuracy of the existing models [27]. The software project is seen as a network in which the nodes represent the classes and the edges denote the relationships between them (aggregation, composition, generalization and dependency). Based on this graph, a series of 10 metrics were computed. Among them, we mention: Barycenter Centrality- sum of shortest distances of node v to all other nodes, Betweenness Centrality - number of shortest paths between all possible pairs of other nodes that go through node v, Closeness Centrality - the mean shortest distance of node v to all the other nodes, Page Rank- suggesting that nodes with more incoming links are more important. Besides these, some custom metrics based on the partial known knowledge are defined: the proportion of known important classes among the neighbours of a class, the shortest distance to known important classes, neighbour existence, etc. The need for these last metrics is that in real scenarios, the amount of labeled data is limited.

The full description of the data sets and the feature extraction can be found in [19] and [27], together with download information.

4.2. **Data Visualization.** Before presenting the classification algorithms, we take advantage of the visualization capabilities of SOM to investigate the data distribution and if any clusters can be already observed.

Figure 3 presents the maps with the activated neurons for two of the datasets: JavaClient and xUML. Each neuron that was activated at least once, was the BMU (Best Matching Unit) for at least one data point, which means that the neuron can be seen as the reduced representation of that data point.



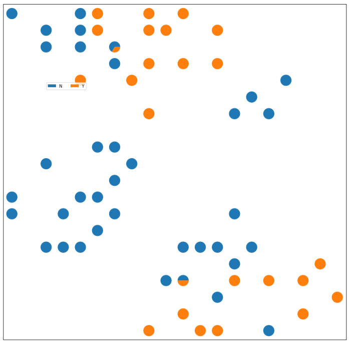


FIGURE 3. Map of activated neurons for JavaClient(left) and XUML(right)

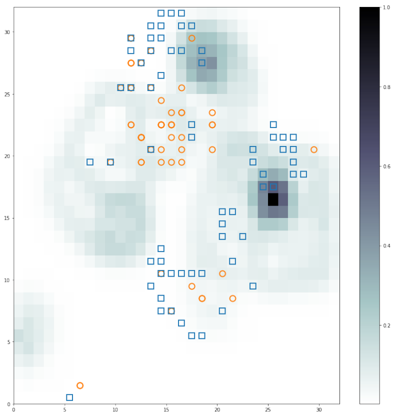
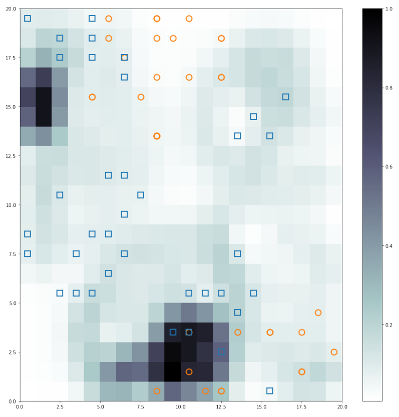



FIGURE 4. U-Matrix for JavaClient(left) and XUML(right)

We illustrate the trained map with the neurons coloured as pie charts, denoting the proportion of positive instances (key classes) and negative instances (non-important classes). The positive instances are marked with orange, while the negative ones are blue. We observe that positive ones are grouped together, with a few exceptions. Also, the neurons which are closer to the ”centre” of the cluster are homogeneous, while the ones closer to the boundaries between the clusters are more mixed: they contain both positive and negative instances.

The U-matrix (Figure 4) can also provide insightful information. With light blue, small distances between neighbouring neurons are represented, while the dark blue stands for large distances between the neurons. The scale on the right side of the map depicts the distances. We label again the associated data points for each neuron. The orange circles represent the positive instances, the blue squares are the negative ones. Again, we observe the neurons with overlapping symbols - which map both positive and negative instances are closer to the cluster boundaries (dark regions).

Overall, for visualisation purposes and dimensionality reduction, we are satisfied with what SOM achieved. The data is separated in groups, but we would like to know exactly which information is encoded in the neurons. Labelling every neuron with the name of the classes they represent may provide some insight. In Figure 5 and Figure 6 we discover the following: each neuron (or neighbourhood of neurons) tends to map classes with the same kind of responsibility or belonging to the same package. Thus, it would be interesting to study other types of software metrics specific to the key class detection problem and assess if they improve the partitioning. We are not discouraged by these findings because the partitioning by design diagram classes is still a satisfactory one, as it can be observed on the coloured maps.

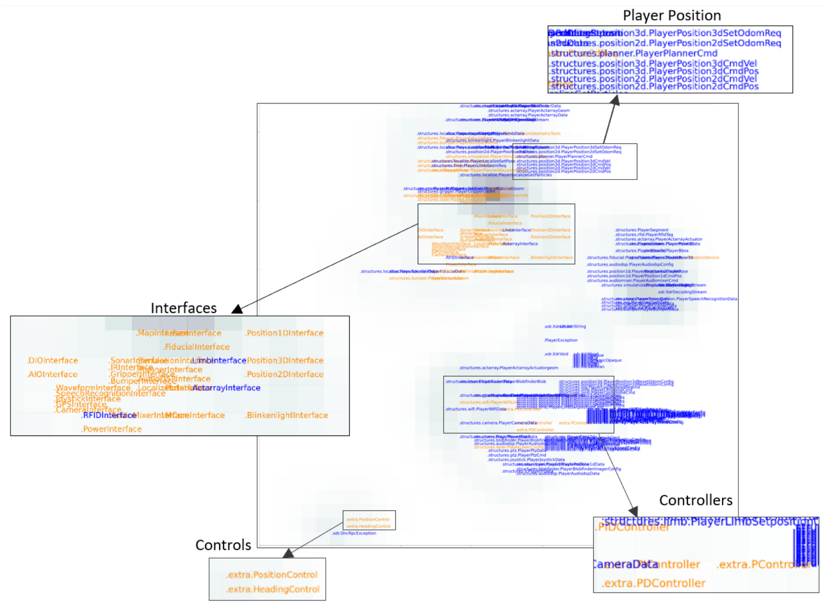


FIGURE 5. Insight of JavaClient Project with SOM

We present a "zoomed in" map representation where we mark some of the groups that we found: interfaces, controllers, factories, etc. JavaClient is an application in robotics. We observe on the map groups of classes responsible

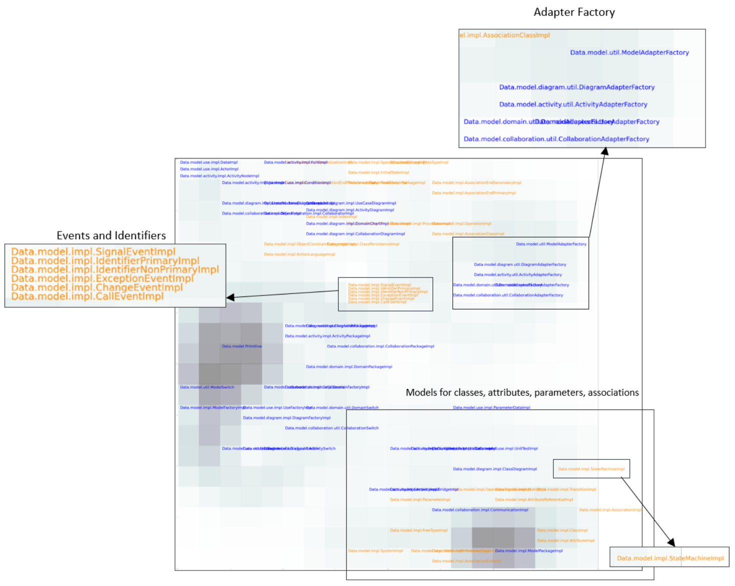


FIGURE 6. Insight of XUML Project with SOM

for the player position and controls. XUML is an application for building executable systems from data models and state machines, so in the corresponding map we find components associated to notions such as classes, attributes, parameters, associations, or events. For this project, the map units seem better separated by the ”In Design” label.

4.3. **Classification with SOM - A Majority Voting Approach.** As discussed in previous chapters, we consider the classes that are on the design diagram as instances from the positive class, and all the others are in the negative class. The current section explains a probabilistic classification algorithm for the trained map units. An unseen data point is assigned to a class by using a type of majority voting. In the literature, other such techniques have been successfully applied for SOM [26], [12].

The algorithm from our own approach has the following steps:

Training the SOM

The first step is nothing else than building the map for the data set. A classical SOM is trained, similar to the one from Section 4.2. The result is a map of trained neurons, which have learned to represent the input data. On the map we find neurons which were activated at least once: they are the best matching unit for at least one input data point. There are also neurons that were never activated, these ones will be ignored in the next steps and we will call them dead neurons (similar to the approach presented in [1]. Dead neurons help their neighbours perform better in training process; however, they are no longer needed after the SOM algorithm is finished.

Build statistics for the trained neurons

Each neuron from the trained map is now analysed. For every unit j, we compute the probability of positive and negative instances that have as best matching unit the neuron j. This is similar to the pie chart mapping from section 4.2. The formulas are the simple probability computations, defined as follows:

$$P_{pos(j)} = \frac{number\_of\_positive\_instances}{total\_number\_of\_mapped\_instances}$$

$$P_{neg(j)} = \frac{number\_of\_negative\_instances}{total\_number\_of\_mapped\_instances} = 1 - P_{pos(j)}$$

Classification

When an unseen instance is going to be classified, it is presented to the trained SOM, which will try to compute the best matching unit for it. There are two cases:

1. The BMU is an activated neuron from the training phase. This means that the percent of positive and negative instances mapped so far by the neuron is known, and we have a good chance that the new instance will belong to the majority class, too.

2. The BMU was not activated in the training phase. This means that no information is available to be able to distinguish if the instance is positive or negative. In this case, we determine the top n closest neurons that were activated and compute the probabilities among them. The reason why we do not choose the closest neighbour is that one single BMU may not reflect the entire neighbourhood. For example, the closest neuron may show a probability of 100% negative, but the rest of the neighbouring neurons may encode only positive instances.

To be observed that this approach remains unsupervised. In the training phase, no information about the actual labels is used when the unit weights are updated. The probabilities are computed after the training has finished and no weights are updated for neurons which are mixed.

4.4. **Classification with SOM - Adding supervised layers.** The second technique is a hybrid approach that combines the SOM with a few traditional neural network layers. The technique is inspired and adapted from [24], in which the authors add one additional layer to the SOM.

For this study, two more layers are added to the classical SOM, which will be responsible with the classification. Some existing approaches use the label information during the SOM training to update the weights, or augment the input vectors with the label information [10], [15]. The present approach is different because it keeps the SOM independent of the label information. As a result, only the additional layers will update their neurons' weights to learn the classification. The SOM organization remains intact. The reason is that we want to preserve the unsupervised and competitive characteristics of the SOM learning step.

The architecture of the proposed system has two components.

SOM component

The first component is the SOM which is trained in the classical unsupervised manner. After the SOM units are organized, we use forward connections to bind them to the additional supervised layers.

In order to be fed forward, the SOM units need to use an activation function. For this, we use the Gaussian similarity, so the activation of the unit j of the SOM map has the following formula:

$$a_j = e^{\frac{-d_j^2(x)}{2\sigma^2}}$$

where $d_j^2(x)$ is the distance between the unit j and the input data x and σ is the width of the Gaussian.

Additional layers

The second component is responsible for learning to perform classification, and consists of the two additional neuron layers.

One of the layers is fully connected to the SOM units and is using ReLU activation. The output of a neuron l from this layer is:

$$o_l = ReLU(\sum_j w_{jl} \times a_j + bias_l)$$

where w_{jl} is the connection weight between the neuron l and SOM unit j and ReLU is the Rectified Linear Unit activation function computed as:

$$ReLU(x) = max(0, x)$$

The other layer is the output layer, which has a single neuron connected to the previous ReLU units. This neuron uses sigmoid activation to output the probability of the positive class. The sigmoid function is a common choice for binary classification and has the following formula:

$$sigmoid(x) = \frac{1}{1 + e^{-x}}$$

To train the two layers, cross entropy loss function is used with gradient descent. The cross entropy cost is popular among state of the art models. It is preferred in [24] and [6] as it leads to faster convergence and to better local optimum than the squared error function. The cross entropy formula for the binary classification is:

$$C(y, o) = -ylog(o) + (1 - y)log(1 - o)],$$

where y denotes the true label and o is the prediction.

4.5. **A comparison between the two approaches.** The common part of the two proposed algorithm consists of the unsupervised SOM component. Both strategies are meant to transform SOM for classification tasks.

The majority voting strategy operates on the trained units, by associating simple probabilities to each unit based on the labels of the instances mapped to each neuron. The classification is performed using these probabilities.

The second strategy is in fact a hybrid algorithm, combining two ML models: SOM and neural networks. The classification step is more complex than the aforementioned technique, because it requires learning. The unsupervised SOM units are fed to the neural network which will perform supervised training.

The two architectures are presented side by side in Figure 7.

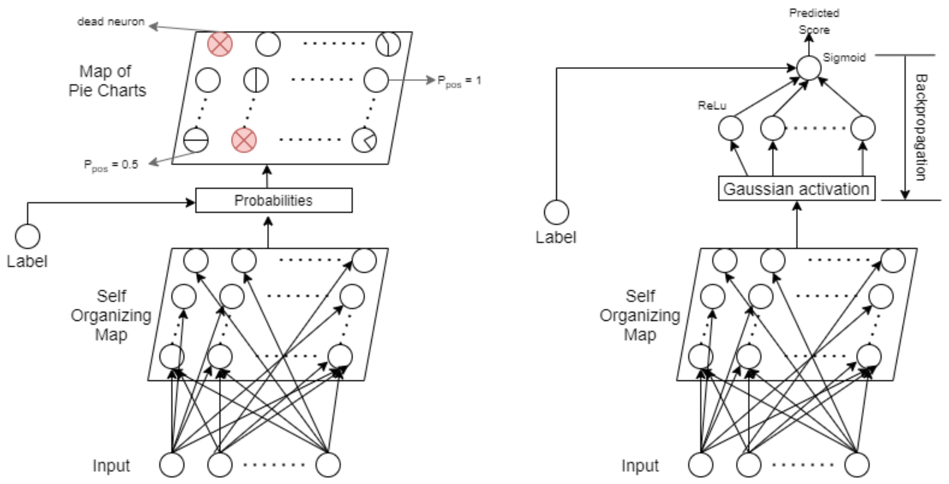


FIGURE 7.
Comparison between the proposed models: SOM with majority voting (left) and SOM with additional layers (right)

5. Experimental Settings and Performance Results

In the following, the two proposed classification algorithms are evaluated for the key class detection problem. Besides, we explain the experimental settings involved in our study.

5.1. **Experimental Settings.** To implement the SOM, we used the *MiniSom* Python library. For the majority voting algorithm, we extended *MiniSom* with custom methods where necessary. For the additional neural layers we used the *Keras* library. The connections between the SOM and the rest of the layers have been implemented from scratch.

For the SOM component we used a traditional rectangular map topology. We performed some tests with a hexagonal topology which brought no significant improvement, therefore we decided not to include it.

Regarding the algorithm based on majority voting, our experiments showed that a map of 5x5 or 7x7 neurons is sufficient for the smaller data sets. For the larger ones we discovered that we need a map with the size around 20x20. The number of neighbours n used to classify unseen instances mapped to a dead BMU was set to 5 by default.

In the second model, the number of neurons on the additional layer connected to the SOM was set to 2/3 of the size of the SOM map. In addition, a dropout strategy from *Keras* is used on this layer.

Because the data sets are imbalanced, leave one out validation is the chosen strategy. In this way, the training set contains as many positive instances as possible. Moreover, each and every data instance is used once for testing. For the supervised component, a strategy for initializing the bias of the output layer was used:

$$b = log(positive\_instances/negative\_instances)$$

To help the training even more, the classes are weighted, telling the model to pay more attention to the positive class, which has a greater weight :

$$weight_0 = (1/negative\_instances) * (total\_instances/2)$$
$$weight_1 = (1/positive\_instances) * (total\_instances/2)$$

The performance will be assessed in terms of the AUC. There are two motivations for using this measure. Firstly, AUC is preferred for highly imbalanced data because it does not favour models that predict the majority label for all data points. Secondly, this metric was also used by Osman and Thung [19] [27], so reporting the same performance measure will help comparing the models.

However, some scientists claim that AUC should not be used when comparing one model to another and should only be used to determine if a ML model is better than random guessing. In absence of any other performance measures for the approach by Thung et al [27], we can only rely on the AUC score. Their experiment is performed in *Weka*, and computes the AUC with the help of a function from the same library.

To evaluate SOM for our novel approach, the *scikit-learn* library for performance metrics was used. For the AUC we applied the *roc_auc_score* function which determines the AUC based on the prediction scores. The implementation uses Riemann integrals for approximating the area under the ROC curve. The thresholds on the curve are also computed automatically by this function. Besides, we also calculated the precision and recall scores. To be noted that the AUC relies on the class probabilities, while for the precision and recall, the predictions are transformed in discrete labels. A default probability threshold of 0.5 was used (the predictions with probabilities greater than 0.5 are considered positive). In a real application, the developers should adjust this threshold, based on the level of detail they prefer for the generated diagram.

5.2. **Results Analysis.** Osman [19] and Thung [27] conducted various research in the domain, with performance ranging from 0.60 to 0.90 (AUC), but we compare our results with their best performance on each data set. Moreover, we also mention the results obtained by the K-means approach employed by Yang et al. [30].

The final performance results are presented in Table 2 and Table 3. Our methods are compared to the random forest optimistic classifier from Thung et al [27] (Baseline 1) and to the K-means approach with ensembles [30] (Baseline 2). Despite the fact that the average AUC is slightly lower than the Baseline 1, SOM achieved better results on some of the datasets. Particularly for xUML project, an outstanding 0.95 AUC was obtained. Compared to the K-means approach, our results are significantly better. However, their study has advantages as well, employing a strategy which requires only a small amount of labelled data.

In general, our approach with additional layers is better than the majority voting one. The surprise was for the Wro4J project which was very difficult for the neural network, but outperformed the literature with the majority voting strategy.

The greatest challenge remains the highly imbalanced data sets. Also, the precision of the neural network is quite low, while the recall is significantly higher. This means that most of relevant classes were successfully retrieved, but the model tends to consider many false positives as well. This could be slightly adjusted by tuning the classification probability threshold.

Project	Baseline 1 [27]	Baseline 2 [30]	SOM & maj voting	SOM & supervised layers
ArgoUML	0.757	0.658	0.71	0.73
JGAP	0.797	0.835	0.67	0.77
JPMC	0.813	0.553	0.76	0.78
JavaClient	0.862	0.774	0.88	0.89
Mars	0.845	0.776	0.76	0.83
Maze	0.767	0.584	0.65	0.61
Neuroph	0.915	0.894	0.88	0.88
Wro4J	0.763	0.680	0.79	0.74
xUML	0.905	0.814	0.94	0.95
Average	0.825	0.730	0.78	0.80

Table 2. AUC achieved by our SOM-based techniques compared to other existing models

Project	SOM & maj voting		SOM & supervised layers	
	Precision	Recall	Precision	Recall
ArgoUML	0.42	0.25	0.14	0.64
JGAP	0.36	0.28	0.23	0.72
JPMC	0.54	0.54	0.52	0.50
JavaClient	0.71	0.81	0.70	0.80
Mars	0.23	0.21	0.13	0.65
Maze	0.74	0.52	0.54	0.78
Neuroph	0.50	0.58	0.47	0.63
Wro4J	0.55	0.55	0.33	0.73
xUML	0.78	0.84	0.92	0.87
Average	0.54	0.51	0.44	0.70

Table 3. Precision And Recall obtained with SOM

6. Concluding Remarks and Further Improvements

This study has proved that SOM is a good challenger for the common classification algorithms. We conclude that it is not only a versatile tool for learning and visualization, but also extensible for different ML tasks.

In general, the software engineering tasks are challenging to tackle from a machine learning perspective, and the problem presented in this research is not an exception. With the help of the Self Organizing Maps, the distribution of the data can be assessed, as well as the relevance of the features that were used to represent it. In the future, it would be interesting to research for other software metrics that would better reflect the degree of importance of a class within a OOP system, but any study related to software measures is a research topic on its own.

The SOM algorithm applied on the important classes detection problem was able to converge quite fast to an acceptable map representation. Also, the majority voting strategy achieved a satisfactory performance for classification, despite its simplicity. The additional neural network did not need many layers and neurons, as we found that one hidden layer (besides SOM map) and a single neuron for the output is enough to learn the classifier. By adding a few more layers, the model is prone to achieve even better results if properly tuned.

Other classification strategies based on SOM are worth exploring. There is the possibility to transform SOM in a completely supervised algorithm, by adjusting the map weights based on the label information and loss function, but for this article the choice was to keep the map organization unchanged.

On a practical note, the purpose of search-based software engineering is to find automatic solutions for everyday tasks. Therefore, a possible application would be to develop an IDEE plugin which integrates the ML model to generate automatic condensed diagrams. In this way, the developers can have a quick initial summary, so that the time spent understanding and navigating through a new project is reduced.

References

[1] Abaei, G., Selamat, A., and Fujita, H. An empirical study based on semi-supervised hybrid self-organizing map for software fault prediction. *Know.-Based Syst. 74*, 1 (Jan. 2015), 28–39.

[2] Ceylan, E., Kutlubay, F. O., and Bener, A. B. Software defect identification using machine learning techniques. In *32nd EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO'06)* (2006), pp. 240–247.

[3] Czibula, G., and Czibula, I. Unsupervised restructuring of object-oriented software systems using self-organizing feature maps. *International Journal of Innovative Computing, Information and Control 8* (03 2012).

[4] do Nascimento Vale, L., and de Almeida Maia, M. Key classes in object-oriented systems: Detection and assessment. *International Journal of Software Engineering and Knowledge Engineering 29*, 10 (2019), 1439–1463.

[5] Fernández-Sáez, A. M., Genero, M., Chaudron, M. R., Caivano, D., and Ramos, I. Are forward designed or reverse-engineered UML diagrams more helpful for code maintenance?: A family of experiments. *Information and Software Technology 57* (2015), 644 – 663.

[6] Golik, P., Doetsch, P., and Ney, H. Cross-entropy vs. squared error training: a theoretical and experimental comparison. In *INTERSPEECH* (2013).

[7] Hammad, M., Collard, M. L., and Maletic, J. I. Measuring class importance in the context of design evolution. *2010 IEEE 18th International Conference on Program Comprehension* (2010), 148–151.

[8] Iqbal, T., Elahidoost, P., and Lucio, L. A bird's eye view on requirements engineering and machine learning. In *2018 25th Asia-Pacific Software Engineering Conference (APSEC)* (12 2018).

[9] Jha, S., Kumar, R., Son, L., Priyadarshini, I., Sharma, R., Long, H., and Abdel-Basset, M. Deep learning approach for software maintainability metrics prediction. *IEEE Access PP* (04 2019), 1–1.

[10] Kohonen, T. The 'neural' phonetic typewriter. *Computer 21*, 3 (1988), 11–22.

[11] Kohonen, T. The self-organizing map. *Proceedings of the IEEE 78*, 9 (1990), 1464–1480.

[12] Lau, K., Yin, H., and Hubbard, S. Kernel self-organising maps for classification. *Neurocomputing 69* (10 2006), 2033–2040.

[13] Lin, J.-C. Automatically estimating software effort and cost using computing intelligence technique. *Computer Science & Information Technology 2* (10 2012), 381–392.

[14] MacDonell, S. G. Visualization and analysis of software engineering data using self-organizing maps. In *2005 International Symposium on Empirical Software Engineering, 2005.* (2005), pp. 10 pp.–.

[15] Mattos, C., and Barreto, G. Artie and muscle models: building ensemble classifiers from fuzzy art and som networks. *Neural Computing and Applications 22* (01 2013), 49–61.

[16] Onet-Marian, Z., Czibula, I., Czibula, G., and Sotoc, S. Software defect detection using self-organizing maps. *Studia Universitatis Babeș-Bolyai Informatica LX* (01 2015), 55–69.

[17] Onet-Marian, Z., Czibula, I.-G., and Czibula, G. A hierarchical clustering-based approach for software restructuring at the package level. *2017 19th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)* (09 2017), 239–246.

[18] Șora, I. Helping progran comprehension of large software systems by identifying their most important classes. In *ENASE* (Department of Computer and Software Engineering University Politehnica of Timisoara, Romania, 2015).

[19] Osman, M. H., Chaudron, M. R. V., and v. d. Putten, P. An analysis of machine learning algorithms for condensing reverse engineered class diagrams. *2013 IEEE International Conference on Software Maintenance* (2013), 140–149.

[20] Osman, M. H., Zadelhoff, A., and Chaudron, M. UML class diagram simplification: A survey for improving reverse engineered class diagram comprehension. *Empirical Studies on the Effects of Modeling in Software Development* (01 2013), 291–296.

[21] Paramshetti, P., and Phalke, D. Survey on software defect prediction using machine learning techniques. *International Journal of Science and Research (IJSR) 3*, 12 (2014).

[22] Pedrycz, W., Succi, G., Musilek, P., and Bai, X. Using self-organizing maps to analyze object-oriented software measures. *Journal of Systems and Software 59* (10 2001), 65–82.

[23] Pedrycz, W., Succi, G., Reformat, M., Musilek, P., and Bai, X. Self organizing maps as a tool for software analysis. *Canadian Conference on Electrical and Computer Engineering 1* (02 2001), 93 – 97 vol.1.

[24] Platon, L., Zehraoui, F., and Tahi, F. Self-organizing maps with supervised layer. *12th International Workshop on Self-Organizing Maps and Learning Vector Quantization, Clustering and Data Visualization (WSOM)* (06 2017), 1–8.

[25] Singer, J., Elves, R., and Storey, M.-A. Navtracks: supporting navigation in software maintenance. *IEEE International Conference on Software Maintenance, ICSM 2005* (10 2005), 325 – 334.

[26] Sousa, R., Rocha Neto, A., Cardoso, J., and Barreto, G. Robust classification with reject option using the self-organizing map. *Neural Computing and Applications 26* (01 2015).

[27] Thung, F., Lo, D., Osman, M. H., and Chaudron, M. R. V. Condensing class diagrams by analyzing design and network metrics using optimistic classification. *Proceedings of the 22nd International Conference on Program Comprehension* (2014), 110–121.

[28] Vanmali, M., Last, M., and Kandel, A. Using a neural network in the software testing process. *Int. J. Intell. Syst. 17* (01 2002), 45–62.

[29] Wiggerts, T. Using clustering algorithms in legacy systems remodularization. In *Proceedings of the Fourth Working Conference on Reverse Engineering* (1997), pp. 33–43.

[30] Yang, X., Lo, D., Xia, X., and Sun, J. Condensing class diagrams with minimal manual labeling cost. In *Proceedings - 2016 IEEE 40th Annual Computer Software and Applications Conference, COMPSAC 2016* (United States of America, 2016), vol. 1, IEEE, Institute of Electrical and Electronics Engineers, pp. 22–31. International Computer Software and Applications Conference 2016, COMPSAC 2016 ; Conference date: 10-06-2016 Through 14-06-2016.

[31] Zimmermann, T., Weisgerber, P., Diehl, S., and Zeller, A. Mining version histories to guide software changes. *Proceedings of the 26th International Conference on Software Engineering* (2004), 563–572.

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

Email address: `meic2001@scs.ubbcluj.ro`