BABEŞ BOLYAI UNIVERSITY
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE

SCIENTIFIC REPORT

January 2016 - December 2016

MACHINE LEARNING FOR SOLVING SOFTWARE MAINTENANCE AND EVOLUTION PROBLEMS

–Învăţare automată în probleme privind evoluţia şi întreţinerea sistemelor informatice–

Project leader: Assoc. prof. CZIBULA István-Gergely

Project code: **PN-II-RU-TE-2014-4-0082**
Contract no.: **263/01.10.2015**

2016

Contents

Chapter 1

Introduction

In this report we will present the original scientific results which were obtained for achieving the objectives proposed in the project's work plan for the year 2016. The first scientific objective is related to the *development of new classification algorithms for identifying entities with defects in software systems*. The second objective is connected to the *development of unsupervised learning methods for software packages restructuring*. The third objective of the current project is related to conducting a literature review on *hidden dependencies identification* and proposing a computational model for this problem.

Chapter 2 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 fuzzy 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 fuzzy self-organizing maps for software defect detection and confirm the potential of our proposal. Section 2.2 introduce a novel approach for predicting software defects using *fuzzy decision trees*. Through the fuzzy approach we aim to better cope with noise and imprecise information.A fuzzy decision tree will be trained to identify if a software module is defective or not. Two open source software systems are used for experimentally evaluating our approach. The obtained results highlight that the *fuzzy decision tree* approach outperforms the non-fuzzy one on almost all case studies used for evaluation. Compared to the approaches used in the literature, the *fuzzy* decision tree classifier is shown to be more efficient than most of the other machine learning-based classifiers.

Chapter 3 approaches the problem of *software restructuring at package level* which has a major importance in the field of software architecture, since refactoring increases the internal software quality and is beneficial during the software maintenance and evolution. As the requirements for grouping application classes into software packages are hard to identify, clustering is useful, since it is able to uncover hidden patterns in data. In this chapter we are investigating software refactoring at the package level by using hierarchical clustering. Two approaches are proposed in order to help software developers in designing well-structured software packages. The first approach takes an existing software system and re-modularizes it at the package level using hierarchical clustering, in order to obtain better-structured packages. The second method we propose considers a certain structure of packages for a software system and suggests the developer the appropriate package for a newly added application class. The experimental evaluations are performed on two open source frameworks and the algorithms have proven to perform well in comparison to existing similar approaches.

The literature review we have conducted in the direction of *hidden dependencies identification* in software systems is presented in Chapter 4.

The main scientific results we have obtained during the period January 2016 - December

2016 are:

- An unsupervised learning based approach using *fuzzy self-organizing maps* (FSOM) and a supervised learning based approach using *fuzzy decision trees* for software defect detection.
- A hierarchical clustering based approach for software restructuring at the package level.
- **11** scientific papers (**9** papers are published, **1** is accepted for publication and **1** is accepted for the second revision). From the published papers, **8** are indexed ISI (**2** in SCI-E journals and **6** in ISI proceedings) and **2** are BDI indexed papers.

We mention that the 2015 *impact factor* of our ISI publications is 2.978.

Chapter 2

Approaches for software defect detection

In this section we present our original approaches which were introduced in the direction of detecting software defects in existing software systems, using *fuzzy self-organizing maps*. The original approaches were introduced by Marian, Mircea and Czibula in [72] and by Czibula, Marian and Ionescu in [30].

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.

2.1 An approach using *fuzzy self-organizing maps*

In order to detect faults in existing software systems, Czibula, Marian and Ionescu introduced in [30] 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.

Software defect detection is a problem intensively investigated in the literature and an active area in the software engineering field, as shown by a systematic literature review published in 2011, which collected 208 fault prediction studies published between 2000 and 2010 [42]. Detecting software faults is a complex and difficult task, mainly for large scale software projects. In the search-based software engineering literature there are a lot of machine learning-based approaches for predicting faulty software entities, for example, [40], [76] and [66]. 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). Much more, 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.

2.1.1 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 [119]. Clark and Zubrow consider in [25] 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 [25]. 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 a software.

Many of the machine learning-based software defect predictors existing in the literature have been built using historical data collected by mining software repositories [56]. Unfortunately, there are studies carried out in the defect prediction literature (like [10]) which have revealed that defect data extracted from change logs and bug reports may contain noise [56]. 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 [39]. Therefore, there is a need to build classifiers which can cope with the lack of information, imprecision and noise.

Fuzzy techniques 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-based approaches (the previously mentioned problem of imbalanced data).

2.1.2 Background

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

2.1.2.1 Fuzzy self-organizing maps

Self-organizing maps (SOMs) [98] are unsupervised learning based models from the neural networks literature that are trained using unsupervised learning to produce a two-dimensional representation of the input space (of training samples), called a *map* [34]. The map consists of an input layer (an input neuron for each dimension of the input data) and an array (usually two-dimensional) of neurons on the computational (output) layer. Each neuron from the input layer is connected to every output neuron and each connection is weighted.

The self-organization process consists of mapping the input instances on the neurons from output layer in order to maintain the topological relationships from the input space. The *topology preservation* is a main characteristic of a SOM and it means that similar input instances are mapped on neurons that are neighbors on the output map [61]. The algorithm that is usually used for training the map is the Kohonen algorithm [98]. After training, the map is able to provide clusters of similar data items [62], being appropriate for data mining tasks that require classification [62]. The SOM can be also used as effective tool for visualizing high-dimensional data.

Different approaches were developed in the literature in order to combine the theory of *self-organizing maps* with the theory of *fuzzy* sets introduced by Zadeh [117].

Tsao et. al introduced in [101] a *fuzzy* Kohonen clustering network, combining the Fuzzy c-Means clustering (FCM) model and the Kohonen network model. This hybridization provided an optimization of the FCM, leading to an improved convergence and accuracy of the obtained results. The authors show that the proposed method can be viewed as a Kohonen type of FCM [101], the "self-organizing" character being given by the size of the updated neighborhood and the learning rate which are automatically adjusted during the learning process.

Lei and Zheng discuss in [63] the combination of ANN and fuzzy sets, and introduce a *fuzzy* self-organizing feature map based on Kohonen's algorithm [63]. An output node from the map corresponds to a cluster and for each output node, a fuzzy set is defined to represent all vectors contained in the cluster corresponding to that node. Compared with the classical Kohonen algorithm, the *fuzzy* approach introduced in [63] replaces the distance between an input instance x and a neuron j on the map with a membership measure of x to the cluster corresponding to neuron j. The authors conclude that the resulting method, unlike the classical SOM method, is able to process inexact or fuzzy information.

Khalilia and Popescu approached in [53] the problem of clustering relational data, i.e., the problem of clustering a set of objects described by pairwise dissimilarity values. The authors proposed an algorithm, FRSOM, which is a combination of the relational SOM approach [45] (the extension of the SOM to handle relational data) and the relational fuzzy clustering algorithm presented in [46] (the extension of the fuzzy c-means algorithm to deal with relational data). The authors highlight in [53], through numerical results, that FRSOM is able to discover substructures in the data that are hard to find by the crisp relational SOM.

A very different approach was introduced by Vuorimaa in [109], a map where the nodes were replaced by fuzzy rules. The exact rules for each node are learned using the regular SOM algorithm. After the map is trained, i.e., the rules were learned, when a new instance is presented to the map, the firing strength of each rule is computed, and these strengths are used as weights to compute one final output for the map. Thus, for each input instance the map will produce one single output value.

2.1.2.2 Related work

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 the experimental part of this paper) .

An approach that uses a combination of self-organizing maps and threshold values is presented in [5]. 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 [75]. After an attribute selection based on the Information Gain [78] 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 nondefective entities are quite similar, they are close to each other on the map. These observations led us to the use of fuzzy self-organizing maps, which can handle such situations.

There are several approaches in the literature that use different clustering algorithms to group defective and nondefective entities. One such approach is presented in [17], where K-Means algorithm is used and the centers of the clusters are found using Quad Trees. Varade and Ingle in [106] use K-Means as well, but they use Hyper-Quad Trees for the cluster center initialization. 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 [22] where the Xmeans algorithm from Weka

[41] 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 nondefective entities. The Xmeans algorithm (together with a second clustering algorithm that is capable of automatically determining the optimal number of clusters, EM) is used by the authors in [85] as well, together with different attribute selection techniques implemented in Weka.

Yu and Mishra in [114] 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 [81] 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 [6] 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 [67]. They compare logistic regression with six machine learning approaches: Decision Trees, Artificial Neural Networks, Support Vector Machines, Cascade Correlation Networks, GMDH polynomial networks and Gene Expression Programming. The models were evaluated on two Ar datasets, and the best performance was obtained using Decision Trees.

2.1.3 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 [75], the software system $\mathcal{S}oft$ is viewed as a set of instances (called *entities*) $\mathcal{S}oft = \{e_1, e_2, ..., e_n\}$. A set of software metrics will be used as the feature set characterizing the entities from the software system, $\mathcal{M} = \{m_1, m_2, ..., m_l\}$. Therefore, an entity $e_i \in \mathcal{S}oft$ from the software system can be represented as an 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 software entity e_i).

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

Before applying the fuzzy SOM approach, the data is *preprocessed*. First, the data is

normalized using the *Min-Max* normalization method, and then a feature selection step will be 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.

2.1.3.1 The *fuzzy* SOM model. Our proposal

The dataset preprocessed 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 the *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* [59].

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 the clusters 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 neighbors is kept from the classical SOM, but if the input instance has a larger membership degree (level) to a neighboring 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 neighborhood 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 the feature selection step) 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 an 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 $u_{ik} \in [0, 1], \forall 1 \leq i \leq c, 1 \leq k \leq n$. 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 (2.1) (as for the *fuzzy* c-means clustering algorithm [59]). 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 [59].

$$u_{ik} = \frac{1}{\sum_{j=1}^{c} \left(\frac{||x_k - w_i||}{||x_k - w_j||} \right)^{\frac{2}{m-1}}} \tag{2.1}$$

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

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.

3.2 Updating. After identifying the "winning neuron", update the connection weights of the winning unit and its neighboring 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 ($\forall 1 \leq j \leq c$), its weights w_{ji} ($\forall 1 \leq i \leq p$) will be updated with a value Δw_{ji} computed as in Formula (2.2)

$$\Delta w_{ji} = \eta \cdot T_{jj^*} \cdot (e_{ti} - w_{ji}) \cdot u_{jt}^m \tag{2.2}$$

where η is the learning rate and T_{jj^*} denotes the neighborhood function usually used in the classical Kohonen's algorithm [98] and whose radius decreases over time.

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

If we are looking to the 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 neighborhood 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 [52] 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 neighbors. If one interprets these distances as heights, the U-Matrix may be interpreted as follows [52]: 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 it was created 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 at 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 are computing the confusion matrix for the two possible classes (*non-defect* and *defect*), considering that the *defective* class is the *positive* one and the *non-defective* class as the *negative* one.

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

Table 2.1: Description of the datasets used for the experimental evaluation.

For computing the values from the confusion matrix, we are using the known labels (classes) of the training entities.

Since defect prediction data are 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 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 [36] (larger AUC values indicate better classifiers).

2.1.4 Computational experiments

In this section we provide an experimental evaluation of the FSOM model (described in Section 2.2.3) 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.

2.1.4.1 Datasets

The datasets used in our experiments are publicly available for download at [31] 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 [75]. 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 *not*.

We depict in Table 2.1 the description of the *Ar1*-*Ar6* datasets used in our case studies. For each dataset, the number of *defects* and *non-defects* are illustrated, as well as the *difficulty* of the dataset. The measure of *difficulty* for a dataset was introduced by Boetticher in [19] and is computed as the percentage of entities for which the nearest neighbor (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 neighbor is non-defective.

From Table 2.1 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 Figures 2.1 and 2.2, which depict a two dimensional view of the data obtained using t-SNE

[104]. T-distributed Stochastic Neighbor 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.

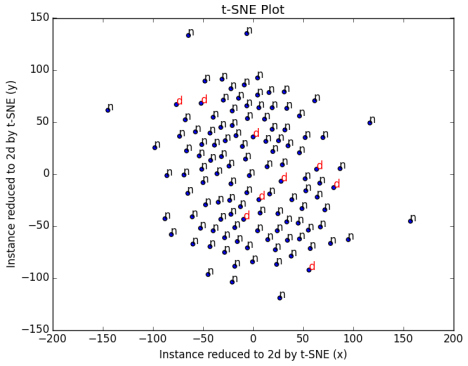

Figure 2.1: t-SNE plot for the *Ar1* dataset.

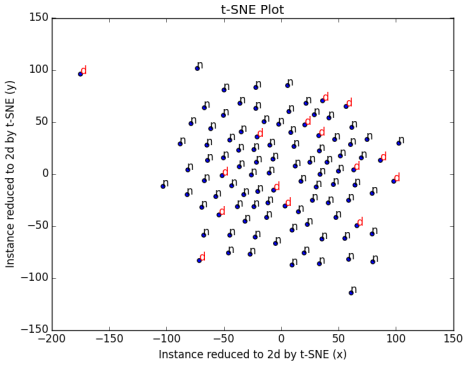

Figure 2.2: t-SNE plot for the *Ar6* dataset.

We can see from Figures 2.1 and 2.2 that, for both *Ar1* and *Ar6* datasets, the fault detection problem we are approaching in this paper is not an easy one, since it is very hard to discriminate between the defective and the non-defective entities (the non-defective entities are marked with black n and the defective ones with red d). We have shown the t-SNE graphs only for the *Ar1* and *Ar6* datasets, but the same situation appears for all datasets we are working with.

2.1.4.2 Results

For the *fuzzy* self-organizing map, we used in our experiments the *torus* topology, since it is shown in the literature that this topology provides better neighborhood than the conventional one [55]. The parameters used for building the map are the following: 200000 *training epochs* 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 [59]).

For the feature selection step, we have used the analysis that was performed in [75] 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 [75] 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* [75]. The previously mentioned features (software metrics) will also be used in our FSOM approach.

We are presenting in the following the results we have obtained by applying the FSOM model (see Section 2.1.3.1) on the *Ar1*, *Ar3*, *Ar4*, *Ar5* and *Ar6* datasets. After the data is preprocessed, the FSOM algorithm introduced in Section 2.1.3.1 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 will be computed.

Figures 2.3, 2.4, 2.5, 2.6 and 2.7 depict the U-Matrix visualization of the best FSOMs obtained on the five datasets used in the experimental evaluation. On each neuron from the

maps we represent the training instances (software entities) which were mapped (using the FSOM algorithm) 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.

Figure 2.3: U-Matrix for the *Ar1* dataset.

Figure 2.4: U-Matrix for the *Ar3* dataset.

Visualizing the U-Matrices from Figures 2.3, 2.4, 2.5, 2.6 and 2.7, one can identify two distinct areas: one containing lightly colored 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 2.1), 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.

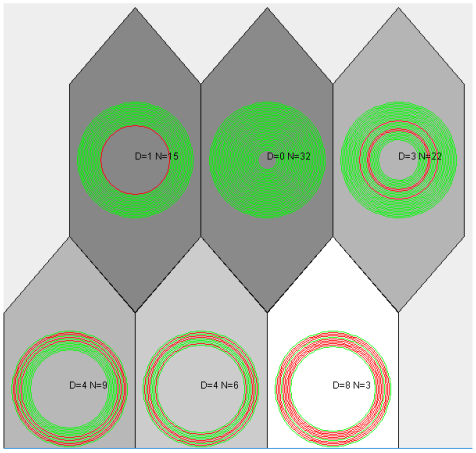

Figure 2.5: U-Matrix for the *Ar4* dataset.

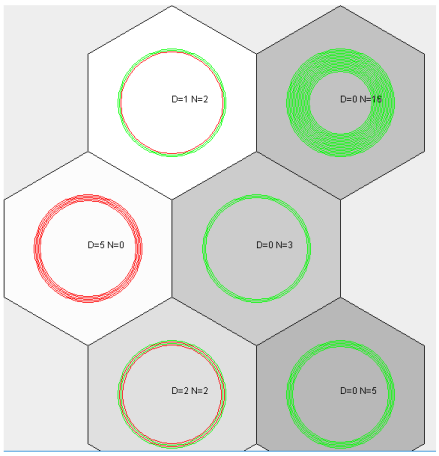

Figure 2.6: U-Matrix for the *Ar5* dataset.

Table 3.6 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*).

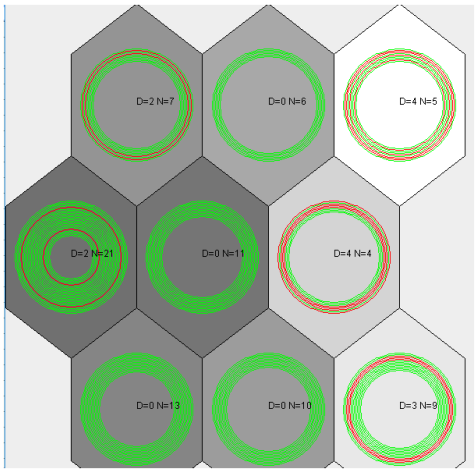

Figure 2.7: U-Matrix for the *Ar6* dataset.

Dataset	rows x columns	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

Table 2.2: Results obtained using FSOM on all experimented datasets.

2.1.5 Discussion and comparison to related work

As presented in Section 3.6 and graphically illustrated in Figures 2.3, 2.4, 2.5, 2.6 and 2.7, 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 [36]. However, it is usually suitable for methods which, instead of directly returning 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 possible the computation of the AUC measure.

Table 2.3 presents the values of the AUC performance 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, presented in Section 2.2.2.2. 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 from 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 measure are marked with bold in the table.

Approach	Ar1	Ar3	Ar4	Ar5	Ar6
Our FSOM	0.829	0.87	0.80	**0.93**	0.762
SOM [75]	0.695	0.87	0.74	0.92	0.726
SOM with Threshold [5]	n/a	**0.88**	**0.95**	0.84	n/a
K-means with Quad-Trees [17]	n/a	0.70	0.75	0.87	n/a
Clustering Xmeans [85]	n/a	0.84	0.69	0.86	n/a
Clustering EM [85]	n/a	0.82	0.69	0.80	n/a
Clustering Xmeans [22]	n/a	0.70	0.75	0.87	n/a
Genetic Programming [6]	0.530	0.67	0.65	0.67	0.630
Multiple Linear Regression [6]	0.550	0.61	0.62	0.55	0.590
Binary Logistic Regression [114]	0.551	0.87	0.73	0.39	0.722
Logistic Regression [81]	0.734	0.82	0.82	0.91	0.640
Logistic Regression [67]	0.494	n/a	n/a	n/a	0.538
Artificial Neural Networks [67]	0.711	n/a	n/a	n/a	0.774
Support Vector Machines [67]	0.717	n/a	n/a	n/a	0.721
Decision Trees [67]	**0.865**	n/a	n/a	n/a	**0.948**
Cascade Correlation Networks [67]	0.786	n/a	n/a	n/a	0.758
GMDH Network [67]	0.744	n/a	n/a	n/a	0.702
Gene Expression Programming [67]	0.547	n/a	n/a	n/a	0.688

Table 2.3: Comparison of our AUC values with the related work.

We would like to mention that the results from [6] 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 case of the cross-project defect prediction approach, [114], 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 2.3 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 [75]. 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 with respect to the crisp one.

Analyzing the results from Table 2.3 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 2.1). 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 larger than most of the AUC values from the literature.

Figure 2.8 depicts, for each dataset we have considered, the AUC value obtained by our FSOM and the average AUC value reported in related work from the literature for the dataset (see Table 2.3). 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 existing related work.

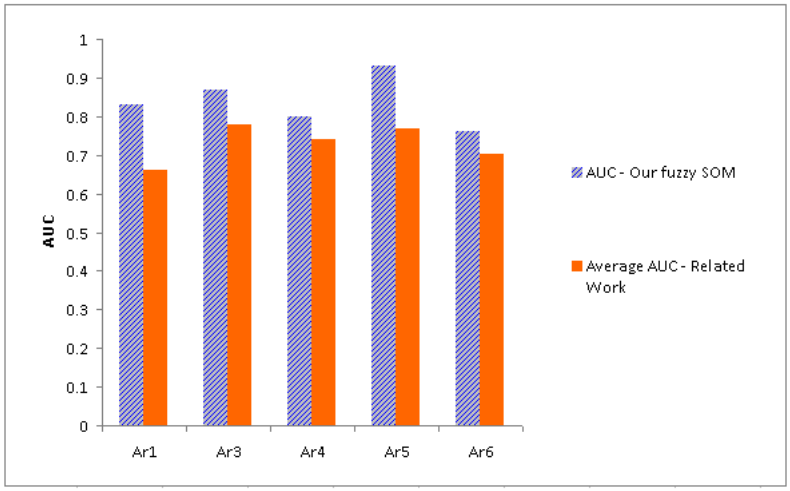

Figure 2.8: Comparison to related work.

2.1.6 Conclusions and future work

A fuzzy self-organizing feature map has been introduced in this section 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 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 (like *fuzzy decision trees* [116]), as well as identifying software metrics appropriate for software fault detection [90].

2.2 An approach using *fuzzy decision trees*

Software quality assurance is a major issue in the software engineering field and is used to ensure the software quality. In order to increase the effectiveness of quality assurance and software testing, defect prediction is used to identify defective modules in an upcoming version of a software system and is useful for assigning more effort for testing and analysing those modules [48].

Most of the machine learning based classifiers existing in the defect prediction literature are supervised. From this perspective, the problem of accurately predicting the defective modules is a hard one, because of the imbalanced nature of the training data (the number of *non-defects* in the the training data is much higher than the number of *defects*). Thus, it is hard to train a classifier to recognize the defects, when a small number of defective examples were provided during training. A major challenge in defect prediction is to increase the number of correctly identified defects and to minimize the number of misclassified defects. Much more, it is not easy to identify the relevant software metrics which would be able to discriminate between *defects* and *non-defects*.

In order to deal with the above mentioned problems, we are introducing in this section [72] a supervised machine learning method based on *fuzzy decision trees* for detecting defects in existing software systems. As far as we know, our approach is novel in the defect prediction literature. The experimental evaluation of the fuzzy decision tree is performed on two open

source software systems and shows that our proposal provides better results than most similar existing ones.

2.2.1 Motivation

Software defect detection represents the activity through which software modules which contain errors are identified. Certainly, the discovery of such defective modules plays an important role in assuring the quality of the software development process. An activity which is also connected to maintaining the software quality is the code review. Reviewing the existing code is time consuming and costly and it is frequently used in the agile software development. *Software defect detection* can be helpful in the code review process to point out parts of the source code where it is likely to identify problems.

From a supervised learning perspective, the problem of identifying defective software entities is a complex and difficult one, mainly because the training data is highly imbalanced. Obviously, a software system contains a small number of defective entities, compared to the number of non-defective ones. Thus, a supervised classifier for defect detection will be trained with a set of defective examples which is much smaller than the set of non-defective ones. This way, the classifier would be susceptible to learn to assign the majority class, namely the non-defective class. That is why, the field of software defect prediction is a very active research area, being a continuous interest in developing performant classifiers which are able to handle the imbalanced nature of the software defect data.

Several studies that have been performed in the defect prediction literature [10] have shown that defect data extracted from change logs and bug reports may be noisy and imprecise [56]. Our previous research in the defect prediction field (like [75]) reinforced the idea that it is very hard to find a crisp separation between the defective and non-defective entities, in most situations defective entities seem to be very similar to non-defective ones. The self-organizing map used in [75] revealed that the defect data contains some uncertain areas (overlapping zones between defects and non-defects) that can lead crisp classifiers to erroneous predictions. That is why we consider that the *fuzzy* approaches would be a good choice for trying to alleviate the previously mentioned problems.

2.2.2 Background

The main characteristics of *fuzzy decision trees* as well as existing approaches for software defect prediction are presented in this section.

2.2.2.1 Fuzzy decision trees

Fuzzy decision trees [103] have been investigated in the soft computing literature as a hybridization between the classical decision trees [78] and the *fuzzy logic*. The classical algorithms for building decision trees (ID3, C4.5) were extended toward a fuzzy setting [50] by considering aspects of fuzziness and uncertainty. At each internal node of the fuzzy tree, all instances from the data set are used, but each instance has a certain membership degree associated. At the root node, all instances have the membership degree 1. Each internal node contains an attribute (selected using Information Gain - Formula (2.6)) and has one child node for each fuzzy function associated to the selected attribute. Each of these child nodes will contain all instances, but the membership degree of each instance from the parent node will be multiplied by the value of the fuzzy function for the given instance. A leaf node from the fuzzy decision tree, instead of containing a single class (target value) as in the classical approach, contains the proportion of the cumulative membership values with respect to the total cumulative membership for each of the classes.

A fuzzy decision tree is used differently when a new instance has to be classified (tested) than a traditional one. The test instance will be considered to belong to all branches of

the fuzzy decision tree with different degrees given by the branching fuzzy function. A final fuzzy membership value will be obtained, this way, for each leaf node in the tree. All the memberships for the leaf nodes are summed for each target class. The class having the maximum associated membership value will be considered as the final classification for the testing instance.

Naturally, the fuzzy decision tree approach conceptually incorporates the crisp approach when the membership degrees of the fuzzy sets used in the process describe crisp memberships. The classic decision tree is therefore a subclass of the fuzzy decision tree and the performance of every fuzzy variant will be at least as good as the crisp correspondent.

However, the problem of defect prediction is very challenging due to the imbalanced nature of the training data sets. Usually the defective entities inside a software project are significantly scarcer than the non-defective ones and therefore the classification using decision trees is not an easy task to be solved, because the Entropy and the Information Gain measures, which play a fundamental part in the decision process, are strongly dependent on the balance in size between the target classes used in training.

Another problem occurs when the probability distributions of the attributes inside the data set are computed separately on each of the target classes. It would have been preferred that the attributes exhibit a normal Gaussian trend as this will aid the decision process, but the probability analysis of the data sets quickly revealed that most of the attributes fall under a lognormal distribution with many values crowded towards 0. In this case it is highly difficult for any form of decision tree to discriminate properly between the two classes as the instances in both groups tend to have the same behaviour and are perfectly disparate throughout the domain making a clear group delimitation a true challenge. Even in the fuzzy perspective it is very difficult to decide between one class and the other as the defective vs. non-defective groups overlap significantly enough to deem many of the instances to be classified undetermined.

2.2.2.2 Literature review

Software defect detection is a well-studied problem, there are many different approaches presented in the literature that try to identify the defective entities in a software system. A literature study published in 2011, [42], found that 208 papers were published on this subject between 2000 and 2010 and since then the number of papers has increased. Most of these approaches are supervised, meaning that they require some training data in order to build the model. There are several openly available data sets that can be used for training, and in this section we are going to present some approaches from the literature that use for the experimental evaluation the same data sets that we have used: *JEdit* and *Ant*.

Okutan and Yildiz present in [82] an approach that uses Bayesian Networks and the K2 algorithm for defect detection. Besides the already existing software metrics they add two new metrics to the data set: lack of coding quality (LOCQ) and number of developers (NOD). For the experimental evaluation, they use 9 publicly available data sets (including *JEdit* and *Ant*) and the implementation of the K2 algorithm from Weka [41]. Based on the generated Bayesian Networks they investigate the effectiveness of different software metric pairs for defect detection, and conclude that the LOC-RFC, RFC-LOCQ, RFC-WMC pairs are the most effective.

Multivariate Logistic Regression is used by Malhotra in [65] to detect the defective entities in the *Ant* system. The authors first detect and remove outliers from the data, then apply the Multivariate Logistic Regression using 10-fold cross validation. The built model includes two metrics from the data set: RFC and CC.

While defect detection is usually considered as a binary classification problem, the authors in [24] consider it a regression problem and they try to predict the exact number of defects in each entity. They compare six different regression methods, Linear Regression, Bayesian

Ridge Regression, Support Vector Regression, Nearest Neighbors Regression, Decision Tree Regression, Gradient Boosting Regression, and conclude that Decision Tree Regression gives the best results in terms of precision and root mean square error. The authors also investigate the difference between within-project defect prediction (when the prediction model is built based on previous version of the same software system) and cross-project defect prediction (when the model is built on other projects). Unlike other such studies, they conclude that cross-project defect prediction models are comparable to within project defect prediction models with respect to prediction performance.

The authors in [21] introduce a cross-project defect prediction approach as well, but they formulate the problem as a multi-objective optimization problem where two different objectives have to be considered: the number of defect prone entities detected and the cost of analyzing the predicted defect prone classes. Their approach is based on a multi-objective Genetic Algorithm, and was tested using 10 different data sets, including *JEdit* and *Ant*. They conclude that the multi-objective approach achieves better performance than the single-objective approach they used for comparison.

Scanniello et al. present an approach, where the classes from the software system are first clustered, to identify clusters of strongly connected classes, then Stepwise Linear Regression is used to build a defect detection model for each cluster separately [93]. Compared to the approach where all classes are used together to build a detection model, this approach can provide a more accurate detection of the number of faults for each class.

2.2.3 Methodology

In this section we introduce our *fuzzy* decision tree based classifier for detecting defective software entities in existing software systems.

As we have previously introduced in [75], the entities from a software system (classes, methods, functions) may be represented as high-dimensional vectors representing the values of several software metrics applied to the considered entity. Thus, a software system $\mathcal{S}$ is viewed as a set of entities (instances) $\mathcal{S} = \{e_1, e_2, ..., e_n\}$ [75]. A set of software metrics will be used as the feature set characterizing the entities from the software system, $\mathcal{M} = \{m_1, m_2, ..., m_l\}$. Therefore, an entity $e_i \in \mathcal{S}$ may be visualized as an l-dimensional vector, $e_i = (e_{i1}, e_{i2}, \ldots, e_{il})$, where e_{ij} represents the value of the software metric m_j applied to the software entity e_i.

As in a supervised learning scenario, the label (class) associated for each entity is known (D=defect, N=non-defect). The first step before applying the *fuzzy* decision tree based learning approach is the *data preprocessing* step. Then, the preprocessed training data will be used for building (training) the *fuzzy* decision tree based classifier. The built classification model will be then tested in order to evaluate its performance. These steps will be detailed in the following.

2.2.3.1 Data preprocessing

During this step, the data set representing the high dimensional software entities will be preprocessed. A feature selection step will be used in order to identify a subset of software metrics that are relevant for predicting software defects.

The data sets that will be used for the experimental evaluation of our approach were created for open-source object-oriented software systems. In these data sets each entity corresponds to a class from the software system and contains data to identify the module (name of the system, version of the system, name of the class) and the value of 20 different software metrics plus the number of bugs in the given entity.

During the data preprocessing step, we first transform the number of bugs into a binary attribute, to denote whether the entity is defective or not. The value 0 will be used for non-defective entities and 1 will be used for the defective ones. In order to reduce the number of

software metrics in the data set, we have used the findings of a systematic literature review conducted on 106 papers [89], which studies the applicability of 19 different software metrics for the task of software fault prediction. Out of the metrics reported by the study as having a strong positive effectiveness on software fault prediction, three can be found in our data sets, so we have decided to eliminate the other metrics. The metrics kept after the preprocessing are: WMC, CBO and RFC.

In order to build the fuzzy sets for the selected software metrics, we have taken inspiration from the work of Filó et al. [96]. They have used 111 software systems written in Java and computed the value of 17 different software metrics for each class of the systems. For each metric they have identifies thresholds to group the value of the metric in three ranges: *Good/Common*, *Regular/Casual*, and *Bad/Uncommon*. The threshold between the first two ranges was computed as the 70 percentile of the data, while the second threshold was considered at the 90 percentile. Since the study presented in [96] contains thresholds for only one of the software metrics that we are using, we have decided to compute our own thresholds.

We have taken all data sets from the Tera-Promise repository [31] that belong to the *Defect* category and use the same software metrics as the data sets used for the experimental evaluation. In case of data sets with multiple versions, we have taken the last version. In this way, we have built a data set containing 6082 instances coming from a total of 30 projects. We have computed the 70 and 90 percentile for the metrics and used these values as thresholds for building two trapezoidal membership functions for the *non-defect* and *defect* classes. The first function measures the membership of a given software metric value to the class of non-defective entities, while the second one measures the membership to the class of defective entities. The two fuzzy membership functions for the WMC software metric are illustrated on Figure 2.9. Formulae 2.3 and 2.4 describe the equations used to compute the membership degree of a software metric value to the *non-defect*, respectively *defect* fuzzy sets. The exact threshold values used for all three metrics are presented in Table 2.4.

$$\mu_{non-defect}(x) = \begin{cases} 1, & x < a \\ \frac{b-x}{b-a} & a \leq x \leq b \\ 0, & x > b \end{cases} \tag{2.3}$$

$$\mu_{defect}(x) = \begin{cases} 0, & x < a \\ \frac{x-a}{b-a} & a \leq x \leq b \\ 1, & x > b \end{cases} \tag{2.4}$$

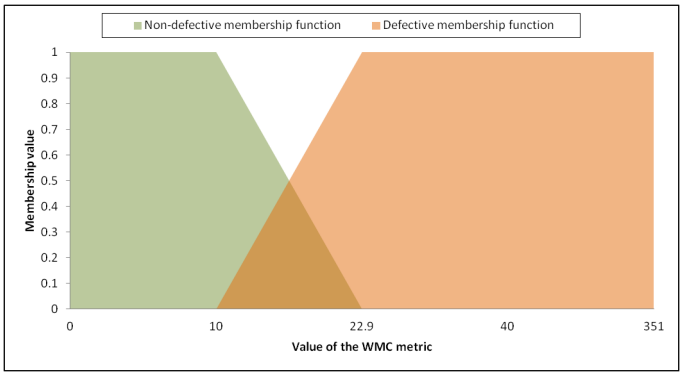

Figure 2.9: Fuzzy membership functions for the WMC software metric.

2.2.3.2 Training

During the training process the fuzzy decision tree is built from the data set that was preprocessed as presented in the previous section. Defect detection data sets are usually imbalanced,

Software Metric	a	b
WMC	10	22.9
CBO	10	20
RFC	30	66

Table 2.4: Threshold values used to build the fuzzy membership functions for the used software metrics.

which can influence the training process and lead to a *fuzzy decision tree* where each leaf node predicts that the given instance is non-defective. In order to reduce the imbalance of the data set, we enhance it before the training process, by adding extra defective instances from other data sets. These instances are used only for the training process, they are not considered during the testing.

Building the *fuzzy decision tree* (*FuzzyDT*)

The building process for the fuzzy decision tree proposed in the current paper resembles the one for a crisp variant of a decision tree with several alterations to cope with uncertainty and data imbalance. Both of these aspects have an important effect on the proposed fuzzy decision tree version shaping it into a custom variant tailored to solve the given problem as accurately as possible.

In order to manage the uncertainty of software defect prediction the defective and non-defective concepts needed to be formalized as fuzzy sets with respect to each of the attributes inside the data set. The method of fuzzy set construction for both of the target classes was presented in the previous section, but it must be added that an intensive selection process was necessary to highlight the attributes that may have a beneficial impact on the fuzzy decision process as many attributes in the data set were naturally not suited to aid any form of classification. This is an aspect that contributes to the difficulty of the defective/ non-defective classification task. It must be mentioned that in order for the fuzzy approach to work, the fuzzy membership functions employed in the decision process need to be handled very delicately preferably their devise being the fruit of a collaboration with software engineering experts. If the fuzzy membership functions do not map accordingly onto real contexts, the whole decisional process will be affected.

Once the fuzzy membership functions are constructed for each attribute, separately on each target class, the real fuzzy decision tree construction may commence. At this point, the other major problem discussed in the introduction occurs: data imbalance. Due to the scarceness of defective instances, the defective target class will be clearly imbalanced with respect to the non-defective target class on the studied attributes. In the classic fuzzy decision tree approach, the fuzzy entropy and fuzzy information gain measures are very biased with respect to data imbalance and this impacts the decision process in the sense that there is a clear inclination towards labeling instances as non-defective simply because the training set contains a significantly increased number of non-defective instances. This is an important issue with deep implications in the decisional process and therefore finding a way to deal with data imbalance was imperiously necessary.

A solution to the imbalance problem was proposed in [64]. Instead of choosing to follow other rather simplistic approaches that directly affect the data set such as over-sampling or under-sampling, which in our opinion are not fit for the present problem because the discrepancy between defective and non-defective instances is too high, the authors propose a way of coping with the imbalance by transforming the entropy and information gain measures. In this way, from a constructional point of view, the only alteration will be changing the entropy and information gain formulae to a form that takes the imbalance into account and includes it in the computation, therefore attenuating its impact. Let us consider, in the

following, that the *defective* class is the *positive* one and the *non-defective* class is the *negative* one. As we have mentioned in Section 2.2.2.1, each internal node from the tree stores all the instances from the training data set $\mathcal{D}$, but each instance has a certain membership degree. The *entropy* measure at a *node* from the *fuzzy* tree is computed as in Formula (2.5) and generalizes the *entropy* computation from the *crisp* case.

$$Entropy(node) = -\frac{m_+}{mm} \cdot \log \frac{m_+}{mm} - \frac{m_-}{mm} \cdot \log \frac{m_-}{mm} \tag{2.5}$$

where m_+ represents the sum of the membership degrees for the instances from $\mathcal{D}$ belonging to the *positive* class, m_- sums the membership degrees for the instances from $\mathcal{D}$ belonging to the *negative* class and mm is the sum of m_+ and m_-.

For computing the *information gain* of an attribute a with respect to the set of instances stored at an internal node *node* from the *fuzzy* tree, a kind of confusion matrix at that node is computed. We denote by F_+^a and F_-^a the *fuzzy* functions associated to attribute a and to the *positive* and *negative* class, respectively. By TP^{Fuzzy}, FP^{Fuzzy} and FN^{Fuzzy} we express the values which generalize (for the *fuzzy* case) the components of the confusion matrix for the crisp case. More exactly, these values are computed as follows:

- TP^{Fuzzy} sums the membership degrees for the instances i belonging to the *positive* class multiplied with the result of applying the function F_+^a on the value of attribute a in instance i.
- FN^{Fuzzy} sums the membership degrees for the instances i belonging to the *positive* class multiplied with the result of applying the function F_-^a on the value of attribute a in instance i.
- TN^{Fuzzy} sums the membership degrees for the instances i belonging to the *negative* class multiplied with the result of applying the function F_-^a on the value of attribute a in instance i.
- FP^{Fuzzy} sums the membership degrees for the instances i belonging to the *negative* class multiplied with the result of applying the function F_+^a on the value of attribute a in instance i.

We use the following notations:

- $m = TP^{Fuzzy} + TN^{Fuzzy} + FP^{Fuzzy} + FN^{Fuzzy}$.
- $p = TP^{Fuzzy} + FN^{Fuzzy}$.
- $pp = TP^{Fuzzy} + FP^{Fuzzy}$.

Using the previous notations, the new formula for the *information gain* measure is presented in Formula (2.6).

$$IG(node) = Entropy(node) - \frac{pp}{m} \cdot E_1 - \frac{m - pp}{m} \cdot E_2 \tag{2.6}$$

where

$$E_1 = -\frac{TP^{Fuzzy}}{pp} \cdot \log \frac{TP^{Fuzzy}}{pp} - \frac{FP^{Fuzzy}}{pp} \cdot \log \frac{FP^{Fuzzy}}{pp}$$

and

$$E_2 = -\frac{TN^{Fuzzy}}{m - pp} \cdot \log \frac{TN^{Fuzzy}}{m - pp} - \frac{FN^{Fuzzy}}{m - pp} \cdot \log \frac{FN^{Fuzzy}}{m - pp}.$$

2.2.4 Testing

After the *fuzzy decision tree* was trained (as described in Section 2.2.3.2), a new instance will be classified as shown in Section 2.2.2.1.

For evaluating the overall performance of the *FuzzyDT* model, a *leave-one out* cross-validation is used [110]. In the *leave-one out* (LOO) cross-validation on a data set with n software entities, the *FuzzyDT* model is trained on n-1 entities and then the obtained model is tested on the instance which was left out. This is repeated n times, for each entity from the data set.

During the cross-validation process, the confusion matrix [87] for the two possible outcomes (*non-defect* and *defect*) is computed. We are considering that the *defective* class is the *positive* one and the *non-defective* class is the *negative* one. The confusion matrix contains four values, the number of *True Positives* (TP), *True Negatives* (TN), *False Positives* (FP) and *False Negatives* (FN). For computing the values from the confusion matrix, we are using the known labels (classes) for the training instances.

Since the software defect prediction data are highly imbalanced (the number of *defects* is much smaller than the number of *non-defects*) the main challenge in software defect prediction is to obtain a large *true positive rate* and a small *false negative rate*. For defect predictors, the *accuracy* of the classifier (i.e. number of testing instances which were correctly classified - Formula (2.7)), is not a relevant evaluation measure, since the imbalanced nature of the data.

$$Acc = \frac{TP + TN}{TP + TN + FP + FN} \tag{2.7}$$

A more relevant evaluation measure for the performance of the software defect classifiers is the *Area Under the ROC Curve (AUC)* measure [36] (larger AUC values indicate better defect predictors). The AUC measure is usually used in case of approaches that output a single value which is transformed into a class label using a threshold. For such approaches, modifying the value of the threshold can lead to different values of the *Probability of detection* (Formula (2.8)) and the *Probability of false alarm* (Formula (2.9)) measures. For each threshold, the point (Pf, Pd) is represented on a plot, and AUC measures the area under this curve.

$$Pd = \frac{TP}{TP + FN} \tag{2.8}$$

$$Pf = \frac{FP}{FP + TN} \tag{2.9}$$

In case of approaches where the output is directly the class label, there is only one (Pf, Pd) point, which can be linked to the (0,0) and (1,1) points, and the area under this curve can be computed using Formula (2.10).

$$AUC = (1 - Pf) * Pd + \frac{Pf * Pd}{2} + \frac{(1 - Pf) * (1 - Pd)}{2} \tag{2.10}$$

2.2.5 Experimental evaluation

In this section we provide an experimental evaluation of the FuzzyDT model (described in Section 2.2.3) on two open-source software systems which were previously used in the software defect prediction literature. We mention that we have used our own implementation for FuzzyDT, without using any third party libraries.

Data set	Defects	Non-defects	Difficulty
JEdit	48	319	0.6667
Ant	166	579	0.5723

Table 2.5: Description of the data sets used for the experimental evaluation.

Data set	TN	TP	FN	FP	AUC	Accuracy
JEdit original	305	18	30	14	0.666	0.88
JEdit enhanced	289	27	21	30	0.734	0.86
Ant original	539	60	106	40	0.646	0.80
Ant enhanced	526	84	82	53	0.707	0.82

Table 2.6: Results of the experimental evaluation.

2.2.5.1 Case studies

For the experimental evaluation of the FuzzyDT model we have used two openly available data sets, created for two software systems written in Java: *JEdit* (version 4.2)[1] and *Ant* (version 1.7)[2]. Both data sets are available at [31]. Details about these two data sets can be found in Table 2.5.

The last column of Table 2.5 contains the *difficulty* of the data sets. This measure was introduced by Boetticher in [19] and is computed as the percentage of entities for which the nearest neighbor (ignoring the label of the entity when computing the distances) has a different label. Since our data sets are imbalanced, when computing the difficulty of the data sets we considered only the percentage of defective entities for which the nearest neighbor is non-defective.

For each data set that is used for the experimental evaluation, we will perform two experiments. In the first experiment we are going to use the data set without any modification, while in the second experiment we are going to enhance it by adding to the data set defective entities taken from a different software system. We are adding extra defective entities to reduce the imbalance in the data set. In the literature, two options are usually presented for adding more defective entities: over-sampling (when some defective entities are duplicated) and SMOTE (when new minority-class entities are created using the existing ones) [107]. We believe that using actual defective entities from a different project is better than creating synthetic entities.

For both data sets, we have added as *extra* defective entities, all the defective entities from the *Tomcat* data set, which is also available at the Tera-Promise repository [31]. Consequently, we have added 77 defective entities to both data sets, increasing the percentage of defective entities from 0.131 to 0.282 (for JEdit) and from 0.223 to 0.30 (for Ant).

2.2.5.2 Results

Table 2.6 contains the results of the experimental evaluation. As presented in the previous section, for each data set we have run the FuzzyDT model both for the original data set and the data set enhanced with the defective entities taken from the Tomcat system. We mention that these defective entities were used only for the training of the model, the testing was performed only on the entities from the JEdit and Ant systems.

Besides the AUC performance measure - computed with the Formula (2.10) - we have decided to add to Table 2.6 the entire confusion matrix to allow the computation of any performance measures for our approach, to facilitate the comparison of our results to other

[1] https://terapromise.csc.ncsu.edu/!/#repo/view/head/defect/ck/jedit/
[2] https://terapromise.csc.ncsu.edu/!/#repo/view/head/defect/ck/ant/ant-1.7

approaches. While we argued that accuracy is not a good performance measure in case of imbalanced data sets, we have decided to add it to Table 2.6 to show how different the values of this measure are compared to AUC.

2.2.6 Discussion

In this section we provide an analysis of our approach, as well as a comparison to similar approaches existing in the defect prediction literature.

2.2.6.1 Results analysis

Analyzing the results from Table 2.6 we can observe that, for both data sets, the AUC values are higher for the enhanced version than for the original one. This difference was illustrated on Figure 2.10 as well. We can also observe from Table 2.6 that the values for the *True positives* have increased for the enhanced version in both situations, which means that the number of *False negatives* has decreased. False negatives are defective entities that are classified as non-defective by the approach, and in case of defect detection, such errors are more serious than *False positives*, situations when non-defective entities are classified as defective. In the first case an error in the entity will be missed, while in the second case some time will be wasted to check an entity that contains no defects.

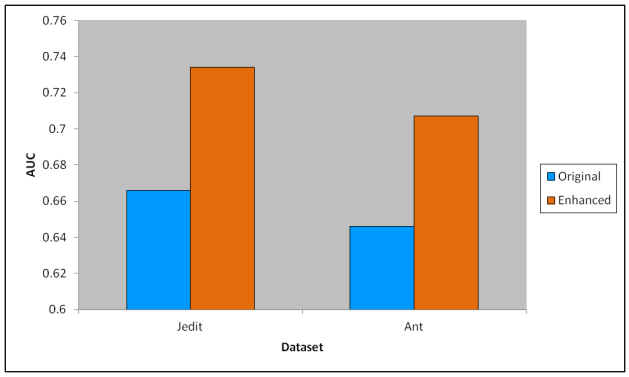

Figure 2.10: Comparison of the AUC values for the original and enhanced data sets.

We can also observe that the values for the accuracy measure are a lot higher than the ones for the AUC. This demonstrates that accuracy is not a suitable performance measure for imbalanced data sets, because we can have high accuracy in cases when only most of the majority class was correctly classified.

Unfortunately, the AUC values are not very high, but the reason for this is the difficulty of the data sets. As presented in the last column of Table 2.5, both data sets have really high values for the *difficulty* metric. The value 0.66 in case of the JEdit data set means that 66% of the defective entities from the data set have as nearest neighbor a non-defective entity.

2.2.6.2 Comparison to related work

In this section we compare the results for our approach to the results reported in the literature. We selected for comparison existing methods that use for the experimental evaluation the same data sets that we have used: *JEdit* and *Ant*. While for the *Ant* data set most existing approaches use version 1.7, in case of the *JEdit* data set there is not a version used in most existing related work.

Table 2.7 contains a comparison of the results for our approach and results achieved for other approaches reported in the literature. The first two lines of the table contain the results for our approach, both for the original and the enhanced data sets.

Approach	AUC - JEdit	AUC - Ant
FuzzyDT - original	0.666	0.646
FuzzyDT - enhanced	0.735	0.707
Weka - Decision Tree	0.520	0.629
Orange - Decision Tree	0.620	0.654
Multivariate Logistic Regression [65]	n/a	0.754
Logistic Regression - Weka	0.602	0.661
Multi-Objective [21]	(0.13, 0.33) 0.305	(0.51, 0.39) 0.641
	(0.18, 0.64) 0.346	(0.43, 0.77) 0.739
	(0.18, 0.54) 0.369	(0.43, 0.43) 0.633
Bayesian networks [82]	0.732	0.703

Table 2.7: Comparison of the results to similar approaches.

The next two lines contain the results achieved for the Decision Tree classifier using two openly available machine learning software: Weka [41] and Orange [83]. We want to mention that we have used the original data sets because in case of the enhanced data sets we could not find settings for using the extra defective entities only for the training and perform the leave-one-out cross validation on the original entities only. Also, we have taken from the results of these software systems only the confusion matrix, and computed the value of the AUC measure using our formula.

The fifth line contains the results reported in [65] using Logistic Regression with 10-fold cross validation on the Ant data set. The paper reports multiple performance measures, we have taken the sensitivity (which is equal to Pd) and specificity (which is $1 - Pf$) and used them to compute the value of the AUC measure, using the Formula (2.10). We have also run the Logistic Regression classifier from Weka on the data sets and computed the value of the AUC measure from the confusion matrix. These values are presented on the next line of Table 2.7.

The next three lines contain the results from [21], a multi-objective cross-project defect detection approach, which, instead of returning one single solution, computes a whole Pareto-front of solutions. In order to perform different comparisons Canfora et al. present in [21] some (precision, recall) pairs for their approach. From these values we have computed the confusion matrix and the value of the AUC measure. On each line, in front of the AUC value, we have given the (precision, recall) pair for which it was computed. In case of the JEdit system, [21] uses the 4.0 version, not 4.2 like we do. We have run our *Fuzzy DT* model on JEdit 4.0 as well, and we achieved an AUC value of 0.7, which is better than the results reported in [21].

The last line contains the results for the Bayesian networks, an approach introduced in [82]. We have used the Weka implementation for Bayesian networks to replicate the experiments presented in [82] and computed the value of the AUC measure from the confusion matrix.

Comparing the AUC values for approaches presented in the literature to our approach, we can observe that in case of the JEdit data set our approach with the enhanced data set has the highest AUC value. In case of the Ant system, our enhanced approach has the third highest value, but the difference between the first three AUC values are not very big.

For the JEdit data set, the Fuzzy DT for the original data set performed better than the related work in 6 cases out of 7 comparisons while the Fuzzy DT for the enhanced data set performed better in all 7 cases. For the Ant data set, out of 8 comparisons, the Fuzzy DT for the original data set provided better AUC values in only 3 comparisons, while the Fuzzy DT for the enhanced data set performed better in 6 cases. One can observe that, for both JEdit and Ant data sets, the Fuzzy DT for the enhanced data sets outperformed the Fuzzy

Data set	TN	TP	FN	FP	AUC	Accuracy
JEdit	292	25	23	27	0.718	0.864
Ant	535	73	93	44	0.682	0.816

Table 2.8: Results of the Fuzzy DT with over-sampling.

DT for the original data sets.

It is known that coping with imbalanced data sets can be done by under-sampling (removing elements from the majority class) and over-sampling (duplicating elements from the minority class) [23]. The process of creating the enhanced data sets used for the experimental evaluation of the Fuzzy DT is similar to over-sampling, but instead of duplicating existing instances we used defective instances from a different project. To demonstrate the advantage of this approach, we compared it to simple over-sampling on both the *JEdit* and *Ant* data sets. In order to make a fair comparison, for both data sets we have randomly selected 77 defective entities (the same number of defective entities were taken from the *Tomcat* project to create the enhanced data sets) and added them to the data sets. We tested the Fuzzy DT on these data sets, using a leave-one-out validation, where the extra entities were used only for the training step, but not for testing. The obtained results are shown in Table 2.8.

Comparing the AUC values from Tables 2.6 and 2.8 we can conclude that coping with imbalance through adding existing instances from a different project leads to an increased performance compared to simple over-sampling. The result of this is illustrated in Figure 2.11. The AUC values for each data set (JEdit and Ant) are depicted using two bars: the first one corresponds to our Fuzzy DT using simple over-sampling on the data set and the second bar represents our proposal of Fuzzy DT for the enhanced data set. Further investigations will be made in order to consider other methods for handling the imbalance nature of the data sets.

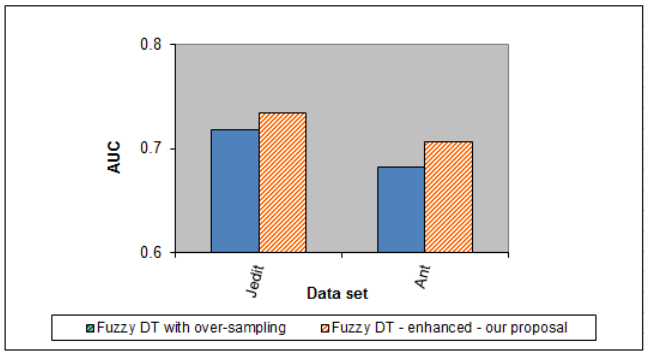

Figure 2.11: Comparison to simple over-sampling.

2.2.7 Conclusions and future work

A fuzzy decision tree model has been introduced for predicting, in a supervised manner, those entities from software systems which are likely to be defective. The experimental evaluation which was performed on two open-source software systems provided results better than most of the similar existing approaches and highlighted a very good performance of the proposed approach . Much more, the *fuzzy* decision tree approach proved to outperform, for the considered case studies, the crisp DT approach.

Further work will be carried out in order to extend the experimental evaluation of the *fuzzy* decision tree approach proposed in this paper. We also aim to investigate a hybridization between the *fuzzy* DT model and *relational association rules* [94], since we are confident that

relations between the values for different software metrics would be relevant in discriminating between defective and non-defective software entities.

Chapter 3

Software packages refactoring using a hierarchical clustering-based approach

The structure of a software system is the subject of many changes during the system lifecycle and it has a major impact on the maintainability of the system. Improper implementations of these changes often imply structure degradation that leads to costly maintenance, this is why continuous software refactoring is beneficial.

Fowler defines in [37] refactoring as "the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure. It is a disciplined way to clean up code that minimizes the chances of introducing bugs".

In the original paper [74] we have approached the problem of software refactoring at the package level and we proposed a clustering-based approach, that would help developers to group application classes from an existing software system into appropriate packages. Clustering [57] is a well known unsupervised learning technique that is very useful in detecting hidden patterns in data. Our approach takes an existing software system and re-modularizes it at the package level using hierarchical clustering, in order to obtain better-structured packages. Considering a certain structure of packages from a software system, the method proposed in this chapter would also be useful for suggesting the developer the appropriate package for a newly added application class.

It is well-known that different software systems can have different architectures [13], and the architecture of the system influences how classes should be divided into packages. Although there is the general "low coupling, high cohesion" rule [108], in case of layered or multitier architectures, for example, classes from different layers have dependencies between them, and they should not be placed in the same package. Thus, when developing a method for grouping classes into packages, we have to consider the architecture of the system. In this paper we will focus on grouping classes into packages in case of frameworks (systems with many abstract classes and interfaces).

The rest of the chapter is structured as follows. Section 3.1 emphasizes the relevance of the problem of software re-modularization at the package level and also gives a motivation of our approach. The fundamentals of clustering, as well as a survey on existing approaches in the software engineering literature in the direction of automatic software packages restructuring are presented in Section 3.2. Section 3.3 introduces the clustering-based approach we propose for software packages restructuring. Section 3.4 provides an experimental evaluation of our approach. An analysis of the method proposed in this paper as well as a comparison with existing similar approaches are given in Section 3.5.

3.1 Motivation

Software systems have become increasingly complex and versatile [51], that is why in order to make them simple to maintain and evolve, it is very important to continuously refactor the code. There is a continuous interest in applying data mining [118], [7] methods in software engineering as mining techniques can support several aspects of the software development life-cycle, such as *software quality* [91].

Refactoring is adopted by the modern software development methodologies, such as extreme programming and other agile methodologies, as a solution for keeping the software structure clean and easy to maintain. Refactoring becomes an integral part of the software development cycle: developers alternate between adding new tests and functionality and refactoring the code to improve its internal consistency and clarity [95].

Nowadays, the software systems are becoming more and more complex, consisting of thousands of application classes which are grouped into software packages. Moreover, they evolve over time and have many releases, which are resolving new functional requirements or are due to technological improvements. Without an appropriate package structure of the software, the system becomes hard to maintain, since its structure may be deteriorated. Thus, *software restructuring* at the package level is an important process in software maintenance and evolution. The cost of software maintenance increases with the complexity of the systems, therefore it is very hard for software developers to decide the appropriate software package in which a newly added application class has to be placed. When the number of application classes is large the class assignment decision is not an easy one, since it involves a good knowledge of the overall system design.

The problem of software packages restructuring arises from practical needs, thus the approach proposed in this paper can be useful for assisting software developers in their daily works of refactoring packages in software systems. Consequently, the approach we propose in this paper would be effective for software developers in assisting them during maintaining complex software systems, as well as through the software evolution [54].

3.2 Background

In this section we present the main aspects related to the *clustering* problem, as well as a literature review on the problem of automatic software packages restructuring.

3.2.1 Clustering

Clustering [43] is a data mining activity that aims at partitioning a set of data (or objects) in a set of meaningful sub-classes, called *clusters*, being considered the most important *unsupervised learning* problem. The resulting subsets or groups, distinct and non-empty, are to be built so that the objects within each cluster are more closely related to one another, than objects assigned to different clusters. Central to the clustering process is the notion of degree of similarity (or dissimilarity) between the objects.

Let $\mathcal{O} = \{O_1, O_2, \ldots, O_n\}$ be the set of objects to be clustered. The measure used for discriminating objects can be any *metric* or *semi-metric* function $d : \mathcal{O} \times \mathcal{O} \longrightarrow \Re$. The distance expresses the dissimilarity between objects.

In this paper we are focusing only on *hierarchical* clustering [35], that is why, in the following, a short overview of the hierarchical clustering methods is presented.

Hierarchical clustering methods represent a major class of clustering techniques [49]. There are two styles of hierarchical clustering algorithms. Given a set of n objects, the agglomerative (bottom-up) methods begin with n singletons (sets with one element), merging them until a single cluster is obtained. At each step, the most similar two clusters are

chosen for merging. The divisive (top-down) methods start from one cluster containing all n objects and split it until n clusters are obtained.

The agglomerative clustering algorithms that were proposed in the literature, differ in the way the two most similar clusters are calculated and the linkage-metric used (single, complete or average). Single link algorithms merge the clusters whose distance between their closest patterns is the smallest. Complete link algorithms, on the other hand, merge the clusters whose distance between their most distant patterns is the smallest [49]. In general, complete link algorithms generate compact clusters, while single link algorithms generate elongated clusters. Complete link algorithms are generally more useful than single link algorithms. Average link algorithms merge the clusters whose average distance (the average of distances between the objects from the clusters) is the smallest. Average link clustering is a compromise between the sensitivity of complete-link clustering to outliers and the tendency of single-link clustering to form long chains that do not correspond to the intuitive notion of clusters as compact, spherical objects [71].

3.2.2 Software remodularization at the package level. Literature review

In the literature, there are several different methods reported for identifying how classes should be grouped into packages. One such method is presented in [9], where clustering is used to find the ideal grouping of classes. The authors present several methods and experimental results of these methods. In the first method they keep the current package structure of the software system, but they check if there are classes which should be moved from one package to another. For each class they count the number of initializations of that class in the existing packages. If a class has a higher number of initialization in a different package than its own, it is suggested to be moved to that one.

The second method presented in [9] uses a vector-based representation of the classes, i.e. every class from the software system is represented as a multidimensional vector. The length of the vector represents the total number of methods in the system and the values in the representation of the classes are the number of calls to the given method. For two such vectors a dissimilarity coefficient is defined, which will be used in the clustering process. They use a hierarchical clustering algorithm and experiment with different linkage-metrics: single, complete and average. They also present a novel clustering algorithm, called *Adaptive k-Nearest Neighbour (A-KNN) Clustering*, which gives similar results to regular clustering algorithms, but has lower complexity.

Another method that tries to automatically divide classes into packages is the one presented in [84]. In this method, software networks are used to represent classes from a software system and their dependencies. They use two kind of dependencies, method accessing attribute and method call dependencies. Dependencies between classes are defined based on these two types of dependencies: if a method from a class accesses an attribute or calls a method from a different class, a dependency is formed between the two classes. On the networks built based on these dependencies a constrained community detection algorithm is applied, which will identify the optimized community structures, that correspond to the ideal package structure.

Even if it can not restructure a whole system, the method presented by Bavota et al. in [14] can divide a package, which has a low cohesion, into several more cohesive packages. They measure cohesion considering both structural and semantic relationships among the classes. Thus, for every pair of classes, they compute a likelihood that the two classes should be together in a package, using a structural software metric (Information-Flow Based Coupling) and a semantic one (Conceptual Coupling Between Classes). Using these likelihoods they extract chains of classes from the package which should form separate packages.

Another direction of research that should be mentioned, is the definition of different metrics, which measure the quality of packages in a software system. Such metrics are defined in

[92], that measure different aspects of the division into packages: coupling, not programming to interfaces, size, system extensibility, API cohesiveness and segregation, common use of method classes, a total of 14 metrics. The only disadvantage of these metrics is that they require the explicit definition of APIs for the packages, but in most systems, such APIs are not defined. Another set of metrics, which overcomes this disadvantage by automatically considering as part of the API those classes that interact with classes from different packages, is presented in [33]. They introduce three coupling metrics and two cohesion metrics defined for packages. These metrics are based on two types of dependencies: *extend* dependencies and *use* dependencies (defined as method call or attribute access). Some metrics are actually pairs, containing one metric defined for *extend* dependencies and one for *use* dependencies (for example: *Index of Inter-Package Usage* and *Index of Inter-Package Extending*).

3.3 Methodology

In this section we introduce the clustering-based approach (*CASP - Clustering Approach for Software Packages Restructuring*) for software re-modularization at the package level.

CASP approach consists of two steps:

- **Data collection** - The existing software system is analyzed in order to extract from it relevant information about application classes, methods, attributes and the existing relationships between them: inheritance relations, aggregation relations, dependencies between the entities from the software system. These information can be extracted from existing documents of the software system, like: source code, byte code, UML diagrams, or other documents that may provide the needed information. All these collected data will be used in the **Grouping** step of our approach.
- **Grouping** - The set of classes from the software system, considering the relevant information extracted at the previous step, are grouped in clusters (packages) using a clustering algorithm (*HASP* in our approach). The goal of this step is to obtain a partitioning of the software system into packages (each cluster from the obtained partition corresponds to a software package).

In the following, we introduce a theoretical model on which our clustering approach is based and a more detailed description of *CASP*.

3.3.1 Theoretical model

Let $S = \{s_1, s_2, ..., s_n\}$ be a software system, where $s_i, 1 \leq i \leq n$ represents an application class from the software system.

Let us consider that:

- Each application class s_i $(1 \leq i \leq n)$ is a set of methods and attributes, i.e. $s_i = \{m_{i1}, m_{i2}, \ldots, m_{ip_i}, a_{i1}, a_{i2}, \ldots, a_{ir_i}$, where m_{ij} $(\forall j,\ 1 \leq j \leq p_i)$ are methods and a_{ik} $(\forall k,\ 1 \leq k \leq r_i)$ are attributes from the application class s_i.
- $Meth(S) = \bigcup_{i=1}^{n} \bigcup_{j=1}^{p_i} m_{ij}$, $Meth(S) \subset \bigcup_{i=1}^{n} s_i$, is the set of methods from all the application classes of the software system S.
- $Attr(S) = \bigcup_{i=1}^{n} \bigcup_{j=1}^{r_i} a_{ij}$, $Attr(S) \subset \bigcup_{i=1}^{n} s_i$, is the set of attributes from the application classes of the software system S.

At the **Grouping** step of our approach, the application classes from software system S have to be re-grouped. This re-grouping is represented as a ***partition*** of S.

Partition into packages of a software system S.

The set $\mathcal{K} = \{K_1, K_2, ..., K_v\}$ is called a **partition into packages** of the software system $S = \{s_1, s_2, \dots, s_n\}$ iff

- $1 \leq v \leq n$;
- $K_i \subseteq S, K_i \neq \emptyset, \forall i,\ 1 \leq i \leq v$;
- $\bigcup_{i=1}^{n} s_i = \bigcup_{i=1}^{v} K_i$ and $K_i \cap K_j = \emptyset,\ \forall i, j, 1 \leq i, j \leq v, i \neq j$.

In a partition $\mathcal{K} = \{K_1, K_2, ..., K_v\}$ of the software system S a cluster K_i represents a software package (group of application classes).

3.3.2 Grouping into packages

In the following we introduce a novel hierarchical agglomerative clustering algorithm (*HASP - Hierarchical Clustering Algorithm for Software Packages Restructuring*), which aims at identifying a **partition** of a software system S, that corresponds to a good structure of packages of the software system. Since the architecture of a software system is an important factor when deciding which classes should be placed in the same package, it is complicated to create a universal model, that works for every architecture. In this paper we are focusing on identifying a good structure of packages for a framework, consequently, the *HASP* algorithm will be suitable for such systems.

In our clustering-based approach, the objects to be clustered are the application classes from the software system S, i.e. $\{s_1, s_2, \dots, s_n\}$. Our focus is to group application classes from S into packages (cluster). In the following, when referring to a cluster, we are considering a group of application classes (a possible software package).

Based on the considerations above, we are going to associate in the following a *score* to a group G of application classes (group that may represent a possible package) in a software system S. This *score* will give a measure of how "good" the software package consisting of the application classes from G is, also considering the other packages from the software system S.

Let us consider that $\mathcal{K} = \{K_1, K_2, ..., K_v\}$ is a partition of the software system S representing a current partitioning into packages of the system. Since we are going to apply a hierarchical clustering-based approach, we have to decide at a given moment in the clustering process to merge into a single cluster two clusters K_i and K_j from the current partition $\mathcal{K}$. Thus, we aim at defining a numerical value, denoted by *score* (Formula (3.1)), indicating the "importance" of the software package obtained by merging the clusters K_i and K_j with respect to the remaining packages from the partition $\mathcal{K}$. At a given moment in the hierarchical clustering process, we will merge the pair of clusters that have the maximum associated score.

3.3.2.1 Selected features

In order to obtain a good package structure, we try to capture important characteristics of a good package: high cohesion, low coupling, high reuse potential. Seven features $F_1, F_2, F_3, F_4, F_5, F_6, F_7$ were identified to be relevant in characterizing how "good" is the software package $K_i \bigcup K_j$ related to the rest of the packages from the partition $\mathcal{K}$. An important part for computing

most of these features is based on the notion of "dependency" (between classes, packages). For a given class C, we consider that the list of classes C depends on is formed by:

(1) All the interfaces implemented by C and the class extended by C, if they exist.

(2) The types of the attributes in class C.

(3) For every method m from class C we consider:

   a. The type of the parameters for method m.
   b. The type of the returned value for method m.
   c. Classes whose attributes are directly accessed in the method m.
   d. Types of the local variables created/used in m.
   e. Classes from which methods are called in m.

We have also considered that not every type of dependency is equally important when grouping the classes into packages, so we have decided to weight these dependency types differently, according to their importance: attribute type (item (2) - strongest), inheritance (item (1)), method/attribute access (item (3c) - weakest). The weights used in our approach for these three dependencies are 6, 4 and 2, respectively. All other dependency types are equally weighted.

Let us denote by $\mathcal{K}^* = \mathcal{K} \setminus \{K_i, K_j\}$ the partition $\mathcal{K}$ without the packages K_i and K_j. Therefore, $score(K_i \bigcup K_j, \mathcal{K}^*)$ depends on the following features:

1. **Feature F_1 - Package cohesion.** This feature counts the number of dependencies between the classes in package $K_i \bigcup K_j$. For every class C from this package we compute the list of classes it depends on and we count how many of these classes are in package $K_i \bigcup K_j$. For this feature we count a dependency multiple times if it appears more than once.

2. **Feature F_2 - Package reuse.** This feature counts the number of the packages from the rest of the system, $\mathcal{K}^*$, depending on the package $K_i \bigcup K_j$ (on at least one class).

3. **Feature F_3 - Package coupling.** This feature counts the number of packages from $\mathcal{K}^*$ on which classes from $K_i \bigcup K_j$ depend on.

4. **Feature F_4 - Name cohesion.** This feature measures the similarity between names of the classes from package $K_i \bigcup K_j$.

5. **Feature F_5 - Dependency similarity.** This feature measures how similar the classes from $\mathcal{K}^*$ on which the application classes of $K_i \bigcup K_j$ depend on are.

6. **Feature F_6 - General coupling.** This feature counts the percentage of pairs of classes (one class from K_i and one from K_j) which have a dependency (in either direction).

7. **Feature F_7 - General name coupling.** This feature computes the percentage of methods with similar names for every pair of classes (one from K_i and one from K_j).

3.3.2.2 A simple example

In order to better understand how these features are computed, in the following, we will present a short, simple example, a system consisting of seven classes, part of an online bookstore. The source code is presented on Figure 3.1, and the list of dependencies for every class from Figure 3.1 is given in Table 3.1.

Let us consider that at a given step in the clustering process, the current partition contains the following four clusters:

```
public abstract class AbstractBook{
   protected String title;
   protected String author;
   protected Integer year;

   public String getDescription(){
      return title + ":" + author+
           + "(" + year + ")";
   }
   // simple constructor with fields as
   // parameters, getters and
   // setters for fields
}

public class AudioBook extends
         AbstractBook{
   private String reader;

   public String getDescription(){
      return super.getDescription() +
        " - AudioBook (" + reader + ")";
   }
   // simple constructor with fields as
   // parameters, getter and
   // setter for reader
}

public class EBook extends
        AbstractBook{
   private String format;
   private boolean blackAndWhite;
   private Integer size;

   public String getDescription(){
      return super.getDescription() +
        " - EBook (" + format + ")";
   }
   // simple constructor with fields as
   // parameters, getters and setters
   // for fields
}

public class PaperBook extends
       AbstractBook{
   private String type;
   private Integer weight;

   public String getDescription(){
      return super.getDescription() +
           " - " +type;
   }
   // simple constructor with fields as
   // parameters, getters and setters
   // for fields
}

public abstract class EBookReader{
   protected List<String> supportedFormats;
   protected String resolution;
   protected String model;

   public abstract boolean supports(EBook b);
  // simple constructor with fields as
  // parameters, getters and setters
  // for fields
}

public class ColorReader extends
        EBookReader{

   public boolean supports(EBook b){
      if(supportedFormats.contains(
             b.getFormat()))
        return true;
      else
        return false;
   }
   // simple constructor with fields as
   // parameters
}

public class BlackAndWhiteReader
       extends EBookReader{

   public boolean supports(EBook b){
      if(!(supportedFormats.contains(
           b.getFormat())))
         return false;
      if(book.isBlackAndWhite())
        return true;
      return false;
   }
   // simple constructor with fields as
   // parameters
}
```

Figure 3.1: Simple Code Example

Class name	Dependencies
AbstractBook	$\emptyset$
AudioBook	AbstractBook (6 times)
EBook	AbstractBook (6 times)
PaperBook	AbstractBook (6 times)
EBookReader	EBook
ColorReader	EBookReader(5 times), EBook(2 times)
BlackAndWhiteReader	EBookReader(5 times), EBook(3 times)

Table 3.1: Dependencies for classes from Figure 3.1

.

- C_1 : EBookReader, ColorReader, BlackAndWhite-Reader
- C_2 : AbstractBook, AudioBook
- C_3 : PaperBook
- C_4 : EBook

The values of the seven features, presented in the previous Section, for the pairs of clusters C_1 - C_4, $C_2 - C_4$ and $C_3 - C_4$ are presented in Table 3.2. For the first pair, C_1 - C_4, consisting of classes EBookReader, ColorReader, BlackAndWhiteReader, respectively EBook, the values are computed in the following way:

- F_1. EBookReader has a single dependency on EBook, ColorReader has five dependencies on EBookReader and two on EBook, BlackAndWhiteReader has five dependencies on EBookReader and three on EBook, and EBook has no dependencies (from the C_1 - C_4 package). This is a total of 16 dependencies.
- F_2. There is no package in the system which uses classes from the C_1 - C_4 package, so this value is 0.
- F_3. The only package on which classes from C_1 - C_4 depend on is C_2, so this value is 1.
- F_4. For this feature we consider every pair of classes and count the common words in their names: EBookReader - ColorReader (1 word), EBookReader - BlackAndWhiteReader (1 word), EBookReader - EBook (2 words, E is a separate word), ColorReader - BlackAndWhiteReader (1 word), ColorReader - EBook (0 words), BlackAndWhiteReader - EBook (0 words). In total, there are 5 common words in 6 pairs, so this value is 0.83.
- F_5. There is no class on which all classes from C_1 - C_4 depend on, so this feature has the value of 0.
- F_6. There is a dependency between every pair of classes, when one class comes from C_1 and the other from C_4, so the value of this feature is 1.
- F_7. There is no common method for the pairs of classes, considered as for F_6, so this value is 0.

Pair of clusters	F_1	F_2	F_3	F_4	F_5	F_6	F_7
C_1 - C_4	16	0	1	0.83	0	1	0
C_2 - C_4	12	2	0	1	0	0.5	0.094
C_3 - C_4	0	1	1	1	1	0	0.091

Table 3.2: Values of features for three cluster pairs

.

$$score(K_i \bigcup K_j, \mathcal{K}^*) = \frac{\sum_{i=1}^{2} w_i \cdot F_i - w_3 \cdot F_3}{|K_i \bigcup K_j|^2 - 1} + \sum_{i=4}^{7} w_i \cdot F_i \tag{3.1}$$

3.3.2.3 Score computation

After the relevant features were identified, the score $score(K_i \bigcup K_j, \mathcal{K}^*)$ of the software package $K_i \bigcup K_j$ is defined as a linear function on these features, as given in Formula (3.1).

When defining this score we have first started from the well-known principle, that packages should have high cohesion and low coupling. We also wanted to consider many different types of dependencies between classes, not just initialization, method call, attribute access like some other methods do. Also, when considering coupling, which should be low, for a given package we differentiate between the package being used by some other package (feature F_2) and the package using another package (feature F_3). From the perspective of a given package, we consider that the former is "good coupling" - when you create a package, you want other packages to use it -, while the latter is "bad coupling" - you want a package to be as independent as possible. Also, we have noticed that aiming just for high cohesion and low coupling will often result in some big packages with absolutely no connection between them (0 being the lowest possible coupling). This is why we introduced the rest of the features, to measure the similarity of the names of the classes, how common the used classes are, how many relations are between the classes of the two packages to be merged and so on.

In Formula (3.1) w_i ($0 \leq w_i \leq 1$) is the weight associated to the feature F_i ($\forall\ 1 \leq i \leq 7$). In order to obtain good values for the weights a grid search procedure [16] will be used (details are given in Section 3.4.1). In defining the score we started from the intuition that in order to obtain good packages the values for the features $F_1, F_2, F_4, F_5, F_6, F_7$ have to be maximized (since they express the cohesion between K_i and K_j), the value for F_3 has to be minimized (since it expresses the coupling between K_i and K_j to the rest of the packages), without favoring packages with a large number of classes.

HASP is based on the idea of hierarchical agglomerative clustering. At a given step, the pair of clusters that have the maximum associated *score* are merged. This means (considering the way the score was defined) that the application classes from the two clusters (packages) are cohesive enough in order to be placed in the same cluster.

The agglomerative hierarchical clustering process is performed until a single cluster is obtained. From all the generated partitions, we need to identify the one that is likely to be the "best" partitioning of the software system S into software packages. For this, we are going to assign an *overall score* to a partition of a software system (set of software packages), score that has to be maximized in order to obtain a better partitioning.

Let us consider a partition $\mathcal{K} = \{K_1, K_2, ..., K_v\}$ of a software system S, where K_i represents a software package. The *overall score* associated to partition $\mathcal{K}$ is defined in Formula (3.2).

$$overallScore(\mathcal{K}) = \frac{\sum_{i=1}^{v} sc(K_i, \mathcal{K})}{v} \tag{3.2}$$

where $sc(K_i, \mathcal{K})$ expresses how well structured is the package K_i within the partition $\mathcal{K}$ and is defined as in Formula (3.3).

$$sc(K_i, \mathcal{K}) = \frac{\sum_{i=1}^{2} w_i \cdot F_i - w_3 \cdot F_3}{|K_i|^2 - 1} + \sum_{i=4}^{5} w_i \cdot F_i \tag{3.3}$$

In Formula (3.3) F_i $(1 \leq i \leq 5)$ are the features that were described above and w_i $(1 \leq i \leq 5)$ are the weights that are associated to these features. We mention that features F_6 and F_7 (from Formula (3.1)) are not considered in Formula (3.3), since these features are characterizing two packages (that will be merged during the clustering process), not a single one.

3.3.2.4 The HASP algorithm

The main steps of the *HASP* algorithm are:

- Each application class from the software system is put in its own cluster (package).
- The following steps are repeated until a single cluster is obtained in the partition:
  - For the current partition, the corresponding *overallScore* is computed.
  - Select the pair of clusters (K_i, K_j) from the current partition $\mathcal{K}$ that maximize the value $score(K_i \bigcup K_j, \mathcal{K}^*)$ (Formula (3.1)) and merge these clusters.

In order to identify the most appropriate np number of clusters, the following analysis is performed. We consider that $K_1, K_2, \ldots K_n$ (n is the number of application classes from the software system) are the partitions generated by the *HASP* algorithm, where K_i represents the partition with i clusters $(1 \leq i \leq n)$. The sequence $os = (os_1, os_2, \ldots, os_n)$ where $os_i = overallScore(K_i)$, is analyzed as follows:

- The local maxima $os_{i_1}, os_{i_2}, \ldots, os_{i_k}$ from the os sequence are computed. For each local maximum os_{i_j} $(1 \leq j \leq k)$ the local minima, that are nearest the local maximum in the sequence before and after it, are computed. Now, we associate to os_{i_j} a value $val(i_j)$ computed as the difference between the local maximum and the mean of the two local minima.
- The position of the maximum value from the sequence $val(i_1), val(i_2), \ldots, val(i_n)$ is considered to be the most appropriate number np of software packages, i.e. $np = argmax_{j=1,n}(val(i_j))$.

With the number of clusters identified using the analysis above, the solution reported by the *HASP* algorithm is the partition containing np clusters, i.e. $\mathcal{K}_{np}$.

We give next the *HASP* algorithm.

`Algorithm HASP is`
Input:
`- the software system` $\mathcal{S} = \{s_1, \ldots, s_n\}, n \geq 2$

Output:
- $\mathcal{K}^{optimal} = \{K_1, K_2, ..., K_v\}$, the partition of packages in $\mathcal{S}$.
Begin
  For $i \leftarrow 1$ to n do
    $K_i \leftarrow \{s_i\}$ //each class is put in its own cluster
  endfor
  $\mathcal{K}[n] \leftarrow \{K_1, \ldots, K_n\}$ //the initial partition
  $nc \leftarrow n$
  //nc is the number of clusters in the current partition
  While $nc > 1$ //until a single cluster is obtained
    $maxScore \leftarrow 0$ //the maximum score
    For $i^* \leftarrow 1$ to $|\mathcal{K}|$-1 do
      For $j^* \leftarrow i^* + 1$ to $|\mathcal{K}|$ do
        $\mathcal{K}^* \leftarrow \mathcal{K}[nc] \setminus \{K_{i^*}, K_{j^*}\}$
        $s \leftarrow score(K_{i^*} \bigcup K_{j^*}, \mathcal{K}^*)$
        If $s > maxScore$ then
          $maxScore \leftarrow s$
          $i \leftarrow i^*$
          $j \leftarrow j^*$
        endif
      endfor
    endfor
    $K_{new} \leftarrow K_i \cup K_j$
    $nc \leftarrow nc - 1$
    $\mathcal{K}[nc] \leftarrow (\mathcal{K}[nc+1] \setminus \{K_i, K_j\}) \cup \{K_{new}\}$
    endif
  endwhile
  @ determine the number np of clusters
  $\mathcal{K}^{optimal} \leftarrow \mathcal{K}[np]$
End.

3.3.3 Assigning application classes to packages

In the following we aim at proposing an algorithm that will provide the appropriate software package for a newly added application class.

Let us consider that $\mathcal{K} = \{K_1, K_2, ..., K_v\}$ is the actual partition into packages of the software system S. A new application class C is added to the system. In order to decide what is the most appropriate software package into which C should be added, we are reasoning as follows. For each software package K_i $(1 \leq i \leq v)$ we compute the score $score(K_i \bigcup \{C\}, \mathcal{K} \setminus \{K_i\})$ obtained by adding application class C to software package K_i. The maximum obtained score will give us the software packages to which application class C should be assigned.

The algorithm is given below.

Algorithm AssignClass is
Input:
- the partition $\mathcal{K} = \{K_1, K_2, ..., K_v\}$ of S
- the application class C
Output:
- rez $(1 \leq rez \leq v)$.
Begin
  $maxScore \leftarrow 0$ //the maximum score

```
For i ← 1 to v do
   s ← score(K_i ⋃{C}, 𝒦 \ {K_i})
   If s > maxScore then
     maxScore ← s
     rez ← i
   endif
endfor
//K_rez is the suggested package for class C
```
End.

In some cases it is possible that the new class should be added into a newly created package. Although the current version of our algorithm does not consider this case, it is possible to add a threshold value t, and report K_{rez} as a solution only if $maxScore > t$, otherwise return $v + 1$ suggesting that a new package should be created. We intend to add this improvement to the future version of the algorithm.

3.4 Experimental evaluation

In our experiments we will consider two open source software systems that will be restructured in packages using the $CASP$ approach introduced in this paper. The reasons for choosing these two software systems are the following:

- They both are frameworks, which is important, because our method was designed to restructure into packages the application classes from a framework.
- They are openly available.
- They were written in Java, and our current implementation of the *Data Collection* step analyzes systems written in Java.
- They have a relatively small number of classes, which allows manual verification and analysis.

3.4.1 Parameters tuning

The optimization of the weights $w_1, w_2, \cdots w_7$ used for computing the *score* (Section 3.3.2) of a software package within a software system is performed by a grid search method on a validation set. Even if it is a method usually used for optimizing parameters of a supervised learning method, we are using the grid search method within an unsupervised learning scenario. As validation set we use a software framework that was introduced in [29] in order to solve combinatorial optimization problems using reinforcement learning [100]. This framework was used in [18, 28, 27] for solving with reinforcement learning several optimization problems. For this software system a good partition into packages is known.

The grid search [16] makes repeated trials for each parameter across a specified interval. For each combination of these parameters, the *HASP* algorithm (Section 3.3.2) is applied on the software system used as validation set. The grid search procedure is guided by the CIP evaluation measure (Section 3.4.2), which expresses how close is a partition to the known partitioning, and thus has to be maximized. Consequently, we are searching for values for the weights which are leading to the partition that is the most similar to the known partition into packages.

In the grid search procedure, we are using the following sequences for the weights: $w_i = (0.2, 0.2 + 1 \cdot 10^{-2}, 0.2 + 2 \cdot 10^{-2}, \ldots, 0.8)$ for $i = 1, 2, 3, 4, 5, 6, 7$.

The best values for the weights obtained through the grid search procedure are: $w_1 = 0.22$, $w_2 = 0.25$, $w_3 = 0.2$, $w_4 = 0.52$, $w_5 = 0.72$, $w_6 = 0.3$, $w_7 = 0.36$. These values will be used in the experiments that will be presented below.

We have performed in [73] a study in order to evaluate how well packages in a software system are structured using the *overallScore* evaluation measure and the weights that were obtained above. Three open source software frameworks [1, 2, 3] were considered for experimentation, and the conclusions of the study were that *overallScore* measure is capable of differentiating between a good and a bad partitioning into packages of a software system, which did not always happen in case of the other metrics taken from the literature. Moreover, *overallScore* has proven to be strongly positively correlated with the good structure of packages from the considered software systems.

3.4.2 Evaluation measure

For evaluating how accurate are the partitions obtained by the $HASP$ algorithm in comparison with a given structure of packages of the software system (considered to be a good structure of packages), we use the CIP evaluation measure.

In the following, let us consider a software system S, a partition $\mathcal{K} = \{K_1, \ldots, K_v\}$ (in our case provided by $HASP$ algorithm) and $\mathcal{K}^{good} = \{K_1^{good}, \ldots, K_q^{good}\}$ (a good structure of packages that is apriori-known).

Cohesion of Identified Packages - CIP.

The cohesion of the software packages from $\mathcal{K}^{good}$ in the partition $\mathcal{K}$, denoted by $CIP(\mathcal{K}^{good}, \mathcal{K})$, is defined as: $CIP(\mathcal{K}^{good}, \mathcal{K}) = \frac{1}{q} \sum_{i=1}^{q} cip(K_i^{good}, \mathcal{K})$. $cip(K_i^{good}, \mathcal{K})$ is the cohesion of package K_i^{good} in partition $\mathcal{K}$ and is defined as: $cip(K_i^{good}, \mathcal{K}) = \frac{\sum_{k \in M_{K_i^{good}}} \frac{|K_i^{good} \cap k|}{|K_i^{good} \cup k|}}{|M_{K_i^{good}}|}$, where $M_{K_i^{good}}$ is defined as: $M_{K_i^{good}} = \{k \mid k \in \mathcal{K},\ K_i^{good} \cap k \neq \emptyset\}$.

For a given software package $p \in \mathcal{K}^{good}$, $cip(p, \mathcal{K})$ defines the degree to which the application classes from the software package p belong together in clusters from the partition $\mathcal{K}$.

It can be easily proven that, if $\mathcal{K}$ is a partition of the software system S and $\mathcal{K}^{good}$ is a good structure of packages of S, then the following inequality holds: $0 \leq CIP(\mathcal{K}^{good}, \mathcal{K}) \leq 1$.

Larger values for CIP indicate better partitions with respect to $\mathcal{K}^{good}$, meaning that CIP has to be maximized. If $CIP(\mathcal{K}^{good}, \mathcal{K}) = 1$, it means that $\mathcal{K}$ is the *optimal* partition, as it coincides with the good structure, $\mathcal{K}^{good}$, of packages.

3.4.3 Experiments

For each software system S considered for evaluation, two experiments are performed. The way these experiments are performed is described below.

Experiment 1

During the first experiment the $HASP$ clustering algorithm introduced in Section 3.3.2 is applied in order to obtain a structure of packages that will be compared against the actual structure of S. The experiment is conducted as follows:

- First, at the *Data Collection* step, the relevant information from the software S is extracted. The evaluated software system is written in Java. In order to extract from the systems the data needed in the *Grouping* step of our approach (Section 3.3) we use ASM 3.0 [11], a Java bytecode manipulation framework. We use this framework in order to extract the structure of the systems (attributes, methods, classes and relationships between all these entities).

- The application classes are grouped in clusters using $HASP$ algorithm and a partition $\mathcal{K} = \{K_1, K_2, ..., K_v\}$ of S is provided. The obtained partition represents a structure of packages in the software system S.

- In order to evaluate the "quality" of the partition $\mathcal{K}$ reported by our algorithm, it is compared with a good partitioning $\mathcal{K}^{good}$ into packages that is apriori-known. We expect $\mathcal{K}$ to be nearly identical to $\mathcal{K}^{good}$. In order to capture the similarity of the two partitions (the one obtained by *HASP* algorithm and the original/known one) the *CIP* evaluation measure is used.

Experiment 2

During the second experiment the *AssignClass* algorithm introduced in Section 3.3.2 is applied in order to identify the software package from the system S where a newly added application class should be placed. The experiment will be detailed for each considered case study.

3.4.3.1 *Commons DbUtils* framework

The first case study considered for evaluation is an open source software framework, *Commons DbUtils* (version 1.5), a library consisting of a small set of classes, which are designed to make working with JDBC easier [1]. It consists of 25 classes, placed in three packages:

- *default package* - contains 11 classes, these are the core classes and interfaces of the system.

- *handlers* - contains 12 classes, implementations for the *ResultSetHandler* interface from the default package.

- *wrappers* - contains 2 classes, two wrappers for the *ResultSet* class from the *java.sql* package.

The exact classes from each package are presented on Table 3.3 and a simplified class diagram of the software system is given on Figure 3.2.

Experiment 1. Results

We applied the $HASP$ algorithm introduced in Section 3.3.2, for the *DbUtils* software system, and it determined that the optimal number of clusters in the final partition is 4. The values for the $overallScore$ measure are presented on Figure 3.3, where the black vertical lines depict the values associated to the local maxima, and the dashed line shows the maximum of these values. The labels under the black lines show the exact values. From Figure 3.3 it can be observed, that the maximal value is for the a partition with 4 clusters. The classes placed in these 4 clusters are presented on Table 3.4.

Even if the resulting packages are not the same as in the original package structure of the system (presented on Table 3.3), it can be observed that they are not very different either. Cluster 3 corresponds to the the packages *wrappers* but it has one extra class, *ProxyFactory*, that was originally in the default package. If we look at these classes and the relationships

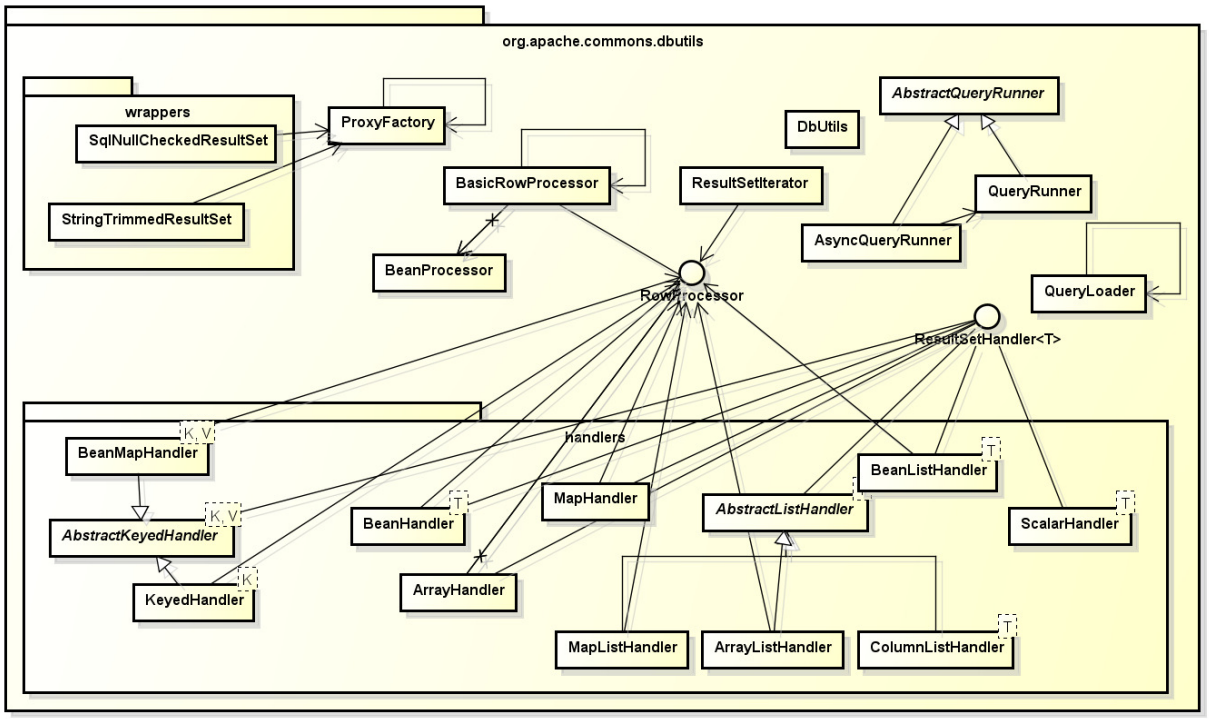

Figure 3.2: The class diagram of the DbUtils framework.

Package	Class Name
default	AbstractQueryRunner, AsyncQueryRunner, BasicRowProcessor, BeanProcessor, DbUtils, ProxyFactory, QueryLoader, QueryRunner, ResultSetHandler, ResultSetIterator, RowProcessor
handlers	AbstractKeyedHandler, AbstractListHandler, ArrayHandler, ArrayListHandler, BeanHandler, BeanListHandler, BeanMapHandler, ColumnListHandler, KeyedHandler, MapHandler, MapListHandler, ScalarHandler
wrappers	SqlNullCheckedResultSet, StringTrimmedResultSet

Table 3.3: Packages and classes in the DbUtils system.

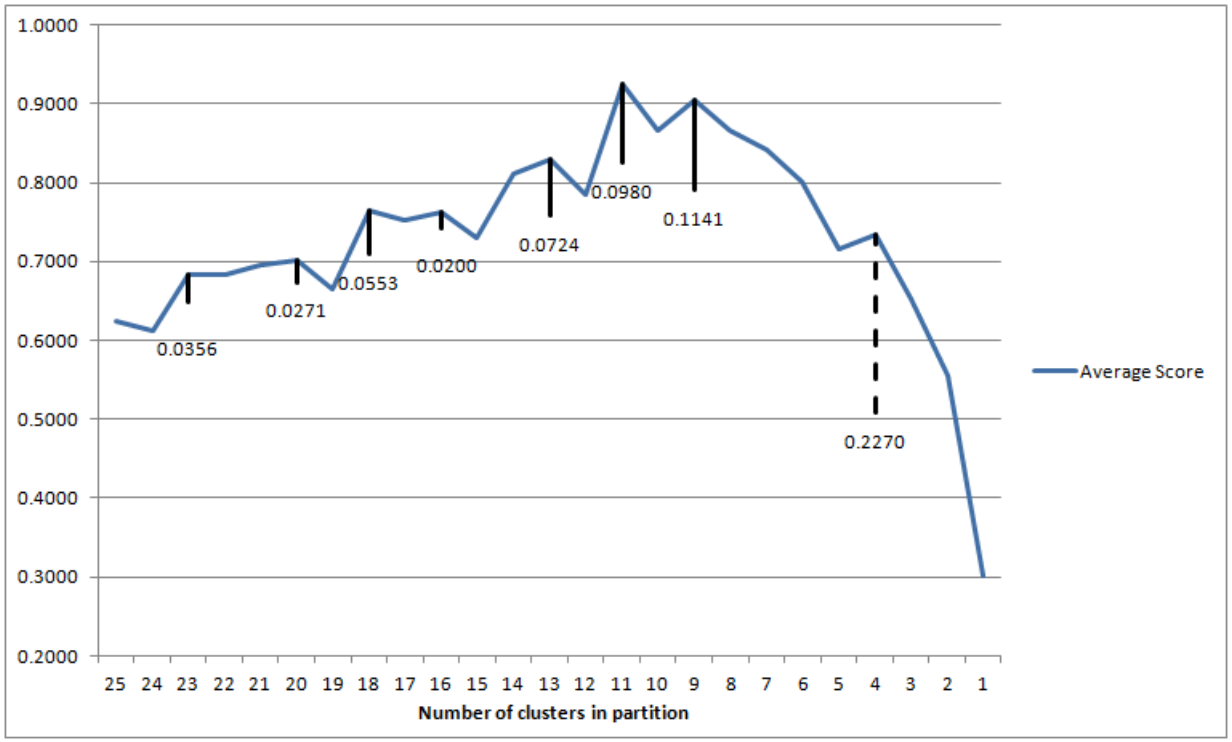

Figure 3.3: The values of the overallScore measure for the DbUtils system.

between them, we can observe that the class *ProxyFactory* uses no class from the system and is only used by the two classes from the *wrappers* package, so it is justified to be moved with those classes.

Cluster 4 corresponds to the *handlers* package in the original structure, but it also has the interface *ResultSetHandler* from the default package. Verifying the dependencies in the system, we can observe that the *ResultSetHandler* interface uses no class from the system. It is used by classes QueryRunner as parameter in some methods, class AsyncQueryRunner in an inner class and is used by every class in the *handlers* package, which either implement it directly or extend a class which implements it. Even if we can not say that interface *ResultSetHandler* has only dependencies with classes from the *handlers* package, it has a larger number of dependencies with the *handlers* package and implement or extend dependencies are, in our opinion, stronger than use dependencies, so we consider that the structure of Cluster 4 is justified as well.

Finally, Clusters 1 and 2 contain the classes which were originally in the default package (without the two classes put into Cluster 3 and 4). Checking the dependencies again, we can see that there is absolutely no dependency between the classes from the two clusters, but there are dependencies inside each cluster. This means, that separating the default package into two separate package does not increase coupling between packages but it increases the cohesion inside the packages, so we believe this division is justified as well.

The results obtained at each step by the *HASP* algorithm on the *DbUtils* software are given in Table 3.5.

Even if we consider that the results of the HASP algorithm can be justified, we computed the value of the CIP metric for the results, considering as $\mathcal{K}^{good}$ the original structure (from Table 3.3). The value of the metric was 0.60813. Obviously, if we consider as $\mathcal{K}^{good}$ the results of the algorithm (from Table 3.4) the value of the CIP metric is 1.

Finally, we have run the *HASP* algorithm on the previous version of the *DbUtils* system, version 1.4, which is very similar to version 1.5, but it does not have the class *BeanMapHandler*

Cluster	Classes
Cluster 1	AbstractQueryRunner, AsyncQueryRunner, QueryRunner, QueryLoader, DbUtils
Cluster 2	BasicRowProcessor, RowProcessor, BeanProcessor, ResultSetIterator
Cluster 3	ProxyFactory, SqlNullCheckedResultSet, StringTrimmedResultSet
Cluster 4	ResultSetHandler, AbstractKeyedHandler, AbstractListHandler, ScalarHandler, ArrayHandler, BeanHandler, BeanListHandler, MapHandler, BeanMapHandler, KeyedHandler, ArrayListHandler, MapListHandler, ColumnListHandler

Table 3.4: Results of the HASP algorithm on the DbUtils system.

in package *handlers*. Our algorithm gave the same results (without the *BeanMapHandler* class) as for version 1.5, the clusters from Table 3.4.

Experiment 2. Results

During the second experiment, several new application classes are added in *DbUtils* system and we aim to determine the software package in which the application classes should be added. For the second experiment, the algorithm introduced in Section 3.3.3, *AssignClass*, will be used. We have decided to remove some application classes from the system, consider the remaining ones as the correct structure and then apply the algorithm from Section 3.3.3 to determine in which package should the removed classes go. In order to thoroughly test the *AssignClass* algorithm, we have decided not to pick some classes randomly, but to try removing every class. Obviously, when we remove a class from the system, we have to remove any other that depend on it (and the classes that depend on these ones, and so on) to be able to consider the remaining classes as an existing system in which we add a new class. Still, there are some classes, which are used by so many other ones, that removing all of them would change the system radically (for example, the *ResultSetHandler* interface on which all classes from the *handlers* package depend on). This is why, we have defined a threshold value for the dependencies: if the number of dependencies is at most 3, then we remove the class and the dependencies (no more than 3) and try to add the class to the system with the *AssignClass* algorithm, otherwise we do not try to remove the given class. The results of this experiment are presented on Table 3.6, where the first column shows the class that is removed, the second column shows the number of dependencies of the class and the last column summarizes the results of the experiment, or contains the corresponding message, if the class was not removed. In order to reference the packages easier, we consider the following names for the packages from Table 3.4: Cluster 1 - *query*, Cluster 2 - *processor*, Cluster 3 - *wrappers* and Cluster 4 - *handlers*.

Being 25 classes in the DbUtils system, it is clear that Table 3.6 contains 25 lines, corresponding to 25 possible runs of the algorithm, one for each class. Out of these, in 5 cases the

Step	Merged clusters		Score
	Cluster 1	**Cluster 2**	
1	ArrayHandler	BeanHandler	2.773
2	AsyncQueryRunner	QueryRunner	2.673
3	BasicRowProcessor	RowProcessor	2.521
4	AbstractKeyedHandler	AbstractListHandler	2.23
5	ArrayHandler, BeanHandler	BeanListHandler	2.05
6	AbstractQueryRunner	AsyncQueryRunner, QueryRunner	1.941
7	ArrayListHandler	MapListHandler	1.92
8	ArrayListHandler, MapListHandler	ColumnListHandler	1.925
9	ArrayHandler, BeanHandler, BeanListHandler	MapHandler	1.838
10	AbstractKeyedHandler, AbstractListHandler	ScalarHandler	1.752
11	SqlNullCheckedResultSet	StringTrimmedResultSet	1.71
12	BasicRowProcessor, RowProcessor	BeanProcessor	1.585
13	ResultSetHandler	AbstractKeyedHandler, AbstractListHandler, ScalarHandler	1.35
14	BeanMapHandler	KeyedHandler	1.28
15	ArrayHandler, BeanHandler, BeanListHandler, MapHandler	BeanMapHandler, KeyedHandler	1.083
16	AbstractQueryRunner, AsyncQueryRunner, QueryRunner	QueryLoader	1.076
17	ProxyFactory	SqlNullCheckedResultSet, StringTrimmedResultSet	1.032
18	BasicRowProcessor, RowProcessor, BeanProcessor	ResultSetIterator	0.991
19	ResultSetHandler, AbstractKeyedHandler, AbstractListHandler, ScalarHandler	ArrayHandler, BeanHandler, BeanListHandler, MapHandler BeanMapHandler, KeyedHandler	0.96
20	ResultSetHandler, AbstractKeyedHandler, AbstractListHandler, ScalarHandler, ArrayHandler, BeanHandler, BeanListHandler, MapHandler, BeanMapHandler, KeyedHandler	ArrayListHandler, MapListHandler, ColumnListHandler,	0.805
21	AbstractQueryRunner, AsyncQueryRunner, QueryRunner, QueryLoader	DbUtils	0.78

Table 3.5: Step-by-step results of the HASP algorithm.

Class name	**Dependencies**	**Result**
AbstractKeyedHandler	2	Class placed correctly in *handlers*
ArrayHandler	7	Too many dependencies
AbstractListHandler	3	Class placed correctly in *handlers*
ArrayListHandler	0	Class placed correctly in *handlers*
BeanHandler	0	Class placed correctly in *handlers*
BeanListHandler	0	Class placed correctly in *handlers*
BeanMapHandler	0	Class placed correctly in *handlers*
ColumnListHandler	0	Class placed correctly in *handlers*
KeyedHandler	0	Class placed correctly in *handlers*
MapHandler	0	Class placed correctly in *handlers*
MapListHandler	0	Class placed correctly in *handlers*
ScalarHandler	0	Class placed correctly in *handlers*
SqlNullCheckedResultSet	0	Class placed correctly in *wrappers*
StringTrimmedResultSet	0	Class placed correctly in *wrappers*
AbstractQueryRunner	2	Class placed incorrectly
AsyncQueryRunner	0	Class placed correctly in *query*
BasicRowProcessor	10	Too many dependencies
BeanProcessor	10	Too many dependencies
DbUtils	3	Class placed incorrectly
ProxyFactory	2	Can not remove, *wrapper* becomes empty
QueryLoader	0	Class placed correctly in *query*
QueryRunner	1	Class placed incorrectly
ResultSetHandler	14	Too many dependencies
ResultSetIterator	0	Class placed correctly in *processor*
RowProcessor	10	Too many dependencies

Table 3.6: Results of the experiments with the *AssignClass* algorithm on the *DbUtils* framework.

class had too many dependencies, meaning that too many classes should have been removed, so the experiment was not performed. We did not perform the experiment for the *ProxyFactory* class either, because removing it and the two dependencies it has, would have meant removing the whole *wrappers* package. We have obtained an accuracy of 84.2 %, since out of the remaining 19 cases, when we have performed the experiment, in 16 cases the algorithm suggested the correct package for the class in the experiment.

3.4.3.2 A *Reinforcement Learning* framework

The second case study considered for evaluation is an open source software framework available at [4], that was introduced in [29] in order to solve combinatorial optimization problems using reinforcement learning [100]. This framework was used in [18, 28, 27] for solving with reinforcement learning several optimization problems: the bidimensional *protein folding* problem, the *DNA fragment assembly* problem, the *temporal ordering* problem.

The RL software framework is realized in JDK 1.6 and has four basic modules: **agent**, **environment**, **reinforcement learning**, and **simulation**.

As in a general agent-based system [111], the *agent* is the entity which interacts with the *environment*, that receives perceptions and selects *actions*. The agent learns using *reinforcement learning* to achieve its goal, i.e. to find an optimal solution of the corresponding optimization problem. Generally, the inputs of the agent are perceptions about the *states* from the environment, the outputs are actions, and the environment offers rewards after interacting with it. The interaction between the agent and the environment is controlled by a *simulation* entity. The environment is assumed to be accessible to the agent, meaning that the perceptions received by it are the states from the environment.

Agent. The agent is the entity that interacts with the environment, receives perceptions (states) from it and selects actions. The agent learns by reinforcement and could have or not a model of the environment. It is the basic class for all the agents. The specific agents will implement the *Agent* interface. The main responsibility of the *Agent* class is to select the most appropriate *action* it has to perform in the *environment*.

Environment. The environment basically defines the optimization problem to solve. The environment has an explicit representation as a space of *states*. It is the basic class for all environments. The specific environments will implement the *Environment* interface. The *Environment* has a function that determines the environment to make a transition from a state to another, after executing a specific action. This function also gives the *reward* obtained after the transition. The environment stores an instance of its current *state*.

ReinforcementLearning. It is the class responsible with the reinforcement learning process. The framework provides implementation for the $Q - learning$ algorithm [100].

Simulation. It is the object that manages the interaction between the agent and the environment. An instance of the simulation class is associated with an instance of an agent and an environment at the creation moment. The *simulation* object is responsible with collecting data, managing the learning process and providing the optimal policy that the agent has learned.

Figure 3.4 shows a simplified UML diagram [47] of the interface, illustrating the core of the RL framework.

The considered software system has ten packages, that are briefly described below. The exact classes that can be found in each package are presented in Table 3.7.

- *action* - package that contains two interfaces, which are abstract representations of the actions that an agent can take.
- *actionselectionpolicy* - a package which contains the *ActionSelectionPolicy* interface and several implementations of it.

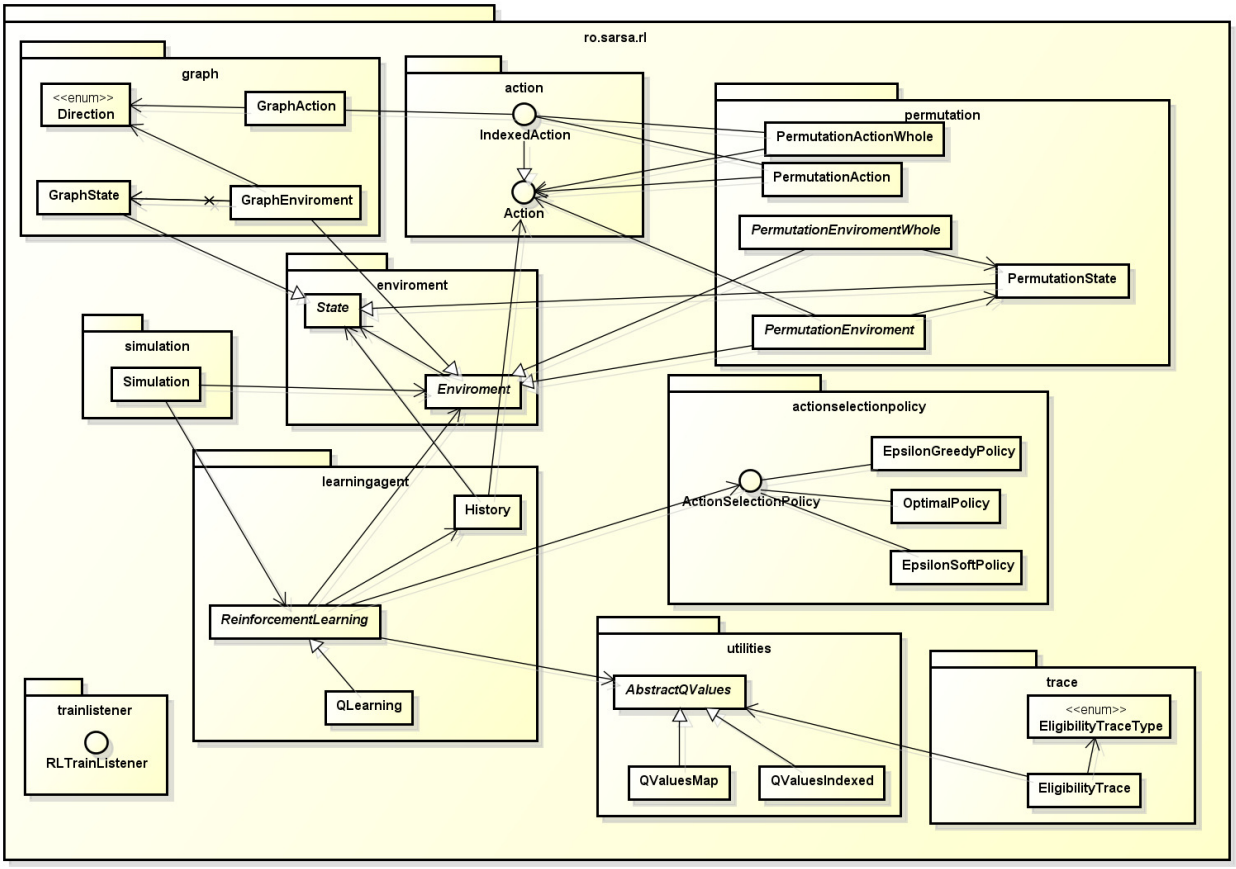

Figure 3.4: The diagram of the RL programming interface

- *environment* - a package which contains two interfaces, representing the *Environment* and a *State* of the environment.
- *graph* - concrete implementations for *Action*, *State* and *Environment* for problems where the solution is a path in a graph.
- *learningagent* - package with the abstract *ReinforcementLearning* class and its implementation for *QLearning*, as well as the *History* class which records the sequence of *States* and *Actions* used through the learning process.
- *permutation* - similarly to the *graph* package, this package contains concrete implementations for problems whose solution may be represented as a permutation. There are two different actions and environments implemented in this package: *PermutationActionWhole* and *Permuta-tionEnvironmentWhole* refer to cases when each state is a whole permutation which is modified, while the other versions refer to the cases when a permutation is built element-by-element.
- *simulation* - contains only the *Simulation* class.
- *trace* - contains a class *ElibigilityTrace* which is a mechanism for handling delayed reward. There are two kinds of traces implemented, *Accumulating* and *Replacing* as denoted by the *EligibilityTraceType* enum from this package.
- *trainlistener* - package with only one interface, *RLTrainListener*, whose implementation can be used to record different information throughout the training process.
- *utilities* - contains the classes related to the Q-values needed for QLearning.

Experiment 1. Results

We applied the *HASP* algorithm, introduced in Section 3.3.2, for the Reinforcement Learning framework presented in the previous section. The values of the *overallScore* measure are presented in Figure 3.5, where the black vertical lines depict the values associated to the local maximums. The dashed line represents the maximum of these values, corresponding to the optimal number of clusters. In case of the Reinforcement Learning framework, this value is for the partition with 11 clusters.

Although the original structure of the system (presented on Table 3.7), consists of only 10 clusters, the difference is small: nine packages correspond exactly in the original structure and the result of our algorithm, the only difference is, that the *permutation* package is divided into two: the first package consists of classes *PermutationAction* and *PermutationActionWhole*, while the other package consists of the remaining 3 classes. Although the result with the 10 clusters would be optimal, this is still a good division. Obviously, all classes that belong to the implementations for learning in a permutation medium should belong together, but generally speaking, environment and state are closer together, than action is to these two. This is also suggested by the fact, that in case of the abstract classes and interfaces, *Environment* and *State* are placed together in a package, but interfaces belonging to the actions have their own package. Also, in case of the other package with implementations for a given medium, *graph*, during the clustering process, classes representing the state and the environment are merged before adding the action class to them. Finally, we also mention, that at the next step, these two clusters would be merged, leading to the optimal 10-cluster version. All of the above lead us to believe that the result of the *HASP* algorithm for the reinforcement learning framework are good. We computed the value of the CIP metric for this result, and it is 0.95, which is a quite high value.

Package	Class name
agent	Action, IndexedAction
actionselection-policy	ActionSelectionPolicy, EpsilonGreedyPolicy, EpsilonSoftPolicy, OptimalPolicy
environment	Environment, State
graph	Direction, GraphEnvironment, GraphAction, GraphState
learningagent	History, QLearning, ReinforcementLearning
permutation	PermutationAction, PermutationActionWhole, PermutationEnvironment, PermutationEnvironment-Whole, PermutationState
simulation	Simulation
trace	EligibilityTrace, EligibilityTraceType
trainlistener	RLTrainListener
utilities	AbstractQValues, QValuesIndexed, QValuesMap

Table 3.7: Packages and classes in the Reinforcement Learning framework.

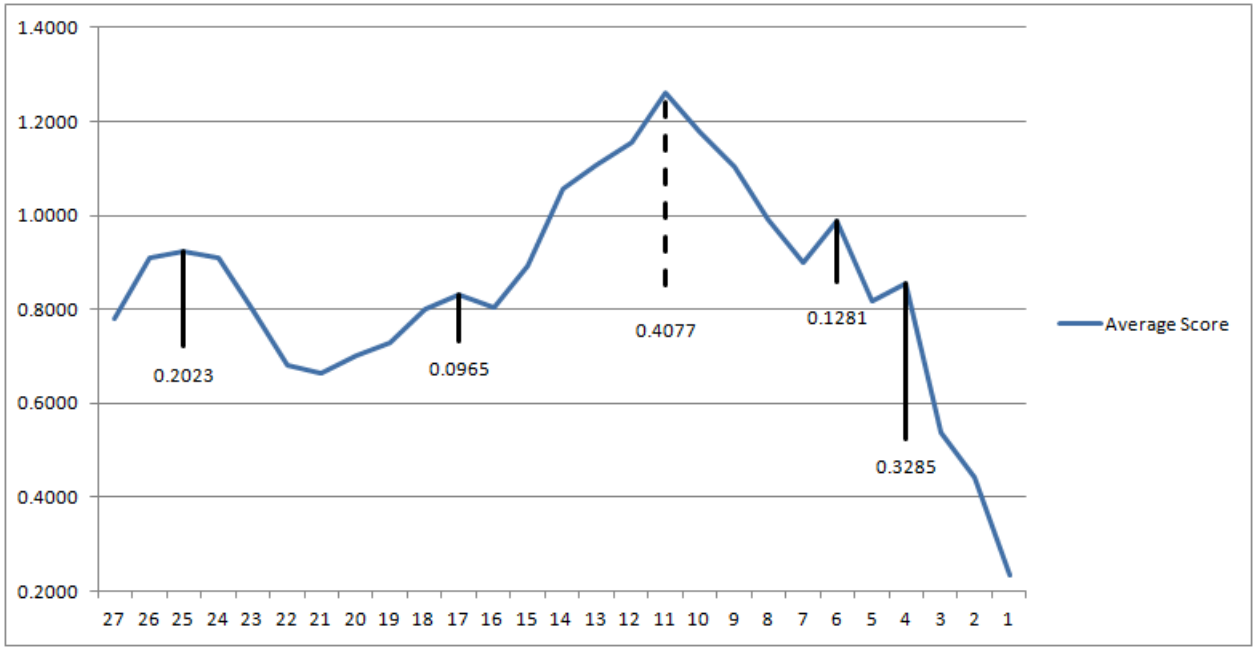

Figure 3.5: The values of the overallScore measure for the Reinforcement Learning system.

Experiment 2. Results

During the second experiment, several new application classes are added in the *RL* framework and we aim to determine the software package in which the application classes should be added. For the second experiment, the algorithm introduced in Section 3.3.3 will be used. Since the previous experiment provided a system with 11 clusters, while the original system had only 10, we have decided to perform this experiment considering as the given system S both cases. So, in Tables 3.8, 3.9 and 3.10 the second column, *Score-10* contains the scores when we consider as S the original structure of the software system, while the column *Score-11* contains the scores when we consider the structure provided by the $HASP$ algorithm. In case of the structure with 11 clusters, the package *permutationaction* refers to the package with the two action classes, while *permutation* denotes the package with the other 3 classes.

The application classes that are added to *RL* framework are:

- $SarsaLearning$ is a class which implements the $SARSA$ learning algorithm, which is similar to $Q-learning$, but uses the value of the actually performed action to determine its update, instead of the maximum available action [100]. $SarsaLearning$ class should belong to the *learningagent* software package, since it extends the *ReinforcementLearning* class, and implements a learning algorithm, just like *QLearning*. The values of the score for each package and the *SarsaLearning* class is presented in Table 3.8. The bold line corresponds to the highest score, i.e. it denotes the package in which the *SarsaLearning* class should be placed.

- $SarsaLambdaLearning$ is a class which implements $SARSA(\lambda)$. λ refers to the use of an *eligibility trace* [97] for obtaining a more general and efficient learning method. The *eligibility trace* is one of the basic mechanisms used in reinforcement learning to handle delayed reward. An eligibility trace is a record of the occurrence of an event such as the visiting of a state or the taking of an action [100]. By associating one of such traces to every possible action in every state, the following temporal credit assignment is implemented: "Earlier states/actions are given less credit for the current temporal difference error". $SarsaLearning$ class should belong to the *learningagent* software package, since is is also an implementation of the *ReinforcementLearning* class, just like the previous *SarsaLearning* class. The values of the score for each package and the *SarsaLambdaLearning* class is presented in Table 3.9. The bold line corresponds to the highest score, i.e. it denotes the package in which the *SarsaLambdaLearning* class should be placed.

- $SoftMaxPolicy$ is a class which implements the $SoftMax$ [100] action selection policy used during the training of the RL agent. $SoftMaxPolicy$ class should belong to the *actionselectionpolicy* software package, since it is an implementation of a learning policy, represented by the interface *ActionSelectionPolicy*, and the other similar implementations are in this package. The values of the score for each package and the *SoftMaxPolicy* class is presented in Table 3.10. The bold line corresponds to the highest score, i.e. it denotes the package in which the *SoftMaxPolicy* class should be placed.

From Tables 3.8, 3.9 and 3.10 we can see that the results are the same for both the 10 cluster and the 11 cluster version. For some of the packages the exact values of the score are the same as well in both cases, for other packages small differences exist, but these do not influence the result, which is correct for every case. These obtained results lead us to the conclusion that for all the newly added application classes, the algorithm introduced in Section 3.3.3 provides the good software package in which the class should belong.

Package	Score - 10	Score - 11
learningagent	**1.328**	**1.328**
environment	1.138	1.169
utilities	1.09	1.09
actionselectionpolicy	1.053	1.053
graph	0.976	0.976
permutation	0.747	0.707
permutationaction	-	0.3
action	0.607	0.638
trace	0.497	0.497
trainlistener	0.013	0.013
simulation	-0.16	-0.16

Table 3.8: Scores for assigning the *SarsaLearning* class to a package.

Package	Score - 10	Score - 11
learningagent	**1.272**	**1.272**
environment	1.195	1.226
trace	1.127	1.127
utilities	1.057	1.057
graph	0.968	0.968
actionselectionpolicy	0.861	0.861
permutation	0.734	0.687
permutationaction	-	0.268
action	0.665	0.696
trainlistener	-0.161	-0.161
simulation	-0.251	-0.251

Table 3.9: Scores for assigning the *SarsaLambdaLearning* class to a package.

Package	Score - 10	Score - 11
actionselectionpolicy	**1.491**	**1.491**
utilities	1.182	1.182
graph	0.976	0.976
environment	0.835	0.866
permutation	0.77	0.725
permutationaction	-	0.355
action	0.632	0.663
learningagent	0.599	0.599
trace	0.522	0.522
trainlistener	-0.1	-0.1
simulation	-0.31	-0.31

Table 3.10: Scores for assigning the *SoftMaxPolicy* class to a package.

3.5 Discussion and comparison to related work

In this section, we aim to provide a detailed analysis of the *CASP* approach we have introduced for software packages remodularization, as well as comparing our proposal to existing similar approaches.

3.5.1 Analysis of our approach

Besides the evaluations that were performed and detailed in Section 3.4, in order to better investigate the effectiveness of the *HASP* algorithm introduced in Section 3.3.2.4, in identifying a good structure of software packages, we have conducted some additional experiments.

For these experiments, we have used the *DbUtils* software system (that was used in the experiments presented in Section 3.4.3.1) and two other open-source software frameworks chosen from the Apache Commons project: *Email*, release 1.3.2 [3] and *EL*, release 1.0 [2]. *Email* is a framework for sending emails, built on top of the Java Mail API, but tries to simplify it. It consists of 19 classes, divided into three packages: *resolver*, *util* and the *default* package. For the *Email* system, the *HASP* algorithm provides a partition with three clusters. In the *HASP* partition there are four classes which are placed in a different package than in the original one.

The *EL* framework is a JSP 2.0 Expression Language interpreter consisting of 57 classes, divided into two packages: *parser* and the *default* package. In this case, there is a big difference between the original partition of *EL* and the partition provided by the *HASP* algorithm. The original partition has two packages, while the *HASP* partition has seven. Out of these seven packages (clusters), one corresponds exactly to the *parser* package from the original partition, while the remaining six packages contain the classes which were originally in the *default* package.

3.5.1.1 Evaluation measures

In order to objectively evaluate the results provided by the *HASP* algorithm, we have decided to use some reference software measures from the literature, designed to evaluate a whole partition or a package from a partition. We have chosen two measures and two multi-objective approaches, that evaluate a whole partitioning of a software system: *Modularization Quality*, *Evaluation Metric Function*, *Maximizing Cluster approach* and *Equal-size Cluster approach*. Most of these measures consider the dependencies between classes and packages. When computing the values of these measures, we considered the dependencies the same way as for the *score* measure, but we do not weight them.

MQ, or *Modularization Quality*, is a measure introduced by Mancoridis et al. in [70], which was used in many different approaches for the evaluation of a software system's partitioning [88], [69], [77], [68], [32], [12] and [44]. The *MQ* measure for a cluster (or package) is computed considering the number of intra-cluster (or intra-package) relations and the number of inter-cluster (or inter-package) relations. The *MQ* value for a partition is the sum of the values computed for the clusters. A similar measure is *EVM*, or *Evaluation Metric Function*, introduced in [102], and used in [44], [12] for the evaluation of a software system's partitioning. For each cluster (package) from the partition, *EVM* rewards each class-pair that has a dependency between them and penalizes those ones which do not have a dependency between them. Then, similarly to *MQ*, the value for the whole partition is the sum of values for clusters. For both measures higher values correspond to better partitions.

In [88] Praditwong et al. present software module clustering as a multi-objective problem, and introduce two different sets of objectives, which could be used for building a set of Pareto-optimal solutions [79]. The first approach is *Maximizing Cluster approach*, consisting of the following five values:

- the sum of intra-cluster relations for all clusters;
- the sum of inter-cluster relations for all clusters;
- the number of clusters;
- the value of the MQ measure;
- the number of isolated clusters (clusters with only one class).

Out of these five values the second and the fifth should be minimized, while the others should be maximized. According to [88] the *Maximizing Cluster approach* tries to capture the characteristics of a good modularization. In a similar manner is defined a second approach, called *Equal-size Cluster approach*, which contains a set of five values as well:

- the sum of intra-cluster relations for all clusters;
- the sum of inter-cluster relations for all clusters;
- the number of clusters;
- the value of the MQ measure;
- the difference between the maximum and minimum number of classes in a package.

Similarly to the first approach, the second and the fifth values should be minimized, while the other three should be maximized.

3.5.1.2 Results analysis

The values of the *MQ* and *EVM* evaluation measures are presented in Table 3.11. For each of the three considered case studies, we computed the value of these measures both for the whole partition (denoted by *Partition*) and for the packages separately. In case of the *DbUtils* and *EL* systems, the original and the *HASP* partitions contain a different number of clusters (packages), in both cases the *HASP* partition having a higher number. In both cases, the *default* package from the original partition is split into more packages (two for *DbUtils* and six for *EL*) in the *HASP* partition. For these cases, we added the values computed for the packages from the *HASP* partition and compare the sum to the value of the *default* package from the original partition (for example, for the *DbUtils* system, we compare the values for the *default* package from the original partition, with the sum of the values for the two corresponding packages from the *HASP* partition).

In Table 3.11 the best values for the evaluation measures are marked with bold. We can see that the *MQ* measure has an equal or a higher value for the *HASP* partition in all cases, except one (the *default* package for the *Email* system). The *EVM* measure has mostly negative values, but this is understandable, because usually there are a lot more class-pairs in a package, which do not have a dependency, than pairs which do. Similarly to the *MQ* measure, in case of the *EVM* measure, in most cases the *HASP* partition has a better values. The only exception is the *resolver* package of the *Email* system. We can notice that even if in case of the *Email* system, we have two cases when the value of a package was better for the original partition, the values computed for the whole partition are better in case of the *HASP* partition. Thus improvements in other packages compensate for the lower values for those packages.

The values for the two multi-objective approaches are presented in Table 3.12, the best values being marked with bold. When comparing two sets of objective functions, we can say that one *dominates* the other, if it is better in at least one function and not worse in all the others. This is the case for the *DbUtils* and *Email* systems, the *HASP* partition dominates

Case study	Evaluated entity	MQ		EVM	
		Original	HASP	Original	HASP
DbUtils	*Partition*	1.1559	**3.0917**	-78	**-39**
DbUtils	*wrappers* package	0	**1**	-1	**1**
DbUtils	*handlers* package	0.6154	**0.7917**	-42	**-40**
DbUtils	*default* package	0.5405	**1.3**	-35	**0**
Email	*Partition*	1.7231	**1.7714**	-33	**-21**
Email	*resolver* package	0.8	**0.8571**	**-2**	-24
Email	*util* package	0	**0**	-1	**-1**
Email	*default* package	**0.9231**	0.9142	-30	**4**
EL	*Partition*	1.8015	**4.2415**	-998	**-210**
EL	*parser* package	0.8182	**0.8182**	-3	**-3**
EL	*default* package	0.9833	**3.4233**	-995	**-207**

Table 3.11: Comparison of the values for the *MQ* and *EVM* measures for the original and *HASP* partition of the three case studies.

Measure	DbUtils		Email		EL	
	Original	HASP	Original	HASP	Original	HASP
Maximizing Cluster	22	**29**	22	**22**	**127**	60
	34	**20**	6	**6**	**8**	142
	3	**4**	3	**3**	2	**7**
	1.1559	**3.0917**	1.7231	**1.7714**	1.8015	**4.2415**
	0	**0**	0	**0**	0	**0**
Equal-size Cluster	22	**29**	22	**22**	**127**	60
	34	**20**	6	**6**	**8**	142
	3	**4**	3	**3**	2	**7**
	1.1559	**3.0917**	1.7231	**1.7714**	1.8015	**4.2415**
	10	**10**	10	**7**	43	**20**

Table 3.12: Comparison of the values of the *Maximizing Cluster* and *Equal-size Cluster* approaches for the original and *HASP* partitions of the three case studies.

the original for both approaches. In case of the *EL* system, we have *non-dominating* multi-objective function values, the original partition has better values for the first two objective functions, while the *HASP* partition has better values for the third, fourth and, in case of the *Equal-size Cluster* approach, fifth objective function.

For the *DbUtils* and the *Email* systems all used measures suggest that the *HASP* partition is better, but in case of the *EL* system we do not have a clear answer. This is why we have decided to use one more metric for investigating this system. We have chosen the *Distance* metric, which is computed for each package separately, and measures the distance of the package from the main sequence: $A + I - 1 = 0$, where A denotes the *Abstractness* of the package and I denotes its *Instability* [33]. Since the *Distance* metric gives a separate value for each package and the two partitions we want to compare have a different number of packages, we decided to take the average of the absolute values for the metric for each partition. In this way we achieved a value of **0.5186** for the original partition and **0.4845** for the *HASP* partition. Since we talk about the distance from the main sequence, we consider that lower values correspond to a better partitioning, so the *Distance* metric suggests that the *HASP* partition is better than the original one.

Therefore, taking into account all evaluation measures for the considered case studies (including the evaluation for the *Distance* metric), the *HASP* partition provided better values

than the original partition or equal values for **25** measures out of the **29** evaluation measures (equal values were obtained 4 times). As it was explained above, the remaining four cases do not actually represent situations in which the *HASP* partition is worse than the original one. All these results indicate a very good efficiency of the *HASP* algorithm.

We can conclude that *CASP* approach introduced in this paper would be useful for assisting software developers in their daily work of refactoring software systems into packages. Moreover, our approach is useful in different scenarios:

- It takes an existing software and provides a partitioning of application classes into software packages.
- It is useful during the evolution of software systems. Considering an existing structure of packages of a software system, when the system evolves and new application classes are added, our method would suggest the developer the appropriate package where the new application class should be added.

Considering the results of all the experimental evaluations performed, we can say that our method and the features that we have defined are capable of finding a good package structure for a software system which is a framework. For other architectures, we will investigate features that would be relevant for the restructuring of the system.

3.5.2 Comparison to related work

In Section 3.2.2 we have presented some approaches from the literature that are similar to our work. Out of the presented works, only two are capable of restructuring a whole system: [9] and [84]. Both methods use as case study an open source software system, called *Trama*, but they report slightly different results for it. We can not use *Trama* as a case study, because it is not a framework, it is a system with a layered architecture (a package for persistence, another for business and one for GUI).

Out of the two mentioned methods, the closest method to the one presented in this paper is the approach presented by Alkhalid et al. in [9], which uses hierarchical clustering, just like our method. This work presents two approaches and based on the description given in [9], we applied these approaches on the *DbUtils* system. In the first approach the existing packages are kept, but classes can be moved between packages if they are initialized more times in a different package. Applying this method on the *DbUtils* system, the original package structure is kept, because there are only a few initializations in the system: there is a total of 15 initializations, but 11 of them are in cases when the constructor of a subclass calls the constructor of the superclass. This small number of initializations means that there are many possible structures for the classes, for which this approach would not suggest any changes. Moreover, considering initialization the only dependency seems restrictive, because interfaces and abstract classes are never initialized in a system.

The second method presented in [9] uses a vector-space representation and is based on method-call dependencies. This representation seems better than the previous one, but we still think that more types of dependencies should be considered. In [9] three linkage metrics are used, single, complete and average, and a new algorithm is introduced, which is considered to have the same results as the clustering algorithms, but its big advantage is the lower complexity. We tried the single, complete and average linkage algorithms for the vector-space representation created for classes from the *DbUtils* system. We have used as a stopping criteria the point where the distance between all clusters was one. The results of these experiments are summarized on Table 3.13, where the first column describes the clustering method used, the second contains the number of clusters when the algorithm stopped and the last two columns contain the value of the CIP metric presented in Section 3.4.2. The value denoted by **CIP 1** (represented in the third column of the table) is the value of the

Method	Clusters	CIP 1	CIP 2
[9] - single link	14	0.4322	0.30
[9] - complete link	17	0.4115	0.2953
[9] - average link	14	0.4322	0.30
HASP	4	**0.608**	**1**

Table 3.13: Comparative results on the DbUtils system.

CIP metric when we consider $\mathcal{K}^{good}$ to be the original structure of the DbUtils system. The value denoted by **CIP 2** (represented in the fourth column of the table) is the value of the CIP metric when we consider as $\mathcal{K}^{good}$ the partition given by the $HASP$ algorithm. We have added in the last line in Table 3.13 the value of the CIP metrics for the results of the *HASP* method as well, in order to make the comparison of the results easier.

One thing to observe from Table 3.13 is that the algorithms stopped with a too high value of clusters. In a system with 25 classes, 14 or 17 packages seem to be a lot. Also, in case of the *handler* package, which originally has 12 classes, 8 of them are grouped into a cluster, but the rest of them are in separate clusters. Also, the package with the 8 handler classes contains the *ResultSetIterator* class as well, which has nothing to do with these classes (no direct dependency) except for the fact that it uses methods from the *ResultSetHandler* class just like the handler classes do.

Even if probably we could not reproduce the algorithms presented by Alkhalid et al. in [9] perfectly, and the actual results would differ a little, we still think that our method is better. For example, in case of software systems with abstract classes and interfaces, where considering just initialization or method call as a dependency type is not sufficient to construct the correct packages.

Chapter 4

Hidden dependencies identification

Maintenance activities such as bug fixes, updating existing features and adding new ones make up the majority of time and costs allocated to a software project. Each of these changes usually affects only part of the system, and determining the affected components (classes, modules, methods etc.) is not a trivial problem. Impact analysis tries to identify, given a component of a software system, the other components that would be affected by changes to it [20]. Such methods usually consider only direct coupling between components, but there exists indirect coupling [112] as well, which creates hidden dependencies, that cannot be found using regular coupling measures, but not identifying them can have serious consequences [26].

4.1 Literature review

There are approaches which use previous versions of the software system and try to identify those classes which were changed together in connection with the same bug report [38]. Gall et al. introduce in [38] an approach, called CAESAR, that uses information about previous versions of a system to discover logical dependencies and change patterns among modules. The proposed method is experimentally evaluated on 20 releases of a large Telecommunications Switching System. Information such as version numbers of programs, modules, and subsystems together with change reports are used for identifying common change patterns of software modules. CAESAR determines hidden dependencies which are not obvious in the source code, like modules that should be restructured. Instead of using the lines of code for the previous versions of the software, the authors use structural information about programs, modules, and subsystems, together with change reports for the releases and their version numbers. The method proposed in [38] was capable to identify bugs which were fixed in one versions of the system and which have appeared again in later versions in other parts of the software.

One of the early works is [115], where Yu and Rajlich transform System Dependence Graphs into Abstract System Dependence Graphs, to determine which class pairs have hidden dependencies. The paper discusses how hidden dependencies impact the process of change propagation and also discusses an algorithm that indicate the possible presence of hidden dependencies. Hidden dependencies are considered to be design faults which contradict the rule "if a class A is unaware of the existence of class B, it is also unconcerned about any change to B". More exactly, a dependence between Class A and B is a hidden dependence, if: (1) class A and B are not neighbors in the ASDG, i.e there is no direct dependence; and (2) there is a third class C, which is dependent on both classes, and there is data flow inside the class C that occurs between the instance of class A and instance of class B. A simple algorithm for determining hidden dependencies is introduced, and a JAVA example consisting of three classes collaborating to manage a session is considered.

While traditional coupling measures cannot be used for finding hidden dependencies,

[86] presents how a conceptual coupling measure that considers identifier names, comments and other textual elements of code can be used for impact analysis and can find hidden dependencies as well. [86] reports precision and recall around 20%. Besides, some existing approaches rely on historical data, which is not always available (and knowledge extracted from it cannot be used for other projects), or on the creation of different graphs which can be expensive for large systems.

In case of large software systems, computing Abstract System Dependence Graphs can be expensive, so other approaches were introduced which are based on the order in which different methods are called (call trace): if a method is always called after another method, there might be a dependency (hidden or not) between the classes where these methods are [105]. Vanciu and Rajlich propose in [105] a dynamic technique for identifying hidden dependencies. It is based on computing "execute completely after" relations which are filtered based on pre- and postconditions that are generated dynamically. For evaluation, open source software systems like JUnit, Drawlets and Apache FtpServer are used. The authors show that hidden dependencies exist even in well-designed software, like the ones considered for evaluation. For the case studies used for evaluation, the technique proposed in [105] obtained a precision between 46% and 59% for discovering hidden dependencies.

In 2014, Kirbas et al. [58] have investigated the influence of the evolutionary coupling on defect proneness. A positive correlation between evolutionary coupling and defect measures, such as number of defects and defect density, have been confirmed by numerical experiments performed for a large financial legacy software system. Two evolutionary coupling measures derived from modification requests (MR) have been used in this study.

In 2015, Kouroshfar et al. [60] have studied the effects of architecturally dispersed co-changes on software quality. It has been experimentally shown that the changes involving multiple architectural modules are more correlated with defects than the intra-module co-changes. The study corroborates the relevance of considering architecture in predicting software defects.

In 2016, Akbarinasaji et al. [8] have proposed a suite of six metrics of logical dependency among source files in a software system. The impact of these metrics on defect prediction performance has been evaluated by applying two learning models, the Logistic Regression and the Naive Bayes, on three different software projects. The metrics have been used as features of the training data, their values being derived from the timestamp information in the change history of files. The experimental results have confirmed that, if the values of logical dependency are high, they significantly improve the performance of the defect prediction models.

Bell studies in [15] the influence of hidden dependencies identification for software testing. The author shows that increasing the efficiency and the effectiveness of testing through a good knowledge of the hidden dependencies between tests contribute to improving the software reliability. In real software systems, there are hidden dependencies between tests, which makes the testing process harder. In such situations, the tests cannot be run in parallel, since they are not independent (i.e. a test outcome is influenced by the execution of other test). It has been shown in the software engineering literature [15] that these dependencies are often difficult and hidden from the software developers. Bell develop in [15] a software system called VMVM for detecting different types of dependencies between tests and use the detected information to significantly reduce the testing time (with around 60% in average). VMVM is a Java implementation of a technique called Unit Test Virtualization, a technique which isolates the side-effects of each unit test from other tests. It is based on a hybrid static-dynamic analysis and automatically identifies the code segments that may create side-effects. These segments are isolated in a container similar to a virtual machine.

Chapter 5

Conclusions

We have presented in this report the original scientific results which were obtained for achieving the objectives proposed in the project's work plan for the year 2016. Our main scientific objectives were related to the *development of new classification algorithms for identifying entities with defects in software systems*, *unsupervised learning methods for software packages restructuring* and *hidden dependencies identifocation using unsupervised learning methods*.

The report presented in the first chapter the original results we have obtained in the direction of *software defect prediction* using fuzzy *self-organizing maps* and *fuzzy decision trees*. The experimental evaluation which was performed on open-source software systems provided results better than most of the similar existing approaches and highlighted a very good performance of the proposed approaches. Further work will be carried out in order to extend the experimental evaluation of the *fuzzy* decision tree. We also aim to investigate a hybridization between the *fuzzy* DT model and *relational association rules* [94], since we are confident that relations between the values for different software metrics would be relevant in discriminating between defective and non-defective software entities.

In the second chapter we have introduced a novel hierarchical clustering-based approach, that can be used for software remodularization at the package level. A hierarchical clustering algorithm *HASP* is introduced in order to group application classes from a software system in packages. The experiments, which were performed on two open source case studies, have demonstrated the potential of our proposal. We have also emphasized the advantages of our approach in comparison with similar existing approaches. Future work will be made in order to extend the experimental evaluation of the algorithms proposed in this paper on other open source and real software systems. We will investigate features that would be relevant for restructuring software systems with different architectures than frameworks, as well as alternative methods to decide the appropriate number of clusters (software packages) [80]. Extensions of the proposed methods to fuzzy approaches [99] as well as other clustering methods [113] may be further analyzed.

The third chapter presented a literature review on the problem of *hidden dependencies identification*. Novel machine learning based models and methods for solving this problem represent the major objective of the project in 2017.

Bibliography

[1] Commons dbutils. http://commons.apache.org/proper/commons-dbutils/index.html.

[2] El. http://commons.apache.org/proper/commons-el/.

[3] Email. http://commons.apache.org/proper/commons-email/.

[4] Reinforcement learning framework. http://www.cs.ubbcluj.ro/~gabis/rl.

[5] 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.

[6] Wasif Afzal, Richard Torkar, and Robert Feldt. Resampling methods in software quality classification. *International Journal of Software Engineering and Knowledge Engineering*, 22(2):203–223, 2-12.

[7] Rezwan Ahmed and George Karypis. Algorithms for mining the evolution of conserved relational states in dynamic networks. *Knowledge and Information Systems*, 33(3):603–630, 2012.

[8] Shirin Akbarinasaji, Behjat Soltanifar, Bora Çağlayan, Ayse Basar Bener, Andriy Miranskyy, Asli Filiz, Bryan M. Kramer, and Ayse Tosun. A metric suite proposal for logical dependency. In *Proceedings of the 7th International Workshop on Emerging Trends in Software Metrics*, WETSoM '16, pages 57–63, New York, NY, USA, 2016. ACM.

[9] A. Alkhalid, M. Alshayeb, and S.A. Mahmoud. Software refactoring at the package level using clustering techniques. *IET Software*, 5(3):274–286, 2011.

[10] Jorge Aranda and Gina 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*, ICSE '09, pages 298–308, Washington, DC, USA, 2009. IEEE Computer Society.

[11] ObjectWeb: Open Source Middleware. http://asm.objectweb.org/.

[12] Márcio De Oliveira Barros. Evaluating the importance of randomness in search-based software engineering. In *Proceedings of the 4th International Conference on Search Based Software Engineering*, pages 60–74, 2012.

[13] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley Professional, 3rd edition, 2012.

[14] Gabriele Bavota, Andrea De Lucia, Andrian Marcus, and Rocco Oliveto. Software remodularization based on structural and semantic metrics. In *17th Working Conference on Reverse Engineering*, pages 195–204, 2010.

[15] Jonathan Bell. *Making Software More Reliable by Uncovering Hidden Dependencies*. PhD thesis, Graduate School of Art and Sciences, Columbia University, 2016.

[16] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *J. Mach. Learn. Res.*, 13:281–305, February 2012.

[17] A. Beszedes, T. Gergely, J. Jasz, G. Toth, T. Gyimothy, and V. Rajlich. Computation of static execute after relation with applications to software maintenance. In *2007 IEEE International Conference on Software Maintenance*, pages 295–304, Oct 2007.

[18] Maria-Iuliana Bocicor, Gabriela Czibula, and Istvan-Gergely Czibula. A reinforcement learning approach for solving the fragment assembly problem. In *Proceedings of the 2011 13th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, SYNASC '11, pages 191–198, Washington, DC, USA, 2011. IEEE Computer Society.

[19] Gary D. Boetticher. *Advances in Machine Learning Applications in Software Engineering*, chapter Improving the Credibility of Machine Learner Models in Software Engineering. IGI Global, 2007.

[20] Lionel C. Briand, Juergen Wuest, and Hakim Lounis. Using coupling measurement for impact analysis in object-oriented systems. In *Proceedings of the IEEE International Conference on Software Maintenance*, ICSM '99, pages 475–482, Washington, DC, USA, 1999. IEEE Computer Society.

[21] Gerardo Canfora, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, Annibale Panichella, and Sebastiano Panichella. Multi-objective cross-project defect prediction. In *Proceedings of the 6th International Conference on Software Testing, Verification and Validation*, pages 252–261, 2013.

[22] 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.

[23] Nitesh V. Chawla, Kevin W. Bowyer, Lawrence O. Hall, and W. Philip Kegelmeyer. Smote: Synthetic minority over-sampling technique. *J. Artif. Int. Res.*, 16(1):321–357, 2002.

[24] Mingming Chen and Yutao Ma. An empirical study on predicting defect numbers. In *Proceedings of the 27th International Conference on Software Engineering and Knowledge Engineering*, pages 397–402, 2015.

[25] B. Clark and D. Zubrow. In *Software Engineering Symposium*, pages 1–35, Carreige Mellon University, 2001.

[26] Daniel Conte de Leon and Jim Alves-Foss. Hidden implementation dependencies in high assurance and critical computing systems. *IEEE Trans. Softw. Eng.*, 32(10):790–811, October 2006.

[27] Gabriela Czibula, Iuliana M. Bocicor, and Istvan-Gergely Czibula. Temporal Ordering of Cancer Microarray Data through a Reinforcement Learning Based Approach. *PLoS ONE*, 8(4):e60883+, April 2013.

[28] Gabriela Czibula, Maria-Iuliana Bocicor, and Istvan Gergely Czibula. An experiment on protein structure prediction using reinforcement learning. *Studia Universitatis Babes-Bolyai Informatica*, LVI(1):25–34, 2011.

[29] Istvan Gergely Czibula, Gabriela Czibula, and Maria-Iuliana Bocicor. A reinforcement learning based framework for solving optimization problems. *Studia Universitatis Babes-Bolyai Informatica*, LVI(3):3–8, 2011.

[30] Istvan-Gergely Czibula, Gabriela Czibula, Zsuzsanna Marian, and Vlad-Sebastian Ionescu. A novel approach using self-organizing maps for detecting software faults. *Studies in Informatics and Control*, 25(2):0 – 20, 2016.

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

[32] D. Doval, S. Mancoridis, and B. S. Mitchell. Automatic clustering of software systems using a genetic algorithm. In *Proceedings of the Software Technology and Engineering Practice*. IEEE Computer Society, 1999.

[33] Stéphane Ducasse, Nicolas Anquetil, Usman Bhatti, and Andre Cavalcante Hora. Software metrics for package remodularization. Technical report, Institut National de Recherche en Informatique et en Automatique, 2011.

[34] 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.

[35] MarkJ. Embrechts, ChristopherJ. Gatti, Jonathan Linton, and Badrinath Roysam. Hierarchical clustering for large data sets. In Petia Georgieva, Lyudmila Mihaylova, and Lakhmi C Jain, editors, *Advances in Intelligent Signal Processing and Data Mining*, volume 410 of *Studies in Computational Intelligence*, pages 197–233. Springer Berlin Heidelberg, 2013.

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

[37] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[38] Harald Gall, Karin Hajek, and Mehdi Jazayeri. Detection of logical coupling based on product release history. In *Proceedings of the International Conference on Software Maintenance*, ICSM '98, pages 190–198, Washington, DC, USA, 1998. IEEE Computer Society.

[39] David Gray, David Bowes, Neil Davey, Yi Sun, and Bruce Christianson. The misuse of the nasa metrics data program data sets for automated software defect prediction. In *Proceedings of the Evaluation and Assesment in Software Engineering*, pages 96–103, 2011.

[40] A.A. Shahrjooi Haghighi, M. Abbasi Dezfuli, and S.M. Fakhrahmad. Applying mining schemes to software fault prediction: A proposed approach aimed at test cost reduction. In *Proceedings of the World Congress on Engineering 2012 Vol I*, WCE 2012,, pages 1–5, Washington, DC, USA, 2012. IEEE Computer Society.

[41] 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.

[42] Tracy Hall, Sarah Beecham, David Bowes, David Gray, and Steve Counsell. A systematic literature review on fault prediction performance in software engineering. *IEEE Transactions on Software Engineering*, 38(6):1276–1304, 2011.

[43] Jiawei Han. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[44] Mark Harman, Stephen Swift, and Kiarash Mahdavi. An empirical study of the robustness of two module clustering fitness functions. In *Proceedings of the 7th Annual Conference on Genetic and Evolutionary Computation*, GECCO '05, pages 1029–1036, 2005.

[45] A. Hasenfuss and Barbara Hammer. Relational topographic maps. In Michael R. Berthold, John Shawe-Taylor, and Nada Lavrac, editors, *Advances in Intelligent Data Analysis VII, Proceedings of the 7th International Symposium on Intelligent Data Analysis*, volume 4723. Springer, 2007.

[46] Richard J. Hathaway and James C. Bezdek. Nerf c-means: Non-Euclidean relational fuzzy clustering. *Pattern Recognition*, 27(3):429–437, 1994.

[47] http://www.omg.org/technology/documents/formal/uml.htm. UML webpage.

[48] Rui hua Chang, Xiaodong Mu, and Li Zhang. Software defect prediction using non-negative matrix factorization. *JSW*, 6(11):2114–2120, 2011.

[49] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: a review. *ACM Computing Surveys*, 31(3):264–323, 1999.

[50] C. Z. Janikow. Fuzzy decision trees: issues and methods. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 28(1):1–14, 1998.

[51] Yuan Jiang, Ming Li, and Zhi-Hua Zhou. Software defect detection with rocus. *J. Comput. Sci. Technol.*, 26(2):328–342, 2011.

[52] 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.

[53] M. Khalilia and M. Popescu. Fuzzy relational self-organizing maps. In *2012 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE)*, pages 1–6, June 2012.

[54] Taghi M. Khoshgoftaar, Yudong Xiao, and Kehan Gao. Software quality assessment using a multi-strategy classifier. *Information Sciences*, 259(0):555 – 570, 2014.

[55] 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.

[56] Sunghun Kim, Hongyu Zhang, Rongxin Wu, and Liang Gong. Dealing with noise in defect prediction. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 481–490, New York, NY, USA, 2011. ACM.

[57] Younghoon Kim, Kyuseok Shim, Min-Soeng Kim, and June Sup Lee. Dbcure-mr: An efficient density-based clustering algorithm for large data using mapreduce. *Inf. Syst.*, 42:15–35, 2014.

[58] Serkan Kirbas, Alper Sen, Bora Caglayan, Ayse Bener, and Rasim Mahmutogullari. The effect of evolutionary coupling on software defects: An industrial case study on a legacy system. In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '14, pages 6:1–6:7, New York, NY, USA, 2014. ACM.

[59] Frank Klawonn and Frank Hppner. What is fuzzy about fuzzy clustering? understanding and improving the concept of the fuzzifier. volume 2810 of *Lecture Notes in Computer Science*, pages 254–264. Springer, 2003.

[60] Ehsan Kouroshfar, Mehdi Mirakhorli, Hamid Bagheri, Lu Xiao, Sam Malek, and Yuanfang Cai. A study on the role of software architecture in the evolution and quality of software. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR '15, pages 246–257, Piscataway, NJ, USA, 2015. IEEE Press.

[61] 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.

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

[63] Peng Lei and Hu Zheng. Clustering properties of fuzzy kohonen's self-organizing feature maps. *Journal of Electronics*, 12(2):124 – 133, 1995.

[64] Wei Liu, Sanjay Chawla, David A. Cieslak, and Nitesh V. Chawla. N.: A robust decision tree algorithms for imbalanced data sets. In *In: Proceedings of the Tenth SIAM International Conference on Data Mining*, pages 766–777, 2010.

[65] Ruchika Malhotra. A defect prediction model for open source software. In *Proceedings of the World Congress on Engineering*, volume II, July 2012.

[66] Ruchika Malhotra. Comparative analysis of statistical and machine learning methods for predicting faulty modules. *Applied Soft Computing*, 21:286–297, 2014.

[67] Ruchika Malhotra. Comparative analysis of statistical and machine learning methods for predicting faulty modules. *Applied Soft Computing*, 21:286–297, 2014.

[68] Ali Safari Mamaghani and Mohammad Reza Meybodi. Clustering of software systems using new hybrid algorithms. In *Proceedings of the Ninth IEEE International Conference on Computer and Information Technology*, pages 20–25, 2009.

[69] S. Mancoridis, B. S. Mitchell, Y. Chen, and E. R. Gansner. Bunch: A clustering tool for the recovery and maintenance of software system structures. In *In Proceedings of the IEEE International Conference on Software Maintenance*, pages 50–59, 1999.

[70] Spiros Mancoridis, Brian S. Mitchell, C. Rorres, Yih-Farn Chen, and Emden R. Gansner. Using automatic clustering to produce high-level system organizations of source code. In *International Workshop on Program Comprehension*, pages 45–53, 1998.

[71] C. Manning and H. Schutze. *Foundation of statistical natural language processing*. MIT, 1999.

[72] Z. Marian, I.G. Mircea, I.G. Czibula, and G. Czibula. A novel approach for software defect prediction using fuzzy decision trees. page to be published, Timisoara, Romania, 2016. IEEE Computer Science.

[73] Zsuzsanna Marian. On evaluating the structure of software packages. *Studia Universitatis Babes-Bolyai Informatica*, LIX(1):58–70, 2014.

[74] Zsuzsanna Marian, Gabriela Czibula, and Istvan Gergely Czibula. Software packages refactoring using a hierarchical clustering-based approach. *Computing and Informatics*, under review:1–30, 2016.

[75] Zsuzsanna Marian, Gabriela Czibula, Istvan-Gergley Czibula, and Sergiu Sotoc. Software defect detection using self-organizing maps. *Studia Universitatis Babes-Bolyai, Informatica*, LX(2):55–69, 2015.

[76] Tim Menzies, Jeremy Greenwald, and Art Frank. Data mining static code attributes to learn defect predictors. *IEEE Transactions on Software Engineering*, 33(1):2–13, 2007.

[77] Brian S. Mitchell and Spiros Mancoridis. On the evaluation of the bunch search-based software modularization algorithm. *Soft Comput.*, 12(1):77–93, 2008.

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

[79] Ankur Moitra and Ryan O'Donnell. Pareto optimal solutions for smoothed analysts. In *Proceedings of the 43rd annual ACM symposium on Theory of computing*, STOC '11, pages 225–234, New York, NY, USA, 2011. ACM.

[80] Stefania Montani and Giorgio Leonardi. Retrieval and clustering for supporting business process adjustment and analysis. *Information Systems*, 40(0):128 – 141, 2014.

[81] Jaechang Nam and Sunghun Kim. Heterogeneous defect prediction. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 508–519. ACM, 2015.

[82] Ahmet Okutan and Olcay Taner Yildiz. Software defect prediction using bayesian networks. *Empirical Software Engineering*, 19(1):154–181, 2014.

[83] Orange data mining. http://orange.biolab.si/.

[84] Wei-Feng Pan, Bo Jiang, and Bing Li. Refactoring software packages via community detection in complex software networks. *International Journal of Automation and Computing*, 10(2):157–166, 2013.

[85] 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.

[86] Denys Poshyvanyk, Andrian Marcus, Rudolf Ferenc, and Tibor Gyimóthy. Using information retrieval based coupling measures for impact analysis. *Empirical Softw. Engg.*, 14(1):5–32, February 2009.

[87] D. M. W. Powers. Evaluation: From precision, recall and f-measure to roc., informedness, markedness & correlation. *Journal of Machine Learning Technologies*, 2(1):37–63, 2011.

[88] Kata Praditwong, Mark Harman, and Xin Yao. Software module clustering as a multi-objective search problem. *IEEE Transactions on Software Engineering*, 37(2):264–282, 2011.

[89] Danijel Radjenović, Marjan Heričko, Richard Torkar, and Aleš Živkovič. Software fault prediction metrics: A systematic literature review. *Information and Software Technology*, 55(8):1397–1418, 2013.

[90] Danijel Radjenovi, Marjan Heriko, Richard Torkar, and Ale ivkovi. Software fault prediction metrics: A systematic literature review. *Information and Software Technology*, 55(8):1397 – 1418, 2013.

[91] D. Rodríguez, R. Ruiz, J. C. Riquelme, and J. S. Aguilar-Ruiz. Searching for rules to detect defective modules: A subgroup discovery approach. *Inf. Sci.*, 191:14–30, May 2012.

[92] Santonu Sarkar, Avinash C. Kak, and Girish Maskeri Rama. Metrics for measuring the quality of modularization of large-scale object-oriented software. *IEEE Transactions on Software Engineering*, 34(5):700–720, 2008.

[93] Giuseppe Scanniello, Carmine Gravino, Andrian Marcus, and Tim Menzies. Class level fault prediction using software clustering. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Testing*, pages 640–645, 2013.

[94] Gabriela Serban, Alina Câmpan, and Istvan Gergely Czibula. A programming interface for finding relational association rules. *International Journal of Computers, Communications & Control*, I(S.):439–444, June 2006.

[95] Gabriela Serban and Istvn Gergely Czibula. Object-oriented software systems restructuring through clustering. In Leszek Rutkowski, Ryszard Tadeusiewicz, Lotfi A. Zadeh, and Jacek M. Zurada, editors, *ICAISC*, volume 5097 of *Lecture Notes in Computer Science*, pages 693–704. Springer, 2008.

[96] Tarcísio G. S.Filó, Mariza A. S. Bigonha, and Kecia A. M. Ferreira. A catalogue of thresholds for object-oriented software metrics. In *First International Conference on Advances and Trends in Software Engineering*, 2015.

[97] Satinder P. Singh and Richard S. Sutton. Reinforcement learning with replacing eligibility traces. *Mach. Learn.*, 22, January 1996.

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

[99] Basma Soua, Amel Borgi, and Moncef Tagina. An ensemble method for fuzzy rule-based classification systems. *Knowledge and Information Systems*, 36(2):385–410, 2013.

[100] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998.

[101] Eric Chen-Kuo Tsao, James C. Bezdek, and Nikhil R. Pal. Fuzzy kohonen clustering networks. *Pattern Recognition*, 27(5):757 – 764, 1994.

[102] A. Tucker, S. Swift, and X. Liu. Variable grouping in multivariate time series via correlation. *IEEE TSMC, Part B*, 31(2):235–245, 2001.

[103] M. Umanol, H. Okamoto, I. Hatono, H. Tamura, F. Kawachi, S. Umedzu, and J. Kinoshita. Fuzzy decision trees by fuzzy id3 algorithm and its application to diagnosis systems. In *Proceedings of the Third IEEE Conference on Fuzzy Systems, 1994. IEEE World Congress on Computational Intelligence.,*, pages 2113–2118 vol.3, 1994.

[104] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of Machine Learning Research*, 9:2579–2605, 2008.

[105] R. Vanciu and V. Rajlich. Hidden dependencies in software systems. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pages 1–10, Sept 2010.

[106] 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.

[107] Nitesh V.Chawla. *Data Mining and Knowledge Discovery Handbook*, chapter Data Mining for imbalanced datasets: an overview. Springer US, 2010.

[108] Oliver Vogel, Ingo Arnold, Arif Chughtai, and Timo Kehrer. *Software Architecture - A Comprehensive Framework and Guide for Practitioners.* Springer, 2011.

[109] Petri Vuorimaa. Fuzzy self-organizing map. *Fuzzy Sets and Systems*, 66:223–231, 1994.

[110] G. Wahba, Y. Lin, and H. Zhang. GACV for support vector machines, or, another way to look at margin-like quantities. *Advances in Large Margin classifiers*, pages 297–309, 2000.

[111] Gerhard Weiß, editor. *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*. MIT Press, Cambridge, MA, USA, 1999.

[112] Hong Yul Yang, Ewan Tempero, and Rebecca Berrigan. Detecting indirect coupling. In *Proceedings of the 2005 Australian Conference on Software Engineering*, ASWEC '05, pages 212–221, Washington, DC, USA, 2005. IEEE Computer Society.

[113] Xun Yi and Yanchun Zhang. Equally contributory privacy-preserving k-means clustering over vertically partitioned data. *Information Systems*, 38(1):97 – 107, 2013.

[114] Liguo Yu and Alok Mishra. Experience in predicting fault-prone software modules using complexity metrics. *Quality Technology & Quantitative Management*, 9(4):421–433, 2012.

[115] Zhifeng Yu and V. Rajlich. Hidden dependencies in program comprehension and change propagation. In *Program Comprehension, 2001. IWPC 2001. Proceedings. 9th International Workshop on*, pages 293–299, 2001.

[116] Yufei Yuan and Michael J. Shaw. Induction of fuzzy decision trees. *Fuzzy Sets Syst.*, 69(2):125–139, 1995.

[117] L.A. Zadeh. Fuzzy sets. *Information and Control*, 8(3):338 – 353, 1965.

[118] Kuan Zhang, David Lo, Ee-Peng Lim, and Philips Kokoh Prasetyo. Mining indirect antagonistic communities from social interactions. *Knowledge and Information Systems*, 5(3):553–583, 2013.

[119] Jun Zheng. Predicting software reliability with neural network ensembles. *Expert Systems with Applications*, 36(2, Part 1):2116 – 2122, 2009.